Context-based Branch Prediction for High-Performance and Low-Power Computing

Dah-Lih Jeng* and Ching-Chuan Chiang**

**Computer Center, Chung Cheng Institute of Technology, National Defense University*
***Department of Computer and Communication Engineering, Ming Chuan University*

ABSTRACT

Advanced processors can simultaneously execute multiple instructions in parallel to achieve better performance. Branches introduce control dependence between instructions. Branch prediction therefore is important for modern processors. Most present predictors use branch history to predict branch outcomes. Using branch history alone results in delay for identifying mispredictions. In this paper, a *context-based branch predictor* is proposed to resolve branch condition for conditional branches. Simulation results of SimWattch from SPECint95 and MediaBench programs reveal that using our method, on average, the CPI can be improved by 2.12%, the IPC can be improved by 2.19%, and power consumption can be improved by 2.12%.

Keywords: branch prediction, misprediction rate

以分支指令前後文為基礎之分支預測器實現高性能、低耗電運算

鄭大立* 江清泉**

*國防大學理工學院電算中心
**銘傳大學資訊傳播工程學系

摘 要

目前處理器大多同時執行數條指令以提升指令階層平行度。然而，平行執行的指令間，分支指令將使指令間產生控制相依性，降低指令間的平行度，進而降低效能。因此目前處理器大都使用分支預測器來處理分支指令所產生之控制相依性。目前分支預測普遍使用分支歷史作為分支預測基礎。以上述分支歷史為基礎，本論文進一步提出以分支指令之資料相依性預測可能分支結果，稱為 Context-based Branch Predictor。我們使用 SPECint95 及 MediaBench 為測試程式，以 SimWattch 進行模擬，結果顯示，由於較早預測分支失誤，在效能方面，每指令時脈數(CPI)平均有 2.12%改善，每時脈指令數(IPC)平均有 2.19%改善，耗電方面有平均 2.12%之改善。

關鍵詞：分支預測、失誤率

文稿收件日期 94.8.29.；文稿修正後接受日期 95.3.27.
Manuscript received August 29,2005；revised March 27,2006. .

I. INTRODUCTION

Superscalar processors can simultaneously execute multiple instructions in parallel. They dominate modern processor market. The key factor to keep high performance for these processors is sustaining high degree of instruction level parallelism (ILP). Branch instructions, however, introduce control dependence between instructions and therefore reduce ILP. Whenever executing branch instructions, the pipeline has to stall and wait for the branch outcomes. *Branch prediction* predicts possible branch outcomes and keeps fetching instructions from target address.

Branches can be divided into conditional and unconditional branches. Evidence shows that the majority of the branches are conditional [1]. The predictions of branches include the branch outcomes and the target address. Once mispredicted, the instructions fetched after the branch have to be squashed. This situation results in waste of cycles and power consumption. Therefore great efforts in literatures [1] are devoted to improve the branch prediction accuracy. In general, most predictors use branch history to predict the branch outcomes. Through experiments, we observe that using branch history alone in prediction for conditional branches, the mispredictions are identified late till the branch conditions are resolved. Identifying mispredictions late degrades the performance and consumes unnecessary power. To solve this problem, we propose a *context-based branch prediction* for conditional branches. Using our approach, experiments show that the performance can be improved and the power consumption can be reduced.

In Section 2 we propose the observation from a motivating example and discuss related work in branch prediction. In Section 3 we propose a *Context-based Branch Predictor* for conditional branches. Section 4 presents the experimental results. Section 5 concludes the paper.

II. BACKGROUND AND RELATED WORK

Branch predictions have been studied extensively. Evers [1] summarizes current branch predictors including Always Predict Taken [1], Backward Taken [1], Forward Not Taken (BTFN) [1], Simple Profiling (profile) [2], Last-Time [3], Two-Bit Counter (2bc) [3] and Two-Level Adaptive Branch Predictors [4, 5]. These predictors use the branch address to access the prediction history. Predictions are made according to branch history. Using branch history alone in predictions could result the high misprediction rate. According to Evers [1], the misprediction rates of this type vary from over 40% for the worst static predictors to less than 4% for the best and costliest.

Consider a five-stage pipelined RISC machine using the Two-Level Adaptive Branch Predictor of Alpha [5] that can simultaneously fetch and decode four instructions per cycle. We use IF, DA, EX, WB and CT to imply instruction fetch, decode, execution, write back and commit pipeline stages. According to the branch prediction results, instructions in the predicted direction are fetched into pipeline. When predicting correctly, the performance gains from the prefetching of instructions. Once mispredicted, the instructions prefetched after the branch have to be squashed. For example, to illustrate this, in Figure 1, test program **`SPECint95/go`** is compiled into Alpha assembly. Figures 1 and 2 demonstrate two snapshots of pipeline executing program **`SPECint95/go`** in cycle 76 and 82, respectively. In cycle 76, the conditional branch **`ap`** is predicted not-taken according to branch history from its PC address. Three consecutive instructions **`aq`**,…,**`as`** following branch instruction **`ap`** are fetched into pipeline. In cycle 82, the conditional branch instruction **`ap`** is evaluated to be taken and mispredicted. The instructions **`aq`**,…,**`ba`** entering the EXE/ WB pipeline stages have to be squashed, which causes the flushing of 11 instructions.

This kind of misprediction penalties is very common. There are two important issues in the above misprediction penalties. First, branch history is retrieved by PC address during instruction fetch stage. Whenever newly conditional branch instruction is fetched into the pipeline, there is no sufficient past information of this branch. Therefore the misprediction rate for the newly conditional branch instruction could be high. Secondly, branch results will not be available until the branch conditions being resolved in the execution pipeline stage. Therefore the correct branch results will not be fed into the branch prediction unit the write back pipeline stage. The time delay in updating the branch prediction unit can result in high misprediction penalties. As we can see in Figures 1 and 2, from cycle 76 to cycle 82, there are 11 additional instructions coming into pipeline after the conditional branch instruction **`ap`**.

```
@cycle = 76
ap = bne r1,0xfffffffffffffff8
aq = bis r31,r31,r31
ar = ldq  r27,-32720(r29)
as = stl  r16,-16680(r29)
 [IF]     [DA]     [EX]     [WB]     [CT]
  ap
  aq
  ar
  as
```

Fig.1. Pipeline status of program **`SPECint95/go`**.

```
@cycle = 82
 [IF]     [DA]     [EX]     [WB]     [CT]
                    at       ap//mispredict
                             aq       am
                             ar       an
                             as       ao
                             au
                             av
                             aw
                             ax
                             ay
                             az
                             ba
```

Fig.2. Pipeline dump of program **`SPECint95/go`** to demonstrate misprediction penalties.

Several researches have been proposed to improve history based branch predictions. In general, there are two techniques being introduced to improve predictions. First, Hybrid Branch Predictions combine conventional history based prediction with other prediction techniques to improve misprediction rate. McFarling [6] proposed hybrid branch predictors which composed of two component predictors and a selector that decides which one is used to predict each branch. Lon and Henry [7] improved the hybrid prediction accuracy by replacing the selection mechanism with a fusion mechanism. Falcon [8] introduced the prophet/critic hybrid predictors. The prophet uses the branch history to predict its direction. The critic gives a critique of the prophet's prediction. Secondly, dynamic data dependence tracking was implemented to alleviate the late resolve of branch conditions. Chen [9] proposed a hardware method, data dependence chain, to implement dynamic data dependence tracking and a value-based branch prediction. However, his method required a long latency to make a prediction and also required a complex structure to implement the dynamic data tracking.

The misprediction penalties and branch prediction rate in Figures 1 and 2 can be improved if we can provide more information to predict the possible outcomes of conditional branches between the retrieving and updating of branch prediction. In this paper, we propose a new hardware solution so that the branch results are predicted according to branch instruction context. Data dependence of branch in the same instruction window is checked on the fly. The result of data dependence checking is combined with the prediction result from conventional predictors. Branch prediction results are updated in instruction decode stage, which is between the retrieving and updating of branch predictions. Using our method, branch prediction rate and misprediction penalties can be improved.

III. CONTEXT-BASED BRANCH PREDICTION FOR CONDICTIONAL BRANCHES

In this section we propose a hardware technique to improve the branch prediction accuracy and misprediction penalties. There are two contributions in our methods. First, different from other branch prediction methods, our prediction is made according to data dependence of branch instructions. Secondly, our prediction is conducted in the instruction decode stage. Using our predictor, once conditional branches are detected to be mispredicted, instructions in the wrong path will not proceed to reduce branch misprediction penalties. The details of our branch prediction are discussed as follows.

The instruction format of the conditional branches is shown in Figure 3. The source register **Ra** is defined as *decision register*. During instruction execution stage, the decision register is tested for specified relationship. For example, instruction **bne** tests decision register for not equal to zero. For these conditional branches, the instruction context that has data

Opcode	**Ra**	**Branch_disp**

Fig.3. Instruction Format of the Conditional Branch.

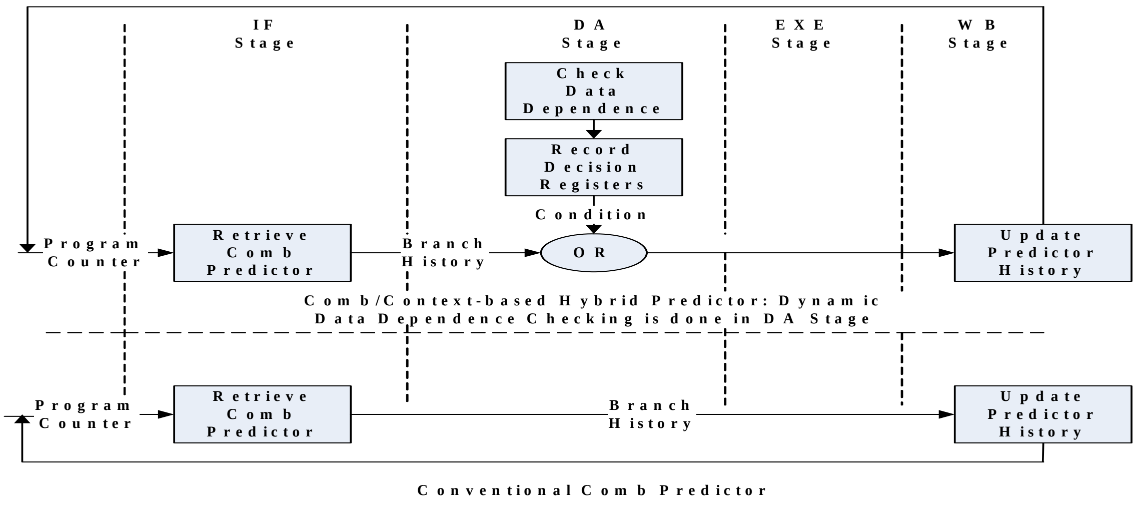

Fig.4. The Implementation of Context-base Hybrid Predictor.

dependence on these decision registers could influence on branch conditions. Therefore in instruction decode stage, we add a 32-bit flag register FLAG to record the writing of these decision registers. In an instruction window, any instruction that will write into registers will set the related flag register.

In the instruction decode stage, once we detect the execution of conditional branch instructions, the related flag register FLAG is checked. If the FLAG is set, it implies that in the same instruction window, there exist other instructions that are data dependent with the decision registers of conditional branches. We claim these instructions as *coupling* instructions. Since branch condition can not be resolved until these coupling instructions finish execution. On the other hand, if the FLAG is clear, it implies that there exists no instruction that is data dependent with the conditional branch instructions. We claim these instructions as *isolated*, since conditional branch do not have to wait for other instructions to finish execution. Therefore, for these isolated instructions, we will test the decision register for its specified relationship.

Take Figures 1 and 2 as examples. In Figure 1, in cycle 76, conditional branch **ap** has no data dependence with other instructions in the same instruction window. Therefore, conditional branch **ap** is eligible for our prediction. The algorithm of our method is shown in Fig. 4.

The implementation of our approach is shown in Fig. 5. Conventional history based prediction is implemented in IF stage. In DA stage, dynamic data dependence tracking is implemented for instructions in the same instruction window. The writing of decision registers is recorded in the FLAG register. For each conditional branch, the FLAG register is checked to test if the decision register is data dependent with other instructions. The dependence tracking result is recorded in CONDITION flag. If there is no other data dependence with the decision register, the CONDITION flag is set and the branch condition can be resolved. Oppositely, the CONDITION flag is clear if the decision register is data dependent with other instructions in the instruction window.

Algorithm: Context-based Branch Predictor

IW	Instruction window;
LOAD/COMP	Load/arithmetic instructions;
COND_BRANCH	Conditional branch inst;
Flag(i)	Flag to record the writing of destination register;
RESOLVE_COND	Function resolving the branch condition;
Condition	Boolean variable recording the outcome of RESOLVE_COND;
PAST_HIST	The branch history;
Branch_Pred	Boolean variable for branch prediction;
UPDATE_PREDICT	Function that updates the prediction status;

```
Begin
/* Initialization */
/* Set the flag to record the writing of the destination register */
(1)    For(each inst $I_i$ in IW) do
       {
(2)        If(($I_i$ ==LOAD)||( $I_i$ ==COMP))
(3)                Flag(Dest_Reg of $I_i$) $\leftarrow$ 1;
(4)        else if($I_i$ ==COND_BRACH)
             {
(5)                  If(Flag(Decision_Reg of $I_i$)==0)
/* For conditional branch, if the decision reg is not written, then we resolve branch condition */
(6)    Condition$\leftarrow$RESOLVE_COND($I_i$,Decision_Reg);
/* The Condition and the PAST_HIST are OR'd, and the result is deposited into Branch_Pred */
(7)                Branch_Pred$\leftarrow$(Condition||PAST_HIST);
/* Renew the predictor status */
(8)                UPDATE_PREDICT(Branch_Pred);
(9)          } # end of (4)
       } # end of (1)
end
```

Fig.5. Context-based Branch Prediction Algorithm.

The relationship between conventional history based predictors and our context-based predictor follows. In DA stage, the prediction results of conventional predictor in IF stage and the CONDITION flag are OR'd and are updated into the branch history. The combination of Comb/Context-based prediction results is demonstrated in Table 1. Once the CONDITION flag is set, there is no data dependence with the decision register, the branch condition can be resolved. So we take prediction from context-based predictor. Oppositely, if the CONDITION flag is clear, the decision register is data dependent with other instructions and we rather take predictions from original history based predictions. In our method, the history based prediction used in IF stage can be any predictors mentioned in Section II. In this paper, we choose Comb predictor as our history based predictor, which is also the combination of Two-Level Adaptive [4, 5] and Bimod [10, 11].

Table 1. Prediction Result Combination of Comb/Context-based Predictors

Comb Predictor Result	Context Predictor Result	Meanings
0	1	Data dependence resolved, take prediction from data dependence.
1	0	Data dependence not resolved, take original prediction.
1	1	Data dependence resolved, take prediction from data dependence.
0	0	Data dependence not resolved, take prediction from data dependence.

IV. EXPERIMENTAL RESULTS

In this section, we compare the performance of two branch predictors, *comb* and *context-based*. The former is the combination of Two-Level Adaptive [4, 5] and Bimod [10,11], which are used in most literatures and certain machines [5]. We use Wattch in our simulation. Wattch is developed by DBrooks [14] using SimpleScalar's *sim-outorder* [15] cycle-accurate model. We configure Wattch to use Alpha as the target machine. Test programs are compiled and statically linked for the Alpha instruction set using the Compaq Alpha C compiler. The instruction execution of Wattch is divided into five pipeline stages: fetch (IF), decode (DA), execution (EX), write back (WB) and commit (CT). The configuration of processor core and the branch predictor is shown in Table 2.

Table 2. Configuration of Simulated Processor

Processor Core	
Instruction Windows	16-RUU, 8-LSQ
Issue Width	4 instructions per cycle
Function Units	4 IntALU, 1 IntMul/Div
	4 FPALU, 1 FPMul/Div
	2 MemPorts
Base Branch Predictor	
Branch Predictor	1k-entry comb
Branch Target Buffer	512-entry, 4-way
Return Address Stack	8-entry

Figure 1 demonstrates snapshot of pipeline executing program **`SPECint95/go`**. In Figure 1, instruction **`ap`** is defined as *isolated* conditional branch. There is no other instruction writing into the *decision register* **`r1`**. Figure 6 and 7 show the pipeline snapshots using *comb* and *context-based* predictors, respectively. In Figure 6, instructions following conditional branch **`ap`** are still fetched into pipeline. These instructions are finally flushed till **`ap`** is resolved to be taken. In Figure 7, there is no other instruction being fetched into pipeline.

```
@cycle = 77
 [IF]    [DA]    [EX]    [WB]    [CT]
  av      ap      ao      am
  aw      aq              an
  ax      ar
  ay      as
          at
          au
```

Fig.6. Pipeline status of program SPECint95/go using comb Predictor.

```
@cycle = 77
 [IF]    [DA]    [EX]    [WB]    [CT]
  aq      ap      ao      am
  ar                      an
  as
```

Fig.7. Pipeline status of program SPECint95/go using context-based Predictor.

Figure 8 demonstrates the branch prediction rate of two predictors. In general, the prediction rate for branch outcomes can be improved from 0.3% to 0.35%. Figure 9 demonstrates the improvement in instruction per cycle (IPC). IPC can be improved by 5.55% at best and 2.19% on average. Figure 10 shows the improvement in cycle per instruction (CPI). CPI can be improved by 5.25% at best and 2.12% on average. Table 3 shows the comparison of power consumption for these two branch predictors. Power can be saved by 5.26% at best and 2.12% on average.

V. CONCLUSION

Conditional branches introduce control dependence between instructions, which degrade performance. Branch predictions are used to cope with the control dependence. However, most branch predictors use only branch history to make prediction, which results in mispredictions for branches. Conditional branches make a decision according to value of *decision registers*. In this paper, a *context-base branch predictor* is proposed to improve the branch prediction rate for conditional branches. Experimental results on SPECint95 [12] and MediaBench [13] programs reveal that the overall prediction rate can be improved by 0.3% to 0.35%. On the other hand, due to the reduction of mispredictions, the performance and the power consumption can both be improved. On average, the IPC can be improved by 2.19%, the CPI can be improved by 2.12%. The power consumption can be improved by 2.12% on average.

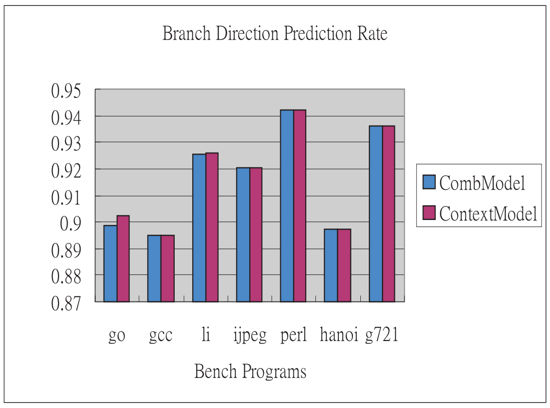

Fig.8. The branch prediction rate improvement.

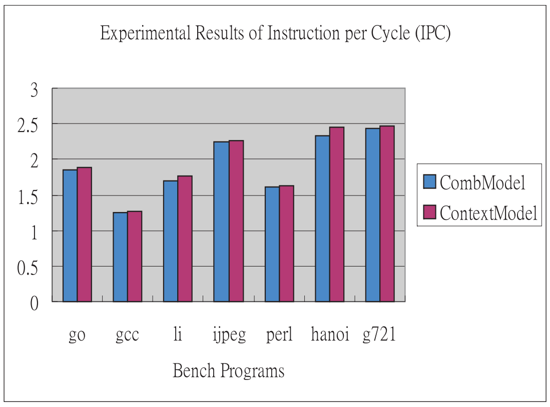

Fig.9. The improvement of instruction per cycle.

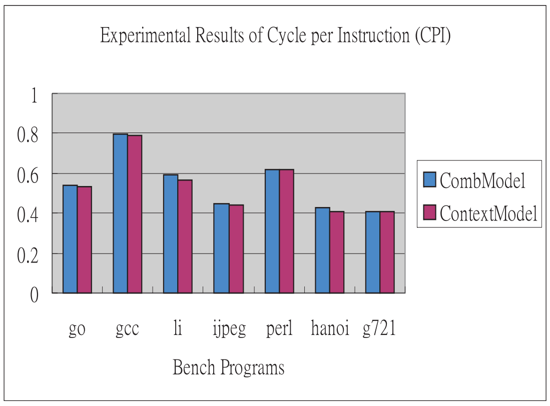

Fig.10. The improvement of cycle per instruction.

Table 3. Power consumption results of two predictors. (Unit: mW)

	Comb Model	Context Model	Power Saved
go	0.2771	0.2726	1.65%
gcc	180.4177	178.6750	0.97%
li	512.7205	490.9900	4.24%
ijpeg	11.2617	11.1668	0.84%
perl	1310.8531	1303.8629	0.53%
hanoi	0.8936	0.8466	5.26%
g721	117.1175	115.5685	1.32%

REFERENCES

[1] Evers, M. and Yeh, T.-Y., "Understanding Branches and Designing Branch Predictors for High-Performance Microprocessors," Proc. of The IEEE, Vol. 89, No. 11, pp. 1610–1620, 2001.

[2] Fisher, J. A. and Freudenberger, S. M., "Predicting Conditional Branch Directions From Previous Runs of a Program," in 5th Int. Conf. Architectural Support for Programming Languages and Operating Systems, pp. 85-95, 1992.

[3] Smith, J. E., "A Study of Branch Prediction Strategies," in 8th Int. Symp. Computer Architecture, pp. 135-148, 1981.

[4] Yeh, T.-Y. and Patt, Y. N., "Two-Level Adaptive Training Branch Prediction," in 24th ACM/IEEE Int. Symp. Microarchitecture, pp. 51-61, 1991.

[5] Kessler, R. E., McLellan, E. J., and Webb, D. A., "The Alpha 21264 Microprocessor Architecture," in Proc. of the 1998 Int. Conference on Computer Design, pp. 90-95, Oct. 1998.

[6] McFarling, S., "Combining Branch Predictors," Technical Report TN-36, Compaq Western Research Lab., June 1993.

[7] Loh, G. H. and Henry, D. S., "Predicting Conditional Branches with Fusion-based Hybrid Predictors," Proc. of Int. Symp. on Parallel Architectures and Compilation Techniques, pp. 165-176, Sept. 2002.

[8] Falcón, A., Stark, J., Ramirez, A., Lai, K., and Valero, M., "Prophet/Critic Hybrid Branch Prediction," Proc. of 31st Int. Symp. on Computer Architecture, pp. 250-261, June 2004.

[9] Chen, L., Dropsho, S., and Albonesi, D., "Dynamic Data Dependence Tracking and its Application to Branch Prediction," Proc. of Int. Symp. on High-Performance Computer Architecture, pp. 65-76, Feb. 2003.

[10] Lee, C.-C., Chen, I.-C. K., and Mudge, T. N., "The Bi-mode Branch Predictor," in Proc. of 30th ACM/IEEE Int. Symp. Microarchitecture, pp. 4-13, Dec. 1997.

[11] Bate, I. and Reutemann, R., "Efficient Integration of Bimodal Branch Prediction and Pipeline Analysis," Proc. of Int. Symp. on Embedded and Real-Time Computing Systems and Applications, pp. 39-44, Aug. 2005.

[12] Welcome to SPEC. The Standard Performance Evaluation Corporation. http://www.specbench.org/.

[13] Lee, C., Potkonjak, M., and Mangione-Smith, W. H., "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," Proc. of Int. Symp. on Microarchitecture, pp. 330-335, 1997.

[14] Brooks, D., Tiwari, V., and Martoncsi, M., "Wattch: A Framework for Architectural-level Power Analysis and Optimizations," Proc. of 27th Annual Int. Symp. on Computer Architecture, pp. 83-94, June 2000.

[15] Burger, D. and Austin, T. M., "The SimpleScalar Tool Set, Version 2.0." Technical Report CS-1342, University of Wisconsin-Madison, June 1997.