Computer Design and Organization
Branch prediction

For the statement of the problem, i.e., why branches introduce bubbles in the pipeline and the performance consequences in RISC machines, see your textbook. With pipelines getting deeper (i.e., having more stages) and with the possibility of issuing several instructions in the same cycle, the performance penalty of either stalling the pipe(s) or recovering from a wrong prediction increases dramatically. Hence, the tremendous effort that has been invested in good predictive schemes (this effort is still continuing).

Many studies show the frequency of branches to be between 25 and 35%. In scientific programs most branches are backwards and taken (because of the importance of loops) while in integer type of applications, as exemplified by the Specint benchmarks and desktop applications, about 3/4 of the branches are conditional and, among those 2/3 are forward. Forward conditionals have about a 50-50 chance of being taken but with a bimodal distribution of almost always taken or almost always not taken, thus being amenable to predictive techniques. Backward branches are taken most of the time.

Besides avoiding some of the bubbles in the pipeline, a good prediction mechanism is needed to enhance instruction-level parallelism (ILP) in superscalar, out-of-order execution and VLIW machines (we'll define these terms later in the quarter). Any speculative execution scheme requires a sophisticated branch prediction mechanism if the extra resources are going to be used profitably. Good prediction should also allow intelligent prefetching in the I-cache.

The reduction of the CPI penalty caused by branches requires knowledge of one or more of the following:

- There is a branch
- Is the branch taken/not taken
- If taken what is the target address
- What is the instruction at the target address

Static techniques to reduce the branch penalty without extra hardware:

Some of these techniques can be the default for dynamic techniques or can be combined with the dynamic techniques.

1. *Assume branch not taken* (Result of the test and PC of the target should be computed as soon as possible to reduce penalty of misprediction). Rationale "easy". Note however that any prediction technique requires "squashing", i.e., cancelation of instructions in progress in case of a wrong prediction, thus introducing some complexity, or the introduction of no-ops, thus reducing the potential performance gain. The "branch not taken" assumption is a "good" method for CISC processors since squashing can be done early in the pipe. The danger of modifying state by previous instructions is thus minimized.
2. *Assume branch taken.* OK if one can compute the target address quickly, even before knowing the result of the test (can happen if tests are based on CC's – condition codes – set by many instructions). Rationale: branches are taken more often than not taken (between 60% and 70% of the time).

3. *Static prediction.* Various possibilities, e.g.: Backward taken/Forward not taken, BTFNT (used in the HP PA-RISC with insertion of a squashable instruction in the case of a misprediction; The Alpha 21064 uses it in conjunction with a dynamic 1-bit history table and a branch return stack –see below); One bit in instruction for taken/not taken set at compile-time; Prediction depending on opcode and direction (used in some IBM mainframes); Prediction depending on profiler.

4. *Branch delay slots filled by compiler.* This was very popular for early RISC designs (IBM 801, Berkeley RISC's, first generation MIPS). This is a fine technique if branch delays are small. Filling 1 slot can be done about 70- 80% of the time; filling more than one slot can be done rarely (e.g., less than 25% for the second slot), hence this method is not good for deep pipelines and superscalar processors.

Dynamic hardware prediction:

This type of prediction is present in all modern high-performance microprocessors.

1. *Branch prediction table – BPT* (1 bit/entry, 2 bits/entry, n bits/entry). An entry within a Branch Prediction Table is associated with branch instructions either in a table indexed by the PC (with possibility of hashing) or in a cache-like manner (i.e., with a tag identifying the static branch instruction). Depending on the implementation, an entry can be a shift register (1 to n bits) of the past history or a saturating counter. The 2 bits/entry saturating counter is the most popular design; it has been shown that higher values (3 or more bits/entry) do not improve the performance. Prediction success is of the order of 80-90%.

2. *Branch Target Buffers –BTB.* A BTB is an extension of a branch prediction table. It is organized like an I-cache but only instructions corresponding to branches are stored in the BTB. In addition to the tag, each entry has at least a prediction bit (in general, it's a 2-bit saturating counter), and the address of the target instruction when the branch is taken. In more expensive designs, the target instruction itself could be kept in the entry, thus saving an I-fetch stage in the pipeline (not implemented in any micro as far as I know). As with other cache-like structures, the hit rate in the BTB, and hence the prediction accuracy and/or performance enhancement, depends on usual parameters (size, associativity), context-switching occurrences etc.

3. *BTB within the I-cache.* Each line in the I-cache can hold a branch address and prediction bits. The advantage is that the address tag is shared and that it is easy to add the prediction hardware. The drawback is that only one branch/cache line can be accommodated (in an I-cache with 32 byte lines, i.e., about 8 instructions, the probability of having more than 1 branch is extremely high). Also there can be contention between the fetch unit and the branch unit that modifies history and target addresses.

4. *Branch return stacks.* When a target address is stored in the BTB, most of the "wrong" target addresses are caused by RETURN instructions. To avoid this, a call-return stack can be added to the BTB with a special bit/entry indicating that the stack should be used for address prediction.

5. *Two-level adaptive and correlated schemes.* In the simplest two-level scheme GAg the first level is the history of the last k branches encountered (e.g., a *G*lobal vector, or rather shift register, of k bits) and the second level is the branch behavior of the last s occurrences of the possible patterns (i.e., a *g*lobal pattern table of 2^k entries, each s-bit wide; in fact what is used at the second level is the 2-bit counter scheme or a slight variation of it). A more elaborate scheme PAg is to replace the shift register by a *P*er-branch-address table, with an entry (shift register) per branch (with usual replacement) and the most elaborate PAp is to consider a table of patterns per branch entry. Another possibility is to use a *G*lobal shift register and a table of patterns per branch entry; this could be called a GAp scheme. If there are too many branch addresses, then the pattern table

can be partitioned into sets GPs, i.e., a set consists of several branches determined by either their addresses or their opcode or other compiler generated attributes. This can also be done for the other schemes and therefore we have 9 potential designs $(G, P) \times (g, p, s)$. Depending on how much "real estate" (transistors) is devoted to branch prediction, prediction accuracies can reach the 94-98% range with these methods. As far as I know, only the GAg scheme has been implemented in modern microprocessors.

An interesting problem is how to update the various components of the adaptive schemes without slowing down the pipeline.

6. *Global history with index sharing.* The so-called *gshare* prediction scheme utilizes a one-level structure indexed by an XOR of some of the PC bits and some of the Global history register bits. It has been shown to be very good for large prediction tables. For smaller tables, just using the PC to index works better.

7. Hybrid predictors. Instead of using one predictor, use two. Then a saturating counter table, indexed by the PC, decides which of the 2 predictors to use for a given branch. Once the branch has been executed, each predictor is updated, and the selector is also updated (e.g., no change if both correct or incorrect, increment (i.e., give preference to) predictor 1 if only one correct else decrement (i.e, give preference to predictor 2).

Decoupling BPT and BTB

First check the BPT. If the prediction is for "taken" then look in BTB which, in this organization, does not need to store any prediction. If there is a hit, get the target address. Of course all that can be done in parallel, with a mux at the end delivering the goods.

Implementation in current microprocessors

- Pentium Pro: 512 4-way set-associative BTB; GAg(4+x,2). (I don't know where the "x" bits come from) Penalty for miss prediction could be up to 12 cycles.
- Pentium: 4-way set-associative 256 entry BTB; only "taken" branches are entered; uses a 2-bit counter for prediction. UltraSparc: 2 counters/cache line
- Alpha 21064: static prediction BTFNT plus 1-bit history table in the I-cache plus 4 deep branch return stack; looks like dynamic prediction is tested first unless indicated otherwise; penalty for misprediction is 4 or 5 cycles.
- Alpha 21164: 1 2-bit counter/instruction in I-cache. Pipeline slightly rearranged to decrease misprediction penalty by 1 cycle.
- Power PC: uses condition codes and static prediction – 1 bit of the instruction reserved to that effect; 2 cycle worst case penalty if wrong prediction; special registers in the branch unit for testing conditions.
- Power PC 620: direct-mapped BPT with 2K entries + BTB (don't know about size)
- MIPS (SGI) R10000: 512 entries BPT (direct-mapped)