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 * 0.25 = 0.5
- 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!!!!!!!!

Better (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 (easy but not the best scheme)
  - 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 branches:
    0.25 * (0.7 * *delay* + 0.3 * 0) (e.g., if delay =2 (10), yields 0.35 (1.75))
  - The problem is that we are optimizing for the less frequent case!
  - Nonetheless 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 branches:
    0.25 * (0.7 * 1 + 0.3 * *delay*) = 0.33 (0.925) if delay = 2 (10)
  - The 1 is there because you need to compute the branch address

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 often exception conditions: mostly not taken
  - Contribution to CPI more complex to assess
    - Unconditional branches: 0.25 * 0.2 * 1 = 0.05
    - Conditional branches (assuming 100% BT and 50-50% for FNT)

    0.20 (0.33 * 1 + 0.66 * 0. 5* delay + 0.66 * 0.5 * 0) = 0.20 if delay =2 and = 0.73 if delay =10)

  (the first term corresponds to BT and the next two to FNT)
- Prediction based on opcode
  - A bit in the opcode for prediction T/NT (can be set after profiling the application)

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)
BPB improves only the T/T case

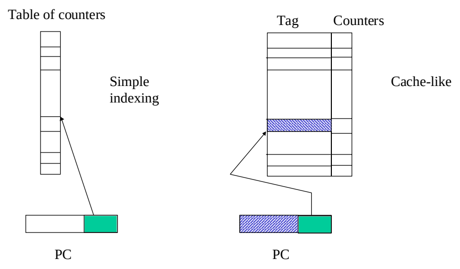
In what's below the number of stall cycles (*delay*) would be 2 or 3 for a simple pipe. It would be larger for deeper pipes.

- Case 1:
  - NT/NT 0 penalty
  - T/T need to compute address: 0 or 1 bubble
- Case 2
  - NT/T *delay* + 1?bubbles
  - T/NT *delay* + 1?bubbles
- Case 3:
  - NT/NT 0 penalty
- Case 4:
  - NT/T *delay* + 1bubbles

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)
  - BPT + 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

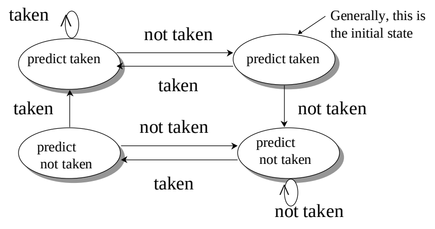


Simplest design

- 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 worthwhile

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 BTB)
    - Mispredict (incorrect branch prediction)
  - These penalties might need to be multiplied by the number of instructions that could have been issued

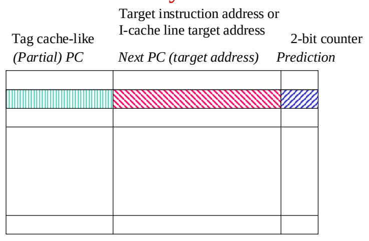
Prediction accuracy

- 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 512 to 1K entries
  - Larger associativity also improves accuracy (in general)
  - Why “in general” and not always? Some time “aliasing” is beneficial

Branch Target Buffers

- BPB: Tag + Prediction
- BTB: Tag + prediction + next address
- Now we predict and “precompute” branch outcome and target address during IF
  - Of course more costly
  - Can still be associated with cache line (UltraSparc)
  - Implemented in a straightforward way in Pentium; not so straightforward in Pentium Pro (see later)
  - Decoupling (see later) of BPB and BTB in Power PC and PA-8000
  - Entries put in BTB only on taken branches (small benefit)

BTB layout


Tag cache-like
(Partial) PC
Target instruction address or
I-cache line target address
Next PC (target address)
2-bit counter
Prediction

During IF, check if there is a hit in the BTB. If so, the instruction must be a branch and we can get the target address – if predicted taken – during IF. If correct, no stall

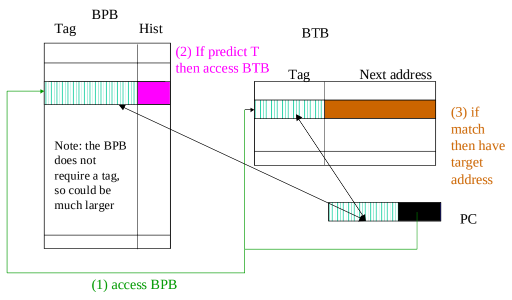
Another Form of Misprediction in BTB

- Correct “Taken” prediction but incorrect target address
- Can happen for “return” (but see later)
- Can happen for “indirect jumps” (rare but costly)
  - Might become more frequent in object-oriented programming a la C++

Decoupled BPB and BTB

- For a fixed real estate (i.e., fixed area on the chip):
  - Increasing the number of entries implies less bits for history or no field for target instruction or fewer bits for tags (more aliasing)
  - Increasing the number of entries implies better accuracy of prediction.
- Decoupled design
  - Separate – and different sizes – BPB and BTB
  - BPB. If it predicts *taken* then go to BTB (see next slide)
  - Power PC 620: 2K entries BPB + 256 entries BTB
  - HP PA-8000: 256*3 BPB + 32 (fully-associative) BTB

Decoupled BTB


BPB
Tag
Hist
(2) If predict T then access BTB
BTB
Tag
Next address
(3) if match then have target address
Note: the BPB does not require a tag, so could be much larger
PC
(1) access BPB

Correlated or 2-level branch prediction

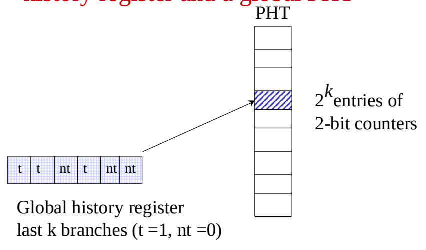
- Outcomes of consecutive branches are not independent
- Classical example

```
loop
 ….
   if ( x = = 2)                     /* branch b1 */
            x = 0;
   if ( y = = 2)                     /* branch b2 */
            y = 0;
   if ( x != y)                      /* branch b3 */
            do this
            else do that
```

What should a good predictor do?

- In previous example if both b1 and b2 are *Taken*, b3 should be *Not-Taken*
- A two-bit counter scheme cannot predict this behavior.
- Needs history of previous branches hence correlated schemes for BPB's
  – Requires history of *n* previous branches (shift register)
  – Use of this vector (maybe more than one) to index a Pattern History Table (PHT) (maybe more than one)

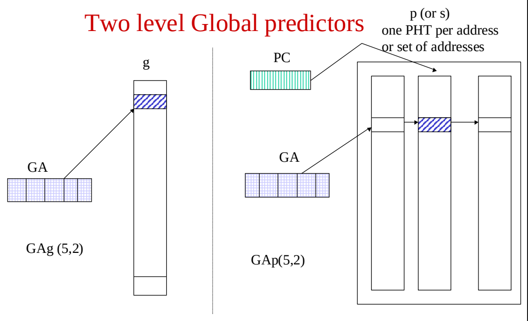
General idea: implementation using a global history register and a global PHT



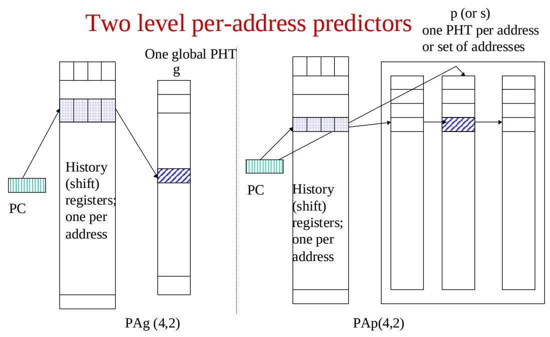
Classification of 2-level (correlated) branch predictors

- How many global registers and their length:
  – GA: Global (one)
  – PA: One per branch address (Local)
  – SA: Group several branch addresses
- How many PHT's:
  – g: Global (one)
  – p : One per branch address
  – s: Group several branch addresses
- Previous slide was GAg (6,2)
  – The "6" refers to the length of the global register
  – The "2" means we are using 2-bit counters

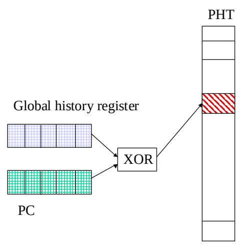
Two level Global predictors



Two level per-address predictors

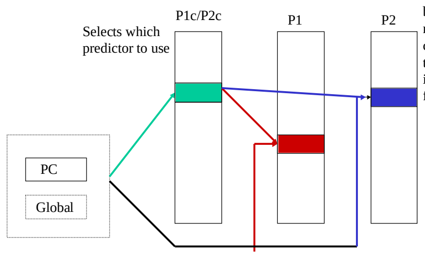

p (or s)
one PHT per address
or set of addresses
One global PHT
g
PC
History (shift) registers; one per address
PC
History (shift) registers; one per address
PAg (4,2)
PAp(4,2)

Gshare: a popular predictor



The Global history register and selected bits of the PC are XORed to provide the index in a single PHT

Hybrid Predictor (schematic)

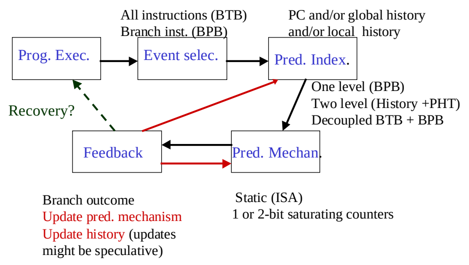


The green, red, and blue arrows might correspond to different indexing functions

Evaluation

- The more hardware (real estate) the better!
  - GA s for a given number of “s” the larger G the better; for a given “G” length, the larger the number of “s” the better.
- Note that the result of a branch might not be known when the GA (or PA) needs to be used again (because we might issue several instructions per cycle). It must be speculatively updated (and corrected if need be).
- Ditto for PHT but less in a hurry?

Summary: Anatomy of a Branch Predictor



Pentium Pro

- 512 4-way set-associative BTB
- 4 bits of branch history
- GAg (4+x,2) ?? Where do the extra bits come from in PC?

Return jump stack

- Indirect jumps difficult to predict except returns from procedures (but luckily returns are about 85% of indirect jumps)
- If returns are entered with their target address in BTB, most of the time it will be the wrong target
  - Procedures are called from many different locations
- Hence addition of a small “return stack”; 4 to 8 entries are enough (1 in MIPS R10000, 4 in Alpha 21064, 4 in Sparc64, 12 in Alpha 21164)
  - Checked during IF, in parallel with BTB.

Resume buffer

- In some “old” machines (e.g., IBM 360/91 circa 1967), branch prediction was implemented by fetching both paths (limited to 1 branch)
- Similar idea: “resume buffer” in MIPS R10000.
  - If branch predicted taken, it takes one cycle to compute and fetch the target
  - During that cycle save the Not-Taken sequential instruction in a buffer (4 entries of 4 instructions each).
  - If mispredict, reload from the “resume buffer” thus saving one cycle