where p_b denotes the probability that a given instruction is a branch, t_b represents the average number of cycles per branch instruction, and t_n represents the average number of cycles per nonbranch instruction.

The average number of cycles per branch instruction can be determined by considering two cases. If the target path is chosen, $1 + c$ cycles are needed for the execution; otherwise, there is no branch penalty and only one cycle is needed.

Thus,

$$t_b = p_t\,(1 + c) + (\,1 - p_t)\,(1) \tag{5.9}$$

where p_t denotes the probability that the t target path is chosen. The average number of cycles per nonbranch instruction is 1. After the pipeline becomes filled with instructions, a nonbranch instruction completes every cycle. Thus,

$$t_{avg} = p_b\,[p_t\,(1 + c) + (\,1 - p_t)\,(1)] + (1 - p_b) * (1) = 1 + p_b\,p_t\,c \tag{5.10}$$

After analyzing many practical cases, J. K. Lee and A. J. Smith have shown the average p_b to be approximately 0.1 to 0.3 and the average p_t to be approximately 0.6 to 0.7. Assuming that $p_b = 0.2$, $p_t = 0.65$, and $c = 3$, then:

$$t_{avg} = 1 + 0.2 * 0.65 * 3 = 1.39 \tag{5.11}$$

In other words, the pipeline operates at 100/1.39 = 72% of its maximum speed when branch instructions are considered.

Sometimes, the performance of a pipeline is represented in terms of throughput. The throughput H of a pipeline can also be expressed as the average number of instructions executed per clock cycle; thus:

$$H = \frac{1}{t_{avg}} = \frac{1}{1 + p_b\,p_t\,c} \tag{5.12}$$

To reduce the effect of branching on processor performance, several techniques have been proposed. Some of the better known techniques are branch prediction, delayed branching, and multiple prefetching. Each of these techniques is explained next.

5.4.6.2. Branch Prediction

With branch prediction, the result of a branch decision is predicted before the branch is actually executed. Based on the prediction, the sequential path or the target path is chosen for execution. Although by choosing the predicted path the branch penalty is often reduced, the penalty can be increased in case of incorrect prediction.

There are two types of predictions, static and dynamic. In ***static prediction***, a fixed decision is made before the program runs for prefetching the instructions from one of the two paths. For example, a simple technique would be to always assume that the branch is taken. This technique simply loads the program counter with the

target address when a branch is encountered. Another technique is to automatically choose one path (sequential or target) for some branch types and another for the rest of the branch types. If the chosen path is wrong, the pipeline is drained and the instructions corresponding to the correct path are fetched; the penalty is paid.

In *dynamic prediction*, during the execution of the program the processor makes a decision based on the information about the previously executed branches. The simplest method is the 1-bit dynamic prediction, which is implemented, for example, in the Alpha 21064 RISC processor. The idea behind this method is to assign a control bit p to a branch instruction when it is first executed. The CPU then will use the value of p to predict the branch; $p = 1$ means that the branch will be taken, and $p = 0$ means that the branch will not be taken. The prediction rule of this method is that the branch will be executed to the same instruction as it did last time it was executed. Thus when perform the iterations of a loop controlled by the branch instruction, once the loop is entered, p predicts that the same path will be followed each time this branch instruction is encountered. An erroneous prediction eventually results when the loop is exited, but the prediction will be correct most of the time. The result of evaluating the branch condition determines the next state of the p bit: p remains unchanged if this result agrees with the latest prediction made by p, and otherwise p is changed.

A better approach is to associate an n-bit counter with each branch instruction. This is known as the counter-based branch prediction approach. In this method, after executing a branch instruction for the first time, its counter C is set to a threshold T if the target path was taken, or T -1 if the sequential path was taken. From then on, whenever the branch instruction is about to be executed, if $C \geq T$, then the target path is taken; otherwise, the sequential path is taken. The counter value C is updated after the branch is resolved. If the correct path is the target path, the counter is incremented by 1; if not, the counter is decremented by 1. If C reaches $2^n - 1$ (the upper bound), C is no longer incremented, even if the target path was correctly predicted and chosen. Likewise, C is never decremented to a value less than 0.

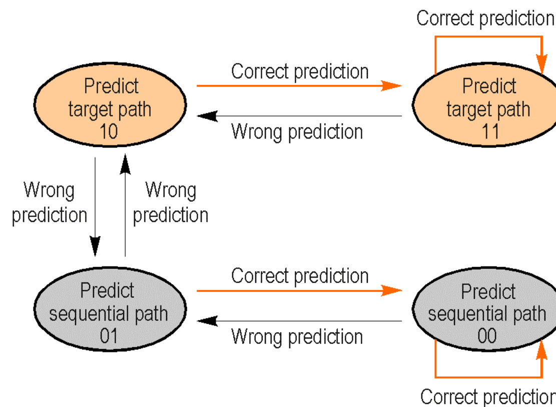

Figure 5.13. Possible states in a 2-bit predictor.

In practice, often n and T are chosen to be 2. Studies have shown that 2-bit predictors perform almost as well as predictors with more number of bits. Figure 5.13 represents the possible states in a 2-bit predictor.

Another variant of the preceding 2-bit predictor is to change the prediction only when the predicted path has been wrong for two consecutive times. Figure 5.14 represents the possible states for such a scheme.

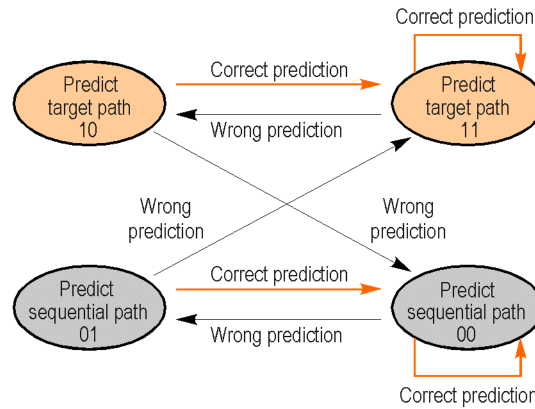

Figure 5.14. Possible states in an alternative 2-bit predictor.

It is convenient to store the branching statistics in a table called *branch history table*, along with the address of the branch instruction and the branch target address. For rapid access, most processors place the branch history table in a small size cache memory called *branch target buffer* (BTB). Often, each entry of this cache memory keeps a branch instruction's address, the branch target address and the prediction statistics (Figure 5.15).

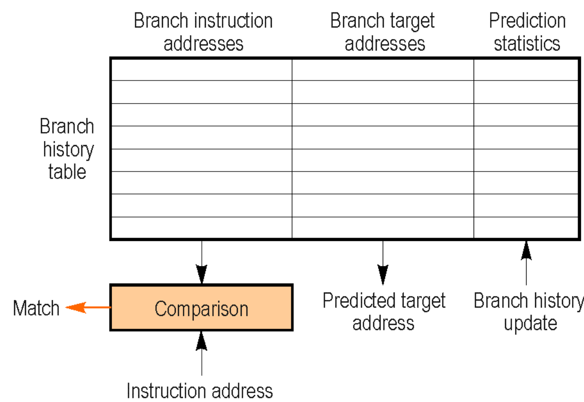

Figure 5.15. Organization of a branch target buffer.

When a branch instruction is first executed, the processor allocates an entry in the BTB for this instruction. Requests for instruction fetch are sent simultaneously to the instruction cache memory and the BTB. If a match is found in the BTB, the predicted branch target address is read from the table. Execution proceeds along the instruction path defined by the branch target address, with all results considered speculative until the result of the branch condition test becomes available. When execution of the branch instruction is completed, its target address is updated in the BTB. The branch instruction's prediction statistics are also updated.

Static prediction methods usually require little hardware, but they may increase the complexity of the compiler. In contrast, dynamic prediction methods increase the hardware complexity, but they require less work at compile time. In general, better results can be obtained using dynamic prediction. Dynamic prediction also provides a greater degree of object code compatibility, since decisions are made after compile time.

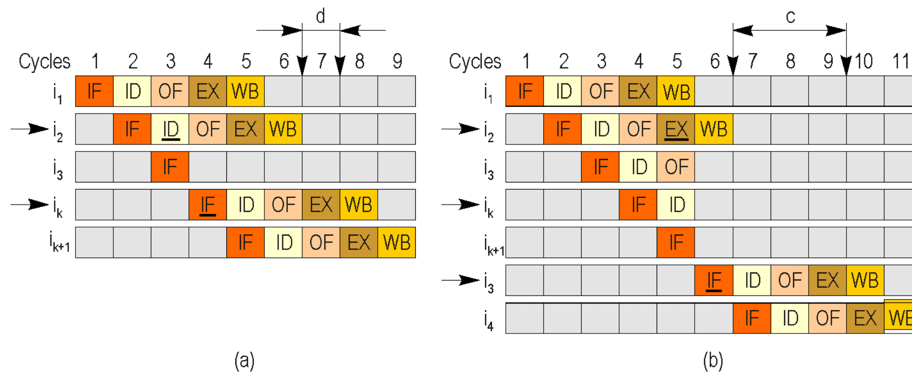

Figure 5.16. Branch penalties when the target path is predicted: (*a*) the penalty for a correctly chosen target path; (*b*) the penalty for an incorrectly chosen target path.

To find the effect of branch prediction on the performance, we need to re-evaluate the average number of clock cycles per branch instruction in Equation (5.8). There are two possible cases: the predicted path is either correct or incorrect. In the case of a correctly predicted path, the penalty is d when the path is the target path (Figure 5.16(a)), and the penalty is 0 when the path is the sequential path. In the case of an incorrectly predicted path for both target and sequential paths, the penalty is c (Figure 5.12 and Figure 5.16(b)). Note that, in Figure 5.16(a), the address of target path is obtained after the decode stage. However, when a branch target buffer is used, the target address can be obtained during or after the fetch stage.

Based on these observations, we have:

$$t_b = p_r\,[p_t\,(1 + d) + (1 - p_t)\,(1)] + (1 - p_r)[p_t\,(1 + c) + (1 - p_t)\,(1 + c)]$$

$$= p_r\,(1 + p_t\,d) + (1 - p_r)\,(1 + c) \tag{5.13}$$

where p_r is the probability of a right prediction. Substituting this term in Equation (5.8), we get:

$$t_{avg} = p_b\,[p_r\,(1 + p_t\,d) + (1 - p_r)\,(1 + c)] + (1 - p_b)\,(1)$$

$$= 1 + p_b\,c - p_b\,p_r\,c + p_b\,p_r\,p_t\,d \tag{5.14}$$

Assume that $p_b = 0.2$, $p_t = 0.65$, $p_r = 0.70$, $c = 3$, and $d = 1$. Then:

$$t_{avg} = 1.27 \tag{5.15}$$

That is, the pipeline operates at 100/1.27 = 78% of its maximum rate due to the branch prediction.

5.4.6.3. Delayed Branching

The delayed branching eliminates or significantly reduces the effect of the branch penalty. In this type of design, a certain number of instructions after the branch instruction is fetched and executed regardless of which path will be chosen for the branch. For example, a processor with a branch delay of k executes a path containing the next k sequential instructions, and then either continues on the same path or starts a new path from a new target address. As often as possible, the compiler tries to place after the branch k instructions that are independent from the branch instruction. If there are not sufficient instructions of this type, NOP instructions are inserted. As an example, consider the following code:

```
i1:    LOAD R1, A
i2:    LOAD R2, B
i3:    BRZ  R2, i7          ; branch to i7 if R2 = 0
i4:    LOAD R3, C
i5:    ADD  R4, R2, R3      ; R4 = R2 + R3
i6:    MUL  R5, R1, R2      ; R5 = R1 * R2
i7:    ADD  R4, R1, R2      ; R4 = R1 + R2
```

Assuming that $k = 2$, the compiler modifies this code by moving the instruction i_1 and inserting an NOP instruction after the branch instruction i_3. The modified code is:

```
i2:    LOAD R2, B
i3:    BRZ  R2, i7
i1:    LOAD R1, A
       NOP
i4:    LOAD R3, C
i5:    ADD  R4, R2, R3
i6:    MUL  R5, R1, R2
i7:    ADD  R4, R1, R2
```

As can be seen in the modified code, the instruction i_1 is executed regardless of the branch result.