Branch Predictor Design and Performance Estimation for a High Performance Embedded Microprocessor

Sang-hyuk Lee, Il-kwan Kim, and Lynn Choi

The Department of Electronics and Computer Engineering
Korea University
Seoul , 136-701
Tel : +82-2-3290-3896
Fax: +82-2-921-0544
E-mail : {hyukii, bitinno, lchoi}@korea.ac.kr

Abstract - AE64000 is a 64-bit embedded processor targeting high-end embedded applications such as HDTV, DVD, and 3D graphics. To achieve a higher performance for the AE64000, we design a branch predictor for the processor, and find the optimum parameters for the design through cycle-accurate simulations on SpecINT benchmarks and embedded applications (Dhrystone and Whetstone).

In AE64000 branch prediction is complicated by the Instruction Folding Unit (IFU) of the processor front-end. By predicting on a Pre-PC in the IFU rather than using a PC in the pipeline core, we can effectively eliminate branch misprediction penalty on a correct prediction. We have developed the AE64000 simulator to evaluate the performance of the designed branch predictor, and selected the optimum branch predictor configuration by considering cost-effectiveness as well as by analyzing the results generated from AE64000 simulator. The selected branch predictor has been implemented in Verilog and is added to AE64000 pipeline.

I. Introduction

AE64000 is a 64-bit high-performance microprocessor that Advanced Digital Chips (ADC) Inc. is developing for performance-demanding embedded applications. The processor has a 5-stage (IF-ID-EXE-MEM-WB) in-order pipeline and separate 8KB instruction and 8KB data caches. AE64000 processor implements the EISC (Extendable Instruction Set Computer) ISA developed by ADC Inc. for an embedded environment.

To reduce code size, EISC uses 16-bit fixed length instructions but can manipulate long immediate data by using a special instruction called LERI. Although the code of EISC can be denser than that of CISC or RISC, the overall performance can be degraded because the use of LERI instructions can increase the number of overall instructions. To avoid this performance degrade, AE64000 has instruction folding unit (IFU) which effectively processes all LERI instructions and passes the remaining instructions to pipeline core. Because IFU needs 2 clock cycles to remove LERI instructions and branch instructions are executed in ID stage, a taken branch causes 3-cycle branch misprediction flush, which can substantially degrade the performance of AE64000 pipeline. To achieve a higher performance for the AE64000 at low cost, we conclude that branch prediction is one of the most effective microarchitecture techniques and must be employed in AE64000 pipeline. In this paper we design a branch predictor for the AE64000, and evaluate the performance of the predictor compared to original AE64000 pipeline.

To validate the performance of the designed predictor and also to select the optimum design parameters, we have developed a cycle-accurate AE64000 simulator and ran simulations on two Integer SPEC95 applications, Dhrystone, and Whetstone benchmarks. Our performance improvements to AE64000 are two-folds. First, by careful redesign of the AE64000 pipeline, we could reduce the branch miss prediction penalty from 3 cycles to 2 cycles, which results in about 10% performance increase to AE64000 without any cost. Second, from the simulation results we found that the last-time predictor with 4-entry TAC is the most cost-effective and can improve the performance of AE64000 by additional 15% with only 3% extra hardware.

II. AE64000

Although AE64000 has a 5-stage in-order pipeline that is similar to DLX [5], the IFU of AE64000 operates like additional stages (called IFU stages) as detailed in Section II.A.

A. Front End of AE64000 Pipeline

Although AE64000 is a 64-bit processor, it uses 16 bit fixed-length instructions to reduce code size. To support operands of any size, EISC has a special instruction called LERI. A single LERI instruction can designate a 12-bit immediate data. Therefore, a 64-bit operand is divided into 6 LERI instructions. To prevent the performance degrade caused by processing additional LERI instructions,

AE64000 has a special component called IFU at the front-end of the pipeline. IFU detects and processes all the LERI instructions by restoring long immediate data from consecutive LERI instructions to a special register called ER, and propagates the remaining (LERI-free) instructions to the pipeline core. IFU fetches 4 instructions at a time and has buffers to store 12 instructions. IFU needs 2 clock cycles for folding LERI instructions. One clock cycle is used for a PPC update, and the other clock cycle for instruction folding. If a branch instruction is recognized in ID stage, branch misprediction penalty (we can say that AE64000 assumes all branches are not taken) is 3 clock cycles.

Fig. 1 shows a code example of a branch instruction and its pipeline diagram. CMP instruction sets the branch condition at EXE stage that is forwarded to ID stage where a branch target address is calculated at the same cycle. If the decided branch condition is taken (misprediction), the ADD instruction in IF stage as well as all the instructions in the IFU are flushed and the next instructions are fetched from the calculated target address.

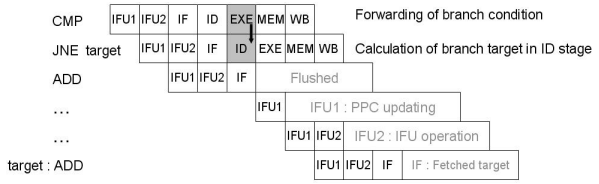

Fig. 1. Branch Execution

B. *Issues in Branch Prediction for AE64000*

There are two issues that complicate the branch prediction for AE64000. First, since four instructions are fetched at a time into IFU, multiple branch instructions can be fetched every cycle. However, a previous study[6] shows that branches occupy only 24% and 5% of dynamic instruction count in integer and floating-point benchmarks respectively. Furthermore, because AE64000 uses LERI instructions, the density of branch instructions is even smaller compared to other processors. Therefore, in case two or more branch instructions are fetched at the same time, in our proposed predictor, we predict only the first taken branch and ignore later branches. In this way, multi-way branch prediction [3] can be avoided. In addition, one study shows that the additional gain that can be achieved by multi-way branch prediction is negligible [3].

The second issue is at which pipeline stage branch prediction should be performed. There are two choices. One is to predict a branch using a PPC in the IFU at IFU1 stage. And, the other is to predict a branch using a PC in the pipeline core at IF stage. It turns out that the branch prediction with PC has a branch penalty of 2 clock cycles due to updating PPC and folding target instructions in IFU even if the branch prediction succeeds. However, the branch prediction with PPC enables IFU to fetch an instruction from the predicted target without a penalty. Therefore, we propose branch prediction with PPC.

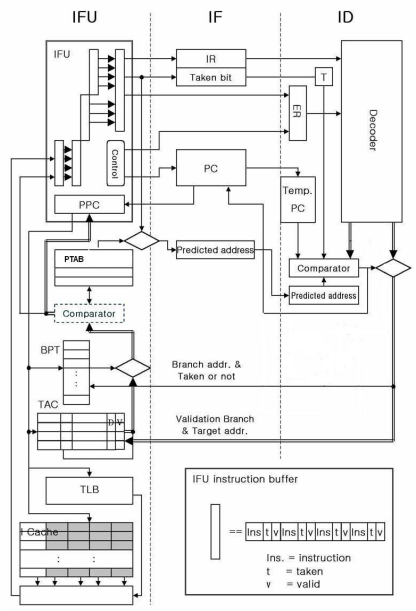

Fig. 2. Diagram of Branch Predictor for AE64000

C. *Reducing Branch Misprediction Penalty for AE64000*

After carefully analyzing the AE64000 pipeline, we found that we can reduce the branch misprediction penalty from 3 cycles to 2 cycles. This can be achieved by updating PPC at the same cycle that PC is updated by adding a multiplexer in IFU.

III. Branch Prediction for AE64000

Fig. 2 is the block diagram of AE64000 pipeline with the proposed branch predictor. We separate branch predictor with target address cache to use area more effectively since BPT and TAC can have different number of entries. In the following, we describe the execution of a branch instruction in each pipeline stage.

A. *IFU stage*

In IFU stage, the recognition of a branch and the prediction of target address and branch condition for the branch are performed.

When an I-cache is accessed in IFU stage, PPC is sent to branch prediction table (BPT) and target address cache (TAC) for branch prediction. A TAC hit implies the existence of a branch among the 4 instructions fetched. And, if the output of BPT is taken, then TAC supplies branch target address to PPC to fetch instructions from the predicted target address at the next cycle.

Since 4 instructions are fetched at a time, we need to determine which instruction is a branch. To do that, the tag field in TAC includes the complete branch address including the lower 3 bits, which is omitted in PPC. On a prediction, this is propagated to IFU, and used to find a branch among 4 instructions. In addition, this is used to mark the

corresponding branch instruction in the instruction buffer.

Since PPC can be updated from the predicted target, on a branch prediction, the next branch to be predicted must be a later instruction after the target. In other words, a branch before the target should not be predicted as a branch. For this, a comparator is used as shown in Fig. 3 to check whether a predicted branch is before or after the target.

The predicted branch condition and the predicted target address should be verified by comparing them with the result of a real branch execution in the pipeline core. Therefore, the predicted branch target address is stored in Predicted Target Address Buffer (PTAB) of IFU, which can contain up to 3 predicted target addresses. When a predicted taken branch instruction is propagated from IFU to the pipeline core, its predicted target address in PTAB is also sent out to the pipeline core. And, both the predicted target address and the predicted branch condition are compared with actual target address and branch condition later in ID stage.

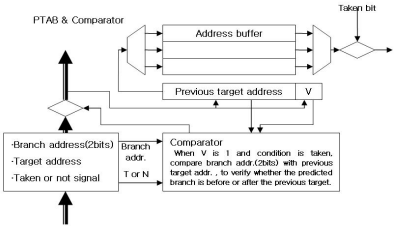

Fig. 3. Predicted Target Buffer and Comparator

B. *IF stage*

In IF stage, both the taken bit (which indicates a taken branch) and the predicted target address are propagated to ID stage. Also, the value of PC is updated with the predicted target address at the next cycle.

C. *ID stage*

In ID stage, the confirmation of a branch condition, the verification of a predicted target address, and updating of BPT and TAC are performed.

The branch condition was decided at the EXE stage of a previous compare instruction and forwarded to the ID stage of the branch instruction. The actual target address is calculated in ID stage. Then, both the target address and the branch condition are compared with predicted target address and predicted branch condition (taken). If either a branch is not taken or the actual target address is different from predicted values, a branch misprediction occurs, and instructions in IF and IFU buffers as well as predicted target addresses in PTAB are flushed. And, a redirection occurs from the actual target address. The result (success or failure) of a branch prediction, the addresses of both the branch instruction and the actual target are sent to BPT and TAC to modify the information in BPT and TAC.

IV. AE64000 simulator

We have developed a cycle-accurate AE64000 simulator to evaluate the additional performance gained by the branch predictor and to determine the optimal design parameters for the branch structures in terms of both cost and performance.

In the following, we simulate the following three AE64000 configurations.

1. The original AE64000 without a branch predictor
2. The optimized AE64000 pipeline without a branch predictor (with a modification to reduce branch misprediction penalty from 3 cycles to 2 cycles)
3. The optimized AE64000 with a branch predictor

A. *Organization of the simulator*

Fig. 4 shows the organization of the AE64000 simulator. The AE64000 simulation model consists of memory, cache, core, and the added branch predictor. The AE64000 compiler translates a benchmark into the memory-mapped AE64000 binaries by using the addresses of RAM, ROM and stack pointer set by user, which is an input to the simulator.

The simulation model is flexible enough to parameterize the various configurations of predictors such as types of branch predictors, size, associativity, and replacement policy (LFU, LRU) of the prediction structures.

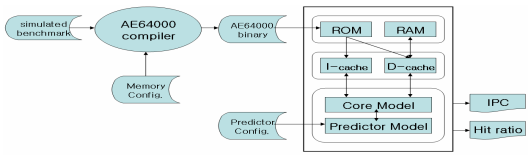

Fig. 4. The block diagram of AE64000 simulator

B. *Simulated branch predictors*

Three kinds of branch predictors are simulated, namely, last-time predictor[5], bimodal predictor[1], and g-share predictor[1]. The last-time branch predictor shown in Fig.5 is organized in much the same way as branch target buffer [2]. All the predictors are accessed through PPC in the IFU.

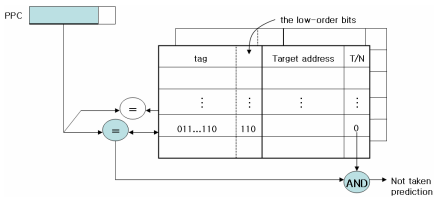

Fig. 5. The last-time branch predictor

V. Performance evaluation and analysis

We assume a cache miss penalty of 4 clock cycles. We also assume that indirect branches are not predicted to reduce the predictor size in an embedded environment. Total of 1 billion instructions are simulated for each benchmark.

A. *Benchmarks and Results*

We simulate Compress and Go applications from

SPECInt95 benchmarks, and a complete suite of Dhrystone and Whetstone benchmarks. Dhrystone and Whetstone are small programs used for embedded processors' test.

Table 1 shows the performance improvement gained by reducing branch misprediction penalty from 3 cycles to 2 cycles on AE64000 pipeline.

Classification	3-cycle misprediction penalty	2-cycle misprediction penalty
Compress	0.6787	0.7429
Go	0.6905	0.7322
Dhrystone	0.6500	0.7222
Whetstone	0.6325	0.7109

Table 1. IPC on different branch miss prediction penalty

By decreasing the miss penalty, IPC increases by as little as 6% in Go application and up to 12% increase in Whetstone benchmark, resulting in 10 % performance improvement on average. This implies that a single cycle reduction on branch miss prediction can affect the performance of AE64000 substantially.

Fig. 6 shows IPC versus the number of TAC entries for last-time branch predictor. The bimodal and g-share branch predictor is not as good as the last-time predictor in cost-effectiveness, thus those results and analyses are not shown in this paper. Note that the branch prediction hit rate combines both branch prediction hit rate (condition prediction) and TAC hit rate (target address prediction). As we expected, IPC increase as we increase the number of TAC entries.

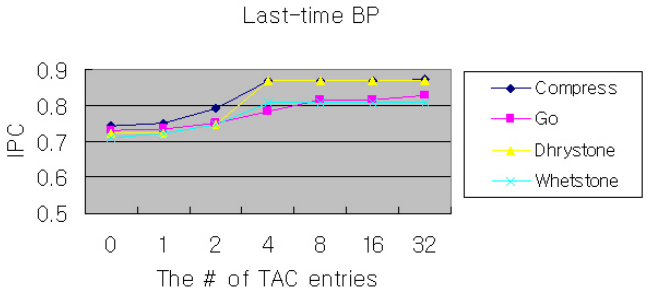

Fig. 6. IPC of last-time predictor

VI. Conclusion

In estimating the hardware cost, we assume 6 NAND gates per bit for BPT and 3K NAND gates for the remaining control part of the added branch predictor. 160K NAND gates are used for the AE64000 pipeline. The cost-effectiveness is defined by the % of performance increase in IPC (dIPC) divided by the % of added hardware cost (dCost) compared to the original AE64000 pipeline.

Fig. 7 shows the average cost-effectiveness of last-time predictor as the number of TAC entries increases. The 4-entry fully associative TAC shows the best cost-effectiveness in the figure.

Fig. 8 represents the ratio when the designed components are added. The blue color represents the average 10% IPC improvement made by the optimization of reducing branch misprediction penalty from 3 to 2 cycles. The light blue color represents the additional IPC increase made by 4-way set-associative 4-entry TAC.

In the optimized pipeline with the last-time predictor, the required cost is about 3500 gates in this configuration. This implies that we can improve the performance of AE64000 pipeline by about 25% with only 3% extra hardware cost.

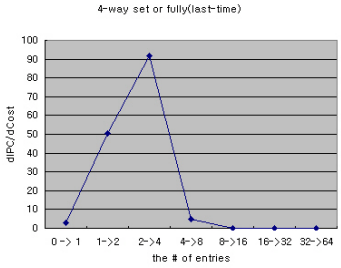

Fig. 7. Cost-effecitiveness of last-time predictor

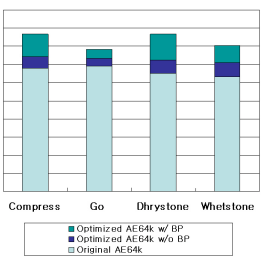

Fig. 8. Performance ratio due to each component

Summary

In this paper, we proposed a branch predictor optimized for AE64000. To validate the design and evaluate the performance improvement possible by the design, we have developed a cycle-accurate simulator for the processor. From the simulation analysis, we conclude that the last-time predictor with 4-entry TAC shows the best performance per cost among the predictors we considered. With a 3% extra hardware cost, we could improve the performance of AE64000 by about 25% (about 10% by reducing the branch misprediction penalty with a careful redesign of the processor front-end and about 15% by adding the branch predictor we proposed).

References

[1] S.McFarling, "*Combining Branch Predictors*", DEC WRL Technical Note TN-36, June 1993

[2] Brian K.Bray, M.J.Flynn, "*Strategies for branch target buffers*", Proceedings of the 24th Annual International Symposium on Microarchitecture, September 1991

[3] T.Y.Yeh, D.Marr, Y.Patt, "*Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache*", in proceeding of the 1993 International Conference on Supercomputing, 1993

[4] John L Hennessy, David A Patterson, "*Computer Architecture: A Quantitative Approach*" 2nd edition, published by Morgan Kaufmann, 1996

[5] T.Y.Yeh and Y.N.Patt, "*Alternative Implementations of Two-Level Adaptive Branch Prediction*", in Proceedings of the 19th Annual ACM/IEEE International Symposium on Computer Architecture, 1992

[6] The Standard Performance Evaluation Corporation, http://www.specbench.org