Branch prediction in the Pentium family

How the branch prediction mechanism in the Pentium has bee uncovered with all its quirks, and the incredibly more effective br prediction in the later versions.

By Agner Fog

What is branch prediction?

Imagine a simple microprocessor where all instructions are handled in two steps: decoding and executio microprocessor can save time by decoding one instruction while the preceding instruction is executing. ' assembly line-principle is called pipelining. In advanced microprocessors, the pipeline may have many s that many consecutive instructions are underway in the assembly line at the same time, one at each stage pipeline.

The problem now occurs when we meet a branch instruction. A branch instruction is the implementation if-then-else construct. If a condition is true then jump to some other location; if false then continue with instruction. This gives a break in the flow of instructions through the pipeline because the processor doe which instruction comes next until it has finished executing the branch instruction. The longer the pipeli longer time it will have to wait until it knows which instruction to feed next into the pipeline. As moder microprocessors tend to have longer and longer pipelines, there has been a growing need for doing some about this problem.

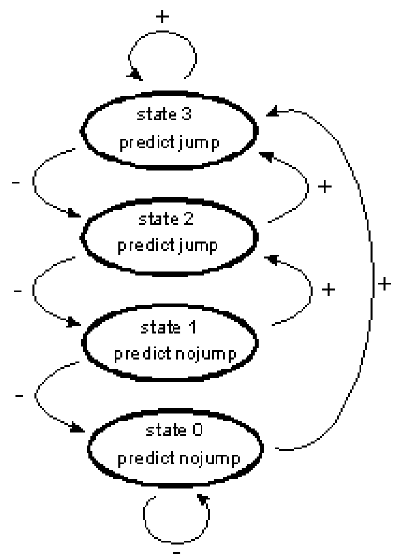
The solution is branch prediction. The microprocessor tries to predict whether the branch instruction wil not, based on a record of what this branch has done previously. If it has jumped the last four times then are high that it will also jump this time. The microprocessor decides which instruction to load next into pipeline based on this prediction, before it knows for sure. This is called speculative execution. If the pr turns out to be wrong, then it has to flush the pipeline and discard all calculations that were based on thi prediction. But if the prediction was correct, then it has saved a lot of time.

The detective work

Intel manuals have never been very specific about how the branch prediction works. However, since mispredictions are expensive in terms of execution time, I found it important to know how the predictio order to optimize my programs. I started to do a lot of experiments together with some clever persons I on the Internet, most importantly Karki J. Bahadur at the university of Colorado, and Terje Mathisen in the guy who reverse engineered system software to find out how to get access to the performance monit counters on the Pentium chip. Well, my first finding was that the Pentium predicts a branch instruction t it has jumped any of the last two times. This fitted all my experiments, but Karki pointed out that a bran jumps every third time is predicted one time out of six, where, according to my first model, it should ne predicted correctly. Then followed a series of new experiments until Karki and I independently came ou same state diagram, shown in fig. 1a. While we agreed on this mechanism, we disagreed on the interpre in particular on why it was asymmetric. In the meantime, another guy had found an old article in Microp Report claiming that the mechanism was a symmetric one as illustrated in fig 1b. My opinion was that t designers had actually intended the mechanism to be as in fig. 1b and that the asymmetry was a bug. Bu and Terje maintained that there had to be an intention behind this asymmetry. It didn't convince them th demonstrated how the symmetric mechanism was superior to the asymmetric one in almost all cases.

Now I discovered a powerful tool to dig deeper into this mechanism. The Pentium has a set of test registers that make it possible to read or write directly into the area that holds the history information for all branches, the branch target buffer (BTB). I had found this information on the home page of another hacker, Christian Ludloff. His page was shut down (rumors say that this was due to pressure from Intel) but fortunately I had downloaded his page before it was too late. Having direct access to the BTB, I was able to see exactly what happened: When a branch does not have an entry in the BTB it is predicted to not jump. The first time it jumps it gets an entry in the BTB and immediately goes to state 3. The complication is that the designers have equated state 0 with 'vacant BTB entry'. This makes sense because state 0 is predicted to not jump anyway. But since it cannot distinguish between state 0 and a vacant BTB entry it will go to state 3 next time the branch jumps rather than to state 1. This is where the quirk comes from. Apparently, somebody at the design labs has done a lot of research to find a good branch prediction scheme, and then somebody else has messed it all up by letting state 0 mean vacant BTB entry without realizing the consequence. And the consequence is that a branch which seldom jumps will have three times as many mispredictions as it would with the symmetric design.

Figure 1 -- State diagram for branch pre mechanism

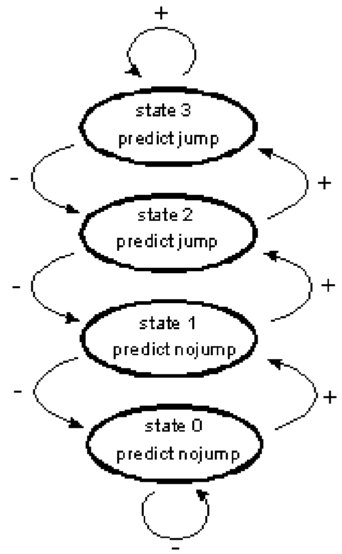


a. asymmetric design in the Pentium:

The state follows the +arrows when the brancl instruction jumps, and the -arrows when not ju The branch instruction is predicted to jump ne in state 2 or 3, and to not jump when in state (

mispredictions as it would with the symmetric design.

Karki and Terje were still not convinced that this design was a blunder. The convincing proof came when I discovered that tight loops behave differently. In a small loop the microprocessor doesn't have enough time to update the BTB entry for a branch instruction before it meets the same branch instruction again. In order to avoid a delay, it bypasses the BTB and reads the branch prediction state directly from the pipeline. And in this case it is actually able to go from state 0 to state 1, as in fig. 1b.



b. symmetric design:

This is how the branch prediction *should* worl state is incremented when jumping (+arrows) decremented when not jumping (-arrows).

More quirks

We soon found that there were more strange things about the Pentium's branch prediction. We couldn't sense of what happened when more branch instructions came close after one another. This time Karki aı came with the 'wild' ideas that led to the solution, while I played the role of the sceptic. After a hectic p where we exchanged results by E-mail every day, we found that the BTB information may actually be s several instructions ahead of the branch it refers to. If there happens to be another branch in between the BTB information is likely to be misapplied to somewhere in the wrong branch. This can lead to many fu phenomena: a branch instruction can have more than one BTB entry; two branches can share the same E so that one branch is predicted to go to the target of the other one; an unconditional jump instruction car predicted to not jump; and a non-jumping instruction can be predicted to jump. I will not go into detail v these quirks here, but you can find it all on my homepage. None of these quirks are fatal, because all mispredictions eventually get corrected.

A much more powerful mechanism

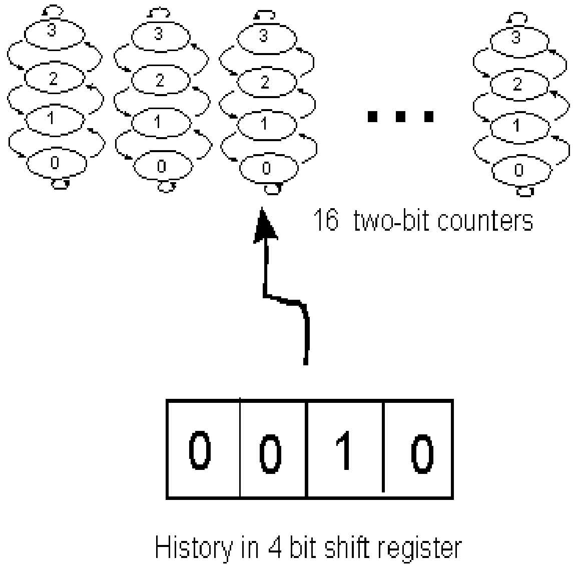
The later processors in the Pentium family: the Pentium MMX, Pentium Pro, Pentium II, Celeron, and X have a much more advanced branch prediction mechanism. I will refrain from more detective stories hei right to the mechanism.

This mechanism is based on the same fundamental idea of the state diagram in fig 1b. This is simply a two-bit counter with saturation. The counter is incremented

Figure 2 - Two level branch prediction in MMX, Pentium Pro, and Pentium

counter with saturation. The counter is incremented when jumping and decremented when not jumping. The branch instruction is predicted to jump next time if the counter is in state 2 or 3, and to not jump if in state 0 or 1. This mechanism makes sure that the branch has to deviate twice from what it does most of the time before the prediction changes.

The improvement in the later processors comes from the so-called two-level branch prediction. The first level is a shift register that stores the history of the last four events for any branch instruction. This gives sixteen possible bit patterns. You get a pattern of 0000 if the branch did not jump the last four times, and a pattern of 1111 after four times of jumping. The second level in the branch prediction mechanism is constituted of sixteen 2-bit counters of the type in fig. 1b. It uses the 4-bit pattern in the first level to choose which of the sixteen counters to use in the second level. See fig. 2.



Level two consists of 16 two-bit counters of tl fig. 1b. Level one is a four bit shift register stc history of the last four events. This four bit pa used to select which of the 16 two-bit counter: for the next prediction.

The advantage of this mechanism is that it can learn to recognize repetitive patterns. Imagine a branch tl every second time. You can write this pattern as 01010101 where 0 means no-jump and 1 means jump. . 0101 always comes an 0. Every times this happens, the counter with the binary number [0101] will be decremented until it reaches its lowest state. It has now learned that after 0101 comes an 0 and will ther make this prediction correctly the next time. Similarly, counter number [1010] will be incremented until three so that it will always predict a 1 after 1010. The remaining fourteen counters for this branch are ne as long as the pattern is the same.

This mechanism is quite powerful as it can handle complex repetitive patterns like 00101-00101-00101. can handle any repetitive pattern with a period of up to five, most patterns of period six and seven, and e patterns with periods as high as sixteen. To see if a pattern of period *n* can be handled without mispredic write down the *n* 4-bit sub-sequences in the pattern. If they are all different, then you will have no mispı after an initial learning time of two periods.

But the two-level mechanism is more powerful than that. It is also extremely good at handling *deviation* regular pattern. If a branch instruction has an *almost regular* pattern with occasional deviations, then the processor will soon learn what the deviations look like, so that it can handle almost any kind of recurren deviation with only one misprediction.

Furthermore, it can handle a situation where you alternate between two different repetitive patterns. Ass you have given the processor one repetitive pattern until it has learned to handle it without mispredictioı another pattern. And then return to the first pattern. If the two patterns do not have any 4-bit subsequenc common, then they do not use the same counters, so the processor doesn't have to re-learn the first patte Therefore, it can handle the transitions back and forth between the two patterns with a minimum of mispredictions.

Conclusion

The first microprocessor in the Pentium family introduced a simple one-level branch prediction mechan many ludicrous quirks. The later versions, Pentium MMX, Pentium Pro, Pentium II, etc. have longer pi and therefore a higher need for effective branch prediction. This need has been met by the incredibly po two-level mechanism with its ability to learn and recognize repetitive patterns and even deviations from regular patterns. This mechanism is also quite economical in terms of chip area as the history of a branc stored in only 32 bits.

The most important shortcoming of the two-level branch prediction is that it is not very good at predicti branch pattern of a loop control. If, for example, you have a program with a loop that always repeats ten then the control instruction at the bottom of the loop will branch back nine times and fall through the te the cost of one misprediction. For the Pentium Pro and Pentium II, where branch misprediction costs a l it may acually be advantageous to replace a loop that executes ten times with two nested loops that exec and two times, in order to avoid mispredictions.

Technical details can be found at my homepage: www.announce.com/agner/assem.

Back to Books and Articles home page
