Area-Aware Optimizations for Resource Constrained Branch Predictors Exploited in Embedded Processors

Babak Salamat
School of Information and Computer Sciences
University of California, Irvine
Irvine, CA USA
bsalamat@ics.uci.edu

Amirali Baniasadi, Kaveh Jokar Deris
Department of Electrical and Computer Engineering
University of Victoria
Victoria, BC Canada
{amirali, kaveh}@ece.uvic.ca

***Abstract*— Modern embedded processors (e.g., Intel's XScale) use small and simple branch predictors to improve performance. Such predictors impose little area and power overhead but may offer low accuracy. As a result, branch misprediction rate could be high. Such mispredictions result in longer program runtime and wasted activity. To address this inefficiency, we introduce two optimization techniques: First, we introduce an adaptive and low-complexity branch prediction technique. Our branch predictor removes up to a maximum of 50% of the branch mispredictions of a bimodal predictor. This results in improving performance by up to 16%. Second, we present front-end gating techniques and reduce wasted activity up to a maximum of 32%.**

I. INTRODUCTION

Our goal in this work is to improve performance and power dissipation in embedded processors. This is done by a) improving the branch prediction accuracy and b) stopping wrong path instructions from entering the pipeline. We introduce efficient, simple and low-overhead techniques to reduce branch misprediction rate and speculation cost considerably.

Traditionally, using branch prediction has been restricted to high-performance processors. However, in recent years, embedded processors such as the Intel's XScale processor have used simple predictors to improve ILP [7].

Branch prediction is essential as it provides steady instruction flow at the fetch stage. Unfortunately, predictors are not perfect and make mispredictions. Such branch mispredictions result in longer program runtimes and energy wasted down the mispredicted instruction path. It is expected that as embedded processors exploit deeper pipelines, branch misprediction cost will also increase. To avoid an increase in misprediction cost designing more accurate yet area- and power-efficient branch predictors is necessary.

Previous study has introduced several highly accurate branch prediction techniques for high-performance processors [2, 8, 11, 13, 14]. Unfortunately, embedded processors perform under resource, area and power constraints and do not afford the complex techniques used in highly accurate branch predictors.

To address this inefficiency in this work we present two classes of optimization techniques and make the following contributions:

First, we introduce an Adaptive, Small and Low-complexity branch prediction technique, referred to as ASAL. ASAL relies on periodic branch instruction behavior measurement to pick the best predictor for different program phases. Accordingly, we reduce the number of branch mispredictions considerably.

Second, we propose a set of power-efficient and area-aware pipeline gating methods and reduce misprediction cost while maintaining performance. As pipeline gating relies on accurate branch confidence estimation [10], we also introduce low-overhead branch confidence estimation techniques.

The rest of the paper is organized as follows. In Section 2 we discuss ASAL. In Section 3 we explain area-aware pipeline gating in more details. In Section 4 we present methodology and results. In Section 5 we review related work. Finally, in Section 6 we offer concluding remarks.

II. ASAL

Accurate branch prediction requires using different information regarding instruction PC and local and global history simultaneously. While some branch instructions are better predicted using global history, there are others that are more accurately predicted if local history is used. The bimodal predictor [11] (referred here as BMD) uses instruction PC to make branch predictions while global gshare [11] (referred to as GG) is an example of a predictor that makes predictions based on global history. Figure 1 shows the schematic of each predictor.

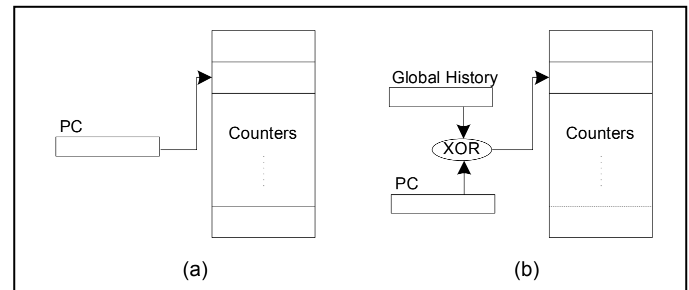

Figure 1. a) BMD: Instruction PC is used to access predictor. b) GG: Instruction PC is xor'ed with the global history to access predictor.

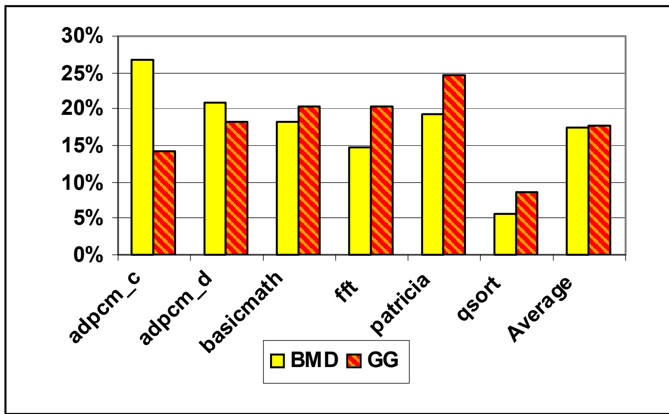

Figure 2. Misprediction rate for BMD and GG predictors. *Lower is better*.

In Figure 2 we report the misprediction rate for a subset of MiBench benchmarks [5] studied here, assuming either 128-entry BMD or a 128-entry GG is used. We pick 128-entry predictors as this is the same size used by the XScale embedded processor. As reported, mispredictions are quite frequent and can be as high as 27% (e.g., for adpcm_c). This is partially due to the simplicity of the predictors used. As presented in Figure 2, while BMD does a better job for branch instructions in some applications (e.g., fft), for others (e.g., adpcm_c) GG is a better choice. In other words, while branch instructions within one application are better predicted using global history, branch instructions within another application will benefit if global history is not taken into account.

One way to pick the right predictor for each branch instruction is using the combined predictor and exploiting multiple predictors [11]. However, in addition to the overhead associated with using multiple predictors, picking the best predictor per branch requires additional structures (e.g., the selector) which may not be affordable in an embedded processor. Moreover, combined predictors rely on pre-decided configurations and follow the one-size-fits-all approach which appears to be non-optimal across all applications.

To avoid such problems, we avoid picking the best predictor per-branch. Rather our goal is to find the right predictor for the right interval.

We investigated branch instruction behavior change during regular intervals. Our study shows that no single predictor performs best across all applications. Moreover, even within an application, the best prediction scheme may change from one interval to the other.

Based on the above, our goal is to find the best prediction scheme (GG or BMD) for each interval. Once the appropriate scheme is identified, we use the same area and resources used by the previous scheme to implement the alternative scheme. Note that we pick BMD as it is already being used by commercial processors and GG as it has a structure similar to BMD (see Figure 1) and is highly accurate for many applications. As such, switching from one scheme to another comes with very little complexity and could be achieved in a fast and efficient manner (more on this later).

We introduce both static and dynamic techniques. In our static approach, we use previously recorded accuracy for BMD and GG during fixed intervals (e.g., 2 million instructions) to pick the best scheme. In our dynamic approach, we dynamically measure how each predictor performs during short evaluation periods. We use the result of the evaluations to pick one of the two predictors for longer periods.

A. Static ASAL

Static ASAL (S-ASAL) relies on profiling. We store the accuracy of BMD and GG during 2M instruction intervals of execution of a certain application. Then we use this data to select the more accurate scheme for each interval. A possible implementation of this technique is done by adding an instruction to the instruction set to perform switching between predictors. A customized compiler can use the profiling information to add the instruction to the appropriate locations of the application code. The processor uses a single bit to switch the prediction scheme when necessary.

B. Dynamic ASAL

Dynamic ASAL (D-ASAL) evaluates how BMD and GG perform during short evaluation periods. D-ASAL enters the evaluation period after executing a pre-decided number of instructions. During the evaluation period, D-ASAL evaluates BMD and GG by using each to predict a fixed number of consecutive branch instructions executed in the application. At the end of the evaluation period, D-ASAL compares the number of mispredictions caused by each of the schemes. ASAL picks the scheme with the less number of mispredictions and uses the scheme during the next interval. Once one interval is finished, D-ASAL repeats the same evaluation process again.

The evaluation period is measured in terms of the number of branch instructions executed. Our study shows that, for the 128-entry predictor, best results are achieved when the evaluation period is 1024 branches. To decide the interval time, we tested many possibilities. The best result was achieved for intervals of two million instructions. Our experiments also show that the result is better if we do not reset the predictors at the beginning of the evaluation periods.

C. Hardware Implementation

Both S-ASAL and D-ASAL are used to find the better prediction scheme. Thus the hardware structure of branch predictor is the same for both methods.

Figure 3 shows the schematic of ASAL. As presented the same table is used to implement both BMD and GG. A MUX is used to access this table using the PC (if BMD is being used) or PC xor’ed with global history (if GG is being used). The GG Branch History Register and PHT are not updated speculatively. The MUX is controlled by the evaluation process. The evaluating algorithm is shown in Figure 4.

The overhead associated with ASAL is negligible. ASAL requires a maximum of three counters to count the number of mispredictions and keep record of evaluation periods and time intervals. For a 256-entry predictor, where the evaluation period is 2048 branches and the time interval is 2M instructions, we need two 11 bit counters to store

mispredictions in the evaluation phase and a 21 bit counter to keep track of evaluation periods and time intervals. The overall storage area required is 43 bits which is only 8% of the table size.

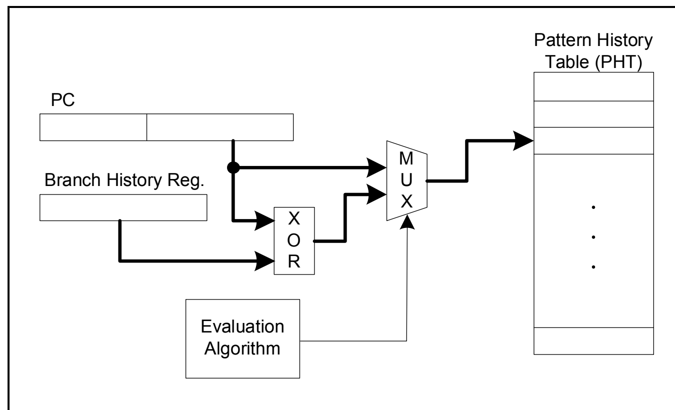

Figure 3. Schematic of ASAL

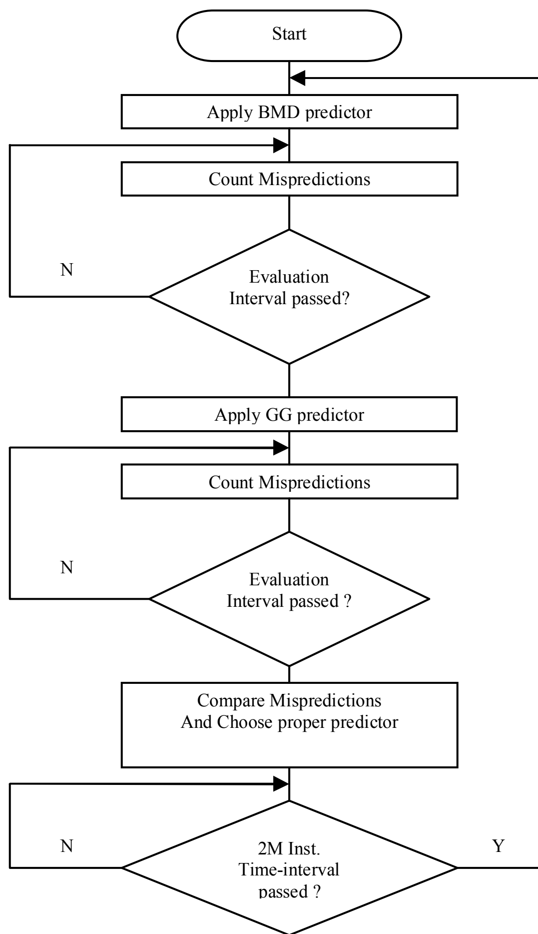

Figure 4. The evaluation algorithm

III. AREA-AWARE PIPELINE GATING

Possible branch mispredictions result in fetching instruction from the wrong path. We refer to the mistakenly fetched instructions as wasted activity (WA). Note that mispredicted instructions do not commit and are flushed as soon as the mispredicted branch is resolved. Our study shows that WA can be more than 23% for embedded applications. Figure 5 reports WA for the applications studied here.

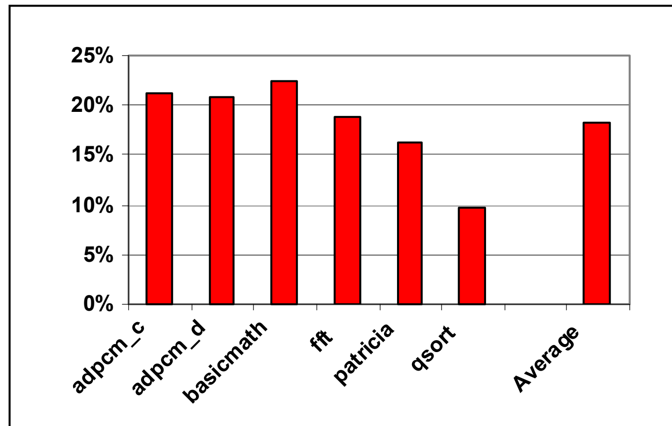

Figure 5. Extra Wasted Activities. Lower is better.

We stall instruction fetch when there is a high chance that the fetched instructions will be flushed. We gate the pipeline front-end when there is low confidence in the executed instructions. As such we need mechanisms to identify low confidence branches.

Previously suggested estimators rely on complex structures which are not affordable in an embedded processor. To apply front-end gating in the embedded space we introduce low-overhead techniques.

A. *Static Confidence Estimation*

The static confidence estimation technique is based on profiling. We execute each benchmark and keep a record of the low confidence branches. We consider a static branch as low confidence if its misprediction rate is over 25%. We gate the pipeline when there are at least two low confidence branches inside the pipeline.

B. *Dynamic Confidence Estimation*

1) History-based Confidence Estimation: In this method we assume that recently mispredicted branches are more likely to be mispredicted in the future. As such we keep track of recently fetched branch instructions' confidence using a very small 16-bit structure. This structure is a PC-indexed 8-entry table where there is a 2-bit saturating counter associated with each entry. The 2-bit counter is incremented for accurately predicted branches. We reset the associated counter if the branch is mispredicted. We look up this structure at fetch and in parallel to probing the branch predictor. If the 2-bit counter is not saturated we consider the branch as low confidence. The table and counters are updated at the commit stage and after the actual outcome of a predicted branch is known.

Previously suggested pipeline gating methods gate the front-end if the number of low-confidence branches exceeds a pre-decided threshold [10]. We take into account instruction

behavior changes in embedded applications and add a level of adaptivity and decide the gating threshold dynamically. To do this, we use the number of in-flight branch instructions and average misprediction rate.

For applications with small number of branches, aliasing is low and our confidence estimator is more likely to do a more effective job. This is particularly true if average misprediction rate for an application is low. As such, for applications with lower number of branches and low misprediction rate, we gate the pipeline if the number of low-confidence branches exceeds one. For applications with higher number of branches and higher misprediction rates, we gate the pipeline if the number of low confidence branches exceeds two.

Accordingly, we measure instructions per branch (IPB) every 256 instructions and set the threshold to two if IPB drops below 4 (indicating a high number of branch instructions) and if the misprediction rate is above 10%.

2) Predictor-based Confidence Estimation: In the second method we assume that the saturating counters which are already being used by the branch predictor indicate branch instruction confidence. By using the already available structures we minimize the hardware overhead.

At fetch, we mark a branch as low confidence if the corresponding counter is not saturated. We gate the pipeline if the number of low-confidence branches exceeds a dynamically decided threshold. We increase the gating threshold from 1 to 2 if IPB drops below 4.

3) Combined Confidence Estimation: Each of the two methods discussed above captures a different group of low-confidence branches. To identify a larger number of low-confidence branches, we combine the two techniques: a branch is considered low-confidence if either the history-based or predictor-based confidence estimator marks it as low-confidence. By using this technique we achieve higher WA reduction while maintaining performance.

C. Area Overhead

The history-based technique uses an 8-entry confidence estimator containing eight 2-bit counters. We also need an 8-bit counter to count the instruction intervals, a 6-bit saturating counter to count the number of branches in each interval and a 3-bit saturating counter to keep track of mispredictions. The total area requirement is equivalent to 33 bits.

The predictor-based method uses an 8-bit counter and a 6-bit saturating counter to keep track of instruction intervals and the number of mispredicted branches respectively. Thus, the total required area is only 14 bits.

The combined method uses the same structures used by the history-based technique. It also uses the already available branch predictor counters. Thus, the area overhead is 33 bits.

IV. Methodology and Results

In this section, we present our analysis and simulation results for ASAL and pipeline gating methods. We report predictor accuracy in 4.1. We report wasted activity reduction in Section 4.2. Performance is reported in Section 4.3.

As explained earlier, Intel's XScale processor uses a BMD predictor. As such, we compare ASAL to this predictor. We also report how replacing the BMD with a GG with the same size impacts mispredictions rate.

An alternative to ASAL is using the same resources to implement the combined scheme [11]. As explained earlier, the combined predictor may not be the best choice for embedded processors. Nonetheless, to provide better understanding, we also compare ASAL to a combined predictor which uses the same overall area.

We used a subset of MiBench benchmark suite compiled for MIPS instruction set. All benchmarks were run to termination. We performed all simulations on a modified version of the SimpleScalar v3.0 tool set [1]. Configuration of the processor modeled was similar to that of Intel's XScale processor. Table 1 shows the configuration used. To show how ASAL impacts different predictor sizes we report for 128- and 256-entry predictors. Table 2 also shows the configurations of the branch predictors used.

TABLE 1. PROCESSOR BASE CONFIGURATION

Pipeline Length	5 stages
Issue Width	In-Order: 2
Functional Units	1 I-ALU, 1 F-ALU, 1 I-MUL/DIV, 1 F-MUL/DIV
BTB	128 entries
Main Memory	Infinite, 50 cycles
Inst/Data TLB	32 entries, fully associative
L1 - Instruction/Data Caches	32K, 32-way SA, 32-byte blocks, 1 cycle
L2 Cache	None
Load/Store queue	8 entries
Register Update Unit	8 entries
Branch Mispred. Penalty	4 cycles

TABLE 2. BRANCH PREDICTORS' CONFIGURATIONS

	128 entries	**256 entries**
Combined	32-entry BMD / 64-entry GG with 6-bit history / 32-entry selector	64-entry BMD / 128-entry GG with 7-bit history / 64-entry selector
Gshare	128 entry GG with 6 bit history	256 entry GG with 7 bit history
Bimodal	128-entry	256-entry

A. Prediction Accuracy

Branch behavior may change during execution of a single application and also on context switching when one application is replaced by another. To simulate a realistic context switching scenario, we simulated context switching for random intervals between 100K and 1M instructions [12]. At the point

of context switch, we loaded all predictor tables and branch history pattern structures with values obtained from predictor tables generated by other applications. Our study shows that similar accuracy improvements could be achieved for a single application and in the absence of context switching. Therefore, and in the interest of space, we only report predictor accuracy in the presence of context switching.

In Figure 6, bars from left to right report misprediction rates for BMD, GG, combined (CMB), S-ASAL and D-ASAL. For most benchmarks ASAL outperforms other schemes eliminating up to half of the mispredictions (i.e., for adpcm_c). Moreover, while GG and CMB improve prediction for some benchmarks (e.g., adpcm_c and adpcm_d) they do deteriorate it for others (e.g., basicmath, patricia and fft). ASAL, either reduces the number of mispredictions considerably (up to 50%), or maintains it at the same level compared to bimodal.

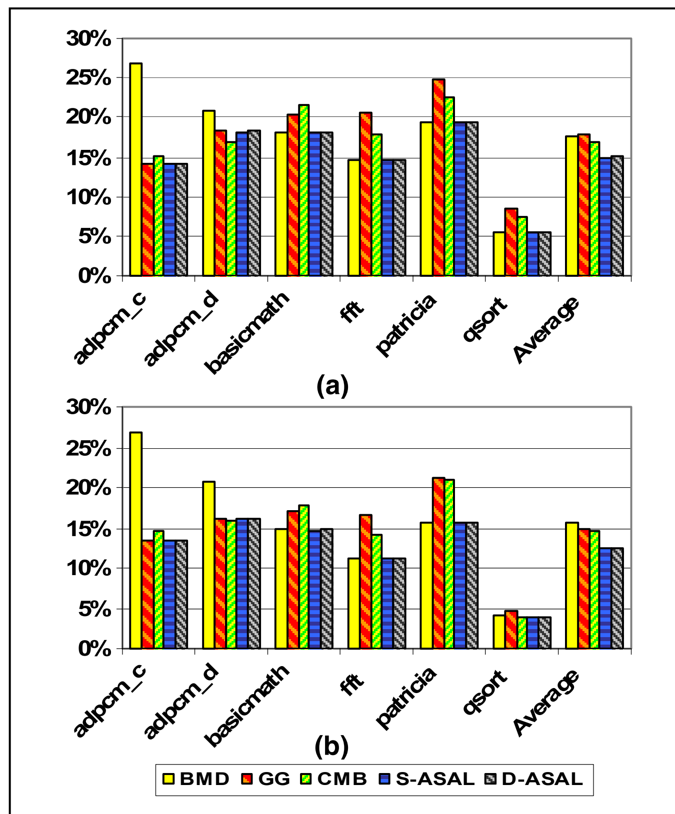

Figure 6. Bars from left to right show misprediction rate for BMD, GG, CMB, S-ASAL and D-ASAL for (a) 128-entry (b) 256-entry. *Lower is better*.

B. Wasted Activity Reduction

Figure 7 shows WA reduction for the proposed front-end gating techniques compared to XScale. On average, the combined method has the highest WA reduction. Maximum WA reduction is 32% (for adpcm_c).

C. Performance

Figure 8 reports performance for a processor that uses 128-entry GG, CMB and D-ASAL compared to an XScale-like processor. Average performance gains for GG, CMB and D-ASAL are 2.1%, 2.3% and 3.2% respectively. Maximum performance improvement is 16% (for adpcm_c).

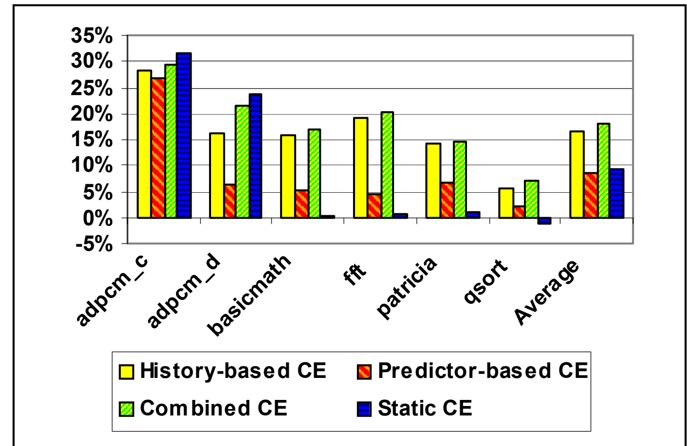

Figure 7. WA reduction (a negative value means an increase in the amount of WA). *Higher is better*.

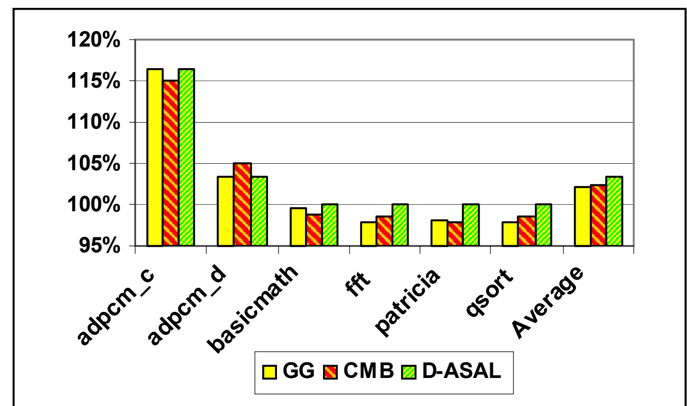

Figure 8. Performance for 128-entry GG, CMB and D-ASAL compared to BMD. *Higher is better*.

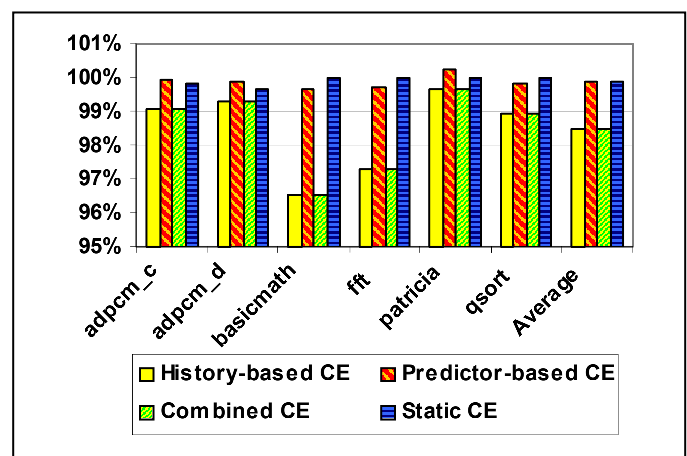

Figure 9. Performance after applying pipeline gating compared to a conventional processor. *Higher is better*.

In Figure 9 bars from left to right report performance for history-based, predictor-based, combined and the static pipeline gating methods compared to a processor similar to XScale using a bimodal predictor that does not use pipeline gating. Reportedly, static method has the lowest amount of IPC loss which is less than 0.1%. Among the dynamic techniques predictor-based method shows the lowest amount of performance. Average performance loss is 0.14% for this technique. Note that we improve performance for patricia when using the predictor-based technique. Our studies show that it is a result of an increase in the I-Cache hit rate after applying pipeline gating.

V. Related Work

Previous work has suggested highly accurate branch predictors [2, 6, 8, 9, 11, 13, 14] for high-performance processors. Our work is different as it focuses on embedded processors.

Pasricha and Veidenbaum [12] studied the effect of context switches on small predictors and proposed methods for storing/restoring predictor tables. Dhodapkar and Smith [3] proposed methods to store/restore significant bits of predictor counters on context switches. ASAL is different from both studies as it does not store and restore branch predictor tables. Instead, we select the better prediction scheme based on branch behavior in the new interval or context.

Juan et al. [9] proposed configuring the history length of predictors dynamically to reduce misprediction rate. We pick different prediction schemes instead of focusing on the history length.

Manne et al. [10] introduced pipeline gating for high-performance processors. Our study is different as we propose adaptive low overhead techniques for embedded processors.

VI. Conclusion

In this work we presented different performance and power optimization techniques for embedded processors. We proposed ASAL as a low-complexity but adaptive technique to reduce misprediction rate (up to 50%) and improve performance (up to 16%) in embedded processors.

We also proposed low-overhead front-end gating techniques for embedded processors. We showed that by using simple confidence estimation techniques, it is possible to reduce the number of mispredicted instructions fetched by up to a maximum of 32% while maintaining performance.

Acknowledgment

This work was supported by the Natural Sciences and Engineering Research Council of Canada, Discovery Grants Program, Canada Foundation for Innovation, New Opportunities Fund and the University of Victoria Fellowship.

References

[1] Burger, D. C., Austin, T. M., “The SimpleScalar tool set,version 2.0,” Computer Architecture News, 25(3):13–25, June 1997.

[2] Chang, P. Y., Hao, E., Yeh, T. Y., Patt, Y., “Branch classification: a new mechanism for improving branch predictor performance,” In MICRO27, pp. 22-31, New York, NY, USA, 1994.

[3] Dhodapkar, A. S., Smith, J. E., “Saving and Restoring Implementation Context with co-Designed Virtual Machines,” Workshop on Complexity-Effective Design, Jun. 2001.

[4] Grunwald, D., Klauser, A., Manne, S., Pleszkun, “A. Confidence estimation for speculation control,” Proc. 25th Ann. Int’l Symp. Computer Architecture, Jun. 1998.

[5] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., Brown, R. B., “MiBench: A free, commercially representive embedded benchmark suite,” IEEE 4th Annual Workshop on Workload Characterization. 2001.

[6] Huang, M. C., Chaver, D., Pinuel, L., Prieto, M., Tirado, F., “Customizing the Branch Predictor to Reduce Complexity and Energy Consumption,” In IEEE micro. Sep. 2003

[7] Intel, Intel XScale Microarchitecture. 2001.

[8] Jimenez, D., “Idealized piecewise linear branch prediction,” Proceedings of the First Workshop Championship Branch Prediction in conjunction with MICRO-37, December 2004.

[9] Juan, T., Sanjeevan, S., Navaro, J. J., “Dynamic History-Length Fitting: A third level of adaptivity for branch prediction,” Proc. of the 25th Ann. Int’l Symp. Computer Architecture, Jun. 1998.

[10] Manne, S., Klauser, A., Grunwald, D., “Pipeline Gating: Speculation Control for Energy Reduction,” Proc. 25th Ann. Int’l Symp. Computer Architecture, June 1998, 132-141.

[11] McFarling, S., “Combining Branch Predictors,” Tech. Note TN-36, DECWRL, Jun. 1993.

[12] Pasricha, S., Veidenbaum, A., “Improving Branch Prediction Accuracy in Embedded Processors in the Presence of Context Switches,” In 21st Int’l Conf. Comp. Design, Oct. 2003.

[13] Seznec, A., “The O-GEHL branch predictor,” Proceedings of the First Workshop Championship Branch Prediction in conjunction with MICRO-37, December 2004.

[14] Tarjan, D., Skadron, K., Stan, M., “An ahead pipelined alloyed perceptron with single cycle access time,” In Proceedings of the 5th Workshop on Complexity-Effective Design, 2004.