Reducing Dynamic Branch Predictor Lookups by Dynamically Collecting Non-branch Instructions

Guan Cheng Fu Jong Jiann Shieh
Department of Computer Science and Engineering
Tatung University
g9606022@ms.ttu.edu.tw shieh@ttu.edu.tw

***Abstract*—**Deeply pipelined architecture uses the dynamic branch predictor to enhance the performance. The dynamic branch predictor of the traditional scheme is exercised every cycle and then causes many power consumptions. For this reason, we intended to eliminate unnecessary lookups for dynamic branch predictor. In this paper, we proposed a scheme to dynamically collecting non-branch instructions between adjacent branch instructions during execution stage. We save the power by using the number we collected to filter the unnecessary lookups. Simulation results show that the dynamic branch predictor power is reduced by 63.79% and 76.58% in average for *SPECint2000* and *SPECfp2000* respectively with negligible performance loss.

Keywords—dynamic branch predictor, dynamically collecting non-branch instructions, power consumption.

I. INTRODUCTION

When designing the high-end processor, many designers use the dynamic branch predictor to improve ILP (Instruction-level Parallelism). Dynamic branch predictor uses cache-like BTB (branch target buffer) to store branch and target address, then using the predict bit of the direction predictor to decide the branch taken or not. Furthermore, the dynamic branch predictor can reduce the penalty of control hazard.

After simulating it shows only 13% and 8% of instructions are branch instructions in *SPECint2000* and *SPECfp2000* respectively. The conventional dynamic branch predictor is exercised every cycle. Lookup the branch target buffer is the main power consumption of the dynamic branch predictor, if we lookup BTB in every cycle then the power consumption will become very considerable. Consequently, we try to eliminate unnecessary lookups that will save the power.

The rest of this paper is organized as follows: Section II gives some related works about dynamic branch predictor. Section III describes the simulation environment and benchmarks and section IV proposes our scheme of dynamically collecting non-branch instructions, section V summarizes the paper.

II. RELATED WORK

In recently years, there are two ways to lower power consumption of dynamic branch predictor. One is a software profiling technique, which extracts the control-flow information during compile/link time. The other way uses extra hardware to dynamic filter unnecessary lookups.

Petrov and Orailoglu [1] proposed an application customizable branch target buffer (ACBTB), which is a software profiling technique. The proposed technique utilizes application-specific information regarding the control-flow structure of the program's major loops. Such information is used to completely eliminate the power hungry branch target buffer (BTB) lookups which normally occur at every execution cycle. Shuai Wang, Jie Hu and Sotirios G. Ziavras [2] proposed a filtering scheme to reduce the access to the BTB to achieve a significant dynamic energy in the BTB while maintaining the performance. They also studied the leakage behavior and its control in their BTB Access Filtering (BAF) design. Sung Woo Chung and Sung Bae Park [3] propose a low power branch predictor, which is based on the *gshare* predictor, by accessing the BTB only when the prediction from the PHT is taken. To enable this, the PHT is accessed one cycle earlier to prevent the additional delay. As a side effect, two predictions from the PHT are obtained at one access to the PHT, which leads to more power reduction. The proposed branch predictor reduces the power consumption, not requiring any additional storage arrays, not incurring additional delay (except just one MUX delay) and never harming accuracy. Authors in [4] introduce Branchless Cycle Prediction (BLCP) who uses a simple efficient structure to predict cycles where there is no branch instruction among those fetched, at least one cycle in advance. The proposed avoids accessing BTB during such cycles.

III. SIMULATION ENVIRONMENT

Before proposing our novel architecture, we describe the simulation environment and the benchmarks we use in this section. For the simulation, we used the *SimpleScalar* [5] 3.0d tool set simulator integrating with *Wattch* [6] version 1.02. *Wattch* augments the *SimpleScalar* cycle-accurate simulator (sim-outorder) with cycle-by-cycle tracking of power dissipation by estimating unit capacitances and activity factors. The *Wattch* is an architectural simulator that estimates CPU power consumption. The power estimates are based on a suite of parameterizable power models for different hardware structures and on per-cycle resource usage counts generated through cycle-level simulation. This paper uses the baseline configuration as shown in Table 1, which resembles, as much

as possible, the configuration of an Alpha 21264 [7] processor.

TABLE 1
Simulated Processor Configuration

Processor Core	
Instruction window	64 RUU; 32 LSQ
Fetch queue size	4 Instructions
Decode width	4 Instructions per cycle
Issue width	4 Instructions per cycle
Commit width	4 instructions per cycle
Function units	4 Int ALU 1 Int Mult/Div 1 FP ALU 1 FP Mult/Div
Memory Hierarchy	
L1 I-cache size	64KB, 2-way, 32B blocks, LRU
L1 D-cache size	64KB, 2-way, 32B blocks, LRU
L2 I-cache size	64KB, 2-way, 32B blocks, LRU
L2 Unity-cache size	2MB, 4-way, 32B blocks, LRU
I/D-TLB	128-entry, LRU
Dynamic Branch Prediction	
Direction predictor	Bimod 2K table
BTB	1024-entry, 2-way
Return address stack	32-entry
Penalty	
L1 hit	1 cycle
L2 hit	12 cycles
Branch mis-pred	1 cycle
Memory access	108 cycles
TLB miss	30 cycles

IV. DYNAMICALLY COLLECTING NON-BRANCH INSTRUCTIONS

In this section we propose a scheme to dynamically collecting non-branch instructions. As results, the number of the dynamic branch predictor's lookups is reduced. We collect the number of non-branch instructions between adjacent branch instructions and then update it to the corresponding entry of branch instruction from the BTB. When branch instruction is executed, the number of non-branch instructions is read out from the BTB. After that, the numbers of upcoming non-branch instructions are available, which will help to determine accessing the dynamic branch predictor or not. In our study, we intend to develop a way to reduce unnecessary dynamic branch predictor lookups. Accessing the dynamic branch predictor only depends on the number of non-branch instructions between adjacent branch instructions we collected during the execution stage.

Figure 1 shows block diagram of dynamically collecting non-branch instructions which is attached to the typical pipelined architecture. The non-branch instructions between the branch instruction and its subsequent branch instruction are collected by attaching a simply counter during execution stage. After that, the number of non-branch instructions collected for the correspondent BTB's entry of branch instruction is updated. At instruction fetch stage, the number of non-branch instructions from BTB is read out and updated to *Seq_bc*, which is used to determine accessing the dynamic branch predictor or not. When fetching non-branch instruction at the same time *Seq_bc* will be decreased by one and will not access the dynamic branch predictor. If *Seq_bc* is equaled to zero that means the next fetched instruction is a branch instruction and should be accessed the dynamic branch predictor.

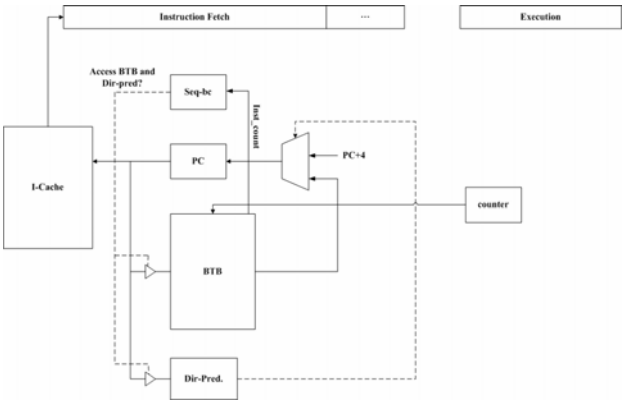

Figure 1. Dynamically collecting non-branch instructions architecture

In this section, we describe flow of dynamically collecting non-branch instructions. At first, we simply add two columns in BTB entry as shown in Figure 2. *Inst_count* records the number of non-branch instructions we collected and *Modify* records status that the counter should enable or not. The default number of *Inst_count* is zero and the default status of *Modify* is valid. *Modify* is valid if dynamic collection of non-branch instructions is not finish, otherwise, *Modify* is invalid. Different from the traditional scheme, we obtain not only the target address from BTB, but also the *Inst_count*. After getting the *Inst_count*, we will stop accessing the dynamic branch predictor.

Tag	Target Address	Inst_count	Modify

Figure 2. Proposed New BTB

Figure 3 shows the flowchart of dynamically collecting non-branch instructions during execution stage. There is an extra register named *Prev_branch* to record the branch instruction that is executed and a simply counter to count non-branch instructions between adjacent branch instructions. If a branch instruction is encountered in execution stage then *Prev_branch* is checked for null or not. When *Prev_branch* is null, the present branch instruction is updated as *Prev_branch* and reset the counter. Otherwise, when *Prev_branch* is in the BTB and *Modify* of the corresponding branch entry is valid, the counter is copied to *Inst_count* of corresponding BTB entry, reset the counter and invalidate *Modify*. Besides, if the instruction executed is not a branch instruction then the counter should be increased.

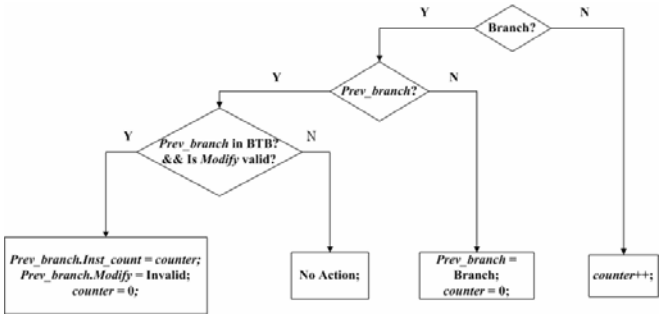

Figure 3. The flowchart of dynamically collecting non-branch instructions

V. SIMULATION RESULT

This section describes the results of our proposed scheme by simulation. The programs from the SPECcpu2000 suite and *SimpleScalar* v3.0 tool set are used to simulate our scheme. Figure 4 shows the percentage reduction in dynamic branch instruction lookups for the proposed scheme, the dynamically collecting non-branch instructions, with respect to the Alpha-21264 architecture for each selected *SPECint2000* benchmarks. Our proposed scheme can reduce the number of dynamic branch predictor lookups in average for *SPECint2000* is 88.17%. Figure 5 shows the percentage reduction in power consumption of dynamic branch predictor for the proposed scheme. The average power reduction for *SPECint2000* is 63.79%.

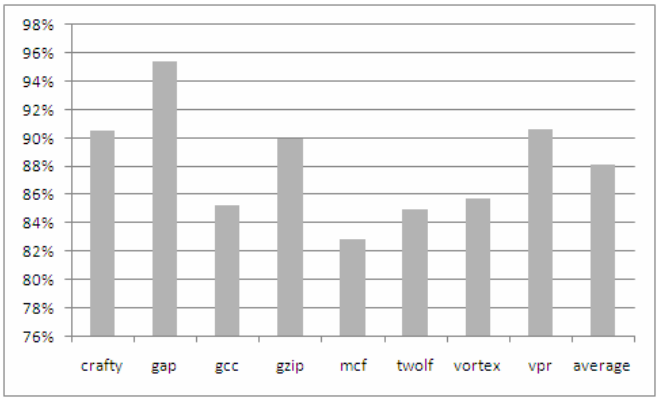

Figure 4. The percentage reduction of branch lookup in *SPECint2000*

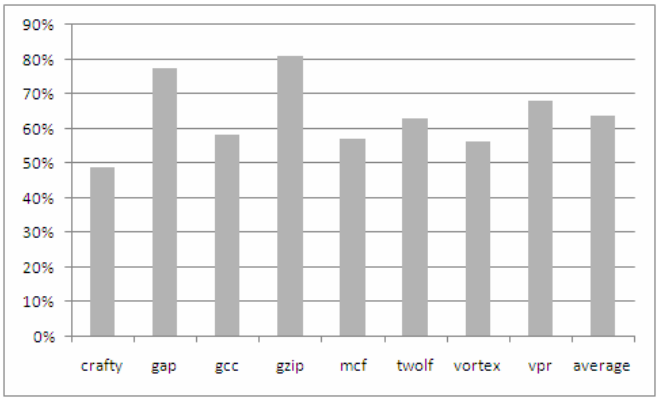

Figure 5. The percentage reduction of branch power in *SPECint2000*

Figure 6 shows the percentage reduction of branch lookup for proposed scheme in *SPECfp2000*. The proposed scheme can reduce the number of the dynamic branch predictor lookup in average for *SPECfp2000* is 93.03%. Figure 7 shows the average branch power reduction for *SPECfp2000* is 76.58%.

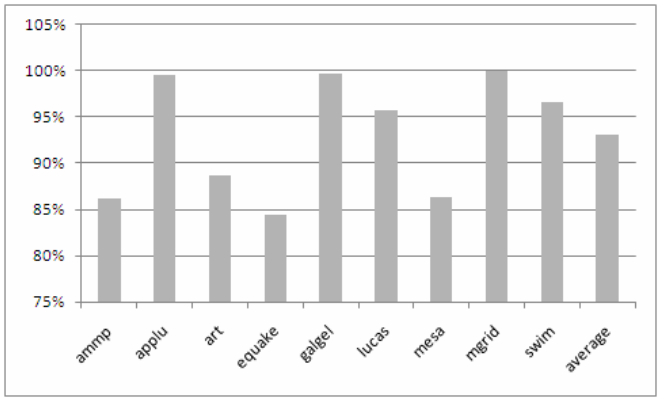

Figure 6. The percentage reduction of branch lookup in *SPECfp2000*

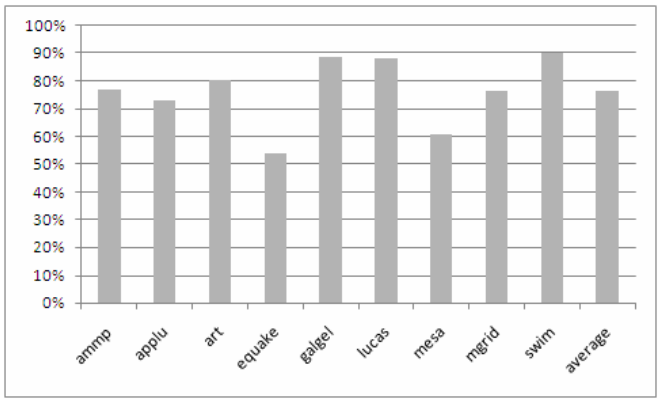

Figure 7. The percentage reduction of branch power in *SPECfp2000*

The metric used to evaluate performance is cycle per instruction (CPI). The performance penalty is 1 cycles for a miss-predicted branch in our proposed scheme. Figure 8 and Figure 9 show the performance for *SPECint2000* and *SPECfp2000* respectively. The performance of traditional branch lookup scheme for *SPECint2000* in average is 2.0489 and our proposed is 2.0498. Besides, the CPI of traditional in average for *SPECfp2000* is 1.2125 and our proposed is 1.2205.

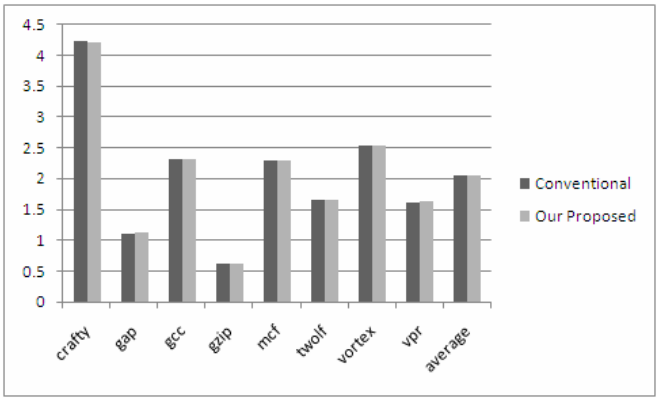

Figure 8. The CPI of different schemes in SPECint2000

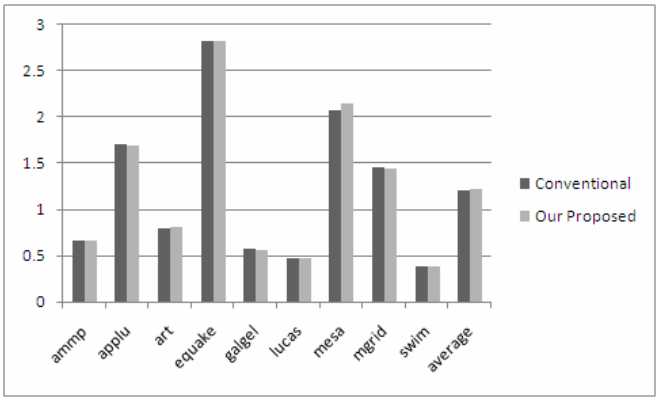

Figure 9. The CPI of different schemes in SPECfp2000

VI. Conclusion

In the paper, we proposed a scheme to dynamically collect non-branch instructions which can eliminate unnecessary dynamic branch predictor lookup. Traditional scheme differ from our proposed scheme is that traditional scheme accesses dynamic branch predictor every cycle. Our scheme filters unnecessary dynamic branch predictor lookup by the number of non-branch instructions that we collected during execution stage. We simply add two columns in BTB entry. The first column is *Inst_count* which is used to record the number of non-branch instructions between the branch instruction and the subsequent branch instruction. The second column is *Modify* which is used to determine to enable the counter, which counts the number of non-branch instructions executed. The experimental results show the percentage reduction in dynamic branch predictor lookup for our scheme in average for *SPECint2000* and *SPECfp2000* are 88.17% and 93.03% respectively. The dynamic branch predictor power is reduced by 63.79% and 76.58% in average for *SPECint2000* and *SPECfp2000* respectively.

ACKNOLODGMENT

This project is supported by Tatung Company (Taiwan) under contract: B98-I07-070

References

[1] Petrov, P., Orailoglu, A., "Low-power Branch Target Buffer for Application-Specific Embedded Processors," *IEE Proceedings on Computers and Digital Techniques*, vol. 152, no. 4, pp. 482 – 488, September 2003.

[2] Shuai Wang, Jie Hu, Sotirios G. Ziavras., "BTB Access Filtering: A Low Energy and High Performance Design," ISVLSI 2008: 81-86.

[3] Sung Woo Chung, Sung-Bae Park, "A Low Power Branch Predictor to Selectively Access the BTB," *Asia-Pacific Computer Systems Architecture Conference*, pp 374-384, September 2004.

[4] Deris, K.J., Baniasadi, A., "Branchless Cycle Prediction for Embedded Processors," *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 928 – 932, April 2006.

[5] Burger, D.C., Austin, T.M., "The SimpleScalar tool set, version 2.0," *Computer Architecture News*, vol. 25, no. 3, pp. 13 – 25, June 1997.

[6] Brooks, D., Tiwari, V., Martonosi, M., "Wattch: a framework for architectural-level power analysis and optimizations," *Proceedings of the 27th International Symposium on Computer Architecture*, pp. 83 – 94, June 2000.

[7] Kessler, R.E., McLellan, E.J., Webb, D.A., "The Alpha 21264 Microprocessor Architecture," *Proceedings 1998 International Conference on Computer Design*, pp. 90 – 95, October 1998.