Abstract

Accurate branch prediction can improve processor performance, while reducing energy waste. Though some existing branch predictors have been proved effective, they usually require large amount of storage or complicate the processor front-end. This paper proposes a novel branch prediction technique called History Artificially Selected (HAS) prediction. It is a hardware technique that bases on the existing branch predictors to detect history noises and avoid noise interferences when predicting branches. It separates the original branch predictor into sub-predictors, each of which performs differently in branch history updating. With the help of some history stacks, one sub-predictor saves and restores the branch history at the entrance and the exit of loops and program subroutines where history noise usually exists. Through using a tournament mechanism, HAS prediction selectively uses the modified branch history to eliminate the history noise interferences and retain those useful history correlations at the same time. Our experimental results show that for three representative branch predictors, gshare, perceptron, and TAGE, it reduces the MPKI by 1.49, 2.85, and 1.10 respectively, resulting in 4.55%, 10.16%, and 4.45% performance improvement. It also reduces energy consumption by 4.02%, 7.78%, and 3.91%, respectively.

Keywords

Branch Prediction, History Noise, Energy Efficient

1. Introduction

Mobile devices demand to use high performance yet low power processors to satisfy various applications. Constrained by the power envelope, processor design prefers those energy-efficient building blocks to the complex blocks that achieve high performance at the cost of area and power[4]. Branch prediction is an essential part to modern processors. Accurate branch prediction can improve processor performance and reduce energy consumption through reducing pipeline flushes and wrong speculative executions.

Researchers have proposed various techniques to improve branch prediction accuracy[1][8][11][12][13]. One of the most effective method is to employ longer branch history to track more distant branch correlation, which enables the branch predictor to have a larger window of previous executed branches[8][13][14]. However, long branch history usually makes them consume a large amount of energy in their large storage or complex control logics.

Recent research has revealed that filtering the noise pollution in branch history is as effective as tracking distant history correlations[11][12]. In a program, some branches’ executions have no contributions to other branches’ directions. These useless correlations become noise in branch history when predicting those uncorrelated branches[5]. Previous works have shown that these noises often reside in loops and program subroutines[11][12], increasing the predictor’s training time and significantly bringing biasing problems. As the noise for each branch is different, the history noise is hard to be detected and filtered. Though some techniques are proved to be effective, they either lose some useful history correlations because of its heuristic method, or require to recompile the program that destroys the binary compatible.

In order to effectively achieve high prediction accuracy without significantly increasing the hardware requirements, a branch predictor that benefits from both the distant history correlations and history noise filtration is necessary. A current commercial processor has already employed an accurate conditional branch predictor that uses long branch history and a series of auxiliary predictors, such as loop predictors and Return Address Stack (RAS). Our goal is to propose a hardware solution that bases on the structures of these existing branch predictors to detect history noises and avoid noise interferences effectively when predicting branches.

In this paper, we propose a novel branch prediction technique called History Artificially Selected (HAS) prediction. HAS prediction employs a tournament mechanism, which selectively uses the modified branch history, to eliminate the history noise interferences and retain those useful history correlations at the same time. It separates the original branch predictor into two sub-predictors, Traditional sub-predictor (T-predictor) and history-Variation sub-predictor (V-predictor). Each sub-predictor has its own global history and acts differently in history updating: the T-predictor records branch histories as long as a branch is committed, whereas the V-predictor erases the branch histories in loops and subroutines by operating the new low-cost structures, the history stacks, when entering and exiting those particular program substructures. The history stacks are added to the existing loop predictor and the RAS. A selector table is required to determine which sub-predictor to provide the branch prediction results according to the execution histories of the previous branch instructions. HAS technique can be used in various kinds of aggressive branch predictors, obtaining the benefits from both the long branch history and the noise filtration.

We implemented the HAS technique in three state-of-the-art branch predictors and performed experiments with various types of applications. All predictors examined benefit from the HAS technique and only a minor hardware modifications are required for its implementation. It reduces the MPKI (Miss-predictions per kilo instructions) of gshare, perceptron, and...
TAGE by 1.49, 2.85, and 1.10 respectively, resulting in 4.55%, 10.16%, and 4.45% performance improvement compared with each original predictor. It also reduces energy consumption by 4.02%, 7.78%, and 3.91%, respectively.

The contributions of this paper are as follows:

1) To our knowledge, HAS prediction is the first technique that uses only hardware components to artificially avoid unnecessary noise correlations. It retains those useful history correlations and filters the history noise at the same time.

2) HAS technique utilizes the existing hardware units to detect and filter the history noise correlations. It does not bring extra energy consumption significantly.

3) It is unnecessary for HAS technique to modify the program binary or recompile the program.

2. Related Work

Global branch history can be used to produce accurate branch predictions because branches often correlate with previously executed branches. Modern processors use longer branch history to track more distant branch correlations. Longer branch history enables predictors to view a larger window of previously executed branches and learn their correlations. Previous researchers have proposed various techniques to prolong the global branch history. GShare scheme[15] XORs the PC and global history to index the pattern history table. Jiménez et al. utilize the branch history as the input to a neural network in perceptron predictor[8]. Considering the hardware budget, it usually uses 32 to 64 bits of branch history. Seznec et al. propose TAGE[14] predictor to make use of longer history, which is more than 600 bits.

Recent research has proved that not all regions of control flow are correlated with recently executed branches. For these regions, the extra information encoded in the global history does more harm than good[5]. Leo Porter have shown that a simple branch history modification, zeroing technique, can be used to improve the branch prediction accuracy by reducing the amount of noise in branch history[12]. It uses the branch instruction’s address to replace the history recorded in the loop or program subroutine, erasing those histories indiscriminately. An alternative technique, CHS[11], is to use the compiler to detect the program substructures and modify the branch history selectively. It identifies the history correlations by the compiler, and then feeds the information to the predictor by inserting guiding instructions. At runtime, the processor dynamically saves and restores the global history according to the compiler-guided information. Though it is very effective, it requires recompiling the target programs.

3. Motivation

Previous works have proposed two critical issues in branch prediction: (1) distant history correlations, and (2) history noise pollution. Aggressive branch predictors that utilize long branch history, such as perceptron[8] and TAGE predictors[14], have effectively solved distant history correlations. However, history noise pollution is still a critical problem. First, the history noise is difficult to be identified, for a branch history may play different roles to different branch instructions in a program. It has the decisive effects on some branches, whereas it has no contribution to others. Figure 1 shows a code example in gcc06 benchmark. The result of Branch A determines the direction of the Branch C, but has no relationship with the Branch D. In such cases, an effective solution is to provide multiple selections of global history when predicting a branch. For example, one records the all branch histories, while the other filters some histories that are possibly history noises. To different branches, different global history is selected to reduce the noise interferences. Second, the history noise is also hard to be filtered, for it is extremely difficult for the branch predictor to eliminate the impacts that the history noise has already made. For example, in a gshare predictor, a history bit affects lots of pattern history table entries which cannot be recovered in one or two cycles. To solve such issue, multiple predictors can be used to avoid the impacts made by history noises. Third, as the binary compatible is essential to various existing applications, such artificial filtration mechanism should be done by only hardware.

Our goal is to propose an energy-efficient branch prediction technique that uses only hardware to avoid the unnecessary correlations with noise in recorded branch history, while retaining those useful history at same time.

4. History Artificially Selected Prediction

A HAS predictor employs a tournament mechanism that selectively uses the modified branch history to avoid unnecessary correlations with history noise. It dynamically selects the prediction result among multiple branch sub-predictors. Each sub-predictor, which is divided from the original branch predictor, performs as the same as the original predictor, but adopts different strategies of branch history updating. One sub-predictor updates the branch history traditionally, while the other erases the branch histories in particular program substructures – loops or program subroutines, in which branches’ executions are possibly useless to the branches outside. The prediction results are selected among those sub-predictors according to the previous executions of branch instructions. The extended loop predictor and RAS are responsible for detecting the loops and subroutines, and modifying the branch history in those program substructures.

A. Structure

The detailed HAS structure is depicted in Figure 2. T-predictor and V-predictor are two sub-predictors that are
Figure 2: The structure of HAS prediction. The units with grey background are new or have been modified.
program runs. We assume that the branch histories in both sub-predictors are 4'b1010 before entering the program subroutine at line 3. The V-predictor pushes the current branch history into the history stack at the entrance of the subroutine, and pops the history stack to recover its global history at the exit of the subroutine. During the program runs, the T-predictor performs as the same as that of the conventional branch predictor. Based on these behaviors, when predicting the branch at line 4, the two sub-predictors have different branch histories, leading to different prediction results. The final prediction is Not-Taken decided by the selector’s output. When the actual branch result is obtained, each sub-predictor is trained toward Taken, and the selector is trained to select the V-predictor which predicted correctly.

5. Experiments and Results Analysis

We employed gem5[2], a cycle-accurate performance simulator, in x86_64 full-system mode to evaluate the effects of HAS prediction. Table 2 shows the parameters of our baseline processor. Our workload includes the benchmarks of SPEC CPU INT 2006[7] and the DaCapo-9.12 benchmarks[3]. The SPEC CPU INT 2006 evaluation shows the effects of HAS mechanism on various kinds of benchmarks. DaCapo benchmarks are written in Java, which consist of a set of open source, client-side, real world applications with non-trivial memory loads. The DaCapo benchmarks are running on the OpenJDK-1.6.20[17].

In this section, we evaluated the HAS technique with three representative branch predictors – gshare, perceptron, and TAGE. Table 1 lists the baseline structures of each branch predictor and their sub-predictors modifications. The selector table in HAS technique employs 128 section selectors, each of which has 16KB.

A. Analysis of Hardware Costs and Timing Impacts

To estimate the hardware costs and the timing impacts, we used Verilog HDL to model the basic structures of the branch predictors in a fully synthesizable 64-bit superscalar processor. This processor issues and completes three instructions per clock cycle. We used the synthesis tools to evaluate the basic structures and those HAS schemes. Table 3 shows the comparisons of equivalent NAND-gates number. We conclude that the HAS technique neither requires large amount of hardware resources nor increases the length of the critical path. As the TAGE predictor records the longest branch history, it requires the largest area overhead if using HAS technique, but it is only 4.84% compared with the original TAGE predictor. Although HAS schemes slightly increase the length of the branch predictor’s timing path, it does not affect the critical path of the processor front-end, which resides in the BTB.

B. Performance of HAS prediction

Figure 4 (left) shows that HAS prediction reduces the branch MPKI for all three types of branch predictors. The MPKI reductions are 1.49, 2.85, and 1.10 in gshare, perceptron, and TAGE predictor, respectively. We also find that the perceptron using HAS technique can achieve similar MPKI to that of the original TAGE predictor. The MPKI reduction leads to attractive IPC improvement shown in Figure 4 (right). As HAS prediction does not influence the processor frequency, IPC generally reflects its performance impact. The IPC improvements of HAS-GShare, HAS-Perceptron, and HAS-TAGE are 4.55%, 10.16%, and 4.45%, respectively.

From the results, we notice some benchmarks have adverse impacts on MPKI and performance using HAS technique. This is because the sub-predictor selection is determined by the accordance of the sub-predictors’ predictions and the actual execution result. If a sub-predictor is selected now, and it predicts taken, the V-predictor should predict not-taken, but it is affected by the biasing, predicting taken instead. Then it continues to use T-predictor rather than V-predictor, resulting in wrong sub-predictor selection. As the

---

Table 1: Sub-predictor implementations of various branch predictors

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Structural Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GShare</td>
<td>Original Predictor: 15-bit global history, 32K-entry PHT, 1-cycle prediction delay</td>
</tr>
<tr>
<td>Perceptron</td>
<td>Original Predictor: path-based perceptrons [8], 32-bit global history, 32KB path-based perceptron, 2-cycle prediction delay</td>
</tr>
<tr>
<td>TAGE</td>
<td>Original Predictor: 250-bit global history, 64K-bit, 10 predictor tables. Tables are indexed with hash functions of branch history, path history, and PC. 2-cycle prediction delay</td>
</tr>
</tbody>
</table>

Table 2: Baseline processor configuration

<table>
<thead>
<tr>
<th>Pipeline Depth</th>
<th>16 stages; out-of-order execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>4 instructions per cycle</td>
</tr>
<tr>
<td>Regs</td>
<td>Physical Integer Regs: 256; Physical Float Regs: 256;</td>
</tr>
<tr>
<td>Execution Engine</td>
<td>4-wide rename/dispatch/writeback,Load Queue: 64-entry;</td>
</tr>
<tr>
<td>Branch Prediction</td>
<td>Store Queue: 64-entry; 512-entry RU; 128-entry LSQ</td>
</tr>
<tr>
<td>Caches</td>
<td>32KB, 8-way, 4-cycle L1 DCache &amp; ICache; 64B Blocks</td>
</tr>
<tr>
<td>Memory</td>
<td>2MB, 16-way, 10-cycle L2 Cache; 64B Blocks, LRU</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Predictor</th>
<th>Structural Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GShare</td>
<td>Original Predictor: 15-bit global history, 32K-entry PHT, 1-cycle prediction delay</td>
</tr>
<tr>
<td>Perceptron</td>
<td>Original Predictor: path-based perceptrons [8], 32-bit global history, 32KB path-based perceptron, 2-cycle prediction delay</td>
</tr>
<tr>
<td>TAGE</td>
<td>Original Predictor: 250-bit global history, 64K-bit, 10 predictor tables. Tables are indexed with hash functions of branch history, path history, and PC. 2-cycle prediction delay</td>
</tr>
</tbody>
</table>

---

Figure 3. HAS prediction example. GHR presents the global history register.
Figure 4: MPKI reductions (Left) and IPC improvement (Right) using HAS prediction

Table 3: Analysis of Hardware Costs and Timing Path. Estimated by TSMC 65GP (0.9V, 125°C), Typical Case

<table>
<thead>
<tr>
<th>predictor</th>
<th>Equivalent number of NAND gates</th>
<th>Timing Path (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>original</td>
<td>SRAMs</td>
</tr>
<tr>
<td>BTB</td>
<td>735216</td>
<td>69.85%</td>
</tr>
<tr>
<td>gshare</td>
<td>361776</td>
<td>72.66%</td>
</tr>
<tr>
<td>perceptron</td>
<td>1488472</td>
<td>69.60%</td>
</tr>
<tr>
<td>TAGE</td>
<td>663424</td>
<td>37.79%</td>
</tr>
</tbody>
</table>

gshare predictor is easier to be affected by biasings, such situations are more popular.

We also compared the HAS technique with the zeroing technique[12] in each predictor. The comparisons show that the HAS technique outperforms the zeroing technique[12], reducing MPKI by 0.64, 1.63, and 0.90 for gshare, perceptron, and TAGE predictor respectively. It also improves zeroing’s performance by 2.57%, 7.31%, and 3.49% respectively. This indicates that the mechanism using history stacks is superior to that totally inhibits the history updating, for the latter action loses the dynamical execution information of those inside branches, causing adverse impacts when predicting the branches in those program structures, such as sunflow and h264ref. We also notice that HAS technique performs worse
than that of the zeroing technique in some benchmarks, particularly in gshare predictor. This is mainly because the primary issue of those programs for the branch predictor to solve is the biasing problem rather than the history noise in long history. Zeroing technique actually eliminates some possible biasing problems while erasing history indiscriminately. If a branch predictor avoids biasing effectively, it will benefit from HAS technique more obviously. The representative program of such cases is tomcat. It has the greatest performance overhead in the TAGE, and HAS-perceptron performs better than that of HAS-gshare.

C. Impacts on different predictor sizes

Figure 5 shows the impacts when predictor size varies. We conclude that HAS technique generally perform better when working with short history. For example, the TAGE predictor achieves 4.87% performance improvement when using 64-bit branch history, while only achieves 3.02% improvement for 1024-bit branch history. This is because those short-history predictors leaves a large number of distant history correlations to be tracked using HAS technique, whereas in long-history predictor, those correlations has already been utilized. However, even to the long-history predictors, the noise filtering in HAS technique can also benefit them by avoiding the interferences of unnecessary correlations.

D. Prediction distribution of sub-predictors

We use perceptron as the representative predictor in subsection D&E, for it has already been implemented in commercial processors[6][9]. Figure 6 shows the statistics of sub-predictor’s access distribution. Some benchmarks, such as gcc, hmmer and libquantum, have close correlations with global history, so they use T-predictor more frequently than V-predictor. To these programs, their performance has nearly no impact, for they use T-predictor much more frequently.

E. Impacts on different implementation parameters

Table 4 shows the impacts when some HAS implementation parameters varies. First, the sizes of selector table are sensitive to the performance of HAS technique. The results show that the fine-grained sections performs generally better than the coarse-grained. It also shows that medium size of bits can have the best performance improvement. Too many or too few bits make the sub-predictor switch too slowly or too quickly. Second, the depth of history stack in loop predictor may affect the performance. The evaluation shows that 16-entry history stack for loop predictor is sufficient for the examined benchmarks, for there are no more than 16 nested loops in majority of applications. This greatly reduces its hardware costs compared with that of attaching entry to each loop predictor entry.

F. Impacts on energy reduction

Accurate branch prediction can result in fewer pipeline flushes and fewer wrong-path executions, reducing total processor energy consumption. The energy reduction evaluation in this section are based on our 64-bit superscalar processor. The basic power parameters of SRAMs are derived from the CACTI tools [16], and the logic parts are obtained using synthesis tools under TSMC 28nm logic library (tcbn28hpwb12t35). The average energy consumption of total processor using three HAS schemes are reduced by 4.02%, 7.78%, and 3.91% respectively over the baseline processor. The detailed comparisons between HAS-perceptron and zeroing technique are illustrated in Figure 7. The HAS scheme outperforms the zeroing technique by 3.73%.

6. Conclusion

Modern researchers dedicate in designing high energy-efficiency branch predictors. In this paper, we propose the HAS prediction, which employs only hardware to reduce unnecessary noise correlations and retains useful histories in loops and subroutines according to the previously executed branches. It can be applied to various state-of-the-art branch predictors as well. Our experiments show that HAS can improve the branch prediction accuracy significantly without the support of compiler and the use of extra large storage. Meanwhile, it also reduces the energy consumption due to less pipeline flushes and wrong-path executions.

References