Journal of ELECTRICAL ENGINEERING, VOL. 56, NO. 11-12, 2005, 298–305

IMPROVING BRANCH PREDICTORS BY COMBINING WITH PREDICATED EXECUTION

Ali Shatnawi — Mohammed F. Shatnawi *

This paper deals with superscalar processors, which are capable of executing several instructions per clock cycle. Superscalar processors may be considered as the most promising uniprocessor architectures of the post RISC era. Although superscalar processors can be viewed as an evolution of the RISC architectures, they are subject to many more trade-offs than simply the pipeline depth. Executing more than one instruction per cycle embeds many problems. Among the most common problems that architects have to solve in superscalar architectures is the pipeline gaps due to control hazard. Control hazard results from the presence of conditional branches in programs. This problem has been addressed by several researchers during the last decade. In this paper, we present an efficient technique based on a combination of the Branch Target Buffer (BTB) and Predicated Execution. Simulation results show the effectiveness of the proposed method.

K e y w o r d s: superscalar processor, control hazard, branch prediction, predicated execution

1 INTRODUCTION

There are three major types of branch instructions: unconditional branches, conditional branches, and branches in loop-control statements. Unconditional branches are always taken, and hence, the calculation of the target address does not depend on any condition resolution at runtime. Therefore, they can be treated as normal sequential program instructions, with the exception that the program counter (PC) will be loaded with the branch target address. In the case of conditional branches, the processor must resolve some condition(s) before it can determine the exact execution path. A branch in a loop is usually taken and the branch target is most likely deterministic, computable at compile time. Regardless of the branch type, the branch target address must be determined before the execution of the branch instruction has completed.

Figure 1 shows an example of a processor pipeline. Suppose that each pipeline stage takes one cycle to complete. If no data hazard exists among instructions and sequential program flow is maintained, a throughput of one instruction per cycle may be achieved. However, in the case where conditional branches are encountered, the address of the next instruction may not be determined until the condition of the instruction is resolved. Hence, the processor would either stall or speculate on the result of the branch (taken or not taken).

If the branch prediction is correct, stages 1 to 5 will essentially contain valid instructions; otherwise, they will have invalid ones. In the latter case, these instructions need to be flushed out of the pipeline. This discontinuity in the pipeline operation will severely degrade the processor performance and reduce its throughput.

In this paper, we propose a new technique to solve the problem of branch prediction and thus hiding the latency that may result of the discontinuity of instruction flow caused by conditional branches.

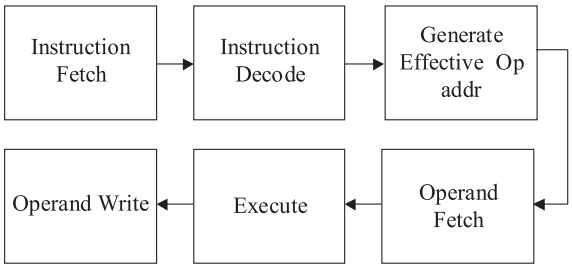

Fig. 1. Example of a Processor Pipeline

This paper will be organized as follows. Section 2 will introduce the problem of control hazard, which will be the topic of this paper, along with a literature survey of the methods used to alleviate the branch problem. In Section 3, we will describe the predicated execution model, a method claimed to reduce the penalty of the branch problem to zero. The contribution or our work to the solution of the branch problem will be discussed in Section 4, based on the model described in Section 5. Section 6 will present some simulation results, and Section 7 will conclude the contribution of this paper.

2 CONTROL HAZARD

A superscalar processor has the ability to run multiple instructions simultaneously in addition to overlapping them during execution [1]. However, instructions are not independent from each other, but rather interrelated. This interrelationship is referred to as data dependency. This dependency prevents some instructions from

* Department of Computer Engineering, Jordan University of Science and Technology, Box 3030, Irbid 22110, Jordan, ali@just.edu.jo

ISSN 1335-3632

proceeding smoothly into the pipeline without stalling. The handling of data dependency is beyond the scope of this paper. Another dependency that may preclude the smooth operations of the pipelines is the control dependency, typically referred to as control hazard. The control hazard is associated with the branch instructions. A branch instruction changes the program counter at a certain stage of its execution. The instruction following a taken branch instruction cannot be determined until the branch target address has been computed.

The effect of control hazard (if not resolved) on the performance of a superscalar processor is very severe, as it prevents the utilization of the processor resources while the decision on the branch is being determined. To enhance the processor performance, high machine parallelism and instruction parallelism are to be sought. A challenge in the design of a superscalar processor is to achieve a good balance between instruction parallelism and machine parallelism. The processor's policies for fetching, decoding, and executing instructions have a significant effect on its ability to discover instructions for concurrent execution [2], [3], [4].

The sole aim of the branch penalty reduction methods is to eliminate or at least alleviate the stalls resulted from unresolved branch conditions or target addresses. The methods used for this purpose fall mainly in two categories, namely, speculation of the branch outcomes and branch delaying until the branch condition is resolved [5].

Branch prediction reduces performance degradation due to control hazards. A pipeline with branch prediction uses some additional logic to *guess* the outcome of a branch decision before it is determined. The pipeline then begins prefetching the instruction stream from the predicted path.

The processor may predict the branch either, statically or dynamically. In static prediction, the processor makes a presumptive prediction to a path that will, most likely, be taken. In static prediction, this presumption is fixed at design (or compile) time. In the dynamic case, the processor makes the guess based on the history of the branch instruction itself. In this case, the processor maintains some history information about the behaviour of each and every branch (the most recent ones for finite buffering). The processor, using a burnt-in branch prediction technique, will then predict the branch decision based on the historical behaviour of the branch.

Static prediction mechanisms do not depend on the history of the branch instruction to predict the branch decision. In some static techniques, the branch is considered to be taken all the time or not taken all the time. In some other techniques, prediction is based on the op-code of the branch instruction. That is, for some branch instructions, the prediction is taken, and not taken for some others. The choice of the op-code is made at compile time based on compiler analysis. An example of an instruction suitable to predict as not taken is a conditional branch. An example of this technique is proposed in [6], which achieved an accuracy of prediction over 75 %, while that proposed in [7] gave an accuracy of about 85 %.

Dynamic branch prediction mechanisms maintain an associative memory table for recently executed branch instructions. If a certain branch instruction has an entry in the table, its recent history is used to determine whether the branch is to be predicted taken or not taken. The use of a different number of history bits with some heuristic algorithms affects the accuracy of the prediction [8]. Examples of dynamic branch prediction techniques can be found in [7], [10], [11], [12].

Lee and Smith found that using a single history gave a prediction accuracy ranging from 80–96 %. Using five bits resulted in an accuracy of 84–98 % [12].

A prefetch buffer is used to prefetch new instructions, as the processor consumes more instructions than the main memory can supply [1]. Thus when the processor seeks an instruction out of the sequential stream, it will be more likely ready in the prefetch buffer. However, instructions following a taken branch are less likely to be found in this buffer. A branch target buffer (BTB) is used to alleviate this problem. The BTB is a special selective cache memory associated with the instruction fetch portion of the pipeline. Each entry in the BTB consists of the address of a branch instruction that had already been encountered, and the address of the target instruction itself. In some architectures, the BTB stores a few instructions following the target instruction. When a branch instruction is encountered, the BTB is associatively searched and the target address is supplied directly to the pipeline.

In 1992, Yeh and Patt provided a new branch prediction method called Two Level Adaptive Branch Prediction (TLABP) [13]. It is an enhanced dynamic method based on profiling and the BTB. This method records the dynamic execution patterns of the branches, uses these 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 new branch pattern recurs, the branch result can highly likely be predicted correctly. The accuracy resulted is about 97 % for some of the SPEC benchmarks. This technique is rather complex and requires high hardware overhead.

The previous methods were mainly hardware based. Delayed branches and some other methods are, on the other hand, software directed. In delayed branches, the branch decision is delayed a few cycles, called the delay slot. That is, some instructions which are supposed to execute regardless of the branch outcome, are used to fill this slot. This approach demands dependency detection by the compiler.

Branch bypass with multiple prefetching is another paradigm for branch prediction. In such methods, both paths of a branch are considered for execution. One path will then be discarded when the branch is resolved. A pipeline containing a branch must prefetch instructions from the two execution paths. In some processors, instructions from the two paths are executed. In this case, the

instructions will not commit their results until the outcome of the branch is resolved. This method is referred to as *Predicated Execution* which was found to result in a great improvement of performance, but at the expense of a more complicated hardware [15], [16]. Some variation of predicated execution can be found in [17].

Branch folding is one of the methods that can reduce the branch penalty to zero. In this method, the pipeline is broken into two stages, which are the fetch-and-decode unit and the execution unit [6]. The fetch-and-decode unit places decoded instructions into a special instruction cache. Each instruction contains a *next-address field*. Thus, any instruction behaves like an unconditional branch, in addition to its normal operation. The fetch-and-decode unit recognizes when an unconditional branch instruction follows a regular instruction. It then folds the branch instruction into the preceding one, thus removing the branch instruction from the execution pipeline.

On the other hand, a conditional branch maintains an alternate next-address field. When a conditional branch is executed, one of the two paths is used for normal execution, while the instructions of the other are placed at the alternate address. When the pipeline finally resolves the branch, it either discards the instructions of the alternate path or flushes itself and restarts execution at the alternate address.

Multiple Independent Instruction Streams in a Shared Pipeline is another method to alleviate the problem of control hazard. The main idea of this method is to interleave process execution, that is, each stage of the pipeline has an instruction from a different process. Since each process is independent, no branch dependencies exist between instructions [19]. In many cases, some calculation may be needed to compute the address of the instruction following a taken branch. Thus, early handling of branches by the instruction fetch unit may be needed to begin prefetching from the appropriate instruction stream. As an example, the Texas Instruments ASC computer uses the Prepare-to-Branch instruction to give the control hardware advance warning that a branch instruction is going to execute. This warning passes the branch target address to the control unit so that the unit can begin instruction prefetching from that address. The effectiveness of the Prepare-to-Branch depends on the precision of the programmer (compiler) in inserting the control instructions at the appropriate points in the instruction stream. An efficient branch prediction technique which theoretically reduces the penalty of control hazards to zero is the predicated execution. Due to the importance of this technique in the context of this paper, it will be addressed in a separate section.

3 PREDICATED EXECUTION

Despite the fact that correct branch prediction could increase the instruction level parallelism (ILP), incorrect prediction often results in large performance penalties [19], [20]. These performance penalties are, in general, attributed to several factors. First, a large number of speculated instructions are executed from the predicted path must be discarded if the speculation turns out to be incorrect. Second, there is some time penalty for recovering from the effects of the improperly initiated speculative instructions. This penalty includes the time needed to drain the pipelines, and that needed to invalidate the appropriate instruction from the processor buffers. Third, it is required to compute the proper target address and start fetching instructions from the new path after a misprediction has occurred. Fourth, the presence of a large number of branches in the instruction stream places a limit on the potential ILP. For example, if 25 % of an instruction stream are branches, an 8 issue superscalar processor must have the capability to execute at least two branch instructions per cycle to achieve its maximum performance. This would demand a great pipeline complexity and the use of multi-ported hardware structures, such as the BTB. Predicated execution support provides a means to remove branches from the instruction stream. A predicate refers to a Boolean operand that is inserted into the instruction code to decide whether the instruction will execute or will be discarded. A true predicate (a predicate with a logical value of one) causes the instruction to execute, whereas a false predicate prevents the instruction from modifying the processor state. An algorithm called if-conversion is used to convert conditional branches into predicate defining instructions, and instructions along alternative paths of each branch into predicated instructions. Those predicated instructions are fetched regardless of their predicate values. Instructions whose predicated values are true are executed and those with false predicate values, are not. As mentioned earlier, this offers the opportunity to improve branch handling in superscalar processors by eliminating frequently mispredicted branches, and thus reducing branch prediction misses. The penalties associated with the eliminated branches are thus removed. The elimination of branches also reduces the need to handle multiple branches in a single cycle for large issue-rate processors. The amount of speculative instructions needed to fill the branches and sustain full processor utilization is also reduced. Predicated execution provides an efficient interface for the compiler to expose multiple execution paths to the hardware. Figure 2 illustrates the concept of predicated execution. Figure 2a comprises the source code. For each iteration of the loop, either the value of j or k is incremented. If-conversion replaces conditional branches in the code with comparison instructions that define one or more predicates. Control dependent (on the branch) instructions are then converted to predicated instructions utilizing the appropriate predicate value.

Using this approach, control dependencies handled by predicates are converted into data dependencies. The assembly code of the source loop without predication is shown in Fig. 2b, and with predication in Fig. 2c. The

```
for( i = 0; i < 100; i++)
    if ( a[i] > 50 ) j = j + 1;
    else k = k + 1;
```
(a)

```
    mov r1,0
    mov r2,0
    ld r3,addr(A)
L1: ld r4, mem(r3+r2)
    bgt r4,50,L2
    add r5, r5, 1
    jump L3
L2: add r6, r6, 1
L3: add r1, r1, 1
    add r2, r2, 4
    blt r1, 100, L1
```
(b)

```
    mov r1,0
    mov r2,0
    ld r3,addr(A)
L1: ld r4,mem(r3+r2)
    pred_gt p1U, p2U*,r4,50
    add r5, r5, 1 (p2)
    add r6, r6, 1 (p1)
    add r1, r1, 1
    add r2, r2, 4
    blt r1, 100, L1
```
(c)

Fig. 2. Illustration of the predicated execution method

values of j and k are placed in registers r5 and r6, respectively. The conditional branch, **bgt**, is replaced by the predicate defining instruction **pred_gt**. The predicate **p1** will have the value of **1** if **r4**$>$**50** and **0** otherwise. The value of **p2** is the complement of **p1**. The two instructions affected by the outcome of the conditional branch, are converted to predicated instructions having **p1** or **p2**, respectively, as their predicates. Depending on the result of the predicate defining instruction, either **r5** or **r6** will be incremented by the predicated add instructions. The semantics of the predicate defining instruction is as follows

Pred_⟨cmp⟩Pout1⟨type⟩, Pout2⟨type⟩, src1,

src2 (Pin) . . . (I)

where **Pout1** and **Pout2** are the output predicates which are set according to the comparison of **src1** and **src2** using ⟨**cmp**⟩ as a comparison relation such as "**eq**", "**ne**" or "**gt**", ⟨**type**⟩ is the predicate type, and **Pin** is the predicate value for the predicate defining instruction itself. The Boolean value written to the predicate register is a function of the result of the comparison, the input predicate of the definition instruction (**Pin**), and the ⟨**type**⟩ field. Types of predicates include, but not restricted to, unconditional, conditional, **OR**, and **AND**.

Based on the model (Cydra 5) introduced in [21], the extension which is assumed to support predicated execution, consists of four major components:

- N x 1-bit predicate register file to store the predicate values.
- An additional source operand for each instruction to specify a predicate for instruction execution.
- A modified decode/issue stage to nullify instructions with a false predicate.
- A set of predicate defining instructions.

The set of predicate defining instructions have an op-code form like that explained in (I).

There are several compiler techniques that could be utilized to exploit predicated execution. One of these is based on an abstract structure called a hyperblock [16] [16]. A hyperblock is composed of multiple basic blocks. In a hyperblock, control may only enter at the first basic block, which is designated as the entry block. Control flow from a basic block to another is replaced by an if-conversion. Basic blocks are appended to a hyperblock in order to maximize the performance as it captures a large fraction of the predicted control flow paths. That is, any block which is likely to flow into should be added to the hyperblock. It is to be noted that including too many blocks will likely result in an overall performance degradation. Exclusion of a block from the hyperblock will result in leaving a branch instruction within the hyperblock. Hyperblock formation should focus on eliminating unbiased branches (highly likely to be either taken or not-taken). Highly biased branches can be left out of a hyperblock, as little performance gain can be achieved by including them. Therefore, careful attention is paid to the size and dependence level of each block to be selected for inclusion in the hyperblock. That is, a neutral branch (not biased towards taken or not taken) may be left in the program if one of the target blocks significantly requires more resources than the others.

In this paper, we relied on the fact that predicated execution has zero time penalty for mispredicted branches, but at the expense of the hardware complexity. With the advances of VLSI technology and the increase in the level of integration, this complexity can be justified. Based on the mentioned facts and observations, we propose a new technique based on a combination of the BTB and the predicated execution to resolve the problem of control hazard

4 PREDICTION WITH PREDICATION

The fact that branch prediction has reached accuracy as high as 95 %, using a two bit history buffer [22], and an accuracy of 98 % using a five bit history buffer, made it impractical to see further enhancements in several applications. However, for some applications, one may not underestimate the 5 % inaccuracy that may be encountered using these traditional techniques. Further, not all applications have the same trend in branch behaviour. Some applications have very predictable branch behaviours like those with scientific nature, while some others behave in a totally unpredictable way. Examples on these are programs found in AI and non-scientific applications. For example, the heavy dependence of AI applications on conditional codes, which suffer from unpredictable behaviour,

```
    bgt r4,50,L2    Pred_gt p1U, p2U*,r4,50
    add r5, r5, 1   add r5, r5, 1 (p2)
    jump L3         add r6, r6, 1 (p1)
L2: add r6, r6, 1   add r1, r1, 1
L3: add r1, r1, 1   add r2, r2, 4
    add r2, r2, 4
```

Fig. 3. Code Segment Representing a Conditional Branch

WB					I1	I2	I3	I4
EX				I1	I2	I3	I4	
D			I1	I2	I3	I4		
F		I1	I2	I3	I4			
	I1	I2	I3	I4				
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Fig. 4. The pipeline behaviour when the branch is correctly predicted

results in a very low percentage of branch prediction accuracy.

Predicated execution on the other hand, totally eliminates the penalty of branch prediction as was shown in previous sections. For each conditional instruction, a predicate defining instruction has to be defined, and a set of predicated instructions have to be generated. This will definitely increase the program size, and consequently the size of the required memory. That is, the formation of a hyperblock is not possible without code replication. Such cost may preclude the full use of predicated execution in some applications. It is also to be noted that if the branch prediction technique is not resulting in high accuracy, there will be no significant performance improvement by the elimination of branches using predicated execution. This motivates the idea of combining the two methods of BTB and predicated execution, thus benefiting from the merits of each. The BTB approach is very well defined and implemented in almost all recent systems. Predicated execution, on the other hand, has a zero branch penalty. Thus, we propose to use BTB as long as it is effective in predicting the behaviour of the branches, and use predicated execution at the instances where BTB can provide high prediction accuracy. The combination of the two methods ensures the use of the easily implemented BTB method most of the time, and the use of predicated execution in a much lesser frequency; thus minimizing the software and hardware overhead and optimizing performance.

5 HARDWARE MODEL

In our architecture mode, there is an issue stage that separates the decode stage from the execute stage. However, this stage is not an existing physical stage, but rather a virtual stage that represents the transfer of instructions from the decode stage to the execution stages. Further, there are two parallel fetch units and two parallel decode units as well. The execute unit is split into three stages execute1, execute2, and execute3.

The principle of *shelving* (attaching a reservation station to each functional unit) is assumed. The execution of an integer instruction is assumed to complete at stage "execute 1", while that of a floating point instruction at the last execute stage.

The proposed doubling of the fetch and decode stages can be implemented by having, for each pipeline in a superscalar processor, a replicate of the two stages. For example, a system of 8 pipelines with a total of 48 stages will have 16 idle stages, that is, 33.3 % of the stages will be idle as long as there are no condition codes resolved by the predicated execution. To alleviate this problem, it is possible to borrow these stages, when needed, from a neighbouring pipeline. Thus, there will be no idle stages during the times where no conditional codes are still unresolved. However, during the times where predicated execution is used, only the two stages of a neighbouring pipeline will be used for only two cycles. The rest of the stages of the lending pipeline can continue their job without the need to flush their contents. After these two cycles, the target instruction which already exists in the lending pipeline can enter the pipeline. In a real system implementation, the processor designer can still consider doubling the stages if the level of integration permits.

A valid question is when to use the predicate approach during program execution. The answer to this question stems from the research conducted in [20], where a classification of the conditional branches of some benchmarks was performed. The study classified branches into easy to predict and problematic. Branches which are easy to predict are handled using BTB, while the problematic ones (having unpredictable nature) are to be handled using predicated execution. Thus a complete system utilizing the proposed model relies on a compiler capable of analyzing branches, and can flag the problematic ones. The architecture model proposed handles branch with the problematic flag set, using predicated execution; otherwise, it will use BTB.

6 EXAMPLE

It will be shown that the application of predicated execution on highly biased branches, (branches that are highly predictable), has very little performance gain. For the code depicted in Fig. 3, the execution of the non-branch instructions in the pipeline will be as shown in Fig. 4 if the prediction was correct, and as shown in Fig. 5 if the prediction was incorrect. However, the execution of these instructions using predicated execution is shown on the pipeline depicted in Fig. 6.

As shown above, with perfect prediction of a path, only 7 cycles are needed to execute the above code, while with an incorrect prediction, 11 cycles are required. Branch elimination using predicated execution results in the same

WB				I1					I2	I3	I4
EX			I1					I2	I3	I4	
D		I1				I2	I2	I3	I4		
F	I1	I2	I3			I2	I3	I4			
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**

Fig. 5. The pipeline behaviour when the branch is not correctly predicted

WB					I2	I3	I4
EX				I2	I3	I4	
D			I2	I3	I4		
F	I1	I2	I3	I4			
D′			I2				
F′		I2					
	1	**2**	**3**	**4**	**5**	**6**	**7**

Fig. 6. The pipeline behaviour when predicted execution is used

Table 1. Instruction and Overhead Costs of the Generated Synthetic Benchmark

Instruction, Overhead	Cost in Cycles	
	Cycle Cost	Overhead Cost
Integer Instruction.	4	
Data Dependency.	4	0
Floating Point Instruction.	6	
Data Dependency.	6	Max (2).
Branch Prediction:		
No Prediction (Integer Instruction)	4	3 + 2
No Prediction (Floating Instruction)	6	5 + 2
Predictable Branch (Integer)	4	0
Predictable Branch (Floating)	6	0
Predicated Execution		
Predicated Execution (Integer)	4	0
Predicated Execution (Floating)	6	0

number of instructions as that of a perfect branch prediction. This reassures the fact that, if the prediction is right, predicated execution results in no gain in performance, whereas there is a great gain in performance in the case of a wrong prediction. Again, it is very important to keep in mind that the application of predicated execution at all incidences of the branches would waste resources without any tangible gain. On the other hand, branch prediction using BTB is very stable and can be easily implemented by hardware. These facts encourage the use of a combination of the two methods rather than solely using predicated execution.

Table 2. Simulation Results I

Sample Size = 200,000 instructions Predictable branches = 35% Floating instructions = 35%			
Branch (%)	CPI w/o Pred	CPI with Pred	Speedup (%)
4	1.61	1.53	5
9	1.7	1.56	9
13	1.78	1.59	12
17	1.86	1.61	16
22	1.93	1.63	18
26	2	1.65	21
31	2.06	1.67	23
35	2.11	1.69	25
39	2.25	1.85	22
44	2.29	1.87	22

Table 3. Simulation Results II

Sample Size = 200,000 instructions Branch instructions = 17.5% Floating instructions = 35%			
% of unpred branches	CPI w/o Pred	CPI with Pred	Speedup (%)
17	1.76	1.64	7
22	1.79	1.63	10
26	1.81	1.62	12
30	1.84	1.62	14
35	1.86	1.61	16
39	1.89	1.6	18
44	1.91	1.6	19
48	1.94	1.59	22
53	1.96	1.58	24
57	1.98	1.58	25

7 SIMULATION RESULTS

As mentioned earlier, some techniques such as data forwarding are used to reduce the latency resulting from data hazards. In the proposed processor model, using internal data forwarding results in no penalty for integer operations, and two cycles for floating point operations. However, for instructions executing two cycles ahead of an executing operation, a penalty of one cycle is encountered. Operations entering the pipeline three cycles ahead or more with respect to the currently executing operation will experience no delay for data dependency. The time needed to drain the pipeline and fetch the correct target instruction is assumed to be two cycles.

Table 1 lists the instruction costs and their overheads in different scenarios. The results of the simulation are depicted in Table 2 and Table 3.

Table 3 shows the results of speedup for different ratios of branch instructions, and Fig. 7 depicts these results. As the percentage of branches increases, the speedup of the proposed method increases. This is due to the fact that the proposed technique is less prone to latency resulting

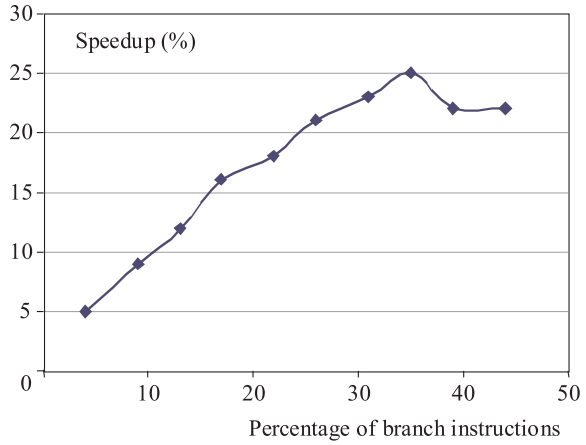

Fig. 7. Speedup versus branches for Simulation I

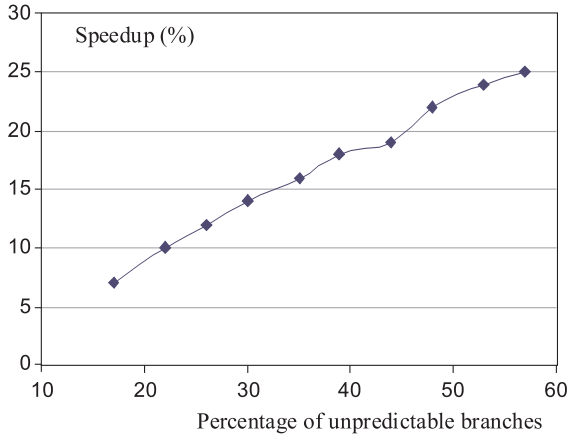

Fig. 8. Speedup versus Unpredictable Branches for Simulation II

from the increase in the number of unpredictable branches with respect to the total number of instructions.

Table 3 shows the enhancement versus the percentage of unpredictable branches in the program. As predicted, the enhancement is higher when the ratio of unpredictable branches increases. Figure 8 shows the linear relation of the enhancement versus unpredictable branches. This shows the effectiveness of the method in applications with unstable or unpredictable branch behaviour. In such applications, the method provides a very efficient solution to the branch problem. The results show the effectiveness of the proposed technique in reducing the execution time as the unpredictable branches increase.

The results and figures clearly show the amount of execution time enhancement and performance gain obtained by using the proposed technique. Predicated execution can give the best execution time but, once again, it is at the expense of the hardware and software overhead.

8 CONCLUSION

Superscalar processors will have a very significant role in modern computing for several years to come. However, superscalar processors are vulnerable to severe performance degradation if no special care is taken to handle data and control hazards. In this paper, we have proposed an efficient solution to the problem of control hazard. Techniques that attempt to reduce the branch penalty to zero (like predicated execution) result in high hardware overhead and complicated compiler algorithms. On the other hand, branch prediction using BTB has proven to be very efficient for several benchmark problems yielding an accuracy of 95 % when using a two bit history buffer. In certain application, where the BTB is inefficient for a percentage of the branches existing in the program, an alternative approach is necessary. To benefit from the merits of the two methods of dealing with the branch problem, we have proposed the technique of combining these two well-know methods. The results obtained using this technique, have been very impressive.

References

[1] DEZSO SIMA: Superscalar Instruction Issue, IEEE Micro, September/October 1997, pp. 28–39.

[2] CHEN, T.-F.—BAER, J.-L.: Effective Hardware-Based Data Prefetching for High Performance Processors, IEEE Transactions on Computers **44** No. 5 (1995), 609623.

[3] HWANG, K.: Computer Architecture and Parallel Processing, McGraw Hill Inc., 1984.

[4] HWANG, K.: Advanced Computer Architecture: Parallelism, Scalability, Programmability, McGraw Hill Inc., 1993.

[5] PATTERSON, D. A.—HENNESY, J. L..: Computer Architecture A Quantitative Approach, Morgan Kaufmann Publishers Inc., 1990.

[6] SMITH, J. E.: A Study of Branch Prediction Strategies, Proc. 8th Annual Symp. On Computer Architecture, 1981.

[7] LILJA, D. J.: Architectural Alternatives for Exploiting Parallelism, IEEE Computer Society Press Tutorial, 10662 Los Vaqueros Circle PO Box 3014, Los Alamitos, CA 90720-1264, Reducing The Branch Penalty in Pipelined Processors, pp. 68–77, 1991. Reprinted from Computer, Vol. 21, No. 7, July 1988, pages 47–55.

[8] MA, R.-L.—CHUNG, C.-P.: Periodic Adaptive Branch Prediction and its Application in Superscalar Processing, The Computer Journal **38** (1995), 457–470.

[9] ESPASA, R.—VALERO, M.: Exploiting Instruction- and Data-Level Parallelism, IEEE Micro (1997), 20–27.

[10] SEZNEC, A.—FELIX, S.—KRISHNAN, V.—SAZEIDES, Y.: Design Tradeoffs for the Alpha EV8 Conditional Branch Predictor, ACM SIGARCH Computer Architecture News, v. 30 n. 2, May 2002.

[11] JIMÉNEZ, D. A.—LIN, C.: Neural Methods for Dynamic Branch Prediction, ACM Transactions on Computer Systems (TOCS) **20** No. 4 (November 2002), 369–397.

[12] EGAN, C.—STEVEN, G.—QUICK, P.—ANGUERA, R.—STEVEN, F.—VINTAN, L.: Two-Level Branch Prediction Using Neural Networks, Journal of Systems Architecture: the EUROMICRO Journal **49** No. 12-15, (December 2003), 557–570.

[13] LEE, J.—SMITH, A. J.: Branch Prediction Strategies and Branch Target Buffer Design, Computer, Jan. 1984, pp. 6–22.

[14] YEH, T.—PATT, Y. N.: Alternative Implementations of Two-Level Adaptive Branch Prediction, In Nineteenth International Symposium on Computer Architecture, 1992.

[15] JIMÉNEZ, D. A.—KECKLER, S. W.—LIN, C.: The Impact of Delay on the Design of Branch Predictors, Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture, p. 67–76, December 2000, Monterey, California, United States.

[16] AKKARY, H.—SRINIVASAN, S. T.—LAI, K.: Recycling Waste: Exploiting Wrong-Path Execution to Improve Branch Prediction, Proceedings of the 17th annual international conference on Supercomputing, June 23-26, 2003, San Francisco, CA, USA.

[17] MAHLKE, S. A.—CHEN, W. Y.—BRINGMANN, R. A.—HANK, R. E.—HWU, W. W.—RAU, B. R.—SCHLANSKER, M. S.: Sentinel Scheduling A Model for Compiler-controlled Speculative Execution, Transactions on Computer Systems **11** (1993).

[18] MENG-CHOU CHANG—FEIPEI LAI: Efficient Exploitation of Instruction-Level Parallelism for Superscalar Processors by the Conjugate Register File Scheme, IEEE Transactions on Computers **45** No. 3 (1996), 278–293.

[19] THOMAS, R.—FRANKLIN, M.—WILKERSON, C.—STARK, J.: Improving Branch Prediction by Dynamic Dataflow-Based Identification of Correlated Branches from a Large Global History, ACM SIGARCH Computer Architecture News, v. 31 n. 2, May 2003.

[20] MAHLKE, S. A.—HANK, R. E.—McCORMICK, J. E.—AUGUST, D. I.—HWU, W. W.: A Comparison of Full and Partial Predicated Execution Support for ILP Processors, ISCA-22, Jun 1995.

[21] MAHLKE, S. A.—HANK, R. E.—BRINGMANN, R. A.—GYLLENHAAL, J. C.—CALLAGHER, D. M.—HWU, W. W.: Characterizing the Impact of Predicated Execution on Branch Prediction, Center for Reliable and High Performance Computing, University of Illinois, Urbana-Champaign, IL 61801..

[22] RAU, B. R.—YEN, D. W. L.—YEN, W.—TOWLE, R. A.: The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs, IEEE Computer **22**(1) (1989), 12–35.

[23] RAVI NAIR: Optimal 2-Bit Branch Predictors, IEEE Transactions on Computers **44** No. 5 (1995), 698–702.

Received 21 November 2004

Ali Shatnawi received the BSc and MSc in electrical and computer engineering from the Jordan University of Science and Technology in 1989 and 1992, respectively; and the PhD degree in electrical and computer engineering from Concordia University, Canada, in 1996. He has been on the faculty of the Jordan University of Science and Technology since 1996. He worked as a director of computer center between 1996 and 1999, vice dean for the faculty of information technology between 2001 and 2002, and as a dean of information technology and president assistant at the Hashemite University between 2002 and 2005. His present research covers hardware design, superscalar architectures, high level synthesis of DSP applications, algorithms, and wireless networks.

Mohammed Fadel Shatnawi received the BSc and MSc in electrical and computer engineering from the Jordan University of Science and Technology in 1995 and 1998, respectively. He worked as a part time lecturer at JUST in 1999. He joined a software company in Jordan as a development lead between 1999 and 2000. He then joined Microsoft in 2000 as a software design engineer and is still in this position to date. In Microsoft he works in the business intelligent (BI) team of SQL server group. His areas of technical interests are BI, ETL (Extract Transform and Load Technology.) He has several patents in the SQL Server product of SSIS (SQL Server Integration Services.)