New Algorithm Improves Branch Prediction

Better Accuracy Required for Highly Superscalar Designs

by Linley Gwennap

Intel's P6 processor *(see **090202.PDF**)* is the first to use a two-level branch-prediction algorithm to improve accuracy. This algorithm, first published by Tse-Yu Yeh and Yale Patt, has the potential to push accuracy well beyond the 90% level achieved by the best processors today. As future processors look to improve performance by increasing the issue rate and/or extending the pipeline depth, the two-level algorithm is likely to become more common.

Branch prediction has been a problem for CPU designers since the advent of pipelining. A pipelined processor must fetch the next instruction before the current one has executed. If the current instruction is a conditional branch, the processor must decide whether to fetch from the target address, assuming the branch will be taken, or from the next sequential address, assuming the branch will not be taken. An incorrect guess causes the pipeline to stall until it is refilled with valid instructions; this delay is called the mispredicted branch penalty.

Processors with a simple five-stage pipeline typically have a two-cycle branch penalty. For a four-way superscalar design, however, this could mean a loss of eight instructions. If the pipeline is extended, the branch penalty usually increases, resulting in the loss of even more instructions. Since programs typically encounter branches every 4–6 instructions, inaccurate branch prediction causes a severe performance degradation in highly superscalar or deeply pipelined designs.

Initial efforts at branch prediction used simple algorithms based on the direction of the branch. Among commercial microprocessors, the MIPS R6000 pioneered the use of compiler "hints" to direct branch prediction. Digital's 21064 was the first microprocessor to store branch history information, with the P6 leading the way to two-level prediction. This article reviews these earlier algorithms before explaining the new two-level method in more detail.

Simple Hardware Can Achieve 65%

For scalar processors with relatively short pipelines, branch prediction is less of a concern. In fact, for processors with a branch delay slot, the branch penalty can be as little as one cycle. The default "prediction" method for simple pipelined designs is to assume that branches are not taken, always fetching sequential instructions. The 486 and most embedded processors use this scheme because of its simplicity and low cost.

It turns out, however, that conditional branches are taken more often than not. Most programs make heavy use of loops, which repeatedly branch to the same address. Simulations show that conditional branches are taken about 60% of the time in the SPECint89 suite and more often in scientific code such as the SPECfp89 benchmarks[1]. Thus, a simple optimization is to always predict branches to be taken.

A better algorithm takes into account the direction of the branch. Backward branches typically complete loop iterations and thus are taken as much as 80% of the time or more. Forward branches are more difficult to predict but tend to be not taken more often than taken. Thus, by simply looking at the direction of the branch (usually available as the sign bit of the offset), a processor can predict backward branches taken and forward branches not taken. This BTFN algorithm succeeds about 65% of the time for SPECint89. MicroSparc-2 and most PA-RISC processors use BTFN.

With appropriate instruction-set hooks, the compiler can improve branch-prediction accuracy. Because it has access to the source code, a good compiler can recognize code sequences that are likely to branch, such as loops, and those that are unlikely to branch, such as exception checking. Current MIPS and PowerPC chips, among others, implement special branch instructions that encode the compiler's prediction in a single bit.

Compilers can take further advantage of these predicted branch instructions by using a technique called profiling or feedback-directed compilation. After the program is initially compiled, it is run using test data to determine the typical direction of each branch; the program is then recompiled to adjust the branch-prediction bits. According to IBM, its compilers achieve 75% accuracy on SPECint92 using this technique.

Dynamic Prediction Uses History

The previous algorithms are classified as static schemes, because any particular branch is always predicted in the same way whenever it is encountered. To achieve greater accuracy, dynamic algorithms take into account run-time information. The processor learns from its mistakes and changes its predictions to match the behavior of each particular branch.

A dynamic algorithm keeps a record of previous branch behavior, allowing it to improve its predictions over time. A simple scheme, published by James Smith in 1981[2], maintains a single history bit for each branch. When a branch is encountered, it is predicted to go the

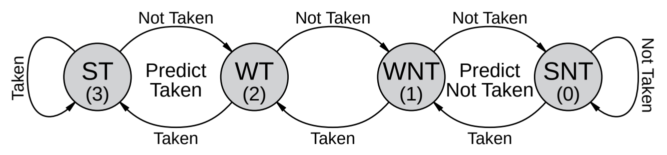

Figure 1. In the two-bit Smith algorithm, the two history bits implement a state machine with four possible states: strongly taken (ST), weakly taken (WT), weakly not taken (WNT), and strongly not taken (SNT). In ST and WT, future branches are predicted taken; in WNT and SNT, branches are predicted not taken.

same way it did the previous time, as indicated by the bit. This technique can push accuracy to 80%.

As a practical matter, there are two ways to implement this scheme. The history bits can be kept in the instruction cache, for example, one per every four instructions. When instructions are fetched from the cache, the history bit comes along. If the bit is set, that group of instructions contains a predicted-taken branch, and the fetch stream is redirected. In this example, the storage overhead would be less than 1% of the cache area.

Although this method—used by Digital's Alpha, AMD's K5, and other processors—provides dynamic prediction with minimal cost, it has some drawbacks. Some groups of instructions will not contain a branch, wasting the history bit. Groups with multiple branches create interference, as the history of one branch overwrites that of another in the same group.

Processors such as Pentium store the history bits in a separate branch history table (BHT), assigning one entry per branch. By avoiding the interference and unused bits of the previous scheme, the BHT offers improved accuracy. Alternatively, similar accuracy is achieved with fewer entries. The BHT, however, must maintain its own set of tags, greatly increasing the amount of storage required.

Given the overhead of tag storage, most processors with a separate BHT store two bits of history per entry instead of just one bit. In this method, also elucidated by Smith[2], the two bits can be thought of as a saturating counter that is incremented when the branch is taken and decremented when it is not; the most-significant bit is used to predict future occurrences. Another way to look at this implementation is as a state machine, which is depicted in Figure 1.

The advantage of the two-bit method is that a single unusual iteration will not change the predicted direction. For example, if a branch has been taken many times in succession, the state machine will be in the Strongly Taken state (3). If the branch is then not taken, the history bits will indicate Weakly Taken but still predict the next iteration as taken. Only if the branch is not taken two or more times consecutively will the prediction change to not taken. This hysteresis effect can boost prediction accuracy to 85% on SPECint92, depending on the size and type of history table that is used.

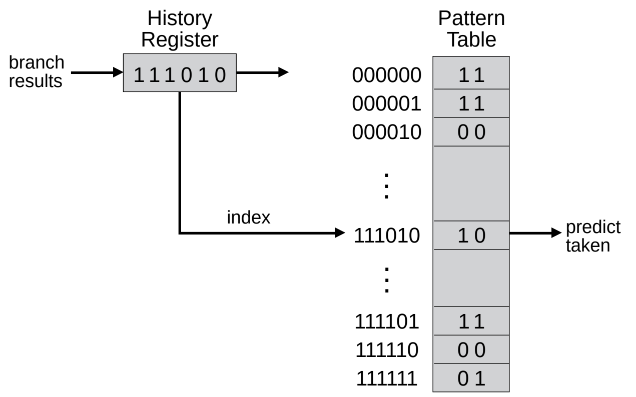

Figure 2. The two-level algorithm uses the contents of a branch history register as an index into a pattern table. Each table entry consists of a two-bit saturating up/down counter. The most-significant bit of the indicated table entry provides the branch prediction.

Two-Level Algorithm Improves Accuracy

Even at 85%, one out of every six branches is mispredicted, a rate that can significantly degrade performance in a highly superscalar design. Consider a four-way superscalar processor with a mispredicted branch penalty of three cycles. With one branch every five instructions, an accuracy of 85% creates a penalty of $(4 \times 3 \times 15\%) \div 5$, or 0.4 cycles per instruction (CPI). This penalty is a major factor for a processor with a peak throughput of four instructions per cycle (0.25 CPI).

For further improvement in prediction accuracy, Intel's P6 designers turned to the two-level algorithm developed by Yeh, a Ph.D. candidate, and Prof. Patt at the University of Michigan. The two researchers first published this technique in 1991[1] and continued refining it[3,4] until Yeh graduated in 1993; he is now employed by Intel on the P7 program.

A simplistic approach to improving the Smith algorithm is to increase the number of history bits beyond two using the same up/down counter. This approach retains more history information but does little to improve accuracy. In fact, it is actually more sluggish than the two-bit design when a branch changes from consistently taken to consistently not taken, or vice versa.

The two-level algorithm instead looks for patterns in an extended history register. For example, suppose a branch has been taken three times in a row, then not taken once. Will the branch return to "taken" on the next iteration, or will it now be consistently not taken? To resolve this issue, the algorithm looks at previous behavior when this same sequence has occurred, using historical data to generate a prediction.

Implementation of this algorithm requires two levels of storage, as Figure 2 shows. For each branch, a history register maintains the state of the last k branches. Unlike a saturating counter, this circuit is a shift register that represents taken branches with a 1 and not-taken branches with a 0. Each branch has a pattern table consisting of 2^k entries. Each entry implements a two-bit saturating up/down counter that tracks the results of previous iterations that occurred when the history register was in a given state.

When a branch is encountered, the contents of the history register are used to index into the pattern table, selecting the entry that corresponds to the recent history of that branch. As in the Smith algorithm, the two bits of that entry indicate the prediction. After the branch is resolved, the result (taken or not taken) is shifted into the history register and used to update the appropriate entry in the pattern table. To improve performance on tight loops, designs may use the prediction to speculatively update these fields, correcting them in the case of a misprediction.

Practicality Forces Simplification

As with simpler dynamic approaches, storing a unique history for every possible branch is impractical, so history is kept for only the most recent branches. The two-level approach, however, requires much more storage than the two-bit design: for each branch, there is a pattern table of 2×2^k bits, as Figure 3 shows. Even a relatively limited implementation with k=4 requires 18 times as much storage as the simple two-bit design.

To allow practical implementations with larger values of k, Yeh and Patt propose reducing the number of pattern tables by combining multiple branches in each table[4]. Branches can be grouped into sets based on their address (using a hashing algorithm), opcode, or some other characteristic. Branches in the same set use the same pattern table. In their taxonomy (see sidebar), Yeh and Patt call this the PAs algorithm, as opposed to the PAp method described above.

Although combining multiple branch histories causes some interference, this loss is compensated by the ability to implement larger pattern tables and thus larger history registers. For a given hardware cost, PAs delivers better accuracy than the full-blown PAp. Yeh and Patt simulated a variety of designs that all use about 8 Kbits of storage; at this size, the most accurate configuration is a $1{,}024 \times 6$-bit branch history table (i.e., k=6) with 16 pattern tables (each of 128 bits). Note that, in this case, the BHT consumes 75% of the storage budget, with the remainder for the pattern tables.

For further simplification, the BHT can be reduced to a single branch history register. This global resource simply tracks the results of the last k branches, regardless of their address. The pattern tables, however, can still be maintained on a per-address basis (GAp) or can be grouped by set (GAs). By eliminating the BHT, the width of the history register (k) and number of pattern tables can be increased within a fixed hardware budget.

Yeh and Patt's Taxonomy

In their paper[4], Yeh and Patt lay out a taxonomy of possible variations to the two-level algorithm, each assigned a three-letter code. The first letter indicates the type of history register (P, S, or G). The second letter is always "A" while the third letter represents the method of selecting a pattern table and is lower case (p, s, or g).

The complete algorithm uses per-address history register (P), that is, a separate history register for each branch. Alternatively, all branches can be combined into a single, global history register (G). An intermediate proposal, presented by Pan, So, and Rahmeh[5], groups branches into sets, with each set having its own history register (S).

Similarly, pattern tables can be assigned on a per-address basis (p) or by sets of branches (s). For ultimate simplification, a global pattern table (g) can be used.

For small implementations such as the 8-Kbit design above, the PAs method provides better results. With a budget of 128 Kbits, however, GAs is superior. Specifically, a GAs design with an 11-bit history register and 32 pattern tables (each of 2×2^{11}, or 4,096 bits) delivers the most accuracy within this storage budget[4].

The above analysis does not include the effect of context switches. The GAs approach has an advantage when context switches occur, because it takes fewer iterations to develop reliable history information from a single history register. This may give the global history register an advantage in real-world designs even for smaller implementations.

Target Addresses Must Also Be Predicted

Predicting whether the branch is taken is only half the battle; for seamless handling of taken branches, the processor must be able to immediately redirect the fetch stream to the target address. This feat can be tricky,

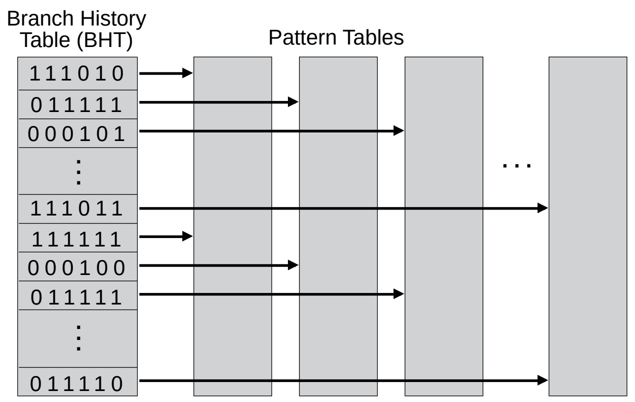

Figure 3. A branch history table consists of a number of entries, each with its own history register. Each history register may map to its own pattern table (PAp), or groups of entries may map to each pattern table (PAs), as shown here, reducing the number of tables.

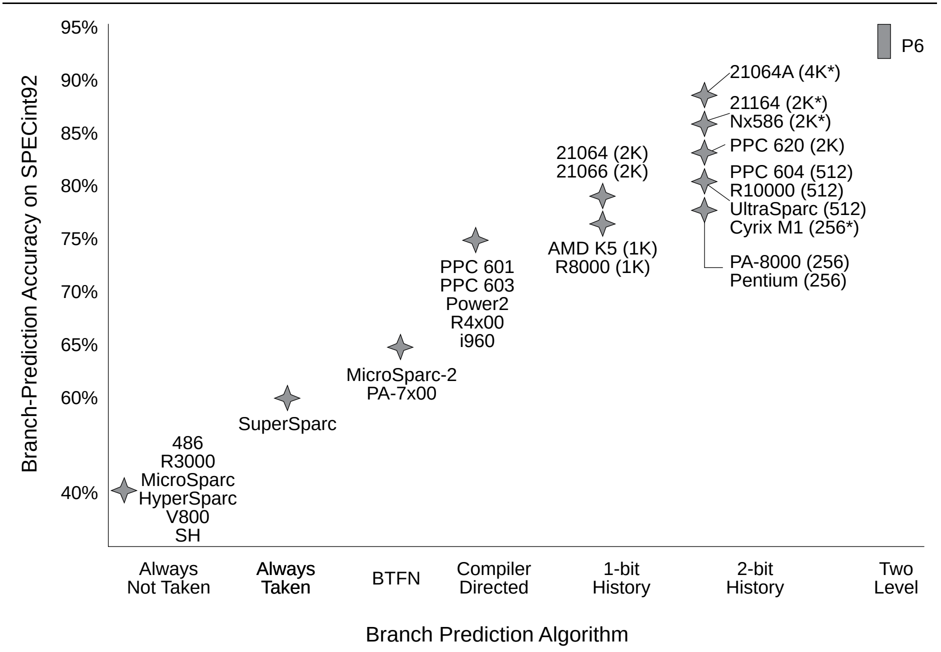

Figure 4. As processors use more complex algorithms, branch-prediction accuracy increases. (Number of history-table entries in parentheses.) *also uses return-address stack.

since typically the branch instruction is not fully decoded in time. If the processor takes an extra cycle to decode the branch and calculate the target, the instructions fetched during that cycle must be discarded if the branch is predicted taken.

Many processors use fetch queues to buffer instructions, hoping that a short stall can be absorbed by instructions already in the buffer. To achieve true zero-cycle branches, a few processors cache predicted target addresses. Most Pentium-class designs use a branch target address cache (BTAC) that contains predicted target addresses. The BTAC is accessed in parallel with the instruction fetch. If the BTAC indicates a branch, the predicted target address is used for the next instruction fetch, redirecting the fetch stream without penalty.

The BTAC is often combined with the BHT, forming a single structure that uses a single set of tags for both target addresses and history bits. This combination is often called a branch target buffer (BTB).

Current BTAC sizes are typically 256 or 512 entries, smaller than typical instruction caches, so addresses that hit in the cache may not be present in the BTAC. If an address misses the BTAC, the instructions at that address may still contain a taken branch; this fact is detected later in the pipeline, when the instructions are decoded. Such a branch will cause a penalty of one or more cycles. Like any cache, the BTAC can be increased in size or associativity to reduce the miss rate.

No matter how large the BTAC is, some branch targets are difficult to predict. Subroutine returns and other register-based branches do not have a fixed target but instead use the contents of a register to determine their destination. To handle these instructions, some processors use special storage for return addresses.

When a subroutine is called, processors such as Cyrix's M1 and all Alpha chips save the return address on a special stack. When the subroutine later returns, this address is taken from this stack and used to redirect the fetch stream, avoiding the need to wait for the return instruction to actually execute. If subroutine calls are nested more deeply than the number of entries in this address stack (typically four), this method cannot provide the correct target address for subroutine returns beyond this limit.

One BTAC variation is to cache target instructions instead of addresses. AMD's early 29K processors, as well as NexGen's 586, have a branch target cache (BTC) to store the first several instructions located at various target addresses. When a branch is encountered that hits in the BTC, the processor begins fetching instructions from the BTC while redirecting the main instruction cache to the new address. Each BTC entry must hold enough instructions to feed the CPU until the main cache can resume supplying instructions.

The BTC technique is useful when the main instruction cache has a latency of more than one cycle, as in the 586, or if there is no other instruction cache, as in the 29K. Like the 29K, some modern processors are connected to a high-bandwidth memory that has a latency of more than one cycle; sequential instructions can be easily prefetched, but taken branches are a problem. Processors such as Digital's 21164 and Hal's Sparc64 use a small (4K–8K) primary instruction cache to provide single-cycle response to branches. In these situations, a BTC can be more effective than a standard instruction cache if only 1K–2K of storage is available.

Alternative Designs Ease Branching

One method of avoiding branch prediction is to follow both the taken and not taken paths simultaneously, delivering 100% prediction accuracy. The only commercial microprocessor to implement this strategy is SuperSparc, which has enough instruction-cache bandwidth to fetch from two streams (on alternate cycles). SuperSparc also has a relatively short pipeline, only four stages, and can resolve pending branches before unneeded instructions are executed.

Few processors have the cache bandwidth to fetch from multiple streams. Furthermore, most processors have longer pipelines, raising the possibility that a second branch could be encountered before the current branch is resolved. In this case, the processor would have to speculate down four (or more) paths at once, a seemingly impractical task.

Regardless of the prediction strategy, keeping the pipeline to a minimal length improves branch performance. If branches are resolved quickly, the percentage of mispredictions is less important. The R10000, for example, uses a six-stage pipeline to keep the mispredicted branch penalty to two cycles, improving performance on code with many branches or with branches that are difficult to predict.

In many advanced processors, however, extra pipeline stages are required to issue several instructions per cycle. As clock speeds increase, some vendors are extending the pipeline to allow multicycle caches. In decoupled designs, branches can stall in execution queues, further increasing the potential branch penalty. These issues all force processor designers to seek improved branch prediction.

P6 Implementation Remains Mysterious

Intel's P6 uses a 12-stage pipeline with a seven-cycle misprediction penalty, making accurate branch prediction crucial to achieving high performance. Intel has not specified the details of its two-level prediction implementation other than to describe the size of the BTB as 512×4 bits. The P6 also has a four-entry return-address stack.

With only 4 bits per entry, the P6 could not implement a two-level algorithm: even with a 2-bit history register, there are not enough bits left for the pattern tables. This description could refer to the first level in the Yeh and Patt scheme, that is, a 512-entry BHT with 4-bit history registers (k=4). With the PAp method, the second level would consist of 512 pattern tables, each with 16 entries of 2 bits each.

The total storage requirement of this BHT would be 36 bits per entry, or 18 Kbits total. Since each BTB entry also contains a 24-bit tag and a 32-bit predicted target address, this added storage requirement would not be onerous. With such a small k value, there is little need for a more compact PAs design.

The total storage for this type of BTB, including the tags and target addresses, is nearly 6 Kbytes. In addition, the two-level BHT requires significant control logic to make predictions, further increasing the die area required. Intel notes that it could have increased the P6 instruction cache to 16K had it used a simpler BTB, but the designers found that the increased performance from more accurate branch prediction more than compensated for the reduction to an 8K cache size.

For older processors, devoting 8K of cache to branch prediction was impractical, considering that it would have reduced the on-chip cache to zero in some designs. But as transistor budgets continue to grow, devoting this amount of the die to branch prediction is quite feasible. When performance demands accurate branch prediction, this large amount of logic is required.

Two-Level Prediction Is Very Accurate

Yeh and Patt found that a two-level BTB similar to the one postulated for the P6 will correctly predict the direction of about 96% of all conditional branches in the SPECint89 suite. This figure does not include mispredicted target addresses, nor does it include the effect of context switches. The P6's branch-target buffer and return-address stack should deliver high accuracy on target addresses as well.

Intel would not quote branch-prediction accuracy for the P6; based on the Yeh and Patt study, we expect that it will achieve 90% to 95% accuracy on programs such as SPECint92 and do even better on SPECfp92. Many real-world applications, however, are notoriously difficult to predict with such accuracy. All processors achieve lower accuracy on these applications, but the P6 will have a greater performance degradation than many other processors due to its longer branch penalty.

As Figure 4 shows, the P6 has the most extensive branch-prediction hardware of any announced microprocessor and is the first to implement the two-level prediction algorithm. Just as Alpha's debut of dynamic branch prediction foreshadowed extensive use of that algorithm in other designs, we expect Yeh and Patt's two-level method to be adopted in future microprocessors, both x86 and RISC, as vendors continue the quest for greater performance. ♦

[1] T. Yeh and Y. Patt, "Two-Level Adaptive Branch Prediction," *24th International Symposium on Microarchitecture* (Nov. 1991), pp. 51–61.

[2] J. Smith, "A Study of Branch Prediction Strategies," *8th International Symposium on Computer Architecture* (May 1981), pp. 135–148.

[3] T. Yeh and Y. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction," *19th International Symposium on Computer Architecture* (May 1992), pp. 124–134.

[4] T. Yeh and Y. Patt, "A Comparison of Dynamic Branch Predictors That Use Two Levels of Branch History," *20th International Symposium on Computer Architecture* (May 1993), pp 257–266.

[5] S. Pan, K. So, and J. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," *5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Oct. 1992), pp. 76–84.