

Periodic Adaptive Branch Prediction and its Application in Superscalar Processing in Prolog

RUEY-LIANG MA AND CHUNG-PING CHUNG

*Institute of Computer Science and Information Engineering, National Chiao Tung University,
Hsinchu, Taiwan 30050, ROC*

Branch instructions create barriers to instruction prefetching, greatly reducing the fine-grained parallelism of programs. Branch prediction is a common method for solving this problem. We first present four lemmata in this paper describing the relationships among branch prediction hit rate and system performance, hardware efficiency, and branch prediction overhead. We then propose a branch prediction method called PAM (Periodic Adaptive Method). An abstract model and detailed implementation of PAM are described. PAM's prediction hit rate as measured by 10 Prolog benchmark programs is 97%. When implemented in a superscalar Prolog system, PAM enhances the degree of system parallelism by 68.8%. PAM can be applied to languages and applications other than the Prolog system we used in this study.

Received July 27 1994, revised July 5 1995

1. INTRODUCTION

1.1. Why superscalar processing in Prolog

Clocksink and Mellish (1981) showed the potential usefulness of Prolog. Prolog can map complex problems into machine code more easily than conventional high-level languages, and is more suitable for processing the applications of fuzzy logic, natural language, theorem proving, expert system and logic inferencing. The execution of these application programs requires high system performance. There are various forms of coarse-grained parallelism in Prolog, generally classified as AND-parallelism (Degroot, 1987; Lin and Kumar, 1988) and OR-parallelism (Kale *et al.*, 1988; Kacsuk, 1994; Naganuma and Ogura, 1994). A more refined categorization is shown in Kale (1987). To exploit as much parallelism as possible for Prolog systems, some researches suggested the integrated AND-OR parallelism. Gupta and Jayaraman (1993) proposed a novel Prolog AND-OR model for shared memory multiprocessor system. This model extends the *binding arrays* method for OR parallel execution and exploits also the AND parallelism. Gupta and Costa (1994) then combined a new abstract model called *composition tree* with paged binding-arrays to develop an AND-OR model that supports full Prolog semantics. Ramkumar and Kale (1992a) proposed a binding environment for exploiting AND-OR parallelism that can be used for both shared memory and distributed multiprocessor system. A parallel compiler based on reduced-OR process model and the binding environment was also proposed in Ramkumar and Kale (1992b). Though lots of Prolog parallel models were proposed, the implementation and bookkeeping overhead still limited the parallel system's speedup. On the other hand, no matter what kinds of parallel models were used, the performance of the uniprocessor is always the key factor

in pursuing higher execution speed. Our research team hence devotes a significant portion of effort to study the design of a high performance Prolog superscalar processor. In this paper, we focus on studying the handling of branch instructions in a Prolog superscalar system and propose a new branch prediction method for solving this problem.

Superscalar processing has recently become a popular way to achieve high single-processor system performance. Various superscalar processors have been constructed to improve uniprocessor performance by exploiting instruction level parallelism. It has been noticed that instructional level parallelism can be combined with Prolog's traditional coarse-grained parallelism to get higher system performance. Several researchers, including Gloria and Faraboschi (1992), Su and Despain (1992), and our own team of Ma and Chung (1994) have designed Prolog superscalar processors.

In this paper, we study a fundamental design issue in superscalar processing in Prolog: branch handling for increasing fine-grained parallelism in Prolog program execution. The proposed scheme is general purpose, although we used a Prolog system as a vehicle in this research.

1.2. Parallelism limitation in Prolog superscalar processing and its solution

Several Prolog superscalar processors have been designed in recent years. Gloria and Faraboschi (1992) constructed a Prolog superscalar system called SYMBOL. Su and Despain (1992) also designed a Prolog superscalar system at the University of Southern California. These systems speed processing up by a factor of 2 over sequential systems. We designed a superscalar architecture (Ma and Chung, 1994) for our RISC Prolog machine, LISCP-II (Chi, 1990), that speeds up processing by a factor of 2.40.

This does not meet our goal of speeding a Prolog superscalar processor up by a factor of 3–4. After extensive simulation, we found that branch instructions were limiting the performance of our system.

Branch instructions are used heavily to implement the complex control behaviors, such as *unification* and *backtracking* of Prolog. The large number of *branches* greatly reduces the system performance. Unification is a critical function of Prolog; lots of dynamic data-type checkings are required in this function and these checkings are implemented as a *multi-way branch*, as will be discussed in Section 5.1. On the other hand, by applying global data flow analysis, Von Roy (1990) found that most complex backtrackings in Prolog are deterministic and they can be simplified as a conditional branch. Thus branches can be used to replace most of the backtracking operations in Prolog. Branch instructions stall the prefetching of instructions across basic blocks, which tend to greatly reduce the performance of pipelined processors. In a superscalar system, instructions are fetched, decoded, sent to an instruction issuing unit and then executed by functional units. Branch instructions, however, create barriers to instruction prefetching by changing the control flow of the program. As a result, the instruction fetch unit must remain idle until the branch condition is resolved. Since branch instructions account for 15–30% of the instructions in a program, the superscalar system instruction fetch unit may be halted frequently, greatly reducing system performance.

Branch prediction is a common method for solving this problem. Much research has been devoted to branch prediction and its effect on pipelined processors. In this paper, we intend to concentrate on the influence of branch prediction and its handling on a Prolog superscalar system. Although the result is indeed general purpose, we conduct this research in a Prolog environment.

1.3. Paper overview

Four lemmata are proposed in Section 3 that model how the branch prediction hit rate greatly influences superscalar system performance. We study the relationship between the branch prediction hit rate and parallelism in Prolog superscalar processing. We also investigate performance reductions due to branch prediction errors; these include recovery overhead, prediction mechanism overhead and system efficiency reduction.

With this background, we then propose a branch prediction method called PAM (Periodic Adaptive Method), which has satisfactory hardware cost and yields a very high prediction hit rate. Five branch patterns are observed from simulation to evaluate the performance of this methods and three existing prediction methods are surveyed.

In Prolog superscalar systems, branch prediction can also speed up the execution of *multi-way branch* and *dereference* instructions. Ten Prolog benchmarks are

used to evaluate the PAM in terms of parameters such as the branch prediction hit rate and performance gain. The results of these evaluations are discussed in detail.

In Section 2, we discuss the influence of branch prediction on superscalar system performance. In Section 3, we present four lemmata to model branch prediction and superscalar system performance. Section 4 presents an abstract model of and implementation techniques for the new prediction method PAM. Five branch patterns are proposed to evaluate PAM and three existing branch prediction schemes. Section 5 discusses some elementary operations in Prolog which can also benefit from the branch prediction mechanism and enhanced instruction level parallelism. Furthermore, the applicability of PAM to superscalar systems designed for other languages is discussed. In Section 6, a set of Prolog benchmarks is used to collect simulation data on PAM and three existing prediction methods. The simulation results show that PAM is a useful branch prediction scheme. The effect PAM has on instruction level parallelism is also discussed. Section 7 concludes the paper.

2. INFLUENCE OF BRANCH INSTRUCTIONS TO SUPERSCALAR SYSTEM PERFORMANCE

Through simulations Lam and Wilson (1992) examined the relationship between program parallelism and the number of instructions that can be issued simultaneously. When a branch prediction method is used, the instructions between a pair of mispredicted branches work like the instructions in a basic block and can be issued together, limited only by data dependencies. The number of instructions between a pair of mispredicted branches is called the misprediction distance. Table 1 shows the relationship between program parallelism and misprediction distance. For each misprediction distance measure, Lam and Wilson computed the harmonic mean of the parallelism. According to their simulations, if no branch prediction is applied, the basic block size (misprediction distance) is only 6.8. The instructions in these short segments tend to be closely related and have many data dependencies; the program parallelism is about 2.14. It is close to the 2.40 speedup we mentioned in Section 1. However, as shown in Table 1, with branch prediction, one can enlarge the misprediction distance and obtain a much higher degree of parallelism.

Johnson (1991) found a close relationship between program speedup and the number of instructions that can be issued together. Johnson used a decoder that decodes two to four instructions per cycle, and drew a graph showing the relationship between speedup and the

TABLE 1. Rough relation between issuing instruction and parallelism (Lam and Wilson 1992)

Misprediction distance	6.8	20	35	40–65
Parallelism	2.14	4	6	8

number of outstanding branches (branches that are decoded but not solved). He found that when the number of outstanding branches is four, the system is near maximum performance. That is, when the decoder rate is two or four instructions per cycle, the decoder can supply a sufficient number of instructions to keep the execution units busy. If, however, we constrain the decoder so that it stops upon encountering a branch instruction (number of outstanding branches is one; no branch prediction) or when less than four branches are encountered (number of outstanding branches is less than four), then the system will eventually be halted by this limitation and system speedup will be limited, because the decoder will no longer be able to supply enough instructions to keep the execution units busy.

In conclusion, branch prediction plays a very important role in the design of today's superscalar systems. A higher level of parallelism depends on the number of instructions that can be issued at the same time and, by relieving control dependencies, branch prediction increases the number of such instructions. Johnson showed that if we wish to issue many instructions together, we must use a decoder that is fast enough and is able to cross several levels of outstanding branches to obtain the maximum speedup. If the branch prediction scheme has a low hit rate, however, the decoder will occasionally decode instructions on mis-predicted paths and the effective degree of instruction parallelism will be low. Therefore, a high branch prediction hit rate is essential to a high performance superscalar system.

3. MATHEMATICAL MODELING AND PERFORMANCE PROJECTION

In this section, we will present four lemmata to clarify and model the relationships among the branch prediction hit rate and system performance, hardware efficiency and branch prediction overhead.

3.1. Model and analysis of relationship between branch prediction and fine-grained parallelism

We first formulate the relationship between the branch prediction hit rate and instruction level parallelism.

LEMMA 1. Assume that the size of a basic block is L , the branch prediction hit rate is r and the number of outstanding branches is K . Then the effective number of instructions that can be issued without control dependency, A , is:

$$A = \frac{1 - r^K}{1 - r} * L \quad \text{if } r < 1 \quad (\text{when } r = 1, A = K * L)$$

Proof. $r = 1$ means that the predictions are always correct. So when we advance by one level of outstanding branches, the effective number of instructions that can be issued without control dependency increases by L .

When r is less than 1, if no prediction is made (number of outstanding branches is 1; $K = 1$), the value of A is L . If the number of levels of basic blocks executable = 2 ($K = 2$), then the effective number of instructions increases by $r * L$, and the K th level of outstanding branches increases the effective instruction number by $r^{K-1} * L$. So if the number of outstanding branches is K , then the total number of effective issuable instructions A is

$$(1 + r + r^2 + \dots + r^{K-1}) * L = \frac{1 - r^K}{1 - r} * L \quad \square$$

The value of A greatly affects system parallelism, as shown in Table 1. If we want to obtain a large A , we must have large K and r . When the value of K increases by 1, the number of instructions in the instruction window increases by L , but only $r^{K-1} * L$ of the L instructions are effective.

The number of run-time active levels of basic blocks (the value of K) is limited by several factors, one very important one being the instruction-issuing mechanism. We assume that the superscalar system has a central window and that the system puts all the decoded instructions in the central window and issues instructions with no data dependency to the execution units. Let the number of entries in the central window (the number of instructions that can be put into the central window, or the size of the central window) be S_{CW} and let the maximum value of K be K_{MAX} . Then

$$S_{CW} \geq K * L, \quad K_{MAX} = \left\lfloor \frac{S_{CW}}{L} \right\rfloor$$

The maximum value of K is limited by the size of the central window, and the value of K at a certain time slice i is $K(i)$. We define the following notation:

$D(i)$: the number of instructions sent from the fetch and decoder units to the central window at time i .

$C(i)$: the instructions issued from the central window to the execution units at time slice i .

$S(i)$: the number of instructions remaining in the central window at time i .

Then

$$K(n) = \frac{\sum_{i=1}^n D(i) - \sum_{i=1}^n C(i)}{L} = \frac{S(n)}{L}$$

for $S(n) \leq S_{CW}$ and $K(n) \leq K_{MAX}$

From this equation, one can see that the size of the central window (S_{CW}), the decoder rate ($D(i)$), the number of different types of execution units (which affect $C(i)$), the data dependency of the program (which affects $C(i)$), and the size of basic blocks (L) all influence the value of $K(n)$ and thus K_{MAX} . An ideal condition is that if the decoder unit is fast enough, the

central window is always full, thus keeping the value of K at K_{MAX} .

LEMMA 2. *The efficiency of the instructions stored in the central window is:*

$$\frac{1 - r^K}{(1 - r) * K}$$

Proof. The effective number of instructions at time n is $A(n)$, the value of which is shown in Lemma 1 to be

$$A(n) = \frac{1 - r^{K(n)}}{1 - r} * L$$

The total number of instructions stored in the central window at time n is $S(n)$ and $S(n) = K(n) * L$. So the effective ratio of instructions is:

$$\frac{A(n)}{S(n)} = \frac{\frac{1 - r^{K(n)}}{1 - r} * L}{K(n) * L} = \frac{1 - r^{K(n)}}{(1 - r) * K(n)} \quad \square$$

Lemma 2 demonstrates the hardware efficiency of the superscalar system when K levels of branches are predicted. All of the $K(n) * L$ instructions are executed in the system, and only $A(n)/S(n)$ of them are effective. So the value of $A(n)/S(n)$ indicates the efficiency of the system hardware, as will be elaborated in the next section.

These lemmata show that the branch prediction hit rate r is very important in superscalar system design and performance enhancement. If we want to achieve the same value of A , as shown in Lemma 1 and Lemma 2, when the value of r is higher, the value of K can be lower. This means that if we devise a better branch prediction method, K can be smaller, i.e. the size of the central window (S_{CW}) and the decoder rate can be smaller. If K is fixed, then there is always the same number of instructions in the central window. In this case, a higher value of r will increase the effective number of instructions (the value of A) in the central window and also the program parallelism. Table 2 lists the impact of r and K on the value of A , as determined by Lemma 1. When the prediction hit rate is low, the increase in the value of A is small even in deep prediction. While when the prediction hit rate is high

TABLE 2. The impact of r and K on the effective number of instructions

r	K					
	1	2	3	8	10	∞
60%	L	$1.6L$	$1.96L$	$2.46L$	$2.48L$	$2.5L$
80%	L	$1.8L$	$2.44L$	$4.16L$	$4.46L$	$5L$
90%	L	$1.9L$	$2.71L$	$5.70L$	$6.5L$	$10L$
95%	L	$1.95L$	$2.85L$	$6.73L$	$8L$	$20L$
98%	L	$1.98L$	$2.94L$	$7.46L$	$9.15L$	$50L$
100%	L	$2L$	$3L$	$8L$	$10L$	∞

L is the size of the basic block.

(90 or 98%), the increase in the number of effective instructions is great.

3.2. Model of performance reduction due to branch misprediction

The first type of overhead incurred with branch prediction is recovery overhead caused by misprediction. After the execution of an outstanding branch, the system will check whether the prediction was erroneous; if an error is found, the system must halt for several cycles and activate the recovery mechanism to restore system status. The following lemma formally describes this overhead.

LEMMA 3. *Assume that the system recovery time of a branch misprediction is E . The average branch instruction recovery overhead (when the dynamic number of levels of execution is K) is:*

$$\frac{K - \left(\frac{1 - r^K}{1 - r} \right)}{K} * E$$

Proof. We first compute the branch instruction misprediction rate, then multiply it by the system recovery time E . The resulting value is the average branch instruction recovery overhead. When $K = 1$ (no branch prediction), the misprediction rate is 0; when $K = 2$, the rate is $1 - r$; for $K = 3$ it is $1 - r^2$; and for any K , the rate is $1 - r^{K-1}$. If we assume the system always executes K levels of code, the average branch misprediction rate is then

$$\frac{(1 - r^{K-1}) + \dots + (1 - r^2) + (1 - r) + (1 - 1)}{K} = \frac{K - \left(\frac{1 - r^K}{1 - r} \right)}{K}$$

So the average branch instruction recovery overhead is

$$\frac{K - \left(\frac{1 - r^K}{1 - r} \right)}{K} * E$$

and the system recovery overhead is

$$\frac{K - \left(\frac{1 - r^K}{1 - r} \right)}{K} * E * I_{\text{br}}$$

where I_{br} is the proportion of branch instructions. $\square$

LEMMA 4. *Assume that the branch penalty is P (the penalty depends on the system pipeline design; typical values are 1–3). The average cycle count of a branch instruction is:*

$$(P + 1) - P \left(\frac{r * (1 - r^K)}{K * (1 - r)} \right).$$

Proof.

Level of branch	1	2	3	4	K
Prediction hit rate	r	r^2	r^3	r^4	r^K
Misprediction rate	$(1-r)$	$(1-r^2)$	$(1-r^3)$	$(1-r^4)$	$\dots (1-r^K)$

This table shows the correct prediction and misprediction rates for each level of branch instruction. Suppose, as in Lemma 3, the program always executes K levels of code. Then the average branch instruction cycle count is:

$$\begin{aligned}
 & [\text{Average prediction hit rate} * 1] \\
 & + [\text{average misprediction rate} * (P+1)] \\
 & = \frac{r + r^2 + \dots + r^K}{K} * 1 \\
 & + \frac{(1-r) + (1-r^2) + \dots + (1-r^K)}{K} * (P+1) \\
 & = (P+1) - P \left(\frac{r * (1-r^K)}{K * (1-r)} \right) \quad \square
 \end{aligned}$$

The branch prediction increases the average cycle count of the branch instructions. As a reference, in a RISC system, this average cycle count is $r * 1 + (1-r) * (P+1) = (P+1) - P * r$.

Another type of overhead is incurred when branch prediction reduces hardware efficiency. As shown in Lemma 2, the hardware efficiency is reduced to $1 - r^K / (1-r) * K$. This is because mispredicted instructions are also executed in the system, thus also consuming registers, cache space and functional units.

3.3. Summary of the four mathematical lemmata

The information in Lemmata 1–4 concerning the effects of branch prediction on system performance is summarized in Table 3. Table 3 shows the effects of branch prediction when the prediction level increases from $K-1$ to K . Notice that the increase in recovery overhead should be equal to the increase in branch penalty, since these two parameters are the same and both are used to recover the system when a misprediction occurs. So the total overhead of a misprediction

TABLE 3. The influence of branch prediction from level $K-1$ to K

Performance factors	Equation
Increase in effective instructions	$\frac{L}{1-r} (r^{K-1} - r^K)$
Increase in system recovery overhead	$\frac{1 - (K * r^{K-1} - (K-1) * r^K)}{(1-r) * K * (K-1)} * E * I_{br}$
Branch penalty increase	$\frac{P * r}{(1-r)} * \left(\frac{1-r^{K-2}}{K-2} - \frac{1-r^{K-1}}{K-1} \right) * I_{br}$
Hardware efficiency reduction	$\frac{1 - (K * r^{K-1} - (K-1) * r^K)}{(1-r) * K * (K-1)}$

TABLE 4. The influence of branch prediction (r) at Level 6

Performance factors	Level					
	Total six levels			Only the sixth level		
	80%	90%	97%	80%	90%	95%
No. of effective instruction increase	22.14	28.11	33.41	1.97	3.54	5.15
Recovery overhead increase	19.3%	11.0%	3.6%	2.87%	1.9%	0.7%
H/W efficiency reduction	38.5%	21.9%	7.21%	5.74%	3.8%	1.4%
Performance	83.8%	90.1%	96.5%	97.2%	98.1%	99.3%

$$\text{Performance} = \left(\frac{1}{1 + \text{recovery overhead}} \right)$$

consists of the recovery overhead and the reduction in hardware efficiency.

3.4. An example

Table 4 shows the performance results when six branch levels are predicted. If the program's basic block size is 6, then its I_{br} is $1/6 = 16.7\%$. Suppose the value of E is three cycles (the same as in the IBM RS/6000 system) and the branch penalty is also three cycles. As shown in Table 4, branch prediction increases the number of effective instructions that can be issued to the central window, while, on the other hand, its recovery overhead also reduces the system performance, as we have pointed out. A high branch prediction hit rate shows its significance in these results: when all six levels of basic block are executed, the system performance reduction for $r = 97\%$ is about $1/3$ times $((100 - 96.5\%)/(100 - 90.1\%))$ the performance reduction of when $r = 90\%$ and $1/4.5$ times that when $r = 80\%$, and the number of effective instructions is increased by a factor of 1.2 and 1.5, respectively.

4. PAM FOR BRANCH PREDICTION

In this section, we will first present some interesting dynamic branch patterns we observed in our simulations and use these patterns to evaluate several branch prediction methods. The detailed simulation results are shown in Section 6. A new branch prediction method called PAM is also proposed, it can be used in Prolog and other languages to obtain high prediction hit rate.

4.1. Analysis of Prolog dynamic branch patterns

We collect all the dynamic execution results of a branch instruction to form a branch pattern, we then classify all possible dynamic branch patterns into five different categories. In the execution of a branch instruction, there are two possible results: either the branch is taken or it is not taken. In the following discussion, a taken branch is represented by a '1' and a not taken branch by a '0.'

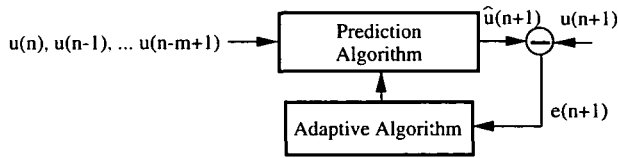


FIGURE 5. An adaptive model for branch prediction.

Another example is shown in Figure 4(2). The pattern is still irregular, although there are some locality characteristics in this pattern. By locality, we mean that some short, regular patterns may appear in the dynamic branches. In Figure 4(2), the patterns '111111' and '010101' occasionally appear in the branch. Thus we should still be able to predict the branch behavior using the locality characteristics.

4.2. PAM

4.2.1. The adaptive model and periodic characteristics for branch prediction

Figure 5 shows an abstract model of the adaptive method of branch prediction. $U(x)$ is the content of bit number x . M bits of branch history from the $u(n-m+1)$ th to the $u(n)$ th branch outcomes for the instruction are used in the prediction algorithm to obtain the prediction value $\hat{u}(n+1)$. When the branch condition is resolved, let the actual branch results be $u(n+1)$. The difference between the prediction and actual results is an error value $e(n+1)$, it is sent to an adaptation algorithm to modify the prediction method parameters. This model uses the difference between the predicted and actual results to guide modification. This procedure trains the prediction algorithm, adapting its parameters to produce increasingly more accurate results.

Figure 6 shows some periodic characteristics of the branches. Let us save the branch results in a shift register and assume that the n th bit of the shift register stores information that is n steps before the current input. For example, as shown in Figure 6(1), when the time stamp is 4, bit 3 stores the data at time stamp 1. So if the branch takes on a periodic pattern, a certain bit in the register will store the information that indicates the branch result. In Figure 6(2), the period of the pattern is 5, so bit 5 is always equal to the branch result and the result can be predicted reliably. Drawing on the adaptive and periodic characteristics of the algorithm and branch

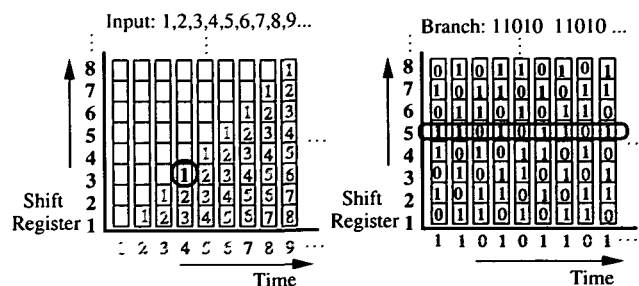


FIGURE 6. Snapshots of the shift register: (1) input is 1,2,3,...; (2) input is 11010....

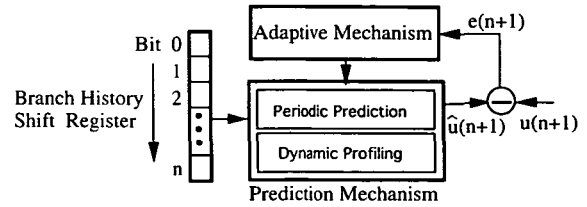


FIGURE 7. A periodic model for PAM.

pattern, we present in the next subsection a branch prediction method called PAM.

4.2.2. The abstract model of PAM

The prediction mechanism of PAM takes into account the periodic characteristic and the dynamic profiling, and an adaptive mechanism is used to modify the information kept in prediction mechanism. Figure 7 shows the PAM model derived from Figure 5. The history register is a shift register that shifts in bits representing recent branch history results. A boundary condition is used to determine whether periodic characteristic or dynamic profiling takes control:

1. If the branch history pattern is regular, periodic prediction method controls prediction.
2. If the pattern is irregular, dynamic profiling controls prediction.

After the branch condition is resolved, adaptive mechanism uses the actual branch result to modify the information in the prediction mechanism.

4.2.3. The design and implementation of PAM

The implementation of PAM is shown in Figure 8. There are four major components: the branch history register, the weight for each bit, the prediction mechanism (a compactor and a counter) and a result adaptation mechanism. When a branch is encountered, its recent history is loaded into the history register and the weights are loaded with the number of consecutive correct predictions for every register bit. The weight is a counter used to record the number of consecutive correct predictions of a certain bit. Each bit has its own weight. Thus a bit with the highest weight value has had the greatest number of consecutive correct predictions and we can use it to predict the next branch result.

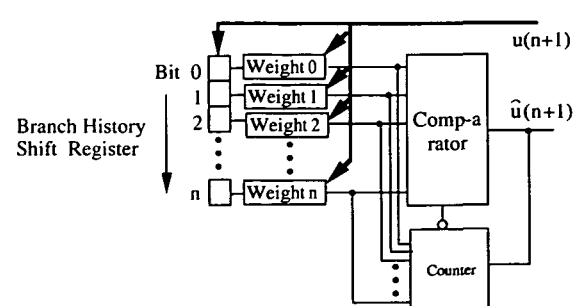


FIGURE 8. The implementation of PAM.

A boundary condition is then used to choose between periodic prediction or dynamic profiling. For example, if the boundary value is 3, then only weights with values greater than 3 can be used in prediction. If no weight is greater than 3, it means the branch history pattern is irregular, thus dynamic profiling is chosen and all register bits sent to the counter, where a majority of bit states provide the outcome prediction. On the other hand, if more than one of the weights has a value greater than the boundary value, the period-based prediction method is used. As illustrated in Figure 6(2), if the branch results show a regular pattern with a period of K , then bit K will always yield correct predictions. Thus the n -bit register and the n weights can be used to record this information from period = 1 to period = n and if the period K is less than n , then the branch prediction will always be correct. In this implementation, the weight values are sent to a comparator and the bit state with the greatest weight value is used to predict the incoming branch. So, if the pattern is predictable, the period-based prediction method is applied, and if the pattern is irregular, the dynamic statistical method of the n most recent results is used.

After the branch condition is resolved, the actual branch result is used to modify the value of the weight. The adaptive algorithm is based on the following rule: if the status of the selected bit is equal to the branch result, then the value of the corresponding weight is incremented; on the other hand, if the bit status is different from the branch result, the weight value will be reset to zero. These adaptive weight values are then used for future predictions.

4.2.4. Design parameters and execution algorithm of PAM

Next, we describe the PAM prediction scheme in detail, after first defining some system parameters:

1. *The number of history bits*: this number determines the hardware cost and the prediction hit rate, so it should be selected carefully.
2. *The boundary weight value*: if all weight values are less than this value, it indicates that the branch pattern is unpredictable, the dynamic profiling should be used.

The PAM algorithm may be stated as follows:

The prediction algorithm:

1. If no weight value is greater than the boundary value then count the number of 1's in all the branch history bit; $\geq n/2$ (n is the number of total history bits) then predict '1'; else predict '0'; or
2. Take the bit status of the bit with the highest weight value as the prediction.

The adaptation algorithm:

Compare the value of each bit in the branch history register against the actual branch outcome:

1. if the bit value is equal to the branch outcome then increment the weight value of this bit; or
2. reset the weight value to zero.

4.2.5. Evaluating PAM with the dynamic patterns

We first evaluate PAM with the five dynamic patterns proposed. For the *always* and *usually* branch patterns, PAM yields a prediction hit rate as high as that of any other method. For *loop* patterns, if the number of history bits is greater than the period of the pattern, then the predictions will be correct throughout; if it is less than the period, the prediction hit rate still satisfactory. Although *recursive* branch patterns, such as that shown in Section 4.1.4, do not repeat regularly, a period-like characteristic still exists, PAM's periodic prediction mechanism can be used to predict these branch patterns well. These results are further discussed in Section 6. Some *random* branch patterns, such as that shown in Figure 4(2), often consist of many short regular patterns. When a short regular pattern recurs, PAM can predict the next branch properly and the prediction hit rate is generally high. Finally, for *random* patterns with no recurring short sub-patterns, PAM can at least use dynamic profiling to predict the branch outcome, which seems to be the only useful method that applies in such cases.

4.3. Survey of several branch prediction methods

4.3.1. Profiling

Profiling is a widely-used static branch prediction method, in which the compiler pre-runs the program using typical input data, and after computing the number of taken and not-taken results, selects the more likely control path as the prediction value. Scientific programs have behave invariantly, so profiling is adequate. For input-data sensitive languages or applications, on the other hand, profiling does not perform satisfactorily. Prolog programs are non-deterministic and their dynamic behavior is difficult to profile, so a static method such as profiling cannot be usefully applied. So, profiling is not a good choice, if we want to obtain a very high prediction hit rate for most applications in Prolog superscalar systems.

4.3.2. The branch target buffer (BTB)

The BTB is a widely-used dynamic prediction method that employs a finite state machine to predict branching outcomes. Finite state machines generally cannot predict dynamic branch behavior well because their state transitions are fixed, making it difficult to predict all branches correctly. In typical simulations such as those by Lee and Smith (1984), BTB performance is very similar to that of profiling, so its prediction hit rate is not high enough for superscalar processing. BTB can predict loop control (the *loop* pattern) and branches that have locality, but in more complex situations, BTB is not

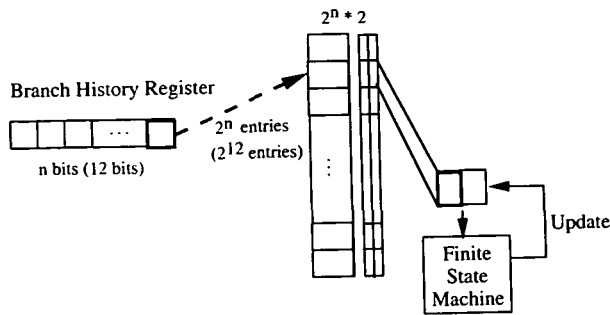


FIGURE 9. An implementation of TLABP.

smart or flexible enough. A more detailed evaluation of BTB in Prolog is presented in Section 6.

4.3.3. Two-level adaptive branch prediction (TLABP)

Yeh and Patt (1992) provided a new branch prediction method called TLABP. It is an enhanced dynamic method based on profiling and the BTB, which is complex and has a very high hardware cost, but which yields a high prediction hit rate. This method records the dynamic execution patterns of the branches, uses these branch patterns to index an entry in a pattern table, and then uses a BTB-like method to update the entry and predict the result. Thus when a branch pattern occurs again, the branch result can be predicted correctly. As shown in Figure 9, TLABP uses an n -bit branch history register (in the implementation shown here, $n = 12$) to record the historical pattern of a branch. All of the patterns can be mapped onto their corresponding locations in the pattern table (2^n entries $\times 2$ bits). Two bits of each entry are used to predict and update the branch result. When a branch instruction is encountered, the n -bit historical pattern will be used to address an entry in the pattern table, the 2-bit value of which is used for prediction and then,

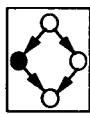
together with the executed branch result, computed in a 2-bit finite state machine to obtain a new prediction value for the next prediction. The implementation in Yeh and Patt (1992) used a large pattern table (2^{12} entries $\times 2$ bits) to obtain a 97% prediction hit rate for the SPEC benchmarks.

4.3.4. Predicated execution

Predicated execution (Pack and Schlansker, 1991) is a technique discussed widely in recent years for handling conditional branches. Its execution model is shown in Table 5. When a *branch* is encountered, predicated execution fetches and executes both possible execution paths. After the branch result is available, the instructions on the invalid path are squashed. To implement this model, *predicated* execution handles instructions based on the value of a Boolean operand called the predicate. This Boolean predicate is to be set or reset by its corresponding branch instruction. When the predicate is true, the associated instructions are executed normally; and when the predicate is false, the associated instructions are treated as a *no_op*.

Table 5 compares the predicated execution with branch prediction. We first compare the execution efficiency of these schemes. As shown in Table 5, predicated execution fetches all the alternative basic blocks to its system. Thus the effective ratio of instructions after an unsolved branch is 50% (because one of the two basic blocks is invalid). While branch prediction fetches only the basic blocks on the predicated path. With the same example, the effective ratio of instructions with branch prediction is r , where r is the branch prediction hit rate. The value of r can be at least 80, 90 or even up to 97% according to our simulation results in Section 6. So the execution efficiency of branch prediction is much higher than that of predicated execution for realistic prediction hit rates is

TABLE 5. Comparison of predicted execution and branch prediction

Factors	Method	
	predicated execution	branch prediction
Execution model	 ○ ● : basic blocks	
Instruction fetching	fetch all basic blocks	fetch predicted basic blocks
Hardware overhead	1. implement predicates 2. support predicated instructions	1. branch prediction mechanism 2. branch information storage
Execution efficiency	$\frac{\text{valid instructions}}{\text{total instructions}}$	$\frac{1 - r^K}{(1 - r) * K}$ (see Lemma 2)
Recovery overhead	(squash the invalid instructions)	$\left[K - \left(\frac{1 - r^K}{1 - r} \right) \right] / K$ (see Lemma 3)

high. Another difference between predicated execution and branch prediction is the recovery overhead. The prediction error incurs recovery overhead in branch prediction; Lemma 3 demonstrates that effect. As for the predicated execution, it squashes the invalid instructions and has no other recovery overhead. Branch prediction has a much higher execution efficiency but incurs extra recovery overhead than predicated execution, as shown in Table 5. If branch prediction has a high prediction rate r , its efficiency is enhanced and overhead is reduced, then it behaves better than predicated execution.

Another problem with predicated execution is the applicability. To implement predication execution, the system must redefine its instruction set. Special hardware and compiler supports are also required. IMPACT (Chang *et al.*, 1991), an experimental superscalar system developed at the University of Illinois, supports special hardware mechanisms for predicated execution. It proposes a 128×2 bits register file for predicates, 25 predicate instructions and a special compiler for this system. While branch prediction also needs extra hardware support, it can be applied in any processors without special compiler support. So branch prediction has a much higher applicability. However, due to the fundamental differences of predicated execution from those of dynamic branch prediction, we review and analyze predicated execution here only for the reference purposes. In the following, we will not discuss this method any further.

5. PAM FOR PROLOG AND OTHER LANGUAGES

5.1. Prolog characteristics and branch prediction

Different from other languages, Prolog has some special characteristics and features. Some instructions other than *branch* may create execution-flow barriers in a Prolog program, and branch prediction mechanism can also be used with them. In a Prolog machine, *multi-way branches* and *dereference* instructions can also benefit from this prediction mechanism.

Dynamic data-type checking is an important feature of Prolog. The data types of *variables* in Prolog are bounded at run time, thus frequent dynamic data-type checking is required. Warren (1983) designed a *switch_on_term* instruction to handle dynamic type checking. In our implementation, a *multi-way branch* instruction is defined for this purpose. Five different data types are defined in Prolog: constant, list, structure, unbounded variable and bounded variable. A *multi-way branch* instruction may branch to five different destinations depending on input-data type, thus creating a branch barrier. As the simulation results in Section 6. (Table 6) show, the *multi-way branch* count is almost the same as that of *branch*. Therefore, we should also apply branch prediction to *multi-way branch* instructions to eliminate the barriers created by *multi-way branches* and

enhance the instruction level parallelism of Prolog programs.

Another type of Prolog instruction that may alter execution flow and which can be predicted is *dereference* instructions. The data in Prolog programs are always represented by pointers, and before a data item is processed, it must first be dereferenced; if the data type is a bound variable, then the *dereference* instruction accesses the memory again to obtain its data type and real value. If a datum is not a bound variable, then the *dereference* instruction does nothing. Warren (1983) showed that these operations are frequently encountered in Prolog programs. Section 6 shows the enhancement in system parallelism gained by applying the prediction mechanism to *dereference* instructions. There is a fundamental difference in predicting *dereference* instruction outcomes, however. If an instruction is predicted to require no dereferencing, then the program continues executing the subsequent instructions, but if it is predicted to require de-referencing, then execution halts until the memory access has been completed.

PAM achieves high prediction hit rates when applied to *multi-way branches* and *dereference* instructions, and the performance gains from predicting these is significant. Related simulation results will be presented in Section 6.

5.2. Applying PAM to other languages

Different schemes employ different hypotheses, theoretical bases, and implementations to predict the outcome of a branch instruction. PAM's design is based on the periodic characteristics and dynamic profiling of branch instruction behavior. Any language (such as FORTRAN and C) having these characteristics can also benefit from PAM. In Section 4.2.5, we evaluated PAM with all of the dynamic branch patterns proposed. These dynamic branch patterns are also found in most languages, and PAM can work well in these languages:

1. Loop and recursion are common control structures in many computer languages. The branch history patterns of these control structures show a periodic characteristic, as discussed in Section 4.2.3, PAM chooses the bit with the largest weight to predict the branch outcome and will get high prediction rates with these branch history patterns.
2. As discussed in Section 4.1.1 and 4.1.2, *always* and *usually* patterns also appear frequently in branch instructions. This is due to exception test, data-type recognition, counter test, redundant branches, etc. PAM's prediction hit rate for these branch types is as high as those of other methods and are all over 97%.
3. If the branch pattern is not one of the four discussed above, then we say it exhibits no regularity at all. However, PAM can cope with even these irregular random patterns by applying adaptive dynamic profiling.

TABLE 6. Dynamic occurrence and occurrence ratio of branch and deference instructions

Benchmark	Execution cycles	Branch		Multi-way-branch		Deference (= 0)
		100%	97%	100%	97%	
Concate	3555	5.23%	4.36%	6.92%	1.74%	2.59%
Fact	121 626	5.94%	1.81%	8.32%	NA	4.32%
Hanoi	126 918	5.64%	1.61%	5.64%	NA	3.22%
Map	135 263	9.09%	6.82%	5.60%	4.16%	3.19%
Queen	182 372	8.92%	2.75%	7.77%	1.51%	3.52%
CKT	4 593 075	6.89%	2.27%	4.83%	1.53%	2.44%
DB	6 759 320	4.90%	4.47%	5.89%	NA	1.77%
EXPERT	4 668 867	4.14%	1.64%	7.53%	1.67%	3.70%
NLP	4 402 997	6.27%	1.54%	4.04%	NA	1.96%
PASER	7 388 366	6.18%	2.85%	7.05%	2.10%	4.04%
Average		6.32%	3.01%	6.36%	2.12%	3.08%
Total				15.76%		

Note: 100%, 97%: 100% includes the all branch instructions, and 97% includes only those branch instructions for which at least 3% of the results are irregular, as compared with the usual outcome.

Yeh and Patt (1992) evaluated profiling with SPEC benchmarks and showed an average hit rate of 91%. PAM also predict irregular branch patterns by applying dynamic profiling, thus its prediction rate is the same as that of profiling in such cases. While for the branch patterns that have periodic nature, the periodic prediction mechanism predicts more accurately than profiling. So PAM should outperform profiling when used with SPEC benchmark programs.

Base on the design philosophy and the evaluation patterns that we have discussed, the branch prediction method PAM can be applied to other computer languages.

6. PERFORMANCE EVALUATION OF BRANCH PREDICTION METHODS

6.1. Benchmark programs and the simulation environment

Ten benchmark programs are used which represent typical Prolog applications. The first five are CKT, a circuit simulator; DB, a database query program; EXPERT, an expert system program for automobile-diagnostics; NLP, a natural-language parsing program; and PARSER, the parser of our Prolog compiler. Concate, Fact, Hanoi, Queen and Map are the other five benchmark programs used in the Berkeley PLM and BAM benchmark sets.

Branch prediction results from the various methods are generated and collected by a Prolog compiler, a superscalar Prolog simulator *SPS* and a branch prediction simulator. These benchmark programs were first compiled by the compiler; their object codes were then executed on *SPS* to obtain their superscalar system performance. The dynamic branch behavior of all branch instructions were collected by *SPS*. These data were then simulated on a branch prediction simulator to get the branch prediction results.

6.2. Simulation parameters: the '100%' and '97%' branch patterns

Each program produces two sets of simulation results, the '100%' and the '97%' results. The first of these is the prediction hit rate for all the branch instructions, and the second counts only the prediction hit rates of those branches in which at least 3% of the branch results are irregular, as compared to the usual outcome. The reason for using the second set of data is explained below. As mentioned earlier, the *always* and *usually* branch patterns have extremely high prediction hit rates, and inclusion of these results in our analysis would blur the significance of this study. The prediction hit rates of the *always* and *usually* patterns are all the same in each method, so excluding these two patterns from the analysis allows us to clearly identify the prediction ability of each method.

Table 6 shows the number of dynamic occurrences and occurrence ratios of *branch* and *dereference* instructions. On average, 6.32% of the *branches*, 6.36% of the *multi-way branch* instructions and 3.08% of the *dereference* instructions do not access memory; 15.76% of the dynamic instructions created barriers to exploitation of fine-grained parallelism. If we apply an advanced prediction method, such as shown in Section 6.3, PAM correctly predicts about 97% of the *branch* and *dereference* instructions, which greatly reduces *branch* and *dereference* barriers. The '97%' dynamic counts of multi-way branches account for only about one-third of the total number of *multi-way branch* instructions and the '97%' dynamic branch counts account for only one-half of the total number of *branches*. That is, too many *always* and *usually* patterns are found in Prolog programs. Thus, the performance differences between the prediction methods may not seem significant. Nevertheless, a smart compiler can be designed to reduce the number of *always* patterns and the differences

TABLE 7. Branch prediction results and overhead of different methods

Benchmark	Method													
	Profiling		BTB		TLABP(12)		PAM(12)		PAM(18)		PAM(24)		PAM(48)	
	100%	97%	100%	97%	100%	97%	100%	97%	100%	97%	100%	97%	100%	97%
Concat	91.9	83.9	91.7	83.4	97.9	95.9	98.5	97.0	98.6	97.1	98.6	97.1	98.7	97.1
Fact	96.5	72.3	95.1	61.8	99.1	93.3	97.7	82.6	98.5	88.2	99.1	93.2	99.7	97.6
Hanoi	92.9	50.0	91.1	37.6	98.5	89.2	98.7	90.6	99.2	94.6	99.3	95.4	99.8	98.8
Map	84.8	80.0	82.5	76.9	89.6	86.4	82.7	77.3	86.8	83.0	89.7	86.6	90.0	89.0
Queen	91.7	67.7	90.2	62.7	97.6	91.0	95.2	82.3	95.3	82.9	95.5	83.5	96.4	86.7
CKT	89.8	68.8	88.6	65.9	96.9	91.0	93.6	80.6	93.6	80.6	93.9	87.3	93.9	87.1
DB	97.0	92.5	97.0	92.5	97.6	95.3	94.0	85.8	98.4	96.2	98.4	96.2	98.5	96.3
EXPERT	92.0	71.7	91.5	70.1	99.1	96.9	98.0	93.1	97.9	92.5	98.7	95.5	99.5	98.3
NLP	97.3	81.6	97.1	80.8	99.7	98.0	99.4	95.9	99.4	96.0	99.4	96.2	99.7	98.0
PASER	84.2	57.8	80.7	48.5	98.2	95.5	97.2	92.6	97.9	94.6	98.5	96.0	99.1	97.6
Average	91.8	72.6	90.5	68.0	97.4	93.2	95.5	87.8	96.5	90.5	97.1	92.7	97.6	94.7
Prediction mechanics	pre-execution		finite state machine		$O(2^n)$		$O(n)$							
Information storage	0		2–3		n		n							

PAM(N): N is the history register width.

100%, 97%: 100% includes the all branch instructions, and 97% includes only those branch instructions for which at least 3% of the results are irregular, as compared with the usual outcome.

n , $O(n)$ and $O(2^n)$: n is the number of history bits used in its branch mechanism.

in the prediction hit rates of the different methods will then become apparent.

6.3. Performance evaluation of branch prediction schemes

6.3.1. Prediction rate and overhead

Table 7 shows the simulation results for the different branch prediction methods. The prediction hit rates of TLABP(12) and PAM(24) (where 12 or 24 is the history register width) are both 97%, and those of profiling and BTB are only about 91%. If the branch prediction level is 6 and the size of an average basic block is 8, from Lemma 1, PAM(24) yields 44.5 effective instructions and BTB yields only 38.4 for the same architecture. The PAM(24) provides about 16% more effective instructions than profiling or BTB, which indicates that a better prediction method is important in a superscalar system design.

From the '97%' column of the table we observe a significant fact. If we ignore the *always* and *usually* patterns of all branches, TLABP(12) and PAM(24) still render high prediction hit rates of about 93%. By contrast, profiling and BTB, prediction hit rates are reduced to only 72.6 and 68%, respectively. So the prediction ability of these two methods is poor, and the overall high prediction hit rates of these two methods are due to an excessive number of *always* and *usually* branches.

Table 7 also shows the overheads for different schemes. PAM and TLABP are both advanced branch prediction schemes, and have high prediction accuracy; thus, the hardware cost is higher. On the other hand,

profiling and BTB are less sophisticated and less accurate. Some analysis is given below:

1. Profiling pre-executes the program, and predicts the execution path with higher hit probability for that branch instruction. So, profiling has a pre-execution overhead.
2. BTB, TLABP and PAM are dynamic branch prediction schemes; they all require two types of extra hardware: the information storage and branch prediction mechanism. These branch prediction schemes store several bits of information for each *branch*. This branch information is to be used by prediction mechanism to predict next branch result of that branch instruction. In common implementation, BTB uses 2–3 bits per branch instruction for this information. TLABP and PAM store n bits of information for each branch instruction, where n is the number of branch history bits. The value of n varies in different implementations, as shown in the following. Hardware costs of branch prediction mechanisms are discussed as following:

- a. One common implementation of the BTB uses a counter as the finite-state machine. Thus, BTB's overhead is the cost of the finite-state machine.
- b. PAM's hardware overhead is $O(n)$, where n is the number of the branch history bits. PAM and TLABP both record branch behavior with history bits, and the number of history bits has a strong influence on prediction hit rate. For each history bit, PAM uses a 4-bit weight register to record the number of consecutive correct predictions. An n -bit comparator and an n -bit counter are used to

TABLE 8. Branch prediction hit rates for different instruction types by different methods

Methods	Branch type					
	multi-way-branch		branch		dereference (= 0)	
	100%	97%	100%	97%	100%	97%
Profiling	95.1	74.3	88.2	72.1	91.2	77.3
BTB	94.7	71.9	86.1	67.0	99.1	92.0
TLAPM	98.1	93.3	96.3	90.5	99.2	95.1
PAM(12)	97.4	90.5	93.2	85.1	98.8	92.6
PAM(18)	98.0	91.8	94.9	88.9	99.0	93.3
PAM(24)	98.2	92.8	95.8	90.7	99.1	93.7
PAM(48)	98.4	95.5	96.3	92.2	99.4	96.7

PAM(N): N is the history register width.

100%, 97%: 100% includes the all branch instructions, and 97% includes only those branch instructions for which at least 3% of the results are irregular, as compared with the usual outcome.

predict the branch result. Table 7 shows the prediction hit rates for PAM from $n = 12$ to 48. Based on these numbers, we chose PAM(24) in our implementation. The hardware cost of PAM(24) is a 24-bit history register, a (24-bit * 4) weight register, a 24-bit comparator and a 24-bit counter. The performance enhancement by PAM(24) in a superscalar system is shown in Section 6.3.2.

- c. By the way of comparison, TLBP uses a pattern table with 2^n entries to store the states of all the n -bit branch patterns. Thus TLBP's hardware cost is $O(2^n)$, where n is also the number of the branch history bits. Yeh and Patt (1992) implemented TLBP with $n = 12$. So the hardware cost of TLBP(12) is a 12-bit history register, a ($2^{12} * 2$ bits) pattern table, and a 2-bit finite-state machine. Both PAM(24) and TLBP(12) have prediction hit rates of 97% in Prolog benchmarks, and both have a higher hardware cost.

Table 8 shows the prediction hit rates for all the *branch* and *dereference* instructions. The prediction hit rates for the *dereference* instructions are uniformly high, so their dynamic conditions seem regular and easy to predict. On the other hand, the prediction hit rates for the *branch* instructions are lower, and TLBP and PAM provide much higher prediction hit rates for *branch* instructions than profiling and BTB do, especially for the '97%' column of branches.

6.3.2. Parallelism enhancement in Prolog by PAM

In Table 9, we show the enhancement in instruction parallelism that results from incorporating the PAM(24) branch prediction method into a superscalar system. A

degree of parallelism of 2.40 is achieved by an RL2N8 superscalar system model, this is a system with register renaming, two load/store units and eight functional units, but no branch prediction unit. When we added the PAM(24) branch prediction unit and used it to predict the *branch* instructions, the execution parallelism jumped to 3.00, an increase of 25%. Furthermore, when we used the branch prediction mechanism to predict the *multi-way branch* instructions as well, the parallelism increased to 3.85, for a total gain in parallelism of 60.4%. When we used PAM to predict the *dereference* instructions as well, the parallelism increased to 4.05, for a total gain in parallelism of 68.8%. We thus conclude that a good branch prediction method can greatly enhance overall system performance, and an increase in superscalar system performance of $4.05/2.40 = 168.8\%$ is likely with moderate machine parallelism.

In summary, all these different branch prediction schemes can be applied in different systems and/or applications. In the early stages of their development, superscalar processors did not support any dynamic prediction schemes. Today, modern superscalar processors, such as the Intel Pentium, DEC Alpha and IBM Power PC, all use BTB for dynamic branch prediction. In these powerful superscalar processors with ample computational power, BTB yields inexpensive hardware cost while effectively enhancing resource utilization. As we have shown in Lemmata and simulations, the branch prediction hit rate will greatly influence hardware efficiency and system performance. With the progress of VLSI technology and computer system techniques, having a branch prediction scheme with high prediction accuracy will become more important with each passing day.

TABLE 9. Execution parallelism versus branch prediction in an RL2N8 superscalar system

Method	Sequential system	No prediction	Branch prediction	Branch + MWB prediction	Add dereference = 0 prediction
Parallelism	1	2.4	3.00	3.85	4.05
Performance enhancement			25.0%	60.4%	68.8%

7. CONCLUSION

In this paper we examined the effect of branch prediction on superscalar system performance and proposed a branch prediction scheme called PAM. We first modeled the effects of branch prediction hit rate on fine-grained parallelism. We also found that the superscalar system performance is highly sensitive to branch prediction hit rate. A good branch prediction scheme can enhance superscalar system performance by 168.8% in an RL2N8 model.

Several existing branch prediction methods were studied, and a new, period-based prediction method called PAM were proposed. PAM provides high performance with moderate hardware cost; in our simulations, the prediction hit rate of PAM was 97%. PAM greatly reduced the number of branch barriers, and it enhanced the program parallelism in a superscalar system by 68.8% compared with designs without branch prediction.

In a Prolog system, the same branch prediction mechanism can also be applied to *multi-way branches* and *dereference* instructions. In this approach, execution barriers created by these instructions can be eliminated, further enhancing the fine-grained parallelism of superscalar Prolog systems.

REFERENCES

- Chang, P. P., Mahlke, S. A., Chen, W. Y. and Hwu, W. W. (1991) IMPACT: an architectural framework for multiple-instruction-issue processors. In *Proc. 18th Ann. Int. Symp. on Computer Architecture*, pp. 266–275.
- Chi, T. (1990) *Design of LISCP-II Processor*. Master thesis, Computer Science and Information Engineering Department, National Chiao-Tung University, ROC.
- Clocksin, W. F. and Mellish, C. S. (1981) *Programming in Prolog*. Springer-Verlag, New York.
- Degroot, D. (1987) Restricted AND-parallelism. In *Proc. Symp. on Logic Programming*, San Francisco, pp. 80–89.
- Gloria, A. D. and Faraboschi, P. (1992) Instruction-level parallelism in prolog: analysis and architecture support. In *Proc. 19th Ann. Int. Symp. on Computer Architecture*, pp. 224–233.
- Gupta, G. and Jayaraman, B. (1993) AND–OR parallelism on shared-memory multiprocessors. *J. Logic Programming*, **17**, 59–89.
- Gupta, G. and Costa, V. S. (1994) Optimal implementation of AND–OR parallel prolog. *Future Generation Comp. Syst.*, **10**, 71–92.
- Johnson, M. (1991) *Superscalar Micro Processor Design*. Prentice-Hall, Englewood Cliffs, NJ.
- Kacsuk, P. (1994) OR-parallel prolog on distributed memory systems. In *Proc. 6th Int. Parallel Architectures and Languages Europe (PARLE 94)*, pp. 453–463.
- Kale, L. V. (1987) ‘Completeness’ and ‘full parallelism’ of parallel logic programming schemes. In *Symp. Logic Programming*, pp. 125–133.
- Kale, L. V., Shu, W. and Ramkumar, B. (1988) *Implementation and Performance of a Parallel Prolog Interpreter*. Technical Report UIUCDCS-R-88-1480.
- Lam, M. S. and Wilson, R. P. (1992) Limits of control flow on parallelism. In *Proc. 19th Int. Symp. on Computer Architecture (ISCA 19)*, pp. 46–57.
- Lee, J. K. F. and Smith, A. J. (1984) Branch prediction strategies and branch target buffer design. *IEEE Comp.*, **6**, 22.
- Lin, Y. J. and Kumar, V. (1988) Performance of AND-parallel execution of logic programs on a shared-memory multiprocessor. In *Proc. Int. Conf. on Fifth Generation Computer Systems*, pp. 851–860.
- Ma, R. L. and Chung, C. P. (1994) The simulation and analysis of instruction level parallelism in prolog superscalar processor. In *Proc. Int. Computer Symp. 1994 (ICS)*, Taiwan, pp. 55–60.
- Nagnuma, J. and Ogura, T. (1994) A highly OR-parallel inference machine (Multi-ASCA) and its performance evaluation: an architecture and its load balancing algorithm. *IEEE Trans. Comp.*, **43**, 1062–1075.
- Pack, J. C. H. and Schlansker, M. S. (1991) *On Predicated execution*. Technical Report HPL-91-58, HP Laboratories, Palo Alto, CA.
- Ramkumar, B. and Kale, L. V. (1992a) Machine independent AND and OR parallel execution of logic programs: part I—the binding environment. *IEEE Trans. Parallel Distributed Syst.*, **5**, 170–180.
- Ramkumar, B. and Kale, L. V. (1992b) Machine independent AND and OR parallel execution of logic programs: part II—compiled execution. *IEEE Trans. Parallel Distributed Syst.*, **5**, 181–192.
- Su, C. L. and Despain, A. M. (1994) An instruction scheduler and register allocator for Prolog parallel microprocessors. In *Proc. Int. Computer Symp. 1992 (ICS)*, Taiwan, pp. 699–706.
- Van Roy, P. (1990) *Can Logic Programming Execute as Fast as Imperative Programming?* PhD dissertation, University of California, Berkeley.
- Warren, D. H. D. (1983) *An Abstract Prolog Instruction Set*. Technique Note 309, SRI International, AI Center.
- Yeh, T and Patt, Y. N. (1992) Alternative implementations of two-level adaptive branch prediction. In *Proc. 19th Int. Symp. on Computer Architecture (ISCA 19)*, pp. 124–134.