IMPROVING BRANCH PREDICTION ACCURACY VIA EFFECTIVE SOURCE INFORMATION AND PREDICTION ALGORITHMS

by

HONGLIANG GAO
B.E. Beijing University of Aeronautics and Astronautics, 2001
M.S. University of Central Florida, 2005

A dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the School of Electrical Engineering and Computer Science at the University of Central Florida Orlando, Florida

Summer Term
2008

Major Professor:
Huiyang Zhou
© 2008 Hongliang Gao
ABSTRACT

Modern superscalar processors rely on branch predictors to sustain a high instruction fetch throughput. Given the trend of deep pipelines and large instruction windows, a branch misprediction will incur a large performance penalty and result in a significant amount of energy wasted by the instructions along wrong paths. With their critical role in high performance processors, there has been extensive research on branch predictors to improve the prediction accuracy.

Conceptually a dynamic branch prediction scheme includes three major components: a source, an information processor, and a predictor. Traditional works mainly focus on the algorithm for the predictor. In this dissertation, besides novel prediction algorithms, we investigate other components and develop untraditional ways to improve the prediction accuracy. First, we propose an adaptive information processing method to dynamically extract the most effective inputs to maximize the correlation to be exploited by the predictor. Second, we propose a new prediction algorithm, which improves the Prediction by Partial Matching (PPM) algorithm by selectively combining multiple partial matches. The PPM algorithm was previously considered optimal and has been used to derive the upper limit of branch prediction accuracy. Our proposed algorithm achieves higher prediction accuracy than PPM and can be implemented in realistic hardware budget. Third, we discover a new locality existing between the address of producer loads and the outcomes of their consumer branches. We study this address-branch correlation in detail and propose a branch predictor to explore this correlation for long-latency and hard-to-predict branches, which existing branch predictors fail to predict accurately.
To my mother and father
ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help and support of a number of people. First of all, I would like to express my deepest gratitude to my esteemed advisor, Dr. Huiyang Zhou. We joined University of Central Florida in the same year. I feel extremely lucky and thankful to be his first PhD student. His encouragement, advice, mentoring and patience have been very helpful to my research. I also want to thank my other dissertation committee members, Dr. Mark Heinrich, Dr. Ronald F. DeMara and Dr. Liqiang Ni, for spending their time to review the manuscript and providing valuable suggestions.

I am fortunate to work within a group of talented students, Yi Ma, Martin Dimitrov and Jingfei Kong. I have enjoyed every moment that we have worked together. I am very grateful for inspiring discussions with all of them. I want especially to thank Yi, who is my best friend and has encouraged and helped me to go through all the difficult times during these five years. My enormous debt of gratitude goes to Martin, who proof-read my dissertation and provided many suggestions to help me improve it.

My graduate studies would not have been the same without my friends in Orlando. My sincere gratitude goes to Yugang Min, Yixiao Yang, Baiyun Chen, Lifang Lou, Mengqian Chen, Fuyu Liu, Min Li, Rui Peng, Yifang Gao, Hua Zhang, Li Miao, Feng Lv, Guoqiang Wang and all members of the Chinese Culture Club of UCF. I appreciate their friendship and I have had a lot of fun with them.
Finally, I am deeply indebted to my mother Zelan He, my father Changquan Gao, my sister Hongying Gao and my girlfriend Adan Niu. Their love and understanding have always been the strongest support to me.
# TABLE OF CONTENTS

LIST OF FIGURES ....................................................................................................................... xi
LIST OF TABLES ........................................................................................................................... xiv

CHAPTER 1.  INTRODUCTION .............................................................................................. 1
  1.1.  A conceptual system model for branch prediction ......................................................... 2
  1.2.  Our solutions to improve branch prediction accuracy .................................................... 2
  1.3.  Contributions .............................................................................................................. 3

CHAPTER 2.  BACKGROUND ................................................................................................ 5
  2.1.  History of Branch Prediction ........................................................................................ 5
  2.2.  Branch Prediction Schemes ........................................................................................ 6
    2.2.1.  Bimodal Predictor ................................................................................................. 6
    2.2.2.  Two-level Predictor ............................................................................................. 6
    2.2.3.  gshare Predictor ................................................................................................. 7
    2.2.4.  Perceptron Predictor ........................................................................................... 7
    2.2.5.  O-GEHL Predictor ............................................................................................. 8
    2.2.6.  TAGE Predictor ................................................................................................. 8
  2.3.  Schemes to Alleviate Branch Misprediction Costs ....................................................... 9

CHAPTER 3.  ADAPTIVE INFORMATION PROCESSING ................................................ 11
  3.1.  Exploiting Correlation with Perceptron Branch Predictors ........................................ 11
  3.2.  Enhancing Branch Correlation Using Adaptive Information Processing ................... 14
3.2.1. Profile-directed Adaptation................................................................. 15
3.2.2. Correlation-directed Adaptation............................................................ 17
3.3. Overall Branch Predictor Structure.......................................................... 18
3.4. Results and Analysis.................................................................................... 21
  3.4.1. Performance for CBP-1 traces ................................................................. 21
  3.4.2. Performance for SPEC2000 Benchmarks................................................. 23
3.5. Summary........................................................................................................ 24

CHAPTER 4. PREDICTION BY COMBINING MULTIPLE PARTIAL MATCHES .... 26
  4.1. Branch Prediction by Combining Multiple Matches................................. 28
    4.1.1. Modeling PPM....................................................................................... 28
    4.1.2. A Study on PPM for Branch Prediction.................................................... 29
    4.1.3. Prediction by Combining Multiple Partial Matches (PMPM)..................... 32
    4.1.4. Pushing the Limit of Branch Prediction Accuracy using PMPM............... 33
  4.2. Idealistic Adaptive-PMPM Predictors....................................................... 36
    4.2.1. Predictor Structure............................................................................... 37
    4.2.2. Prediction Policy................................................................................... 39
    4.2.3. Update Policy....................................................................................... 39
    4.2.4. Results................................................................................................. 40
  4.3. Realistic PMPM Predictors......................................................................... 41
    4.3.1. Predictor Structure............................................................................... 41
    4.3.2. Prediction Policy................................................................................... 43
    4.3.3. Update Policy....................................................................................... 44
    4.3.4. Hardware Complexity and Response Time.............................................. 47
5.4.5. Reduction in Energy Consumption ................................................................. 89

5.5. Limitations and Potential Optimizations ............................................................. 90

5.6. Summary ............................................................................................................. 91

CHAPTER 6. CONCLUSIONS .................................................................................. 92

6.1. Contributions ...................................................................................................... 92

6.2. Future Works ...................................................................................................... 93

LIST OF REFERENCES .............................................................................................. 95
LIST OF FIGURES

Figure 1. A conceptual system model of branch prediction ....................................................... 2
Figure 2. A micro benchmark to illustrate that perceptron weights can be used as a quantitative measure of branch correlation ................................................................. 12
Figure 3. An MAC perceptron predictor with adaptive information processing. (the shaded units are introduced for adaptive information processing) ................................................ 15
Figure 4. Pre-determined input configurations for different types of workloads ................. 16
Figure 5. The overall branch predictor structure ....................................................................... 19
Figure 6. Branch misprediction rates on CBP-1 traces ........................................................... 22
Figure 7. Prediction contribution from individual branch predictors ........................................ 22
Figure 8. Branch misprediction rates on SPEC 2000 benchmarks .......................................... 24
Figure 9. Prediction contribution from individual branch predictors ........................................ 24
Figure 10. Misprediction rates of PPM algorithm ................................................................. 29
Figure 11. Misprediction rate reductions of the confidence-based PPM compared to PPM ...... 30
Figure 12. Misprediction rates of PPM and prediction by the K-th longest confident partial matches .................................................................................................................... 31
Figure 13. Misprediction rates of PMPM-L algorithms, the PPM algorithm and the prediction by the 3rd longest confident partial match scheme ................................................................. 34
Figure 14. Misprediction rate reductions of the best PMPM-L ($L = 11$) and Adaptive-PMPM compared to PPM ................................................................................................................. 36
Figure 15. The overall idealistic Adaptive-PMPM branch prediction scheme ......................... 37
Figure 33. Normalized execution time of the baseline processor augmented with ABC predictors.

Figure 34. Normalized execution time of the dynamic learning and profiling schemes with a 9kB ABC predictor.

Figure 35. Reductions in branch misprediction penalties achieved by ABC predictors.

Figure 36. Normalized execution time of different primary predictors with and without ABC predictors.

Figure 37. Normalized execution time of GTL predictor with ABC predictors.

Figure 38. Performance improvements for processors with different ROB sizes, L2 sizes and memory latencies.

Figure 39. Reduction in energy consumption achieved by a 9kB ABC predictor.
**LIST OF TABLES**

Table 1. Hardware budget............................................................................................................. 20

Table 2. The optimal $L$ for the PMPM-L algorithm. ................................................................. 35

Table 3. Idealistic Predictor Configuration................................................................................... 40

Table 4. Configuration of the 32kB PMPM predictor. ................................................................. 51

Table 5. The major characteristics of the AIP, the O-GEHL and the TAGE predictors. ............. 52

Table 6. Misprediction rates (MPKI) of different branch predictors with the same 32kB storage budget............................................................................................................................................ 53

Table 7. Misprediction rates (MPKI) of the TAGE and the global history only PMPM predictors with 32kB storage budget. ............................................................................................................ 56

Table 8. Misprediction rates (MPKI) of the TAGE-PPM and the PMPM-FIXED predictors with 32kB storage budget. .................................................................................................................... 56

Table 9. Length of the training period and the number of selected load/branch pairs. ................ 75

Table 10. Configuration of the processor. .................................................................................... 81

Table 11. The number of mispredictions for the selected hard-to-predict branches. ................... 84
CHAPTER 1. INTRODUCTION

The front end of a microprocessor has to sustain high throughput so that the out-of-order execution engine has enough instructions to efficiently explore instruction-level parallelism (ILP). The difficulty of maintaining a high instruction fetching rate mainly comes from conditional branch instructions. When a conditional branch instruction is fetched, its outcome is not available until the branch has been executed. Before the branch execution, the front end has no knowledge about which execution path will follow the branch instruction. Instead of stalling the front end and waiting for the branch outcome, a branch predictor is used to speculatively predict the branch outcome. Thus the front end will continue fetching instructions based on the prediction.

Branch prediction technique successfully maintains the front end throughput and yields a tremendous improvement in performance. However, modern microprocessors are using deeper pipelines and larger instruction windows. Therefore, any branch misprediction will result in a larger penalty by executing tens or even thousands of instructions on the wrong path before recovering the processor to the correct path upon the misprediction resolution. Moreover, wrong path executions waste a considerable amount of energy. Motivated by the importance of branch prediction, we focus on searching for methods to achieve high branch prediction accuracy.
1.1. A conceptual system model for branch prediction

As presented by Allen et al. [2], dynamic branch prediction schemes can be described using a generic conceptual system model, as shown in Figure 1. The source information, including branch addresses (PC), local/global histories (LHR/GHR), along with other run-time information, is gathered when executing a program. The information processor extracts a subset of the source information and forms an information vector. Then, the predictor processes the information vector and makes a prediction. Traditionally, various Markov Finite State Machines (FSMs) are used as the predictor [2].

![Figure 1. A conceptual system model of branch prediction.](image)

1.2. Our solutions to improve branch prediction accuracy

As shown in Figure 1, in order to predict branches accurately, we have to extract appropriate runtime information from the program and design an efficient predictor to use the information provided by the information processor. Hence, our solutions focus on the information processor and the predictor components of branch prediction schemes.

Traditionally, branch histories are used as the input information to predictors. The advantage of using branch histories is that the hardware support to track them is simple and they...
are efficient in capturing correlations between branch outcomes. We propose two solutions for branch correlation based predictors. First, we propose a method to dynamically adjust the information processor to select the most useful history bits. This solution reduces the noise impact of useless history bits. Since it uses the limited hardware resource on most correlated information, it also makes the predictor more efficient. Second, we propose a new prediction algorithm named Prediction by Combining Multiple Partial Matches (PMPM). PMPM algorithm improves the prediction accuracy by selectively combining multiple partial matches instead of using the longest match as in Prediction by Partial Matching (PPM) algorithm [8].

We also discovered a novel correlation between load addresses and branch outcomes, which can be used to accurately predict some long-latency and hard-to-predict branches. Based on this correlation, we propose a predictor that dynamically extracts correlation information between load addresses and their consumer branch outcomes to handle hard-to-predict branches.

1.3. Contributions

The contributions of this dissertation are summarized as follows:

1. We show that branch history bits are not equally useful for branch prediction. We demonstrate that selectively feeding the branch predictor with the most useful history bits can significantly improve branch prediction accuracy. We design an adaptive information processor to dynamically learn the importance of different branch history bits in perceptron predictors. The proposed predictor achieves very high prediction accuracy and is the winner of the 1st JILP Championship Branch Prediction Competition (CBP-1) held on 2004 [43].
2. We propose a new branch prediction algorithm called Prediction by combining Multiple Partial Matches (PMPM). In this algorithm, multiple partial matches are selectively combined by an adder tree. We show that PMPM algorithm has higher prediction accuracy than previously deemed optimal PPM algorithm. Based on PMPM algorithm, we design an idealistic predictor to push the limit of branch prediction. It has higher accuracy than the winner predictor of the idealistic track of the 2nd JILP Championship Branch Prediction Competition (CBP-2) held on 2006 [44]. We also design a realistic predictor to implement PMPM algorithm in real hardware and it achieves higher prediction accuracy than other state-of-art branch predictors.

3. A particularly critical group of branches are those hard-to-predict branches that depend on cache misses. We discovered a novel correlation between load addresses and their consumer branch outcomes. We call this address-branch correlation. We present our study of the fundamental program behavior that generates stable address-branch correlation. We design a low cost predictor to explore address-branch correlation and show that it significantly reduces the misprediction penalties of those hard-to-predict branches.
CHAPTER 2. BACKGROUND

In this chapter we describe the background of branch prediction. We review the history of branch prediction and some representative dynamic branch prediction mechanisms. We also provide a review of mechanisms used to alleviate branch misprediction costs.

2.1. History of Branch Prediction

Branch prediction techniques appeared in 1960’s in some main frame computers. For instance, the IBM System/360 Model uses a static branch prediction scheme [3]. In this scheme, a branch prediction is made based on the type of branch. For example, branches used to terminate loops are predicted as taken. Smith proposed a dynamic branch predictor by indexing a table of 2-bit counters with branch address and dynamically training those 2-bit counters to capture branch behaviors in 1981[37]. This predictor is called bimodal predictor and still used as a component in many state-of-art branch predictors. An important breakthrough of branch prediction was the discovery of branch correlation by Yeh and Patt in 1993 [39]. They discovered that the branch outcomes are highly correlated to each other so that a branch outcome could be predicted accurately based on previous branch outcomes. They designed a set of branch predictors to explore branch correlation, which significantly improve branch prediction accuracy. Since the discovery of branch correlation, extensive research has been done to improve branch prediction accuracy. At the same time, various schemes have been proposed to alleviate the
harmful impact of branch mispredictions. In the remaining of this chapter, we will review some of the representative schemes.

2.2. Branch Prediction Schemes

In this section, we review some of the representative branch prediction schemes. These schemes either have significant impact on the research of branch prediction and/or are highly related to our solutions proposed in this dissertation.

2.2.1. Bimodal Predictor

Bimodal branch predictor was proposed by Smith in [37]. In a bimodal predictor, a table of 2-bit counters is used to track history outcomes of each static branch. At the prediction stage, a 2-bit counter is read from the table based on the branch address. Then the branch outcome is predicted based on the higher bit of the counter. When this bit is “1”, the branch is predicted as taken. Otherwise, it is predicted as not-taken. When the actual outcome of a branch is known, its corresponding 2-bit counter is increased by 1 if the branch is taken or decreased by 1 if the branch is not-taken. Comparing to the simple dynamic branch prediction scheme that uses the last outcome of a branch as the prediction, 2-bit counters have been shown to be more robust on the impact of rare outcomes of a branch.

2.2.2. Two-level Predictor

Yeh and Patt proposed a set of predictors to explore branch correlation in [39]. Those predictors are called two-level predictors. They discovered that a branch outcome is highly correlated with previous branch outcomes of itself or other branches. They proposed to use
branch history registers to record branch outcomes and explore branch correlation. Branch history register is a bit vector of branch outcomes. For a taken branch, a “1” is shifted in the register. For a not-taken branch, a “0” is shifted in. If a branch history register is shared by all branches, it is called a global history register. If a branch history register is owned by a particular static branch, it is called a local history register. In order to explore branch correlation, pattern history tables (PHT) are used to track the corresponding branch outcomes of a specific branch history with 2-bit counters. In two-level predictors, branch address is used to locate a PHT at the first level. In the second level, a 2-bit counter is selected by indexing the PHT with the branch history. Many variants of the two-level predictor are proposed and used in real processors.

2.2.3. gshare Predictor

gshare predictor is one of the most successful variants of the two-level predictor. It was proposed by MacFarling in [22]. Instead of using two levels of tables to distinguish different branches and branch histories, gshare predictor uses “branch address XOR branch history” as the index to a table of 2-bit counters. In this way, the table utilization is significantly improved, which makes gshare predictors achieve higher prediction accuracy than two-level predictors with the same hardware storage costs.

2.2.4. Perceptron Predictor

Motivated by the strong learning ability of neural networks, Jimenez proposed to use perceptrons as dynamic branch predictors [16]. In a perceptron predictor, a perceptron is selected based on the branch address. Then branch history bits are fed into the perceptron as inputs. Each perceptron includes a set of weights corresponding to every bit of the branch history. The sign of
the summation of perceptron weights is used as the branch prediction. At the update stage, a perceptron weight is increased if the corresponding history bit agrees with the branch outcome. Otherwise, we decrease the perceptron weight. In this way, each weight successfully captures the positive or negative correlation between a history bit and the branch outcome. Since the number of weights of a perceptron increases linearly with the branch history length, perceptron predictors are able to explore much longer branch histories than previously proposed branch predictors, in which the number prediction counters increase exponentially with the history length.

2.2.5. O-GEHL Predictor

In order to explore very long branch histories, Seznec proposed a multi-table predictor called O-GEHL predictor [28]. In an O-GEHL predictor, prediction counters are stored into several (e.g., 8) prediction tables. Prediction tables are indexed by branch histories with different lengths. The history lengths of prediction tables are constructed as a geometrical series. In this way, very long history length can be used in the table indexed with long history without affecting the table with short histories. In order to get a prediction, one prediction counter is selected from each prediction table and the sign of the summation of these counters is used as the final prediction. Experimental results show that O-GEHL predictors have higher prediction accuracy than perceptron predictors [28].

2.2.6. TAGE Predictor

Prediction by Partial Matching (PPM) is a powerful algorithm originally proposed for data compression [8]. In order to efficiently implement the PPM algorithm for branch prediction, Seznec and Michaud proposed TAGE predictor [35]. In a TAGE predictor, several partially
tagged prediction tables are indexed by global branch histories with different lengths. Similar to O-GEHL predictor, the history lengths are constructed as a geometrical series. After tag comparison, several tables may include matching entries for the current branch. The table corresponding to the longest history match is used as the final prediction. Since TAGE predictor is able to explore very long histories and use the prediction counters very efficiently, it is one of the most accurate branch predictors in the literature.

2.3. Schemes to Alleviate Branch Misprediction Costs

Instead of focusing on improving the branch prediction accuracy, many researchers have also proposed schemes to alleviate branch misprediction costs. In this section, we review a recent proposal for aggressive superscalar processors.

In order to alleviate the harmful impact of branch mispredictions, a scheme named predication has been used to avoid pipeline flushes due to branch mispredictions [2]. In a predication scheme, control dependencies are converted into data dependencies. The processor fetches instructions from both paths of a branch but commits only results from the correct path. The prediction scheme effectively avoids the pipeline flush associated with a branch misprediction. However, it also introduces some extra cost of fetching and executing wrong path instructions when a branch is correctly predicted.

One major limitation of predication is that it can only handle branches in a simple control flow. This limitation was addressed by Kim et al. They proposed the Diverge-Merge Processor (DMP) [20], which significantly improves the performance of previous predication schemes. In DMP, the confidence of a branch prediction is estimated. If a prediction is likely to be wrong, the processor will dynamically predicate the branch and both paths of the branch will be executed.
Branches along each path are predicted by a branch predictor. With the help of the branch predictor to selectively predicate instructions, complex control flows are predicated efficiently by following the frequently executed paths. Since DMP is able to selectively decide which branch to predicate and dynamically predicate branches in a complex control flow, it shows significant performance improvement. In fact, in our analysis of hard-to-predict branches, we found that many of them could be effectively predicated by DMP.
Most branch predictors employ an implicit information processor, by simply extracting some number of bits from GHR, LHR and PC. In this chapter, we describe a new information processor [10], [11], which extracts the information vector from the information source adaptively based on the strength of branch correlation. A key observation of our approach is that perceptron weights can be used as a quantitative measure of correlation between a branch and the input information vector. With such a measure, we can re-assemble the information vector accordingly to maximize the branch correlation. Additionally, since the adaptation is performed at a coarse grain, e.g., at program phase boundaries or at an interval of a certain large number of branches, the adaptation logic is not latency critical.

3.1. Exploiting Correlation with Perceptron Branch Predictors

Many branch predictors, including two-level [39] and gshare [22] predictors, achieve high prediction accuracy by exploiting correlation from history information. Perceptron predictors, moreover, have an additional capability to estimate the strength of such correlation, as illustrated from the example shown in Figure 2.
while (1){
  r1 = Rand(0, 1000); r2 = Rand(0, 1000);
  c1 = r1 > r2; c2 = r1 > 500; c3 = r1<=r2;
  if (c1) count1++;
  if (c2) count2++;
  if (c3) count3++;
  if (c2 ^ c3) count4++;
}

<table>
<thead>
<tr>
<th>Source Program</th>
<th>misprediction rate (MR) and perceptron weights</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B1: MR=50.09%  w1-w8: -1 -1 5 7 -3 -5 -1 -1</td>
</tr>
<tr>
<td></td>
<td>B2: MR=25.45%  w1-w8: 26 0 0 2 2 2 0 0</td>
</tr>
<tr>
<td></td>
<td>B3: MR=0%  w1-w8: -1 -31 1 1 -1 -1 -3 -1</td>
</tr>
<tr>
<td></td>
<td>B4: MR=19.11%  w1-w8: 12 22 -12 10 10 -2 0 -2</td>
</tr>
</tbody>
</table>

Figure 2. A micro benchmark to illustrate that perceptron weights can be used as a quantitative measure of branch correlation.

The micro-benchmark in Figure 2 contains four conditional branches (B1, B2, B3 and B4): B1 is determined by two random values, B2 correlates with B1 as both branches share the same random variable \( r1 \), B3 is simply the inverse of B1, and B4 correlates with both B2 and B3 with an XOR operation. To predict these four branches, a 4-entry perceptron predictor (i.e., each branch has its own perceptron) with 8-bit global branch history (i.e., an 8-bit GHR) [17] is used. After simulating 100M instructions, the perceptron weights (\( w1-w8 \), corresponding to GHR[0] to GHR[7]) of these branches are reported and the misprediction rates are 50.09%, 25.45%, 0% and 19.11% for branches B1-B4 respectively.

Several important observations can be made from Figure 2. First, perceptron weights embody the correlation information and the absolute values of perceptron weights provide an estimate of the correlation strength. As branch B1 finds no correlation from previous branch history, it features small random perceptron weights. Since branch B2 correlates to branch B1 by sharing the same variable \( r1 \), its perceptron weights contain a large \( w1 \) (i.e., strong correlation with GHR[0]) and small random \( w2-w8 \). Branch B3 shares the same condition (inverse) as
branch \( B1 \). Therefore, it has a singular large weight \( w_2 \) (i.e., strong correlation with GHR[1]). Due to its nonlinear correlation with previous branches, branch \( B4 \) has more relatively large perceptron weights.

Secondly, based on the correlation strength, the input to a perceptron predictor can be re-assembled to utilize the predictor resource more efficiently. For the example in Figure 2, we know that GHR[0] and GHR[1] carry the most useful correlation. Therefore, we can reduce the perceptron size to only exploit correlation from a 2-Bit GHR instead of an 8-bit GHR and we found that the perceptron predictor with the shorter history achieves similar or slightly better prediction accuracy (misprediction rates are 50.03%, 25.02, 0%, and 18.21% for branch \( B1-B4 \) respectively). The improvement is due to the reduced noise effect in the perceptron training process from uncorrelated history bits. Note that such correlation-based input/history re-assembling is more flexible than history length adaptation [19]. For example, if it is determined that GHR[0] and GHR[7] carry the most useful correlation, we can still use a perceptron predictor with two weights to capture correlation from the 2-bit history formed with GHR[0] and GHR[7]. Dynamic history length adaptation, on the other hand, would fail to exploit such correlation patterns.

Thirdly, if the perceptron size is fixed (e.g., with 3 weights), we can reconfigure the input to maximize branch correlation by exploring a much larger history-information set and replacing weak correlation bits with stronger ones. In this way, the predictor resource is used more efficiently with strongly correlated inputs from a large information set. For example, rather than using a 3-bit GHR (GHR[2:0]), we can form the input vector to the perceptron predictor with GHR[1:0] and a redundant bit (GHR[0] \( \land \) GHR[1]) since the redundant history is helpful to handle
linearly inseparable data sets [34]. For the example in Figure 2, the misprediction rate of branch $B4$ drops to 0% with this redundant information bit.

From the above observations, we can see that with correlation strength effectively measured, there are new opportunities to further improve the resource efficiency and prediction accuracy of branch predictors. In the next section, we present our design effort to enhance prediction accuracy with a given prediction table budget for the 1st Journal of Instruction Level Parallelism (JILP) Championship Branch Prediction Competition (CBP-1) [43]. In the rest of this chapter, we use a MAC perceptron predictor [34] as the baseline predictor although the proposed idea does not depend on any particular implementation of perceptron predictors.

3.2. Enhancing Branch Correlation Using Adaptive Information Processing

Based on the principles developed in Section 3.1, we propose two types of adaptive schemes to extract input bits from a large set of information to maximize branch correlation. The first is based on static workload profiling. The second is based on dynamic examination of perceptron weights and serves as a safe-net for workload misdetections or phase variations inside a workload.

In a MAC predictor, the perceptron weights are distributed in several tables. For each table, 4-bit information data are selected and used as the column address for the corresponding weight. The 4-bit inputs to all the weight tables simply form the information vector, as shown in Figure 3. The shaded units in Figure 3 are introduced for our proposed adaptation schemes, as described in detail next.
3.2.1. Profile-directed Adaptation

The adaptive information processing method was proposed in our submission to the 1st Journal of Instruction Level Parallelism (JILP) Championship Branch Prediction Competition (CBP-1). Therefore we use the distributed CBP-1 traces to illustrate our design.

The distributed CBP-1 traces are grouped in four categories: floating point (FP), integer (INT), multi-media (MM), and server (SERV) benchmarks. Each type of workload exhibits different behavior and the same information bits, such as PC, LHR, or GHR, carry different correlation strength for different types of workloads. For example, SERV benchmarks have a large number of static branches and their PC bits carry strong correlation when multiple branches share the same perceptron. FP workloads, in comparison, have much fewer static branches and show stronger correlation with global/local history than PC bits.
Because of this workload-dependent behavior, we propose to use static profiling to determine appropriate configurations of the input vector for each type of workloads. Then, at run-time, a workload detector determines the workload type and selects a predetermined configuration accordingly. Since this adaptation is based on profiling, it is called profile-directed adaptation. In our proposed implementation as shown in Figure 3, the inputs to the first multiplexer (MUX1) of each perceptron weight table are the pre-determined configuration and the control of the multiplexer is the workload type determined by the workload detector. Based on CBP-1 traces, the predetermined configurations for each type of workloads are shown in Figure 4. Taking the weight table 1 (labeled ‘wt table 1’) as an example, the inputs to its MUX1 are 4-bit local history (LHR[0:3]) for FP workloads and 4-bit PC (PC[0:3]) for other workloads.

For CBP-1 traces, the workload detector uses the following detection criteria: (a) SERV benchmarks feature a large number of static branches; (b) a small/medium number of static branches, a high number of floating point operations, and a high/medium number of instructions using XMM registers imply FP/MM workloads; and (c) the remaining benchmarks are treated as default INT workloads. More details can be found in the publicly distributed source code [43].

![Figure 4. Pre-determined input configurations for different types of workloads.](image-url)
3.2.2. Correlation-directed Adaptation

Profile-directed adaptation exploits coarse-grain workload behavior (i.e., one configuration for the entire run after the workload type is determined). If a workload exhibits significant phase behavior\(^1\), such a global configuration would fail to fit the changing requirements. To overcome such inefficiency, we propose an additional adaptation scheme based on dynamical correlation examination and it is named *correlation-directed adaptation*.

In correlation-directed adaptation, the strength of correlation is examined at a certain execution interval (or a program phase) and the inputs with weak correlation is replaced with those potentially having stronger correlation. For example, if we find some LHR bits have weak correlation, while GHR bits show stronger correlation, replacing those LHR bits with additional GHR bits would be an appealing choice.

As shown in Figure 3, besides the MUX1, which is used to support profile-directed adaptation, there is another multiplexer (MUX2) for each table. The MUX2 fine-tunes the predetermined workload-dependent information vector based on the strength of branch correlation. For each weight table, the inputs to its MUX2 include the 4-bit output from MUX1, 4-bit GHR, 4-bit LHR, and 4-bit PC. The control bits of MUX2, \(c_i\), are initialized so that the 4-bit output from MUX1 is selected. Then, after a certain execution interval (or a program phase), the strength of branch correlation is examined for the 4-bit information data and the bits with weak correlation are replaced. For example, in order to measure correlation strength of the 4-bit input to weight table 1, all weights in the row selected by \(\text{index1}\) are read and the summation of weights (absolute values) will be used as a quantitative measure of the correlation between the 4 information bits and the current branch instruction. If it turns out that the GHR/PC/LHR bits

\(^1\) The distributed CBP-1 traces show little phase behavior given its limited trace length.
carries the strongest correlation, the 4 information bits with the minimum correlation will be
replaced with 4 more GHR/PC/LHR bits by setting the control bits of MUX2 of the
corresponding table. For example, for a branch in a MM benchmark, if its 4 PC bits (e.g., the
input to weight table 1) have the strongest correlation while its LHR bits (e.g., the input to
weight table 2) have the weakest correlation, the input to weight table 2 will be configured as 4
more PC bits (e.g., PC[4:7]), replacing the LHR bits.

In the design shown in Figure 3, each entry in a weight table has its own control signal $ci$
so that the input vector can be tuned for each entry. For example, the perceptron corresponding
to the first entry of the weight table 1 may choose to exploit correlation from 4 PC bits while the
one corresponding to the second entry exploits correlation from 4 GHR bits. Such flexibility
requires more hardware resource (2-bit control for each entry) and more complex information
feeding logic. A more efficient design is to tune the input configuration at the table level so that
all the entries in the same table will share the same input configuration (i.e., only one $ci$ for each
table). We call these two adaptation designs as per-entry and per-table correlation-directed
adaptation and examine their performance impact in Section 3.4.

3.3. Overall Branch Predictor Structure

For conditional branches with simple always taken/not-taken patterns and loop branches,
a perceptron predictor is not a good candidate due to its complexity, potential latency and power
consumption issues. In addition, such simple branch patterns could pollute perceptron weights
when the same weights are shared with other branches. Therefore, we propose a bias branch
predictor and incorporate a loop branch predictor to handle these special branches, as shown in
Figure 5.
Figure 5. The overall branch predictor structure.

The bias predictor is an array of 2-bit FSMs indexed with PC. Each FSM transits from the initial state ‘11’ into the state ‘01’ or ‘00’ depending on whether the branch is taken or not taken respectively. The state ‘01’ implies the branch is always taken while the state ‘00’ means the branch is always not taken. An FSM transits to the state ‘10’ whenever the actual outcome disagrees with the bias state. In this case, the final prediction will be made by other predictors.

The loop branch predictor is a set-associative cache structure and each entry consists of a loop count, a current iteration count (taken count), a tag, a confidence (increased/decreased when the loop count matches/mismatches) counter, and LRU bits. A branch hits in the loop branch predictor only if there is a tag match and the confidence is high.
The overall prediction priority order is the bias prediction if a branch is biased (i.e., the bias state is not ‘10’), the loop branch prediction if the branch hits in the loop predictor, and then the prediction from the MAC perceptron predictor with adaptive information processing.

During update, the bias predictor is updated first. Only if the bias state is 10 (i.e., not biased), the loop branch predictor and the MAC perceptron predictor are examined. If the branch hits in the loop branch predictor, the MAC perceptron predictor is not updated. In this way, the adverse impacts from bias and loop branches upon the MAC perceptron predictor, including weight pollution and unnecessary updates, are effectively eliminated.

With the branch predictor configuration that we proposed for CBP-1, the hardware budget requirements are summarized in Table 1.

Table 1. Hardware budget.

<table>
<thead>
<tr>
<th>COMPONENT</th>
<th>CONFIGURATION</th>
<th>COST</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bias branch predictor</td>
<td>2293 entries</td>
<td>2293 x 2 = 4586 bits</td>
</tr>
<tr>
<td>Loop branch predictor</td>
<td>24 entries, 8-way set associative</td>
<td>1344 bits</td>
</tr>
<tr>
<td>Information</td>
<td>PC, GHR: 100 bits, LHR: 8 bits, 63 entries</td>
<td>32 + 100 + 8 * 63 = 636 bits</td>
</tr>
<tr>
<td>MAC perceptron predictor with</td>
<td>W0 table: 61 entries, 8 bits each</td>
<td></td>
</tr>
<tr>
<td>adaptive information processing</td>
<td>Other tables’ sizes: 63, 55, 53, 51, 49, 43, 41, 41, 41, 41, 39, 37, 37, 35. Total MAC entries: 597. Each MAC entry has 16 weights. Each weight has 6 bits. Control bits: 2 bits each entry</td>
<td>61 * 8 + 597 * 16 * 6 + 597 * 2 = 58994 bits</td>
</tr>
<tr>
<td>Adaptation Logic</td>
<td>Interval: 100000 conditional branches</td>
<td>22 bits</td>
</tr>
<tr>
<td>Workload Detector</td>
<td>Interval: 10000 instructions and 10000 conditional branches</td>
<td>82 bits</td>
</tr>
<tr>
<td>Total Cost: 65649 bits</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
3.4. Results and Analysis

In this section, we evaluate the performance of the proposed predictor for CBP-1 traces and SPEC2000 benchmarks.

3.4.1. Performance for CBP-1 traces

Figure 6 shows the prediction accuracy of different branch predictors with the same 8KB prediction table size, including a gshare predictor, an MAC perceptron predictor, and our proposed predictor with different adaptation schemes, which include profile-directed adaptation (labeled as ‘profile’), profile-directed with per-table correlation-directed adaptation (labeled as ‘profile+per-table’), and profile-directed with per-entry correlation-directed adaptation (labeled as ‘profile+per-entry’). Compared to the gshare predictor, our proposed adaptive schemes reduce the misprediction rate by 47% on average. Compared to the fine-tuned baseline MAC perceptron predictor, the improvement from adaptation is close to 10%. As the distributed CBP-1 traces do not exhibit strong phase behavior, profile-directed adaptation reaps the most benefit (2.847 mispredictions per 1K instructions) and correlation-directed adaptation only shows small incremental improvements (2.846 and 2.822 mispredictions per 1K instructions for per-table and per-entry correlation-directed adaptation respectively).

Since our overall predictor design contains multiple predictors handling different branches, we investigate the contribution of each predictor in Figure 7, which shows the percentage of predictions (both correct and incorrect) made by each predictor. We can see that the simple bias predictor and the loop predictor filter out a significant amount of dynamic branches and effectively reduce the aliasing and update cost of the MAC perceptron predictor. On average, among all the dynamic predictions, 17.61% (and 0.03%) are correct (and incorrect).
predictions made by the bias predictor, 16.56% (and 0.03%) are correct (and incorrect) predictions made by the loop predictor, and 63.13% (and 2.64%) are correct and (incorrect) predictions made by the MAC perceptron predictor with adaptation.

Figure 6. Branch misprediction rates on CBP-1 traces.

Figure 7. Prediction contribution from individual branch predictors.
3.4.2. Performance for SPEC2000 Benchmarks

In this experiment, we examine the prediction accuracy of the proposed scheme on SPEC 2000 CINT and FP benchmarks [15]. In our simulation, the reference inputs are used. We fast-forward the first 500M instructions and simulate the next 1000M instructions. We also assume that the workload type (FP or INT) can be detected correctly. The profiled FP/INT configurations based on the CBP-1 traces are used for SPEC FP/CINT benchmarks.

From the misprediction rates that are reported in Figure 8, we can see that our proposed predictor outperforms the gshare predictor by 35% on average. Another interesting observation is that the profiled FP and INT configurations based on the CBP-1 traces work well for SPEC benchmarks. We gain little improvements from the additional per-table and per-entry correlation-directed adaptation. This result is promising since profile-directed adaptation is much simpler to implement than correlation-directed adaptation.

Individual contributions from the bias, loop, and MAC perceptron predictor with adaptation are examined in Figure 9. The bias and loop predictors accurately predict a significant amount of dynamic branches of SPEC 2000 benchmarks, similar to the results of CBP-1 traces.
Figure 8. Branch misprediction rates on SPEC 2000 benchmarks.

Figure 9. Prediction contribution from individual branch predictors.

3.5. **Summary**

In this chapter, we take an untraditional perspective to attack the branch prediction problem: rather than a new prediction scheme to better exploit branch correlation, we propose to process the inputs to branch predictors to maximize such correlation. An important observation is
made from perceptron branch predictors that the perceptron weights provide an estimate of the correlation strength. Based on such correlation strength, novel adaptive information processing schemes are proposed to dynamically re-assemble the input information vector so that it contains only those PC/history bits with the strongest correlation. The experiments using CBP-1 traces and SPEC 2000 benchmarks show that our proposed approach achieves significant improvement on prediction accuracy over other competing branch prediction schemes.
Most recent developments for branch prediction mainly focus on two directions, perceptron branch predictors [16], [17] and predictors derived from the Prediction by Partial Matching (PPM) algorithm [8].

PPM algorithm was a powerful algorithm proposed originally for data compression and it has been used to establish a theoretical limit on branch prediction accuracy. The PPM branch predictor uses a set of Markov chains to keep track of branch outcome history. A Markov chain of order $H$ records the transition probability of the next branch outcome for each possible context pattern based on the $H$ immediately preceding branch history bits. A PPM algorithm of order $N$ uses the $N$ immediately preceding bits to search for a match in the highest-order Markov chain. If no match is found (i.e., the pattern has not been recorded yet), it will use $(N-1)$ immediately preceding bits to search for a match in the second highest-order Markov chain. This process continues until the longest partial match is found and the corresponding Markov chain will be used to make a prediction.

In this chapter we present our study on the PPM branch predictor [7] and show that the longest partial match policy used in PPM algorithm can lead to suboptimal results for branch prediction. The reason is mainly due to the time-varying unstable behavior of branches. To overcome such a limitation, we propose to combine multiple partial matches to make a prediction instead of relying on the longest partial match [12], [13]. We call such an algorithm Prediction by Combining Multiple Partial Matches (PMPM). We explored various ways to
combine multiple partial matches and we found that a simple adder tree provides an efficient way to combine multiple partial matches. Furthermore, the adder tree makes it easy to integrate multiple PPM predictors exploiting different types of correlation. For example, it has been known that both local and global branch history carry strong correlation that can be exploited to make a prediction. However, it is difficult to use a single PPM predictor to incorporate both types of history information.

Based on our study of PPM and PMPM, we also found that combining all partial matches can be harmful because the lower order Markov chains are more susceptible to noises generated by branches having near random outcome behavior and combining several longest matches provides more accurate predictions. Therefore, we propose a scheme to selectively combine multiple partial matches. Our simulation results show that using global branch history only, the PMPM scheme significantly outperforms the previously deemed optimal PPM algorithm by up to 13.6% and 6.6% on average.

To apply the proposed PMPM algorithm to practical implementation, we propose a design based on a recently developed PPM-like TAGE branch predictor [35]. Our design has similar hardware complexity compared to the TAGE predictor and it enables efficient integration of local history information. Our experimental results show that with a 32kB storage budget our design achieves higher prediction accuracy, up to 54% and 5.8% on average, than the TAGE predictor. Finally, we implement ahead pipelining to evaluate the impact of the access latency of our proposed PMPM predictor and the results show that our design can provide a prediction every cycle with relatively small accuracy loss.
4.1. Branch Prediction by Combining Multiple Matches

In this section, we present our study on PPM algorithm. We first describe the way we model the PPM predictor. Next, we show why PPM is suboptimal for branch prediction. Based on our analysis, we then propose a novel prediction scheme to combine multiple partial matches to improve prediction accuracy.

4.1.1. Modeling PPM

We model the PPM predictor using the same way as proposed by Michaud et al. [24]. For each branch, we construct a sequence of Markov chains, one for each history length. Since the number of possible patterns grows exponentially with the history length, we limit the maximum history length to 40 and the history information is limited to global branch history only. For each pattern, we use a saturated signed counter within the range \([-4, 4]\) to record the possibility of the next bit transition. When the pattern is encountered the first time, the prediction counter is initialized to 1 if the branch following the pattern is taken and -1 otherwise. Once a prediction counter is initialized, it will be incremented by 1 when the corresponding branch is taken and decremented by 1 otherwise. When a branch outcome is determined we update all Markov chains of the branch. The limited range on prediction counters is due to rapid changes in branch behavior and a larger range results in poor prediction accuracy based on our results.

With the maximum history length of 40, we have up to 41 partial matches with history lengths from 0 to 40 at the prediction stage. When we have no matches, i.e. the first occurrence of a new static branch, not-taken is used as the prediction. When multiple partial matches are available, PPM algorithm uses the sign of the prediction counter associated with the longest partial match as the prediction. If the counter is zero, the prediction is taken.
All simulation results presented in this chapter are based on the 20 distributed traces from the 2nd Journal of Instruction Level Parallelism (JILP) Championship Branch Prediction Competition (CBP-2) [44]. Each trace contains the branches encountered in a phase of 100 million instructions from a wide range of benchmarks. The branches from the library code and system activities are also included in the traces. With the PPM of order 41, the misprediction rates, measured in mispredictions per 1000 instructions (MPKI), are shown in Figure 10.

From Figure 10, we can see that gzip, vpr, mcf, and twolf contain a large number of hard-to-predict branches. Some hard-to-predict branches have random like behavior. Because the PPM predictor is built upon global branch history, those branches also affect the states in Markov chains of other branches, of which the prediction accuracy is reduced.

4.1.2. A Study on PPM for Branch Prediction

In this section, we show that with a simple modification, the prediction accuracy of PPM algorithm can be improved. In the PPM algorithm, the prediction counters record the transition probability of the subsequent branch and we can view them as a confidence of the prediction. If the branch direction is highly biased to one direction for a specific history, the corresponding
counter will likely be saturated at either the largest or the smallest value. On the other hand, if the branch direction is not biased, the counter will keep changing around zero most of the time. In the extreme case, when the counter value is zero, the prediction confidence is the lowest as it is equally possible for the transition to be taken or not-taken based on the history information. As a result, those longest matches with a zero-value prediction counter provide essentially no useful information of the subsequent branch behavior. Based on this observation, we modify the PPM algorithm to eliminate those longest partial matches with a zero-value prediction counter. In other words, we treat those matches with a non-zero value prediction counter as confident matches and we only use the longest confident match to make a prediction. The misprediction rate reductions of such a confidence-based PPM prediction mechanism compared to the PPM algorithm are reported in Figure 11. From the results, we can see that the confidence-based PPM has lower misprediction rates than the PPM scheme for all the benchmarks except vortex.

![Figure 11. Misprediction rate reductions of the confidence-based PPM compared to PPM.](image)

The previous experiment demonstrates that the longest match used in the PPM algorithm is not an optimal choice. Next, we show that any fixed single confident match selection policy is not optimal either. We first generalize the longest confident match to the $K^{th}$ longest confident match, which means that we use the $K^{th}$ longest confident match to make a prediction rather than
the longest confident match. The results, shown in Figure 12, report the average misprediction rates of PPM and the prediction by the $K^{th}$ longest confident partial match with $K$ varying from 1 (i.e., the longest) to 10 (i.e., the 10$^{th}$ longest).

Two important observations can be made from Figure 12. First, the longest confident match is not the most accurate one. A detailed study reveals the following reason: among multiple confident matches, their predictions may not agree with each other and different branches may favor different history lengths for more accurate predictions. Such history-length adaptivity is also reported in [19]. To solve this problem, we propose to combine the correlation found at different history lengths (i.e., combine multiple partial matches) so that we can explore correlation in a wide range of history lengths. Second, several longest confident matches provide the most accurate predictions. As we further increase $K$, the misprediction rate increases. This implies that the Markov chains modeling shorter histories are more susceptible to noises generated by branches with near random outcome patterns. This trend suggests that when we combine multiple partial matches, higher priority needs to be given to longer history matches.

![Figure 12. Misprediction rates of PPM and prediction by the K-th longest confident partial matches.](image-url)
4.1.3. Prediction by Combining Multiple Partial Matches (PMPM)

As discussed in Section 4.1.2, combining multiple partial matches enables the exploitation of correlation at different history lengths. In this section, we explore various ways to combine multiple partial matches to make a prediction.

**Selecting a single match from multiple partial matches**

One way to combine multiple partial matches is to select one single match that is most likely to be correct. The longest match used in the PPM algorithm is one such scheme based on the assumption that the longest history provides the most accurate context information to make a prediction. The confident longest match is another way to select a single match in order to eliminate those longest matches that have no strong correlation.

Another option is to select the most confident match from multiple matches. If there are more than one highly confident match (i.e., a prediction counter saturated at either maximum or minimum value), the tie can be broken by selecting the one with the longest history match.

**Linear combination of multiple partial matches**

Since the prediction counter embeds the prediction confidence, we can use the following linear combination model to combine multiple partial matches. For a set of N Markov chains associated with a branch, the final prediction, P, can be generated as a weighted sum of all the prediction counters for which there is a pattern match, i.e.,

\[
P = \text{sign}(\sum w_i \cdot p_i),
\]

where there is a pattern match at the \(i^{\text{th}}\)-order Markov chain with \(i\) varying from 0 to N. The parameters \(w_i\) and \(p_i\) are the weight and the prediction counter associated with the matching pattern in the \(i^{\text{th}}\)-order Markov chain, respectively. The weights can be trained using the perceptron algorithm [16].
A simplification of linear combination is to sum all the prediction counters to make a prediction, i.e., $P = \text{sign}(\sum p_i)$, where there is a pattern match at the $i^{\text{th}}$-order Markov chain with $i$ varying from 0 to N.

**Majority vote of multiple partial matches**

In this scheme, multiple matches are treated equally and the prediction is computed as follows, $P = \sum [\text{sign}(p_i)]$, where there is a pattern match at the $i^{\text{th}}$-order Markov chain with $i$ varying from 0 to N.

**Selective combination of multiple partial matches**

As discussed in Section 4.1.2, low-order Markov chains are more susceptible to noises than high-order ones. Therefore, when we combine multiple partial matches to make a prediction, we can sample the matches so as to reduce the contribution from those noise-sensitive Markov chains. Such selective combination achieves the similar goal to weighted summation or linear combination by giving higher priority to certain Markov chains. There are many possible selection mechanisms and one simple example would be using the summation of the prediction counters or majority vote corresponding to the $L$ longest confident partial matches.

4.1.4. Pushing the Limit of Branch Prediction Accuracy using PMPM

We performed extensive experiments on the multiple match combination schemes described in Section 4.1.3 and observed the following trends from the experimental results. First, although selecting a single match from multiple matches, achieves higher prediction accuracy than PPM as exemplified in Section 4.1.2, it does not yield as high accuracy as the linear combination of multiple matches in general. Second, the attempt to combine all partial matches
using the simple summation or majority vote results in low prediction accuracy since there are too many matches from low-order Markov chains (or short-history matches) and the final prediction is easily affected by branches with near random outcome patterns. Between these two simple schemes, the majority voting scheme has a lower accuracy since it ignores the confidence information embedded in prediction counters. Third, selective combinations such as the summation of several longest confident partial matches provides highly accurate predictions and we will examine this scheme in more details due to its simplicity and high prediction accuracies. Fourth, compared to selective combination using simple summation, the linear combination of all partial matches does not offer significant additional benefits. The reasons include (a) the learning time of the weights, especially those corresponding to the low-order (or short-history) Markov chains, affects the prediction accuracy; and (b) the value of prediction counters already embeds the confidence information, which is captured by weights redundantly. As a result, weighted summation does not show better results than selective combination using simple summation.

![Figure 13. Misprediction rates of PMPM-L algorithms, the PPM algorithm and the prediction by the 3rd longest confident partial match scheme.](image)

To show the impact of combining multiple longest confident partial matches using summation, we perform the following experiment. We vary the number of longest confident matches that are to be combined using summation and we use PMPM-L to denote the scheme.
that uses the summation of $L$ longest confident matches to make a prediction. The average misprediction rates for different number of matches are shown in Figure 13. For comparison, Figure 13 also includes results of the PPM algorithm and the prediction by the 3rd longest confident partial match scheme, which is the best single match scheme.

From Figure 13, we can see that on average, combining multiple partial matches can provide higher prediction accuracy than utilizing a single partial match. Meanwhile, combining too many partial matches can be harmful since many low-order Markov chains are included, which are susceptible to noises. Furthermore, we observe that the optimal number of longest confident matches to be combined is workload dependent. Some benchmarks require few longest partial matches while others need a much higher number, as shown in Table 2.

Table 2. The optimal $L$ for the PMPM-L algorithm.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>gzip</th>
<th>gcc</th>
<th>crafty</th>
<th>compress</th>
<th>raytrace</th>
<th>javac</th>
<th>mtrt</th>
<th>eon</th>
<th>gap</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal L</td>
<td>19</td>
<td>4</td>
<td>9</td>
<td>8</td>
<td>4</td>
<td>8</td>
<td>2</td>
<td>2</td>
<td>23</td>
<td>24</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>vpr</th>
<th>mcf</th>
<th>parser</th>
<th>jess</th>
<th>Db</th>
<th>mpegaudio</th>
<th>jack</th>
<th>perlmbk</th>
<th>vortex</th>
<th>twolf</th>
</tr>
</thead>
<tbody>
<tr>
<td>Optimal L</td>
<td>18</td>
<td>12</td>
<td>9</td>
<td>7</td>
<td>4</td>
<td>9</td>
<td>4</td>
<td>3</td>
<td>7</td>
<td>11</td>
</tr>
</tbody>
</table>

To push the limit of branch prediction accuracy using the PMPM-L scheme, we propose to dynamically learn the best $L$ for each static branch. In order to do so, we assign $N$ (e.g., $N = 41$) counters, denoted as $C[k]$ ($1 \leq k \leq N$), with each static branch to learn the optimal $L$ for that branch. The counter $C[k]$ keeps tracking the number of correct predictions made with combining $k$ longest confident matches. Then, at the prediction phase, we choose to combine $L$ longest confident matches where $C[L]$ has the highest value among $C[k]$ ($1 \leq k \leq N$). In Figure 14, we compare the reductions in misprediction rates achieved by this adaptive PMPM and the best PMPM-$L$ ($L=11$ based on the results shown in Figure 13) over the PPM algorithm.
Figure 14. Misprediction rate reductions of the best PMPM-L ($L = 11$) and Adaptive-PMPM compared to PPM.

From Figure 14, we can see that although PMPM-L has better prediction accuracy on average than PPM. It may not be the case for individual branches or particular benchmarks. With adaptive PMPM, we can achieve higher prediction accuracy for each benchmark. Compared to PPM, adaptive PMPM reduces the misprediction rate by up to 13.6% ($\text{bzip2}$) and 6.6% on average. More importantly, we can see that the adaptive PMPM achieves higher than average misprediction rate reduction for hard-to-predict benchmarks, $\text{gzip}$, $\text{vpr}$, $\text{mcf}$, and $\text{twolf}$, which demonstrates the effectiveness of combining multiple matches for branches that are hard to predict.

4.2. Idealistic Adaptive-PMPM Predictors

As discussed in Section 4.1.1, the history length is limited to 40 in previous experiments. To further push the limit of prediction accuracy, we need a way to handle much longer global history and to incorporate other types of information such as local branch history. We develop an idealistic Adaptive-PMPM predictor which explores extremely long global history (2000 bits) with acceptable requirements on memory storage and simulation time.
4.2.1. Predictor Structure

The overall predictor structure is shown in Figure 15. We use a per-branch tracking table (PBTT) to record some basic information of each branch. PBTT is a 32k-set 4-way cache structure. Each PBTT entry includes the branch address, LRU bits for replacement, a branch tag, a 32-bit local history register, a meta counter used to select either GHR-based or LHR-based predictions, a bias counter, a simple (bias) branch predictor, and two arrays of counters that are used to learn the optimal numbers of local and global counters to be combined as described in Section 4.1.4. A unique tag is assigned to each static branch and it is used to replace the branch address in index and tag hashing functions. Since there are a large number of highly biased branches, we use a simple branch predictor to filter them out. The simple branch predictor detects fully biased (always taken or always not taken) branches. A branch only accesses the main Adaptive-PMPM predictor when it is not fully biased. For remaining branches, PBTT sends the branch tag, LHR, meta counter, and bias counter to the Adaptive-PMPM predictor.

Figure 15. The overall idealistic Adaptive-PMPM branch prediction scheme.
The Adaptive-PMPM predictor has three prediction tables, as shown in Figure 16. A short-GHR table is used to store information corresponding to the most recent 32-bit global history. A long-GHR table is used to store information corresponding to longer global histories. We also use a LHR table to make local-history-based predictions and the LHR table uses 32-bit local histories. Those three prediction tables are 4-way set-associative structures and each entry has a tag, an LRU field and a prediction counter. In order to capture very long global histories, we use geometric history lengths [28] in the long GHR table. Since the CBP-2 traces include both kernel and user branches, we use two sets of history registers to avoid history pollution, as proposed by Seznec in [30].

Figure 16. An idealistic Adaptive-PMPM predictor.
4.2.2. Prediction Policy

We hash the branch tag, global history, path history, history length to get the index and tag to access each table in the Adaptive-PMPM predictor. We use similar hash functions to those in [35] and [18] with empirically selected prime numbers. The indexes and tags use different primary numbers in their hash functions.

The short GHR table uses global histories with lengths 1 to 32. For longer histories, we use 96 different lengths in order to reduce the storage requirement. Both short and long GHR tables use 32-bit path history. We combine 0 to 128 longest confident matches with the bias counter to generate the GHR-based predictions. Based on the previous accuracy of combining different numbers of global prediction counters, an optimal number $M$ is decided and the GHR-based prediction with $M$ global prediction tables is used as the final GHR-based prediction. The LHR prediction table works the same way as the short GHR table with the selection of the optimal $N$, which is the number of local prediction counters to combine. The final prediction is selected by the meta counter.

4.2.3. Update Policy

The prediction counters are updated if the corresponding entries have a tag match with the current branch. The tag contains both branch address and context information as described in Section 4.2.2. At the same time, we train the accuracy counters corresponding to combining different number of prediction counters based on the actual branch outcome.

In the case of a miss, i.e., the branch history has not been retained in the tables, new entries are allocated in all prediction tables based on the LRU replacement policy.
4.2.4. Results

We simulate the proposed idealistic Adaptive-PMPM predictor with the configuration shown in Table 3.

Our experimental results show that the idealistic Adaptive-PMPM predictor achieves an average misprediction rate of 2.690 MPKI with a maximum global history length of 2000. It outperforms the idealistic GTL predictor (2.718 MPKI) [30], which was the winner of the idealistic track of CBP-2 [44]. The GTL predictor is a hybrid predictor combining a GEHL predictor [28], a TAGE predictor [35], and a loop predictor. The limit of prediction accuracy can be further improved by replacing the TAGE predictor in GTL with our Adaptive-PMPM predictor and the resulting hybrid predictor reduces the average misprediction rate to 2.620 MPKI, a 3.6% improvement over the currently achieved limit of branch prediction accuracy.

Table 3. Idealistic Predictor Configuration.

<table>
<thead>
<tr>
<th>Types of Prediction Tables</th>
<th>Detailed Settings</th>
</tr>
</thead>
<tbody>
<tr>
<td>Short-GHR Prediction Table</td>
<td>4M sets, 2-way</td>
</tr>
<tr>
<td>Long-GHR Prediction Table</td>
<td>32M sets, 2-way</td>
</tr>
<tr>
<td>LHR Prediction Table</td>
<td>1M sets, 2-way</td>
</tr>
<tr>
<td>Maximum # of selected counters for GHR matches</td>
<td>128</td>
</tr>
<tr>
<td>Maximum # of selected counters for LHR matches</td>
<td>32</td>
</tr>
<tr>
<td>Minimum GHR length of the geometric history lengths used by the Long-GHR Table</td>
<td>33</td>
</tr>
<tr>
<td>Maximum GHR length of the geometric history lengths used by the Long-GHR Table</td>
<td>2000</td>
</tr>
<tr>
<td>Range of a prediction counter</td>
<td>[-5, +5]</td>
</tr>
</tbody>
</table>
4.3. Realistic PMPM Predictors

In this section, we present our PMPM branch predictor design for practical implementation. As discussed in Section 4.1, the PMPM algorithm is built upon PPM. Therefore, we choose to develop our design from recently proposed PPM-like branch predictors [23], [35], the TAGE branch predictor [35] in particular. A TAGE branch predictor employs multiple tables to emulate Markov chains of different orders. The Markov chains associated with different branches share the tables and the table entries are partially tagged to reduce aliasing. Each table entry corresponds to a pattern from a Markov chain and it has a prediction counter to record the transition frequency of the subsequent branch following the pattern. Among multiple tag matches found in multiple tables, the one corresponding to the longest history (i.e., longest partial match) is selected to make a prediction. It has been shown that TAGE predictors achieve very high prediction accuracy compared to other state-of-art branch predictors and are regarded as practical for hardware implementation.

4.3.1. Predictor Structure

The overall structure of the proposed PMPM predictor is shown in Figure 17. Here, the same multi-table structure is used as in the TAGE predictor, except that a local history prediction table is incorporated. The prediction and update policies, however, are completely redesigned to implement PMPM. The PMPM predictor also shares similarity to the O-GEHL predictor [28], which also uses multiple prediction tables and an adder tree to generate branch predictions. Compared to O-GEHL predictors, two key differences make a PMPM predictor potentially more accurate. First, in an O-GEHL predictor all potential prediction counters are included in the summation, which lacks the critical selective combination part as we show in Section 4.1.
Second, the PMPM predictor can efficiently integrate local history information to make predictions, which is shown to be difficult in the O-GEHL predictor [28].

In order to reduce the design space and simplify the counter selection logic, we use the reference configuration for the TAGE predictor [35], although it has been shown that further tuning of the number of tables may improve the prediction accuracy of both TAGE and our PMPM predictors. As shown in Figure 17, the PMPM predictor contains a bimodal prediction table [37], seven global prediction tables (labeled as “gtable0” to “gtable6”) indexed by the branch address, global history and path history, and a local prediction table (labeled as “ltable”) indexed by the branch address and local history. Geometrical history lengths [28] are used for the global prediction tables: gtable0 is associated with the shortest global history and gtable6 is associated with the longest global history. Each entry of the global prediction tables and the local prediction table has three fields: a tag, a signed saturated prediction counter (labeled as “ctr”)
and an unsigned saturated counter (labeled as “uctr”) to track the usefulness of the prediction counter. Index and tag hashing functions for the global prediction tables are same as those used in TAGE predictors. The local prediction table uses hashed branch address and local history as the index and the XOR-folded (i.e., \(PC \, ^\uparrow (PC \, >> \, M)\), where \(M\) is the tag width) branch address as the tag.

### 4.3.2. Prediction Policy

The first stage of the prediction phase is to calculate indexes and tags for each prediction table. Then an entry is read out from each prediction table using these indexes. Next, tag comparison is performed except for the bimodal table. Besides the (tag-less) bimodal counter, the prediction counters with a tag match will be selectively added together to make a prediction.

The selection policy for local prediction is as follows: the counter from the local prediction table will be selected only when its corresponding uctr is larger than 1, which means it is a highly useful prediction. For global prediction tables, we may have up to 7 matches. In order to reduce the logic complexity and counter selection latency, we use an approximation of the selection policy described in the PMPM algorithm in Section 4.1. We divide the global prediction tables into groups and select the longest match from each group. For example, when the group size \(S\) equals to 2, the seven tables will be divided into 4 groups: \((gtable0, gtable1), (gtable2, gtable3), (gtable4, gtable5)\) and \((gtable6)\). When \(S\) equals to 1, all matches will be selected. When \(S\) is set to 7, only the longest match will be selected. With the baseline update policy presented in Section 4.3.3, we simulate a 32kB PMPM predictor with \(S\) varying from 1 to 7 and the results are shown in Figure 18. From the results, we can see that the misprediction rate is lowest when \(S\) is 2, which is relatively easy to implement as the selection logic of each group would be a simple...
multiplexer controlled by the two tag comparison results. In remaining experiments in this chapter, we use the selection logic with $S$ equal to 2 for PMPM predictors.

![Figure 18. Average misprediction rates of PMPM predictors with $S$ from 1 to 7.](image)

When $S$ set as 2, up to 4 counters from the global prediction tables and 1 counter from the local prediction table can be selected. The adder tree calculates the sum of those counters and the bimodal prediction counter. When the sum is zero, i.e., the confidence is low, we choose to use the prediction from the bimodal table. If the sum is not zero, the sign of the sum is used as the prediction (i.e., predict taken if it is positive).

### 4.3.3. Update Policy

As shown in previously proposed branch predictors [29], [33], [35], the update policy has a critical impact on the prediction accuracy, especially when storage cost is limited. A well designed update policy needs to achieve two contradictive goals. On one hand, in order to achieve high prediction accuracy we want to record more history patterns for each branch, which means more prediction counters should be assigned to a branch. On the other hand, in order to reduce aliasing we want to assign and update fewer counters for each branch. For those reasons,
the optimal update policy may be dependent upon the storage budget. In our design, the update policy is optimized for a 32kB PMPM predictor.

As described in Figure 17, a PMPM predictor has one bimodal prediction table, seven global prediction tables, and one local prediction table. The update policy decides the operations to be performed on each table. For the bimodal table, the prediction counter is always updated. The update policies of the global prediction tables and the local prediction table are described as follows.

**Update Policy of the Global Prediction Tables**

Similar to the perceptron predictor [16], [17], and the O-GEHL predictor [28], the prediction counters of the global prediction tables are updated only when the overall prediction is wrong or the absolute value of the summation is less than a threshold. We also adopt the threshold adaptation scheme proposed in the O-GEHL predictor to fit different applications. Instead of updating all prediction counters, we only update those counters that have been included in the summation. At the same time, for each of these prediction counters, the associated \( uctr \) counter is incremented when the prediction counter makes a correct prediction.

In the case of a misprediction, besides updating the prediction tables where the branch hits and their counters are used, we also try to allocate a new entry. The new entry will be selected from tables where the branch misses. As \( uctr \) is correlated to the usefulness of a prediction counter, we select one entry that has a zero \( uctr \) from those tables as a new entry allocated for the current branch. In this new entry, the tag will be replaced and the prediction counter \( ctr \) is initialized based on the branch outcome. If there are multiple entries with zero \( uctrs \), we select the one with the shortest history length. If there is no entry with a zero \( uctr \), we don’t allocate a new entry. At last, for each entry corresponding to a longer history than the
longest match, its $uctr$ counter is decremented. This process is termed as “stealing” entries in some previously proposed tagged branch predictors [23], [35].

**Update Policy of the Local Prediction Table**

If current branch hits in the local prediction table, we always update the prediction counter. The $uctr$ counter is decremented if the corresponding prediction counter makes a wrong prediction. If the prediction counter makes a correct prediction, we increment $uctr$ only when the overall prediction is wrong.

If the current branch doesn’t hit in the local prediction table and the overall prediction is wrong, we will try to allocate a new entry in the local prediction table. If the indexed entry has a zero $uctr$, a new entry is allocated in a similar fashion as in the global prediction table. Otherwise, its $uctr$ counter is decremented.

**Optimization upon the Update Policy**

The base update policy described above is also optimized to address the following two issues. First, due to our prediction policy, a newly allocated entry may not be included in the summation even if the same branch occurs immediately after the allocation. Second, for applications with a large number of hard-to-predict branches, some otherwise useful entries can be evicted due to frequent mispredictions.

To address the first issue, we modify the update policy so that in each group of two global prediction tables, a new entry will not be allocated in the table with the shorter history length if the branch hits in the table with the longer history length.

To address the second issue, we use a misprediction counter to periodically detect those applications/program phases with a large number of hard-to-predict branches. For those application/phases, we slightly vary the update policy: on a misprediction, we don’t decrement
the uctr counters in those prediction tables that have tag misses if we already allocate a new entry; and we will decrement the uctr of the longest match if its prediction counter is wrong. In this way, we limit the chance of victimizing useful entries.

The effect of these two optimizations of the update policy is shown in Figure 19. As we can see from the figure, the optimizations give a relatively large benefit (3.1% to 8.4% MPKI reductions) to vpr, mcf and twolf, which have a large number of hard-to-predict branches.

Figure 19. Effect of the update policy optimizations.

4.3.4. Hardware Complexity and Response Time

Similar to TAGE and O-GEHL branch predictors, the proposed PMPM branch predictor’s response time has three components: index and tag generation, prediction table access, and prediction computation. In the proposed PMPM predictor, we use similar index and tag hashing functions to those in the TAGE predictor for the global prediction tables. Given the complexity of those functions, one full cycle is assumed for the index and tag generation for global prediction tables. The tag generation for the local prediction table is a XOR-folding of the branch address. The index generation for the local prediction table has two sequential parts, the local branch history (LHR) table access followed by bitwise XORing of the LHR and the branch address for index. Since we use a small 1K-entry tag-less LHR table and each LHR has only 10
bits, we assume the overall latency for generating index for local prediction table is also one full cycle. Another way to look at this latency would be a table access plus a ‘tag’ match using a single level XOR gate.

The tagged prediction table access is also similar to the TAGE predictor. For small or medium tables, one cycle is assumed to read the prediction tables and generate the hit/miss signal. In the TAGE predictor, the prediction policy is dynamically adjusted to use the longest match or the 2nd longest match if the entry corresponding to the longest match is a newly allocated entry. The selection of those two longest matches has a longer latency than our simple one level counter selection logic. However, after counter selection we have an adder tree with up to six 5-bit inputs to calculate the prediction. Therefore, the overall latency of the prediction computation logic of the PMPM predictor should be similar to the TAGE and O-GEHL predictors. Overall, we expect that the response time of the proposed PMPM predictor should be very close to the TAGE and O-GEHL predictors, which is around three cycles depending on the hardware implementation technology.

4.3.5. Ahead Pipelining

As the prediction latency is typically more than one cycle, we use ahead pipelining [32] to generate one prediction per cycle. Although the original ahead pipelining scheme was designed only for global branch history predictors, we find that the scheme can also be used for the PMPM predictor. Assuming 3-cycle prediction latency, our 3-block ahead pipelining scheme for the PMPM predictor is shown in Figure 20.

In Figure 20 we show four basic blocks ending with branches A, B, C and D respectively. They are fetched in cycle 1 to cycle 4. In order to start fetching the block after branch D in cycle
5, the direction of D has to be known before the end of cycle 4. The 3-block ahead prediction of branch D is initiated at the beginning of cycle 2. In cycle 2, we use the address of B (labeled as ‘PC(B)’), the global history including the result of A (labeled as ‘GHR(A)’), and the path history including the address of A (labeled as ‘path(A)’) to calculate the indexes and tags for the bimodal table and the global prediction tables. PC(B) is also used to read the local history table and calculate the index and tag of the local prediction table. In cycle 3, we read 4 adjacent entries from each prediction table with the indexes calculated in cycle 2. Then, at the end of cycle 3 we have four sets of entries and tags. For each set we calculate a potential prediction in cycle 4. At last, the prediction for D is known before the end of cycle 4 by selecting one prediction from the 4 potential predictions with a 2-bit vector from a hashing function of the branch addresses and directions of B and C. Figure 20 only illustrates the scheme of 3-block ahead predictions. With the similar process, other n-block ahead prediction schemes can be designed by initiating the prediction process n cycles ahead.

As pointed out in [32], an n-block (n > 1) ahead pipelined predictor would have some accuracy loss compared to the ideal 1-block ahead predictor. From Figure 20, we can see that all branches with the same 3-block ahead information will be indexed to 4 adjacent entries in each prediction table, which potentially generates more aliases in prediction tables. Moreover, for predictors using local histories, the local branch history table would also have more aliases. The impact of ahead pipelining on the proposed PMPM predictor will be examined in Section 4.4.3.
1. Use PC(B), GHR(A) and PATH(A) to calculate the indexes and tags for the bimodal table and gtables.
2. Use PC(B) to read the local history and calculate the index and tags of ltable.

1. Read 4 adjacent entries from each prediction table. Tag comparisons may also be finished in this cycle.

1. Calculate 4 potential predictions.
2. Use information of B and C to select out one prediction.

Figure 20. A 3-block ahead scheme for the PMPM predictor.

Another issue with ahead pipelining is the index used to update the local history table. It has been shown in [29] that speculative updates of branch history are important for prediction accuracy and an outstanding branch queue (OBQ) is proposed to manage speculative updates of both global history and local history. With ahead pipelining, global history can be managed in the same way as proposed in [29]. However, for the update of local history, the indexes of the local history table and the OBQ need to use the n-block ahead branch address instead of the current branch address.

4.4. Experimental Results

4.4.1. Methodology

The design space of PMPM predictors is much larger than traditional predictors such as gshare branch predictors [22]. In order to reduce the design space, we place some constraints on the predictor structure: the global prediction tables have same number of entries and the number of global prediction tables is fixed to 7. Also, we use the same geometric global history lengths.
as the TAGE predictor. Based on a 32kB hardware storage budget, the configuration of our PMPM predictor is shown in Table 4.

The 1st JILP Championship Branch Prediction Competition (CBP-1) was held in 2004. In order to examine recent progress on branch predictor design, we choose 3 predictors proposed at or after CBP-1 to compare with the PMPM predictor. The first predictor is the winner (labeled as ‘AIP’ predictor) of CBP-1 [10], which is described in detail in Chapter 3. The other two predictors are the O-GEHL and the TAGE predictors. As the design space is also large for these predictors, we use the optimized configurations in their distributed code. For different predictor sizes, we just scale the prediction tables accordingly. In order to avoid history pollution, two sets of history registers are used to distinguish kernel and user branches in the CBP-2 traces as proposed in [31]. The major characteristics of those three predictors are summarized in Table 5. Prediction accuracies of these predictors and the PMPM predictor are obtained using trace-driven simulation of CBP-2 traces.

Table 4. Configuration of the 32kB PMPM predictor.

<table>
<thead>
<tr>
<th>Branch History</th>
<th>Maximum global history length: 130; Path history length: 16. Local history length: 10; 1k-entry local history table.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bimodal prediction table</td>
<td>Four prediction bits share one hysteresis bit as proposed in [33]. Number of prediction bits: $2^{11}$; number of hysteresis bits: $2^{12}$.</td>
</tr>
<tr>
<td>Global prediction tables</td>
<td>5-bit prediction counters and 2-bit uctr counters for all tables. Global history lengths of gtable0 to gtable6: 5, 9, 15, 25, 44, 75, 130. Tag widths of gtable0 to gtable6: 6, 7, 7, 8, 9, 10, 10 (bits). Each gtable has 2k entries.</td>
</tr>
<tr>
<td>Local prediction table</td>
<td>5-bit prediction counters and 2-bit uctr counters. Tag width: 5 (bits). Number of entries: 1k.</td>
</tr>
</tbody>
</table>
Table 5. The major characteristics of the AIP, the O-GEHL and the TAGE predictors.

<table>
<thead>
<tr>
<th></th>
<th>History Info</th>
<th>Predictor Structure</th>
<th>Prediction policy</th>
<th>Adaptations / Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIP</td>
<td>8 bits local history; 100 bits global history.</td>
<td>14 weight tables for the perceptron predictor with MAC representation [34], one bias branch predictor and one loop predictor.</td>
<td>Fully biased and loop branches will be predicted by the bias branch predictor and the loop predictor. Other branches are predicted by the perceptron predictor.</td>
<td>Adaptively change the input of each weight table to fit different application characteristics.</td>
</tr>
<tr>
<td>O-GEHL</td>
<td>200 bits global history; 16 bits path.</td>
<td>8 components. Geometrical history lengths are used in the index of each table.</td>
<td>Summation of 8 counters.</td>
<td>Dynamic adaptation of the training threshold and dynamic history length fitting.</td>
</tr>
<tr>
<td>TAGE</td>
<td>130 bits global history; 16 bits path.</td>
<td>One bimodal prediction table plus 7 global prediction table. Geometrical history lengths are used in the index of each table.</td>
<td>Longest match.</td>
<td>Periodically resetting usefulness counters, adaptively select between the longest match and the 2nd longest match if the longest match is a newly allocated counter, pseudo random selection on new entry allocation.</td>
</tr>
</tbody>
</table>

4.4.2. Prediction Accuracy

With the same 32kB storage budget, the misprediction rates of the AIP, the O-GEHL, the TAGE and the proposed PMPM predictors are shown in Table 6. Compared to these state-of-art branch predictors, the PMPM predictor achieves the highest accuracy for all traces except gcc and twolf. The AIP predictor is the most accurate one for twolf and the TAGE predictor is the most accurate one for gcc. Compared to the AIP and the O-GEHL predictors, the PMPM predictor reduces the average misprediction rate by 9.8% and 11.7% respectively. Compared to the TAGE predictor, the PMPM predictor has up to 54% (vortex) and an average of 5.8% lower misprediction rate. For gcc, the misprediction rate of the PMPM predictor is 8.2% higher than the TAGE predictor. Among the 20 CBP-2 traces, gcc has the largest branch footprint, which makes it most sensitive to the aliasing problem. Two reasons make the PMPM predictor potentially have more aliases than the TAGE predictor. As the PMPM predictor uses an adder tree to combine multiple prediction counters, it needs larger prediction counters to offset the
noise impact from wrong prediction counters. In order to save cost for a larger prediction counter (5-bit in the PMPM predictor compared to 3-bit in the TAGE predictor) and the local history related tables, we reduce the size of the bimodal table and the tag widths of the global prediction tables. The other reason is related to the policies used in the PMPM predictor. In order to implement the PMPM algorithm, we need to have multiple matches available. As a result, we update more prediction counters. In comparison, the TAGE predictor only updates the prediction counter with the longest match.

Table 6. Misprediction rates (MPKI) of different branch predictors with the same 32kB storage budget.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>gzip</th>
<th>gcc</th>
<th>crafy</th>
<th>compress</th>
<th>raytrace</th>
<th>javac</th>
<th>mtrt</th>
<th>eon</th>
<th>gap</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIP</td>
<td>10.346</td>
<td>4.606</td>
<td>2.713</td>
<td>6.153</td>
<td>0.755</td>
<td>1.129</td>
<td>0.900</td>
<td>0.344</td>
<td>1.994</td>
<td>0.052</td>
</tr>
<tr>
<td>O-GEHL</td>
<td>10.720</td>
<td>4.741</td>
<td>2.583</td>
<td>6.321</td>
<td>1.091</td>
<td>1.217</td>
<td>1.124</td>
<td>0.416</td>
<td>1.925</td>
<td>0.045</td>
</tr>
<tr>
<td>TAGE</td>
<td>10.867</td>
<td>3.396</td>
<td>2.529</td>
<td>5.878</td>
<td>1.064</td>
<td>1.120</td>
<td>1.101</td>
<td>0.379</td>
<td>1.674</td>
<td>0.040</td>
</tr>
<tr>
<td>PMPM</td>
<td>9.678</td>
<td>3.673</td>
<td>2.428</td>
<td>5.526</td>
<td>0.626</td>
<td>1.079</td>
<td>0.704</td>
<td>0.258</td>
<td>1.375</td>
<td>0.037</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>vpr</th>
<th>mcf</th>
<th>parser</th>
<th>jess</th>
<th>db</th>
<th>mpegaudio</th>
<th>jack</th>
<th>perl81k</th>
<th>vortex</th>
<th>twolf</th>
</tr>
</thead>
<tbody>
<tr>
<td>AIP</td>
<td>9.062</td>
<td>11.863</td>
<td>6.073</td>
<td>0.528</td>
<td>2.597</td>
<td>1.134</td>
<td>0.948</td>
<td>0.374</td>
<td>0.233</td>
<td>13.347</td>
</tr>
<tr>
<td>O-GEHL</td>
<td>9.291</td>
<td>10.922</td>
<td>6.047</td>
<td>0.494</td>
<td>2.481</td>
<td>1.218</td>
<td>0.874</td>
<td>0.554</td>
<td>0.363</td>
<td>14.317</td>
</tr>
<tr>
<td>TAGE</td>
<td>9.320</td>
<td>10.124</td>
<td>5.249</td>
<td>0.410</td>
<td>2.472</td>
<td>1.119</td>
<td>0.784</td>
<td>0.437</td>
<td>0.288</td>
<td>13.711</td>
</tr>
<tr>
<td>PMPM</td>
<td>8.859</td>
<td>10.059</td>
<td>5.193</td>
<td>0.388</td>
<td>2.307</td>
<td>1.072</td>
<td>0.687</td>
<td>0.285</td>
<td>0.132</td>
<td>13.407</td>
</tr>
</tbody>
</table>

The aliasing impact is also examined by simulating those predictors with different predictor sizes. The average misprediction rates are shown in Figure 21. From the figure, we can see that all predictors achieve higher prediction accuracy with larger storage budgets. Among these predictors, the PMPM predictor shows the best scalability while it suffers more from aliasing when the budget is limited to 8kB. With a 16kB storage budget, the aliasing impact is
significantly reduced for the PMPM predictor. With a 128kB budget, the improvement of the PMPM predictor over the TAGE predictor becomes 8%.

![Figure 21. Misprediction rates with different storage budgets.](image)

4.4.3. Ahead Pipelining

To evaluate the impact of ahead pipelining on prediction accuracy, we model the ahead pipelining scheme described in Section 4.3.5 for both TAGE and PMPM predictors. The average misprediction rates of the TAGE and PMPM predictors with 1, 2, 3 and 4-block ahead pipelining schemes are presented in Figure 22.

![Figure 22. Misprediction rates of the ahead pipelined 32kB TAGE and PMPM predictors.](image)

From the figure we can see that the prediction accuracy losses of both predictors are relatively small. For the 3-block ahead TAGE predictor, the accuracy loss is only 0.05 MPKI,
while the PMPM predictor has a larger accuracy loss of 0.11 MPKI. Although the PMPM predictor suffers a little more accuracy loss because of the high extra aliasing in the local history table in ahead pipelined predictors, it is still more accurate than the TAGE predictor even when we compare the 4-block ahead pipelined PMPM predictor with the ideal 1-block ahead TAGE predictor.

4.4.4. Global History PMPM Predictor

One advantage of the PMPM algorithm is that it enables efficient integration of different types of history information. Our PMPM predictor takes this advantage by integrating local history based predictions with the global history based predictions. However, due to the trend of using very large instruction windows, the management of speculative local history updates could be challenging. With this consideration, we remove the local prediction table and the local history table so that the resulting PMPM predictor only uses the global branch history. For the same 32kB storage budget, we increase the prediction counter to 6 bits and the tag widths of \( gtable3 \) to \( gtable6 \) are increased by 1 to use the storage saved by removing local history related tables. Table 7 shows the misprediction rates of the TAGE and the global history only PMPM (labeled as ‘PMPM (GH Only)’) predictors with the same 32kB storage budget.

From Table 7 we can see that without local history information, the improvement of the PMPM predictor over the TAGE predictor is significantly reduced, 2.2% reduction of the average MPKI. One important reason is the aliasing problem associated with the PMPM predictor with limited storage budgets. Moreover, for two important policy optimizations used in the TAGE predictor, we don’t find a good adaptation to the PMPM predictor. In the TAGE predictor, if the longest match is a newly allocated entry, the prediction policy will adaptively
choose the longest match or the 2nd longest match depending on previous knowledge. Another optimization is the pseudo-random table selection when allocating a new entry, which makes different tables have different probabilities to be allocated a new entry. In order to get a fair comparison of the longest match policy of the PPM algorithm and the PMPM algorithm with fixed number of selected counters, we simulate a 32kB TAGE predictor without those two optimizations (labeled as ‘TAGE-PPM’) and a 32kB PMPM predictor with the fixed update policy (without update policy adaptation) as used for hard-to-predict applications (labeled as ‘PMPM-FIXED’). The misprediction rates of these two predictors are shown in Table 8.

Table 7. Misprediction rates (MPKI) of the TAGE and the global history only PMPM predictors with 32kB storage budget.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>gzip</th>
<th>gcc</th>
<th>crafty</th>
<th>compress</th>
<th>raytrace</th>
<th>javac</th>
<th>mtrt</th>
<th>eon</th>
<th>gap</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAGE</td>
<td>10.867</td>
<td>3.396</td>
<td>2.529</td>
<td>5.878</td>
<td>1.064</td>
<td>1.120</td>
<td>1.101</td>
<td>0.379</td>
<td>1.674</td>
<td>0.040</td>
</tr>
<tr>
<td>PMPM (GH Only)</td>
<td>10.231</td>
<td>3.645</td>
<td>2.432</td>
<td>5.722</td>
<td>1.074</td>
<td>1.093</td>
<td>1.119</td>
<td>0.383</td>
<td>1.701</td>
<td>0.427</td>
</tr>
<tr>
<td>Benchmark</td>
<td>vpr</td>
<td>mcf</td>
<td>parser</td>
<td>jess</td>
<td>db</td>
<td>mpegaudio</td>
<td>jack</td>
<td>perl</td>
<td>vortex</td>
<td>twolf</td>
</tr>
<tr>
<td>TAGE</td>
<td>9.320</td>
<td>10.124</td>
<td>5.249</td>
<td>0.410</td>
<td>2.472</td>
<td>1.119</td>
<td>0.784</td>
<td>0.437</td>
<td>0.288</td>
<td>13.711</td>
</tr>
<tr>
<td>PMPM (GH Only)</td>
<td>8.891</td>
<td>9.973</td>
<td>5.275</td>
<td>0.421</td>
<td>2.350</td>
<td>1.089</td>
<td>0.802</td>
<td>0.451</td>
<td>0.300</td>
<td>13.379</td>
</tr>
<tr>
<td>Average</td>
<td>TAGE</td>
<td>3.598</td>
<td>PMPM (GH Only)</td>
<td>3.519</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 8. Misprediction rates (MPKI) of the TAGE-PPM and the PMPM-FIXED predictors with 32kB storage budget.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>gzip</th>
<th>gcc</th>
<th>crafty</th>
<th>compress</th>
<th>raytrace</th>
<th>javac</th>
<th>mtrt</th>
<th>eon</th>
<th>gap</th>
<th>bzip2</th>
</tr>
</thead>
<tbody>
<tr>
<td>TAGE</td>
<td>12.177</td>
<td>3.436</td>
<td>2.568</td>
<td>6.527</td>
<td>1.076</td>
<td>1.220</td>
<td>1.131</td>
<td>0.382</td>
<td>1.705</td>
<td>0.044</td>
</tr>
<tr>
<td>PMPM</td>
<td>10.199</td>
<td>4.011</td>
<td>2.462</td>
<td>5.728</td>
<td>1.076</td>
<td>1.082</td>
<td>1.120</td>
<td>0.382</td>
<td>1.746</td>
<td>0.043</td>
</tr>
<tr>
<td>Benchmark</td>
<td>vpr</td>
<td>mcf</td>
<td>parser</td>
<td>jess</td>
<td>db</td>
<td>mpegaudio</td>
<td>jack</td>
<td>perl</td>
<td>vortex</td>
<td>twolf</td>
</tr>
<tr>
<td>TAGE</td>
<td>10.542</td>
<td>11.752</td>
<td>5.393</td>
<td>0.417</td>
<td>2.614</td>
<td>1.230</td>
<td>0.799</td>
<td>0.449</td>
<td>0.295</td>
<td>14.467</td>
</tr>
<tr>
<td>PMPM</td>
<td>8.894</td>
<td>9.978</td>
<td>5.760</td>
<td>0.417</td>
<td>2.352</td>
<td>1.077</td>
<td>0.796</td>
<td>0.452</td>
<td>0.301</td>
<td>13.348</td>
</tr>
<tr>
<td>Average</td>
<td>TAGE-PPM</td>
<td>3.911</td>
<td>PMPM-FIXED</td>
<td>3.561</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This experiment confirms the benefit of the PMPM algorithm over the PPM algorithm. With the fixed policy, the reduction of average MPKI achieves 9.0% over the TAGE predictor. Also, the PMPM algorithm effectively improves the prediction accuracy of those applications with a large number of hard-to-predict branches.

4.5. **Summary**

In this chapter we present our study on the previously deemed optimal PPM algorithm for branch prediction. We show that the longest partial match used in PPM can lead to suboptimal results. To overcome such limitation, we propose to combine multiple partial matches to make more accurate predictions. We explore various ways of combining multiple matches and propose to use an adder tree to selectively combine multiple partial matches. Such combination also enables efficient integration of other types of correlation such as local branch history. Our results show that the proposed PMPM scheme significantly outperforms the PPM algorithm.

We also present a practical PMPM predictor to effectively approximate the PMPM algorithm with limited hardware cost. The PMPM borrows the predictor design of the TAGE predictor, which is a direct approximation of the PPM algorithm. With the redesigned prediction and update policies based on the PMPM algorithm and the integration of local history information, the misprediction rate of the PMPM predictor is 5.8% lower than the TAGE predictor on average (up to 54%) with a 32kB storage budget.
The performance of modern microprocessors for single-threaded applications is mainly constrained by two factors, long-latency memory accesses and hard-to-predict branches. Although large instruction-window processors [1],[9],[21],[38],[42] can effectively exploit memory-level parallelism to overcome the memory wall problem, the pressure is essentially shifted to branch predictors to fetch a large number (thousands) of instructions from correct paths. Particularly, if a program features hard-to-predict branches whose operands depend on long-latency cache-misses, those high-penalty mispredictions become serious performance obstacles and cause substantial energy to be wasted in executing instructions from wrong paths.

It has been shown in the 2nd JILP Championship Branch Prediction Competition (CBP-2) [44] that scaling the state-of-art branch correlation based branch predictor unlimitedly can only reduce the misprediction rates by about 21% compared to a 32kB realistic predictor [30]. In order to further improve prediction accuracy, we have to discover novel locality that can be explored to predict those branches that are not predicted accurately by exploring branch correlation. Pre-execution has been proposed as one solution to handle hard-to-predict branches [6],[26]. Besides the additional cost and complexity, pre-execution is less effective for large instruction window processors since it is difficult for a pre-execution thread to be far ahead of a main thread with a large instruction window.

In this chapter, we focus on hard-to-predict branches that depend on long-latency cache-missing loads and we refer to them as hard-to-predict long-latency branches. When the outcomes of such branches depend on loaded values with irregular patterns, branch prediction
schemes exploiting correlation in branch histories often fail to predict them accurately. In this chapter, we present a novel locality to handle those branches [14]. The new locality is based on the following observation: for some applications, major data structures or key data components tend to remain stable, i.e., the addresses of the key components do not change for a long time. For example, after a linked list is initialized, the address of the ending node remains the same until it is deleted or a new node is appended to it. If a branch is dependent on such stable data, e.g., the branch determining the end of a linked list traversal, the load address instead of the loaded value is sufficient to determine the branch outcome. Therefore, the branch can be resolved once the corresponding load address is known, which is much more promptly than waiting for the loaded value. Since the locality is based on the correlation between load addresses and branch outcomes, we call it address-branch correlation. In a load/branch pair that exhibits address-branch correlation, the load address is referred to as a ‘producer address’ and the branch is referred to as a ‘consumer branch’.

In this chapter, we present a study on address-branch correlation for memory-intensive applications given the importance of long-latency hard-to-predict branches in those applications. An in-depth analysis of benchmark source codes reveals common program patterns that lead to address-branch correlation. We analyze the performance potential of exploiting address-branch correlation and find that a large portion of branch misprediction penalties can be eliminated by focusing on a few (3 to 4) hard-to-predict branches that exhibit address-branch correlation. For example, 57% of the total branch misprediction penalties of *mcf* can potentially be eliminated by exploiting address-branch correlation in 4 branches. We then propose an auxiliary branch predictor, named Address-Branch Correlation (ABC) predictor, to exploit the locality. In the ABC predictor, stable address-branch correlation information is stored in a prediction table.
When a producer address is known, this prediction table is accessed to see whether the address has stable correlation with a consumer branch. If so, the branch outcome is predicted and the prediction is used as either a prioritized one when the branch has not been fetched or an overriding one when the branch has already been fetched using the prediction from the primary branch predictor. Our experimental results show that augmenting a 16kB TAGE branch predictor [35],[31] with a 9kB ABC predictor reduces the execution time by 6.3% (up to 27%) and the energy consumption by 5.2% (up to 24%) of a set of memory-intensive applications, which outperforms a 64kB TAGE branch predictor.

5.1. Address-Branch Correlation

In this section, we present an in-depth study of address-branch correlation. We focus on long-latency hard-to-predict branches, i.e., those branches that incur high performance penalties and can not be predicted accurately using existing branch predictors including perceptron branch predictors [16], [17], and recently proposed predictors based on the Prediction by Partial Matching (PPM) algorithm [23], [35]. We observe that the outcomes of many long-latency hard-to-predict branches show stable correlation with their producer load addresses. Therefore, the branch outcome can be predicted accurately using the load addresses instead of waiting for the loaded values. We use source-code examples to reveal the inherent reason why such address-branch correlation exists. We then analyze a set of memory-intensive benchmarks to examine the potential benefit. All simulation results in this section are based on the processor configuration described in Section 5.3 unless otherwise stated.
5.1.1. Motivation

We first use a microbenchmark to illustrate the correlation between producer load addresses and consumer branch outcomes. The key data structure in the microbenchmark, as shown in Figure 23, is a linked list. The linked list is initialized to contain 10 nodes. Then it is traversed 100000 times. Before each traversal, the linked list is updated by either inserting or deleting a node at a random position except the ending node. The maximum length of the linked list is restricted to 20. Although completely random behavior is rare in real applications, many hard-to-predict branches exhibit similar irregular behavior as the branches in the microbenchmark.

Figure 23. A microbenchmark to illustrate address-branch correlation.

The branch labeled ‘Branch 1’ in Figure 23 is the one of interest. It depends on the pointer-chasing load, “node = node->next” (labeled ‘Load 1’), to determine the end of the linked list. If the linked list has N nodes, the while loop will iterate N times and the branch outcomes would be (N–1) ‘Takens’ followed by one ‘Not-Taken’. Because of the random node insertion/deletion operation, neither gshare [22] nor TAGE branch predictors predict Branch 1 accurately. A 16kB gshare predictor reports a 9% misprediction rate and a 16kB TAGE predictor has a 6.5% misprediction rate. Furthermore, since Branch 1 depends on Load 1, a misprediction can not be resolved until Load 1 returns the value. If Load 1 has a high L2 cache miss rate, the branch
misprediction penalties of Branch 1 will have significant impact on the overall performance. However, as shown in the source code, the traversal always ends at the last node. As long as this node remains stable, i.e., it is not deleted and no new nodes are appended to it, the address of the ‘next’ field of the node (i.e., the load address of Load 1) is sufficient to determine the outcome of Branch 1. Assuming that the address of the ‘next’ field of the last node is X, Branch 1 would be always taken until the load address of Load 1 becomes X. From this example, we can see that strong correlation exists between the address of the producer load and the outcome of the consumer branch and the correlation is a direct result of stable data structures during program execution.

The microbenchmark in Figure 23 highlights the branch-address correlation between a load and a branch in the same loop iteration. We can also extend the scope to extract correlation between load addresses and branch outcomes across different iterations. For example, in the linked list traversal microbenchmark, if the last two nodes (i.e., Node N-1 and Node N) of the linked list are stable, the address of the ‘next’ field of node N-1 also correlates with the outcome of Branch 1 of the next iteration, as shown in Figure 24. In other words, once node N-1 is accessed, it is certain that Branch 1 will be ‘Not-Taken’ in the next iteration. Correlation across iterations is helpful when both node N-1 and node N accesses (dependent loads) are cache misses. Rather than waiting for the ‘next’ field of node N to be loaded, the address of node N-1 provides sufficient information to determine the outcome of Branch 1. We use the term ‘correlation distance’ to describe address-branch correlation across multiple iterations. Address-branch correlation with a distance of k means that the correlation exists between the load and the branch across k iterations. When the load and the branch are in the same iteration, the correlation distance is zero.
After illustrating address-branch correlation using a microbenchmark, we now examine real workloads. As our focus is on long-latency hard-to-predict branches, we select the notorious pointer-intensive benchmark *mcf* from the SPEC CINT 2000 benchmark suite as a representative workload. Using program profiling, we found that the function *refresh_potential* is invoked frequently to refresh a huge tree structure that is larger than typical L2 caches. As a result, the pointer-chasing code in this function has high L2 cache miss rates. Furthermore, the loaded values that determine the conditional branch outcomes are irregular, which makes the branches hard to predict. Overall, frequent cache misses combined with branch mispredictions make this function a critical performance bottleneck of *mcf*. Figure 25 shows the key segments of the function *refresh_potential*, in both C code (Figure 25a) and assembly code (Figure 25b).
The code segment in Figure 25b consists of two hard-to-predict branches, labeled as ‘Branch 1’ and ‘Branch 2’, respectively. Our simulation results show a 17.4% misprediction rate for Branch 2 using a 16kB TAGE branch predictor. The average misprediction penalty of Branch 2, measured as the latency between fetching the branch instruction and resolving the misprediction, is as high as 636 cycles due to dependent cache misses resulting from pointer chasing. Compared to Branch 2, Branch 1 enjoys much lower misprediction rate, 3.1% with a 16kB TAGE branch predictor. However, it still suffers from high misprediction cost, 542 cycles on average. Overall, Branch 1 and Branch 2 contribute 9.5% and 62.1% of the overall branch misprediction penalties of mcf.

Either of the two high-cost branches is dependent on a load instruction, labeled as ‘Load 1’ and ‘Load 2’ in Figure 25b. A trace containing the load addresses and branch outcomes reveals that both of the load/branch pairs exhibit strong address-branch correlation. The correlation between Load 2 and Branch 2 is similar to the linked list traversal microbenchmark.
presented in Section 5.1.1. The load/branch pair (Load 1 and Branch 1) shows a different type of code exhibiting address-branch correlation. Rather than searching for a NULL field, the branch outcome is dependent on comparing the loaded value with a constant non-zero value. By inspecting the source code shown in Figure 25a, we can see why the load/branch pair exhibits strong address-branch correlation. Although the code continuously refreshes the ‘potential’ field of many nodes, the address of each node and the ‘orientation’ field of the nodes, which affects the branch outcomes, remain stable. Therefore, once the load address is known, branch outcomes can be determined since the loaded value is not changing over time. For example, when Load 2 accesses the address 0x100FD9C4, the loaded value (a non-zero value) is used to resolve a misprediction of Branch 2. Next time when Load 2 accesses the same address, we know for sure that the branch outcome is taken as long as the value at address 0x100FD9C4 has not been updated.

Besides pointer-intensive code, we also found stable address-branch correlation in workloads using other types of data structures. One such example is large sparse matrices. As long as the matrix remains stable (i.e., non-zero entries are not reset to zero), the load address provides enough information to determine the outcome of a dependent branch.

5.1.3. Performance Potential

If a program exhibits strong address-branch correlation, the load address can be used to either accurately predict a branch or promptly override a misprediction when the branch has already been fetched. In this section, we examine the performance potential of exploiting address-branch correlation to handle long-latency hard-to-predict branches.
In our experiments, we select memory-intensive benchmarks from the SPEC2000 benchmark suite using the following criterion: the benchmarks that gain at least 40% performance improvement with an ideal L2 cache are considered memory intensive. We used the reference inputs and simulated 300M instructions for each benchmark. The simulation points are determined using the Simpoint toolset [5] except parser. In parser, key data structures are refreshed for each new input sentence. The simulation point of parser selected by Simpoint includes the processing of many different sentences, which results in stable address-branch correlation for short periods. Therefore, for parser we use a simulation point (skip the first 700M instructions) in which a single complex sentence is processed.

We focus on hard-to-predict branches, i.e., those branches that have a large number of branch mispredictions if predicted by the default 16kB TAGE predictor. For each hard-to-predict branch, we perform the following analysis to examine whether it has correlation with its producer load addresses and how much misprediction penalty can be reduced if such correlation exists. As the first step, we identify the producer loads of each hard-to-predict branch. For each dependent load/branch pair, we then generate a trace of its dynamic instances as shown in Figure 26a. For each producer load, the trace contains its load address (labeled as ‘Producer_addr’) and the cycle when the address is generated (labeled as ‘Addr_cycle’). For each consumer branch, the trace includes the cycle when the branch is fetched (labeled as ‘Fetch_cycle’), the cycle when the branch is resolved (labeled as ‘Resolve_cycle’), the prediction from the primary branch predictor (labeled as ‘Branch_pred”), and the actual branch outcome (labeled as ‘Branch_outcome”). An example of such a trace is shown in Figure 26b to illustrate the analysis process.
<table>
<thead>
<tr>
<th>Load:</th>
<th>0xA0B0</th>
<th>0xC0D0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch:</td>
<td>9950</td>
<td>14950</td>
</tr>
<tr>
<td>Load:</td>
<td>0x0C0D0</td>
<td>54950</td>
</tr>
<tr>
<td>Branch:</td>
<td>15000</td>
<td>55000</td>
</tr>
</tbody>
</table>

![Figure 26. The trace of dynamic instances of a dependent load/branch pair.](image)

Based on the trace, a Boolean array, $Addr_{Branch} \_k$, is formed to keep track of the correlation between producer addresses and consumer branch outcomes of a certain correlation distance $k$. Using the example in Figure 26b, to keep track of correlation information of distance 0 for this load/branch pair, we set $Addr_{Branch} \_0[0xA0B0] = T$ and $Addr_{Branch} \_0[0xC0D0] = NT$. To keep track of correlation information of distance 1, we set $Addr_{Branch} \_1[0xA0B0] = NT$. If a producer address correlates to a consumer branch outcome (i.e., $Addr_{Branch} \_k[Producer\_addr] = Branch\_outcome$) and the branch is mispredicted, we assume that this misprediction can be eliminated by exploiting address-branch correlation. The misprediction penalty reduction is calculated as ($Resolve\_cycle$ of the consumer branch – $Addr\_cycle$ of the producer load of correlation distance $k$) if the branch is fetched before the correlated address is available. In this case, the prediction based on the correlated address serves as an early misprediction resolution. If the branch has not been fetched when the correlated address is available, the misprediction penalty reduction is calculated as ($Resolve\_cycle$ of the consumer branch – $Fetch\_cycle$ of the consumer branch) assuming that the prediction will be
used as a prioritized one to direct the instruction fetch unit. The sum of all misprediction penalty reductions is used as the performance potential of exploiting address-branch correlation for this load/branch pair of a correlation distance k. The detailed algorithm is shown in Figure 27.

```
FROM the first TO the last dynamic branch of the trace {
    Search backwards to identify the corresponding producer load with correlation distance k;
    IF ( the producer address has not been observed) { // a new producer address
        Mark the producer address as observed;
        Mark the producer address as correlated;
        Record the current branch outcome as the address-branch correlation based (ABC) prediction for this producer address;
    }
    ELSE IF ( the producer address is marked as correlated ) { // valid ABC prediction exists
        IF ( the ABC prediction based on the producer address is correct ) {
            IF ( the primary predictor mispredicts this branch)
                Increase the estimated reduction;
        }
        ELSE { // a misprediction
            IF ( the primary predictor correctly predicts this branch)
                Reduce the estimated reduction;
                Mark the producer address as uncorrelated;
        }
    }
}
```

Figure 27. The algorithm to calculate the potential reduction in branch misprediction penalties for a load/branch pair with a correlation distance k.

For the sample trace shown in Figure 26b, we can see that exploiting the address-branch correlation information with distance 0, Addr_Branch_0[0xC0D0], can reduce the misprediction penalties by 100 cycles for each misprediction since the address 0xC0D0 is available after the mispredicted branch is fetched. In comparison, exploiting the address-branch correlation information with distance 1, Addr_Branch_1[0xA0B0], can reduce the misprediction penalties by 150 cycles for each misprediction since the address 0xA0B0 is available before the mispredicted branch is fetched. Here, it is worth to point out that the performance potential calculated using our algorithm in Figure 27 is an upper bound of misprediction penalty reduction. The reason is that the beneficial effects of wrong path instructions, especially prefetching, are not considered.
Using the algorithm in Figure 27, we calculated the performance potential of exploiting address-branch correlation for branches with high misprediction penalties. If a branch has multiple correlated producer loads, we select the one with the highest potential reduction. We also explore address-branch correlation of different distances in order to determine the optimal correlation distance for each load/branch pair. In order to show the relationship between correlation distances and potential reductions, we select one load/branch pair with the largest potential reduction from each benchmark and present four representative pairs in Figure 28. The reductions are normalized to the total misprediction penalties of the branch. The figure shows that a large portion of the misprediction penalties could be eliminated if a branch has strong address-branch correlation. The maximum reduction is typically achieved with a correlation distance larger than 0. Larger correlation distances offer higher reduction since mispredictions can be resolved more promptly. On the other hand, longer correlation distances require more stable data components, e.g., a correlation distance $k$ requires the last $(k+1)$ nodes in a linked list remain stable rather than only the last node in the list. If data structures fail to provide such high degree of stability, address-branch correlation will drop dramatically with the correlation distance, as observed from the load/branch pair from the benchmark *vpr*.

![Figure 28](image-url)  
**Figure 28.** Representative load/branch pairs to show the relationship between potential reduction and correlation distance.
After determining the optimal correlation distance for each load/branch pair, we select the top $N$ ($N=1$ to 8) pairs ranked by their potential reductions of each benchmark. The reduction in overall misprediction penalties is reported in Figure 29. The results show that among all the benchmarks, *ammp*, *art*, *mcf* and *parser* show the strongest address-branch correlation. By exploiting address-branch correlation in the top 8 branches, 45% to 57% of the overall branch misprediction penalties could be eliminated. Among the remaining benchmarks, *equake*, *twolf*, and *vpr* show reductions ranging from 11% to 17% while the benchmarks *gcc* and *swim* report very limited address-branch correlation. Another important observation from Figure 29 is that most of the performance benefits can be achieved with only a few (3 or 4) load/branch pairs. Since correlated branches and addresses need to be stored in order to exploit address-branch correlation, limiting the number of load/branch pairs will reduce the associated storage requirement.

Figure 29. Potential reductions in branch misprediction penalties by exploiting address-branch correlation for the top $N$ ($N=1$ to 8) load/branch pairs.
5.2. The Design of an Address-Branch Correlation Based Predictor

In order to explore address-branch correlation, we need to identify load/branch pairs showing strong address-branch correlation. Such identification can be done either statically with program profiling or dynamically with hardware support. In this chapter, we propose a two-step dynamic approach to capture load/branch pairs with strong address-branch correlation. In step 1, we select the hard-to-predict branches and their producer loads. In step 2, we determine whether the selected load/branch pairs have strong address-branch correlation and what correlation distances are optimal. The detailed hardware implementation of this two-step identification is presented in Section 5.2.1.

After the load/branch pairs are identified, their PCs and correlation distances are stored in a small content-addressable memory (CAM), which is used to check whether a dynamic load or branch instruction needs to access an Address-Branch Correlation based predictor (ABC predictor). Since the proposed ABC predictor is only used to handle those long-latency hard-to-predict branches, we use it as an auxiliary predictor to a primary branch predictor such as a gshare or TAGE predictor.

In our proposed ABC predictor, the prediction process starts with a producer load address, which is computed in the Address Generation (AGEN) stage of a producer load instruction. A prediction table, which keeps track of address-branch correlation information, is then accessed with the producer address. If this address correlates to a consumer branch outcome, the prediction table returns a prediction. Based on whether the consumer branch has been fetched, the prediction can be used as either an overriding prediction to resolve a misprediction or a prioritized one to direct the instruction fetch unit. The prediction table is
updated when a consumer branch commits. The detailed design of the prediction table is presented in Section 5.2.2.

Unlike traditional branch predictors, the proposed ABC predictor involves two correlated instructions, a producer load and a consumer branch. As a result, it is necessary to link the dynamic instances of the load/branch pair in order to determine the producer/consumer relationship. In the proposed design, a FIFO structure, named Address-Branch Correlation Queue (ABCQ), is used for this purpose and the associated operations to link load/branch pairs are discussed in Section 5.2.3.

5.2.1. Capturing Load/Branch Pairs with Strong Address-branch Correlation

To capture load/branch pairs with strong address-branch correlation, a two-step approach is used and the associated hardware implementation is shown in Figure 30.

![Figure 30. Hardware structures to capture load/branch pairs with strong address-branch correlation.](image)

The first step is to capture hard-to-predict branches and to find their producer loads. To do so, a Hard-to-predict Branch Tracking Table (HBTT) is used to keep track of branches with
high misprediction penalties. The HBTT is organized as a 16-entry 4-way cache structure. As shown in Figure 30, each entry has two fields: the branch PC (BrPC) and its accumulative misprediction penalties (MisPen). This table is accessed when a mispredicted branch is retired. If the branch hits in the table, its misprediction penalty is added into the MisPen field. If the branch misses in the table, a new entry is allocated. The misprediction penalty is measured in the following way. Besides the renaming table, each shadow map saves the timestamp when a branch is dispatched. When the branch is resolved, the difference between the current timestamp and the one stored in the shadow map is the latency to resolve this branch. This latency is carried with the branch and used at the retire stage to update the HBTT. After training the HBTT for 2M instructions, a branch is determined hard-to-predict if it has an accumulative misprediction penalty larger than 10000 cycles. If there is at least one such branch, we stop training the HBTT and select 5 hard-to-predict branches. Otherwise, we clear the HBTT and train it for another 2M instructions. We performed experiments with varying the training period and found that training 2M instructions provides a good balance between the quickness in capturing hard-to-predict branches and the accuracy to capture the most important ones. The PCs of the selected hard-to-predict branches are copied into the ABC Information Table (AIT) and we start to identify the producer load for each branch in the AIT using a structure named Producer Loads Register File (PLRF) in Figure 30. The PLRF tracks the dynamic data dependency among instructions. It is indexed by the logical register number and the content of each entry is the PC of the load instruction that affects the corresponding register, either directly or indirectly. The PLRF works as follows. When an instruction is retired, it updates the PLRF indexed by its destination register. If it is a load, it updates the PLRF entry with its PC. If the instruction is not a load but has a destination register, the PCs stored in the PLRF entries corresponding to its source registers are
read out. Then the PC, which is closer to the current instruction’s PC, is selected to update the PLRF. For example, if an add instruction with the PC 0x4000ABC8 retires, the PLRF entries corresponding to its two source registers contain 0x4000ABC0 and 0x4000ABB0. 0x4000ABC0 will be used to update the PLRF as it is closer to 0x4000ABC8 than 0x4000ABB0. If a register-writing instruction has no source registers (or using r0, the constant zero), we reset the entry in the PLRF entry corresponding to its destination register. This way, when a branch that hits in the AIT is retired, it can access the PLRF with its source registers to determine its producer load.

The second step is to determine whether the selected load/branch pairs have strong address-branch correlation and which correlation distances are optimal. To do so, for each selected load/branch pair in AIT, we keep track of the benefit by exploiting address-branch correlation with three pre-selected correlation distances (1, 3 and 5). We select those three correlation distances since most optimal distances are from those three or very close to one of them. Each distance is utilized by the ABC predictor (see Sections 5.2.2 and 5.2.3) for at least 512 hits in the predictor and at least 2M instructions. In this period, we don’t actually use the ABC prediction to substitute or override the primary predictor’s prediction. We only use the prediction to estimate how much misprediction penalty could be saved. At the retire stage of a selected branch, if the ABC prediction is correct and the primary predictor is wrong, the Save field of this branch is incremented by the misprediction penalty. On the other hand, if the ABC prediction is wrong and the primary predictor is correct, we decrement the Save field. After the benefits of all three correlation distances are estimated, we compare the Save fields and decide which correlation distance is best for a load/branch pair. The PCs and correlation distances of those load/branch pairs remain in the AIT, which are used to check whether a dynamic load or branch instruction needs to access the ABC predictor.
Depending on the density of hard-to-predict branches, the overall training period ranges from 8M instructions (mcf, parser, twolf) to 148M instructions (equake). The length of the overall training period and the number of load/branch pairs captured for each benchmark are reported in Table 9.

Table 9. Length of the training period and the number of selected load/branch pairs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Training (M Insts.)</th>
<th># of selected pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>13</td>
<td>1</td>
</tr>
<tr>
<td>art</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>equake</td>
<td>148</td>
<td>1</td>
</tr>
<tr>
<td>mcf</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>parser</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td>twolf</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>vpr</td>
<td>9</td>
<td>3</td>
</tr>
</tbody>
</table>

5.2.2. Tracking Address-Branch Correlation with a Prediction Table

We use a prediction table to keep track of address-branch correlation information. The prediction table is organized as a 2-way set-associative structure. Each entry of the prediction table contains three fields, a partial ‘Tag’ containing several bits of a producer address; a 1-bit ‘Correlated’ flag indicating whether the address has correlation to the branch outcome, and a 1-bit ‘Pred’ field keeping the outcome of a consumer branch, as shown in Figure 31.

The prediction table is accessed when a producer address is available. If the producer address hits in the table and the ‘Correlated’ flag is true, a valid prediction is generated using the ‘Pred’ field.

![Figure 31. An Address-Branch Correlation Prediction Table.](image-url)
When a consumer branch is retired, the prediction table is updated with the producer address (obtained from the ABCQ described in Section 5.2.3), the actual branch outcome, and whether the branch is mispredicted by the primary predictor. If the producer address misses in the prediction table and the primary predictor has a misprediction, a new entry is allocated in the prediction table, the ‘Pred’ field is set to the branch’s actual outcome, and the ‘Correlated’ flag is set to true. If the address hits in the prediction table, the ‘Correlated’ flag is set to false when the actual branch outcome doesn’t match with the ‘Pred’ field, indicating that there is no stable correlation with this address.

When a new entry is required at a set where no empty entry is available, an existing entry is replaced only if its ‘Correlated’ flag is false. The reason is that we do not want to replace useful correlation information. To overcome the problem of out-dated correlation, which means that some addresses in the prediction table will not be accessed anymore and will not be replaced, we resort to resetting the table periodically at an interval of 100M instructions. Since long-latency mispredictions incur much higher branch penalties than short-latency ones, among the first 2M executed instructions after each reset, only long-latency mispredictions are allocated a new entry in the prediction table. Short-latency mispredictions compete for the rest of the entries afterwards. We treat a misprediction with more than 100-cycle resolution latency as a long-latency misprediction.

5.2.3. Linking Producer Loads and Consumer Branches

In the proposed design, we use address-branch correlation queues (ABCQs) to link dynamic producer loads and their consumer branches. Each selected load/branch pair has its own ABCQ. As discussed in Section 5.1.3, a few (3 or 4) load/branch pairs are sufficient to reap most
of the performance benefits. Therefore, only a few ABCQs are needed in our design. Each ABCQ maintains information of dynamic instances of a particular branch that has been fetched but has not yet been committed. Each ABCQ entry contains four fields: Prediction Available (PredAvail), Prediction (Pred), Address Available (AddrAvail), and Address (Addr). The first two fields indicate whether a valid ABC prediction is available and what the prediction is, whereas the next two fields show whether a valid producer address is available and what the address is. The Pred field is also used to store the primary predictor’s prediction if an ABC prediction is not available when a consumer branch is fetched. The queue is managed by three pointers: ‘Head’, which points to the dynamic instance of the consumer branch that will retire next; ‘Dispatch’, which points to the dynamic instance of the consumer branch that will be dispatched next; and ‘Fetch’, which points to the dynamic instance of the consumer branch that will be fetched next. Those pointers are updated when the corresponding dynamic branch is retired, dispatched and fetched, respectively. The reorder buffer (ROB) is also augmented with an ‘ABCQ_ptr’ field, which provides a link to the corresponding ABCQ entry. The queue structure and its relationship with ROB are shown in Figure 32.

Figure 32. Linking producer loads and consumer branches using an address-branch correlation queue (ABCQ).
When a producer load of interest is dispatched, the current Dispatch pointer of the corresponding ABCQ is stored in the $ABCQ\_ptr$ field of its ROB entry. When its consumer branch is dispatched, the same pointer is stored in its $ABCQ\_ptr$ field, as shown in Figure 32. Since the ABCQ maintains the branch’s dynamic instances in the program order, it provides an efficient way to link producer loads and consumer branches across multiple iterations. For example, a load can reach its correlated branch with correlation distance 1 using $(ABCQ\_ptr + 1) \% ABCQ\_size$, as illustrated in Figure 32.

When the address of a producer load is known, it is used to access a prediction table to retrieve an ABC prediction. The prediction as well as the address will be used to update the ABCQ entry through the $ABCQ\_ptr$. In other words, the entry $ABCQ[(ABCQ\_ptr + k) \% ABCQ\_size]$ is updated if address-branch correlation of distance $k$ is to be exploited for this load instruction. Since the $Pred$ field may also be used to store the primary predictor’s prediction, we need to check the current ‘Fetch’ pointer of the ABCQ before updating the field with the ABC prediction. The ‘Fetch’ pointer of the ABCQ shows whether the dynamic instance of the consumer branch has been fetched or not. If it has been fetched, the fetch pointer will exceed $(ABCQ\_ptr + k)$ and the $Pred$ field maintains the existing prediction generated by the primary predictor. In this case, the $Pred$ field is compared to the ABC prediction to decide whether to override the existing prediction and redirect the fetch unit. Then, the ABC prediction will be stored in the $Pred$ field. If the dynamic instance has not been fetched, the $PredAvail$ flag will be set to true and the ABC prediction will be used as a prioritized prediction when it is fetched, i.e., when the ‘Fetch’ pointer equals to $(ABCQ\_ptr + k)$. 

78
When a consumer branch is ready to commit, its actual outcome as well as the information maintained in its assigned ABCQ entry, pointed through its $ABCQ_{ptr}$, will be used to update the prediction table. After the branch is committed, its ABCQ entry is de-allocated.

Since an ABCQ allocates an entry for each fetched instance of a particular consumer branch, some of those dynamic instances may be on wrong paths. In order to recover the ABCQs in case of branch mispredictions, their Dispatch and Fetch pointers are checkpointed along with the rename map table when a branch is dispatched. When a branch misprediction is detected, the checkpointed pointers are used to recover the ABCQ’s state.

5.2.4. Hardware Cost

The storage requirement of our design includes two parts: the hardware structures to capture load/branch pairs and the ABC predictor. The HBTT has 16 entries and each entry includes a 32-bit PC and a 24-bit saturating counter to store the misprediction penalty. The AIT has 5 entries and each entry includes two 32-bit PCs, a 3-bit counter to store correlation distance and three 21-bit saturating counters. The PLRF has 64 entries and each entry stores a 32-bit PC. The overall storage requirement of those three tables is 3594 bits (449 bytes).

The storage requirement of the proposed ABC predictor depends on the sizes of the prediction table and ABCQs. In our design, we use a 7-bit partial tag. Therefore, each entry in the prediction table has 9 bits, including the correlation and prediction flags. A larger prediction table can capture more correlated addresses, but results in larger hardware cost and higher access latency. We experimentally found that a 9kB 2-way associative table achieves good balance between performance and hardware cost (See Section 5.4.1).
Each ABCQ entry has three 1-bit fields and an address field. Since the address field is used to access the prediction table, only the index and the partial tag need to be stored. For a 9kB 2-way prediction table, the index needs 12 bits and the partial tag has 7 bits. Therefore, each entry of the ABCQs has 22 bits. We use 64-entry ABCQs since our results show that the performance with 64-entry ABCQs is very close to the performance of queues with infinite number of entries. The reason is that the selected branches are hard-to-predict. If a large number of these branches have been fetched, it is very likely that the front end is already on a wrong path and there is no performance loss to stall the front end. Considering the requirement of training 5 load/branch pairs as described in Section 5.2.1, we use 5 ABCQs and the total cost of the queues is 7040 bits (880 bytes).

5.3. Simulation Methodology

Our simulator infrastructure is built upon the SimpleScalar toolset [5] but our execution-driven timing simulator is completely rebuilt to model MIPS R10000-style out-of-order superscalar processors. The processor configuration is shown in Table 10. We model a large instruction window processor to highlight the performance impact of branch mispredictions since the memory wall problem can be resolved effectively with large instruction windows. The primary branch predictor is a 16kB TAGE predictor which uses 130-bit global branch histories with ideal 1-block ahead configuration. The performance of the proposed ABC predictor is evaluated by using SPEC 2000 benchmarks with the reference inputs and the same selection criterion and simulation points as described in Section 5.1.3. The dynamic load/branch pairs selecting period is also included in the total simulation range of 300M instructions. We exclude
swim and gcc because their estimated branch misprediction penalty reductions are too small (less than 2%) as shown in Figure 29.

Table 10. Configuration of the processor.

<table>
<thead>
<tr>
<th>Configuration</th>
<th>Details</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Cache</td>
<td>32 kB, 2-way; Line size=16 Inst.; Miss penalty=10 cycles.</td>
</tr>
<tr>
<td>Data Cache</td>
<td>32 kB, 2-way; Line size = 64 bytes; Miss penalty = 10 cycles.</td>
</tr>
<tr>
<td>Unified L2 Cache</td>
<td>1024 kB, 8-way; Line size=128 bytes; Miss penalty=300 cycles.</td>
</tr>
<tr>
<td>Primary Branch Predictor</td>
<td>16kB TAGE: 8 prediction tables. Min branch misprediction penalty = 20 cycles.</td>
</tr>
<tr>
<td>Superscalar Core</td>
<td>Reorder buffer: 1024 entries; Dispatch/issue/retire bandwidth: 4-way superscalar; 4 fully-symmetric function units; Data cache ports: 4. Issue queue: 512 entries. LSQ: 512 entries. Rename map table checkpoints: 256.</td>
</tr>
<tr>
<td>Execution Latencies</td>
<td>Addr. Gen.: 1 cycle; Mem. access: 2 cycles (hit in data cache); Int. ALU ops = 1 cycle; Complex ops = MIPS R10000 latencies.</td>
</tr>
<tr>
<td>Memory Disambiguation</td>
<td>Perfect memory disambiguation.</td>
</tr>
<tr>
<td>Hardware Prefetcher</td>
<td>Stride-based stream buffer: 8 four-entry stream buffers with a PC-based two-way 512-entry stride prediction table.</td>
</tr>
</tbody>
</table>

5.4. Experimental Results

5.4.1. Performance

In this experiment, we examine the performance of our proposed ABC predictor. We augment the baseline processor with an ABC predictor and vary the size of its prediction table from 4.5kB to 72kB. The normalized execution time with respect to the baseline processor is reported in Figure 33. For reference, we also include the results of using ideal predictions for those selected branches.
Figure 33. Normalized execution time of the baseline processor augmented with ABC predictors.

Two observations can be made from Figure 33. First, among the memory-intensive benchmarks that we examined, the ABC predictors always result in positive performance improvements and higher performance improvements are achieved in integer benchmarks than floating-point benchmarks, which is expected as even ideal prediction has limited performance impact in floating-point benchmarks. Among the integer benchmarks, \textit{mcf} and \textit{parser} show strong address-branch correlation which confirms the performance potential study shown in Figure 29. Second, although larger prediction tables can capture more address-branch correlation information and result in better performance, a small 4.5kB prediction table is sufficient to keep track of the majority of useful producer addresses. An exception is the benchmark \textit{mcf}, for which increasing the table size has significant performance impact, the execution time is reduced by 15\% with a 4.5kB table and by 41\% with a 72kB table. The reason is that its large working set and pointer-chasing code result in a large number of producer addresses that correlate to consumer branches. On average, the ABC predictor reduces the execution time by 4.1\% with a 4.5kB prediction table and 9.5\% with a 72kB table. Among the different prediction table sizes, a 9kB prediction table reduces the execution time by 6.3\% (up to 27\%), which provides a good tradeoff between performance improvement and hardware cost.
To evaluate the effectiveness of our dynamic learning scheme, we also performed profiling to select the most correlated load/branch pairs and their optimal correlation distances. We use the profiling process described in Section 5.1.3 with reference inputs to select one to five load/branch pairs for each benchmark. The execution phases of the profiling and test runs are same to get the maximum performance of profiling. For each selected load/branch pair, we record the load PC, the branch PC and the optimal correlation distance. The information is loaded at the beginning of the execution phase into AIT to identify those selected load/branch pairs. Figure 34 shows the performance of dynamic learning and the profiling schemes with a 9kB ABC predictor. From the figure, we can see that the profiling scheme achieves maximum overall performance with 3 profiled load/branch pairs, which is only 0.7% higher over the proposed dynamic learning scheme.

![Figure 34. Normalized execution time of the dynamic learning and profiling schemes with a 9kB ABC predictor.](image)

5.4.2. Prediction Accuracy and the Reduction in Branch Misprediction Penalties

In this experiment, we examine the prediction accuracy achieved by the proposed ABC predictors and what fraction of the branch misprediction penalties can be eliminated. For those selected branches a 9kB ABC predictor achieves 96.8% prediction accuracy, which is much
higher than a 16kB TAGE predictor (87.8%) and the idealistic GTL predictor [30] with nearly unlimited storage budget (91.8%). However, since the ABC predictor only provides a prediction when a producer load address is highly correlated with the branch outcome, it does not cover all dynamic instances of those selected branches. For a fair comparison, we report in Table 11 the number of mispredictions of those selected branches using a 16kB TAGE predictor with and without a 9kB ABC predictor. The ABC predictor reduces the mispredictions of those selected hard-to-predict branches by 37.7% on average.

Table 11. The number of mispredictions for the selected hard-to-predict branches.

<table>
<thead>
<tr>
<th></th>
<th>ammp</th>
<th>art</th>
<th>equake</th>
<th>mcf</th>
<th>parser</th>
<th>twolf</th>
<th>vpr</th>
<th>amean</th>
</tr>
</thead>
<tbody>
<tr>
<td>16kB TAGE</td>
<td>58005</td>
<td>257476</td>
<td>197533</td>
<td>2842260</td>
<td>478172</td>
<td>577130</td>
<td>522262</td>
<td>704691</td>
</tr>
<tr>
<td>16kB TAGE + 9kB ABC</td>
<td>24388</td>
<td>117594</td>
<td>191520</td>
<td>1159779</td>
<td>179864</td>
<td>458304</td>
<td>489888</td>
<td>374476</td>
</tr>
<tr>
<td>Misprediction Reduction</td>
<td>58.0%</td>
<td>54.3%</td>
<td>3.0%</td>
<td>59.2%</td>
<td>62.4%</td>
<td>20.6%</td>
<td>6.2%</td>
<td>37.7%</td>
</tr>
</tbody>
</table>

Next, we examine the reduction in overall misprediction penalties achieved by the ABC predictor. Here, we use the interval between the time when a mispredicted branch is fetched and the time when the misprediction is resolved as the misprediction penalty. The reductions are normalized to the misprediction penalty of the baseline processor with a 16kB TAGE predictor and are shown in Figure 35. We can see that although the ABC predictor only provides predictions for one to four (static) selected branches, the proposed ABC predictors successfully reduce overall misprediction penalties and a 9kB ABC predictor can achieve up to 29% and an average of 11% reduction.
Figure 35. Reductions in branch misprediction penalties achieved by ABC predictors.

5.4.3. Impact of Primary Branch Predictors

In this experiment, we vary the primary predictor to evaluate its impact on the proposed ABC predictor. We simulate gshare predictors, which have lower prediction accuracy but are relatively easy to implement, and large TAGE predictors, which provide state-of-art prediction accuracy but have higher implementation complexity. The execution time of different branch predictors, either used alone or combined with a 4.5kB or a 9kB ABC predictor, is shown in Figure 36. Here, the execution time is normalized to the baseline processor with a 16kB gshare predictor.

From the figure, we can see that the ABC predictor can achieve significant performance improvement even if it is used together with a highly accurate primary predictor. When augmenting a 128kB TAGE predictor with a 9kB ABC predictor, the execution time is reduced by 4.2% (up to 15%). For a less accurate branch predictor, such as a 16kB gshare predictor, the ABC predictor is also effective and reduces the execution time by up to 22% and 5.6% on average. Another observation from the figure is that an 8kB TAGE predictor with a 4.5kB ABC predictor outperforms a 32kB TAGE predictor and a 16kB TAGE predictor with a 9kB ABC predictor.
predictor outperforms a 64kB TAGE predictor for those benchmarks. The reason is that the ABC predictor achieves high prediction accuracy for a few long-latency branches while the larger TAGE predictor aims to improve prediction accuracy universally for both long-latency and short-latency branches. Focusing on long-latency branches is more effective in reducing the overall execution time.

Figure 36. Normalized execution time of different primary predictors with and without ABC predictors.

As we discussed in Section 5.1, for branch prediction purpose, the address-branch correlation is fundamentally different from the traditional branch-correlation. Therefore, an ABC predictor should be able to provide higher prediction accuracy even when it works with a very accurate branch-correlation based predictor. In order to show the ability of the proposed ABC predictor to capture the novel address-branch correlation where a traditional branch predictor will fail, we augment an idealistic GTL predictor [30] with ABC predictors. The GTL predictor was the winner of the idealistic track of CBP-2 [44]. It is a hybrid predictor combining a GEHL predictor [28], a TAGE predictor [35], and a loop predictor. The complexity of the predictor is only constrained by two factors. First, the predictor simulator has to be able to run on a machine with 1GB memory. Second, the total simulation time of the CBP-2 traces has to be less than two hours on a 3.06GHz Pentium 4-based computer. The normalized execution time of augmenting the GTL predictor with a 9kB ABC predictor or a 22MB ABC predictor with respect
to the baseline processor with a GTL predictor is reported in Figure 37. As we can see from the figure, the ABC predictor successfully improves the performance by 4.5% (with a 9kB ABC predictor) and 6.6% (with a 22MB ABC predictor) over the idealistic GTL predictor. For mcf, a 22MB ABC predictor achieves 30% performance improvement. The effectiveness of the ABC predictor when working with a GTL predictor confirms our argument that the address-branch correlation is a novel correlation.

![Figure 37. Normalized execution time of GTL predictor with ABC predictors.](image)

5.4.4. Sensitivity Study

In this experiment, we study the sensitivity of the proposed ABC predictor on various architecture parameters. We use the baseline configuration shown in Table 10 and vary the ROB size, the L2 cache size and the memory latency. The performance improvements of a 9kB ABC predictor calculated by the reduction of the geometric mean of the normalized execution time are shown in Figure 38.
As shown in Figure 38a, the ABC predictor provides larger benefit to processors with larger instruction windows. The reason is that a large instruction-window processor can effectively exploit memory-level parallelism to overcome the memory wall problem. Therefore, the pressure is essentially shifted to branch predictors to fetch a large number of instructions from correct paths. The percentage of the total misprediction penalty to the execution time increases from 24% to 30% when we increase the ROB size from 128 to 1024. For the impact of the cache and memory system as shown in Figure 38b and Figure 38c, we can observe that the
ABC predictor provides similar performance benefits (5.5% to 6.5%) with different L2 sizes and memory latencies. Generally, with a larger L2 size or smaller memory latency, the performance improvement is higher since the performance bottleneck is shifted from the latency of memory operations to the branch prediction accuracy. An exception is the mcf with a processor having a 4MB L2 cache. Compared to a 2MB L2 cache, the L2 cache miss rate of the 4MB L2 cache drops dramatically for mcf (from 46% to 26%). Such drop reduces the relative importance of the load dependent branches predicted by the ABC predictor and the performance improvement of mcf is reduced from 26% to 19%.

5.4.5. Reduction in Energy Consumption

Since long-latency branch mispredictions lead to a large number of wrong-path instructions being executed, a reduction in mispredictions reduces the energy being wasted by wrong-path instructions. To analyze the energy consumption effect of the ABC predictor, we port WATTCH [4] and HotLeakage [41] into our simulator to account for both dynamic and static energy consumption. In our experiments, we use the 70nm technology with a clock frequency of 5.6GHz and assume the linear clock gating [4]. The energy consumed by the ABC predictor is taken into account by modeling the prediction table and ABCQs. Since the ABC predictor is only accessed by four selected load/branch pairs, its energy consumption is very small and accounts for less than 0.6% of the total energy consumption of the processor. The overall energy consumption of a processor with a 9kB ABC predictor normalized to the baseline processor is shown in Figure 39. From the figure we can see that the ABC predictor reduces energy consumption by up to 24% in mcf and 5.2% on average. For equake, the energy consumption is increased by 0.4% because the extra energy consumed by the ABC predictor is
larger than its contribution to energy reduction. The energy reductions mainly come from the reduced execution time and the reduced number of instructions fetched and executed by the processor. Our results show that the ABC predictor reduces the average number of fetched instructions by 6.5% and the average number of executed instruction by 2.8%.

![Figure 39. Reduction in energy consumption achieved by a 9kB ABC predictor.](image)

5.5. Limitations and Potential Optimizations

The proposed approach is a specialized approach aiming to exploit the address-branch locality. Our results show that it is effective for benchmarks with heavy pointer chasing since branches with strong address-branch correlation in those benchmarks normally contribute to a large portion of the total mispredictions. However, if the relative importance of those branches is low, there is limited performance benefit. For those benchmarks, for which exploring address-branch correlation is not effective, we can turn off the ABC predictor or reuse the prediction table to predict other branches. Since each entry of the prediction table has a partial tag and two 1 bit fields, we may reuse the table as a tagged branch prediction table working with the primary branch predictor to improve prediction accuracy of all conditional branches.
5.6. Summary

In this chapter we present a novel locality named address-branch correlation (ABC) that can be exploited to handle long-latency hard-to-predict branches. A detailed study of address-branch correlation reveals why stable address-branch correlation exists. The reason is that in many memory-intensive workloads, major data structures or key data components that affect branch outcomes tend to remain stable. If a hard-to-predict branch depends on such stable data, the address of the data rather than the data value is sufficient to determine the branch outcome. Since load addresses are obtained much earlier than loaded values, a misprediction can be detected more promptly especially if the loads result in long-latency cache misses.

We then propose a design to exploit address-branch correlation to reduce misprediction penalties of those hard-to-predict long-latency branches. The proposed predictor dynamically captures correlations between producer load addresses and consumer branch outcomes. Address-branch correlation based predictions are generated when a producer address is known and the prediction is used as either an overriding prediction if the branch has been fetched or a prioritized one if the branch has not been fetched. Our experimental results show that an ABC predictor achieves very high prediction accuracy (96.8%) for those hard-to-predict branches. With a 9kB prediction table, the proposed ABC predictor reduces execution time by 6.3% on average (up to 27%) and also reduces energy consumption by 5.2% on average (up to 24%) for a set of SPEC 2000 benchmarks.
In this chapter we review the contributions of this dissertation and discuss future works related to methods proposed in this dissertation.

6.1. Contributions

As modern processors are designed with deeper pipelines and larger instruction windows, branch prediction accuracy is becoming more critical for both performance and energy efficiency. In this dissertation, we proposed several methods to improve the prediction accuracy of conditional branches.

For traditional branch correlation based branch prediction schemes, we focused on the information processor and the prediction algorithm. We proposed two methods to improve them:

1. An adaptive information processing scheme is proposed to dynamically learn the importance of different input information of a branch predictor and feed the predictor with the most useful information. We showed that using adaptive information processing can significantly improve the prediction accuracy of perceptron predictors.

2. We proposed the Prediction by combining Multiple Partial Matches (PMPM) algorithm, which shows higher prediction accuracy than previously deemed optimal PPM algorithm. In PMPM algorithm, we proposed to selectively combine multiple partial matches instead of using the longest match. We designed both an idealistic PMPM predictor and a realistic PMPM predictor to demonstrate the capability of the
PMPM algorithm. Experimental results show that the idealistic PMPM predictor is more accurate than previously proposed GTL idealistic predictor. The realistic PMPM predictor also achieves higher prediction accuracy than other state-of-art branch predictors.

Besides the study of branch correlation based predictors, we also discovered a novel correlation, named as address-branch correlation. Since long-latency hard-to-predict branches are critical performance bottlenecks of large instruction window processors, we studied this type of branches in detail and found that address-branch correlation could be efficiently explored to predict those branches. With address-branch correlation, we could predict the consumer branch outcome using the producer load address, which is available much more promptly than the loaded value. We demonstrated that there are many branches showing strong address-branch correlation. We designed a practical predictor to explore address-branch correlation and showed that it can accurately predict those hard-to-predict branches even when working with an idealistic branch correlation based predictor.

6.2. Future Works

Although it has become more and more difficult to further improve the prediction accuracy of branch correlation based branch predictors, they are still very important due to their relatively low implementation complexity. In Chapter 3, we demonstrate that appropriate inputs adaptation can significantly improve the prediction accuracy of perceptron predictors. One promising research work is to learn how to dynamically select the most useful input information and adjust the inputs accordingly for PPM/PMPM based predictors, which have higher prediction accuracy than perceptron predictors and are easier to implement in hardware.
As shown in this dissertation, exploring address-branch correlation is very effective to handle long-latency hard-to-predict branches. However, our design of the address-branch correlation based predictor is limited to capture only long term stable correlation. In our study, we observed that there exist many load/branch pairs with address-branch correlation stable for short terms. Further research work needs to be done to capture these short term correlations. Moreover, our design of the address-branch correlation based predictor requires several extra hardware structures that are only useful for address-branch correlation. A design to integrate the address-branch correlation prediction table with the traditional pattern history table would be more resource efficient.


The official CBP-1 website: http://www.jilp.org/cbp/.

The official CBP-2 website: http://camino.rutgers.edu/CBP-2/.