Branch Predictor Design for Low Resource Architectures

Thesis Submitted By

Moumita Das

Doctor of Philosophy (Engineering)

Department of Information Technology
Faculty Council of Engineering and Technology
Jadavpur University
Kolkata-700032, India
2019

Branch Predictor Design for Low Resource Architectures

Thesis Submitted By

Moumita Das

DOCTOR OF PHILOSOPHY (Engineering)

Under the supervision of

Dr. Ansuman Banerjee
Advanced Computing and Microelectronics Unit
Indian Statistical Institute, Kolkata-700108

and

Dr. Bhaskar Sardar
Department of Information Technology
Jadavpur University, Kolkata-700106

Index No.: 244/16/E

1. **Title of the Thesis:** Branch Predictor Design for Low Resource Architectures

2. **Name, Designation and Institution of the Supervisor/s:**

   (a) **Dr. Ansuman Banerjee**
   Advanced Computing and Microelectronics Unit
   Indian Statistical Institute
   Kolkata-700108

   (b) **Dr. Bhaskar Sardar**
   Department of Information Technology
   Jadavpur University
   Kolkata-700106

3. **List of Publications / communications out of this work:**

   (a) **Journals**

      i. **Moumita Das**, A. Banerjee, and B. Sardar, A framework for evaluating branch predictors using multiple performance parameters, Accepted for publication in Inderscience International Journal of High Performance Systems Architecture (IJHPSA).

      ii. **Moumita Das**, M. Chaudhuri, A. Banerjee, and B. Sardar, Enabling Shared Pattern History Tables in Multi-component Branch Predictors with a Tiny Dealiasing Cache, Communicated to IEEE Embedded System Letters (ESL).

      iii. B. Nongpoh, R. Ray, **Moumita Das**, and A. Banerjee, Enhancing Speculative Execution with Selective Approximate Computing, In ACM Transactions on Design Automation of Electronic Systems (TODAES), 2018, Vol. 26, pp. 1-29.

   (b) **Conferences**

      i. **Moumita Das**, A. Banerjee, and B. Sardar, A Shared BTB Design for Multicore Systems, IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2019, Washington, DC, USA, pp. 267-268.

      ii. **Moumita Das**, A. Banerjee, and B. Sardar, A Multi-component Branch Predictor Design for Low Resource Budget Processors, International workshop on Pioneering Processor Paradigms (WP3) 2018, Vienna.

      iii. **Moumita Das**, A. Banerjee, N.K. Singh, and B. Sardar, Performance attacks on branch predictors in embedded processors with SMT support, International Conference on Information Systems Security (ICISS), 2017, pp. 313-322.

      iv. **Moumita Das**, A. Banerjee, and B. Sardar, An empirical study on performance of branch predictors with varying storage budgets, 7th Intl Symp. on Embedded computing system Design, ISED, 2017, pp. 1-5.

v. **Moumita Das**, A. Banerjee, and B. Sardar, A framework for branch predictor selection with aggregation on multiple parameters, International Symposium on VLSI Design and Test (VDAT), 2017, pp. 69-74.

vi. **Moumita Das**, and A. Banerjee, Improving Energy Efficiency of Mobile Execution Exploiting Similarity of Application Control Flow, International Conference on Advances in Mobile Computing and Multimedia (MoMM), 2016, pp. 212-216.

vii. **Moumita Das**, A. Kar, A. Banerjee, and B. Sardar, Attacks on Hybrid Branch Predictors: An Empirical Exploration, International Conference on Computing, Communication and Sensor Network (CCSN), 2016.

viii. **Moumita Das**, B. Sardar, and A. Banerjee, Attacks on Branch Predictors: An Empirical Exploration., International Conference on Information Systems Security (ICISS), 2015, pp. 511-520.

ix. S. Dutta, **Moumita Das**, and A. Banerjee, Enhancing Branch Prediction using Software Evolution, IEEE international conference on Networking, Architecture, and Storage (NAS), 2015, pp. 295-304.

4. **List of Patents:** None

5. **List of Presentations in National / International Conference:**

   (a) **Moumita Das**, A. Banerjee, and B. Sardar, A Shared BTB Design for Multicore Systems, IEEE/ACM International Symposium on Code Generation and Optimization (CGO) 2019, Washington, DC, USA, pp. 267-268.

   (b) **Moumita Das**, A. Banerjee, N.K. Singh, and B. Sardar, Performance attacks on branch predictors in embedded processors with SMT support, International Conference on Information Systems Security (ICISS), 2017, pp. 313-322.

   (c) **Moumita Das**, A. Banerjee, and B. Sardar, An empirical study on performance of branch predictors with varying storage budgets, 7th Intl Symp. on Embedded computing system Design, ISED, 2017, pp. 1-5.

   (d) **Moumita Das**, B. Sardar, and A. Banerjee, Attacks on Branch Predictors: An Empirical Exploration., International Conference on Information Systems Security (ICISS), 2015, pp. 511-520.

CERTIFICATE FROM THE SUPERVISORS

This is to certify that the thesis entitled **Branch Predictor Design for Low Resource Architectures** submitted by Smt. Moumita Das, who got her name registered on 15.04.2016 for the award of Ph. D. (Engg.) degree of Jadavpur University is absolutely based on her own work under our supervision and that neither her thesis nor any part of the thesis has been submitted for any degree/diploma or any other academic award anywhere before.

1. **Dr. Bhaskar Sardar**
   Associate Professor
   Dept. of Information Technology
   Jadavpur University,
   Kolkata-700106, INDIA.
   Dated : June, 2019.

2. **Dr. Ansuman Banerjee**
   Associate Professor
   ACMU
   Indian Statistical Institute,
   Kolkata-700108, INDIA.
   Dated : June, 2019.

To my parents, husband and supervisors

Contents

List of Figures

List of Tables

Acknowledgement

I would like to convey my highest gratitude to my supervisor, Dr. Ansuman Banerjee, Advanced Computing and Microelectronics Unit, Indian Statistical Institute, Kolkata, for his guidance and continuous support, patience and encouragement. He has been my philosopher, mentor, guide, big-brother right from the time I started this journey. Interactions with him have always been of great value, and those will linger in my memory for long. He has literally taught me how to do good research, and motivated me with great insights and innovative ideas. I have quite a few fond memories about my research with him, when we spent day after day dabbling on ideas, without support from experiments, fought through rejections with a positive spirit, and finally celebrated the taste of success when it came. It has been a wonderful journey together and I will cherish every moment of it for days to come.

My deepest thanks to Dr. Bhaskar Sardar, Jadavpur University for his support during my Ph.D. tenure. His careful and meticulous examination of the writeups gave us inputs on many interesting points that were often overlooked by me, but had a lot of value going forward. I thank him for his support for helping me to smoothly handle all administrative tasks related to my Ph.D. tenure.

I would also like to thank Dr. Mainak Chaudhuri, Indian Institute of Technology, Kanpur, India for his constructive comments and various discussions on my research. During my interactions with him, I learnt a lot about how systems research is carried out. His innovative ideas were always of deep insights, and we benefited immensely through these interactions. It was indeed a great pleasure to learn from him.

On the official front, I would like to thank Indian Statistical Institute for accepting my Woman Scientist affiliation. Indeed, this gave me a place to remember for long, my first research lab where I have spent many a sleepless night, delving deep into experiments. My sincere thanks to the faculty and staff of Advanced Computing and Microelectronics Unit, ISI for providing a wonderful friendly atmosphere for my work, and helping me with all admistrative support that I needed to carry out my research. A sincere note of thanks to DST, Government of India, for providing me support through the first few years of my Ph.D. tenure, and to ISI for providing me support later.

Finally, I am very much thankful to my parents and my husband for their everlasting support. Last but not the least, I would like to thank all my friends for their help and support. I thank all those, whom I have missed out from the above list.

Moumita Das
Dept. of Information Technology
Jadavpur University
Kolkata - 700106, India.

List of Abbreviations

BHR	Branch History Register
BHT	Branch History Table
BTB	Branch Target Buffer
BTFN	Backward taken forward not taken
CBP	Championship Branch prediction
DC	Design Compiler
DPI	Degree of Pattern Irregularity
EPL	Effective pattern length
FIFO	First In First Out
GBHR	Global Branch History Register
GEHL	GEometric History Length
KB	Kilo Byte
LBHR	Local Branch History Register
LOC	Lines of Code
LRU	Least Recently Used
LSB	Least Significant Bit
MPKI	Misprediction Per Kilo Instructions

MSB	Most Significant Bit
NTB	Number of taken branch instructions
NT	Not Taken branch
PC	Program Counter
PHT	Pattern History Table
RTL	Register-Transfer Level
SMT	Simultaneous Multithreading
SRAM	Static Random Access Memory
TAGE	TAgged GEometric predictor
T	Taken Branch
VLIW	Very Long Instruction Word
WP	Weakest Precondition

Abstract

In the landscape of computer architecture research, branch prediction has been an important item of both academic and industrial research interest since long. Indeed, as widely acknowledged in this community, designing efficient branch predictors has always been one of the top priority research tasks in computer architecture. Over the past few decades, a wide variety of branch prediction strategies have been proposed in literature, with widely varying insights, structures, philosophies and performance considerations. Fostered by phenomenal research in branch prediction algorithms, modern commercial processors today embed quite sophisticated predictors inside, with significant prediction capabilities and runtime performance on widely varying workloads.

This thesis is an attempt to revisit the problem of branch predictor design from a slightly different lens. In particular, the specific focus of our work is in designing predictors that are particularly suited for low storage environments. Dynamic branch predictors which are invoked during program execution typically store outcome histories of program branch instances, learn interesting patterns from these histories and use them for delivering predictions for future branch instances. As is evident from experiments reported in literature, many of the state-of-the-art dynamic branch predictors that show promise when exercised in high end processors, often fail to meet the accuracy and energy expectations when exercised in storage constrained embedded environments. A thorough scrutiny of the performance debacle reveals the fact that these sophisticated prediction strategies are significantly sensitive to the sizes of the history they learn from, more the size of the history structures these predictors are allowed to operate on, the better is the prediction accuracy, in general. This motivates us to design strategies that can efficiently mitigate these concerns, while maintaining desired prediction accuracies at low storage points.

In our journey through the different chapters of this thesis, we explore three main directions in branch prediction research. On one side, we examine if static analysis, through which we learn program characteristics beforehand between program versions, can lead to enhanced prediction performance. Our explorations reveal a positive performance benefit, and we propose a novel scheme to do away with a number of dynamic predictor invocations, and thereby improve on latency and energy, without compromising on prediction accuracy. As our second theme of exploration, we explore the problem of multi-component predictor design that can deliver better prediction accuracy by combining several dynamic components to deliver a prediction. Indeed, this is quite useful in general, since branch instances of a given application exhibit widely varying dependencies on the outcome history patterns, and it is usually not possible to learn these and predict their outcomes with a single predictor

component. However, the price to be paid for such multi-component predictors is in terms of the storage cost, since these components are typically trained to operate on individual history storage tables, which makes them difficult to exercise at low storage points. This is where our research provides two novel design schemes, with a proposal to unify the storage tables of the multiple components, and still deliver the expected prediction accuracy performance. In particular, we look at predictor designs with two or more components, and propose suitable modifications to make them work with shared predictor tables, while maintaining high prediction accuracy. The final theme of our research is around branch predictor evaluation. On one side, we develop a framework for branch predictor selection with consideration of multiple metrics of evaluation, namely, accuracy, latency, energy etc. This is an useful aid for prediction strategy selection for a given application domain, wherein multiple performance metrics, not just one, are important elements of concern. Our framework provides an useful exploration aid in this direction. On the other side, we examine the different components of a branch predictor design with respect to their resilience against faults and attacks. One of our key contributions in this direction is an attack scheme that can definitively slow down a benign application in a concurrent multi-threaded execution environment. We believe that such studies can lead to positive benefits for predictor design.

In summary, we believe that this thesis makes some important contributions in the space of branch predictor design for low storage processors. On one hand, our strategies for storage consolidation entail predictor component designs that can work on shared predictor tables. On the other hand, inputs from static program analysis help us synthesize better strategies for prediction. This gives us a unique advantage, as we demonstrate through extensive experiments on architecture workloads.

Chapter 1

Introduction

In high performance pipelines, branch instructions disrupt the smooth flow of instruction fetching and execution. This causes serious performance degradation as the fetched instruction and the following instructions must wait until the result of the condition is available. To alleviate this issue, out of order execution was introduced to make use of this idle time and let other non-dependent instructions advance with their execution. To further enhance the efficiency of pipelines, the processor at runtime often predicts the direction which a branch instruction will take. Branch predictors thus play a critical role in achieving high performance in all modern pipeline processors. A branch predictor at runtime predicts the direction which a branch instruction will take and begins fetching, decoding and executing instructions using this prediction even before the result of the branch condition is known. This greatly reduces delay and decreases CPI (cycles per instruction) of a program. The branch predictor is therefore, a crucial piece inside any modern pipelined processor. This has a flip side as well. If the branch predictor gives an incorrect prediction, all the fetched instructions have to be flushed and the alternative path that was expected to be not taken and all its following instructions have to be fetched into the pipeline. This in turn implies that the entire branch prediction effort has been a waste and the mis-prediction penalty

manifests in terms of the energy and processor cycles wasted as a result.

Developing efficient branch predictors has always been one of the top priority research tasks in computer architecture. While on one hand, a simple branch predictor component is easy to implement and light to support at runtime, its accuracy may often be limited. On the other hand, a sophisticated predictor may guarantee better expected prediction success, but at the cost of more energy and performance footprint, and this may elongate the pipeline critical path, often leading to a performance slowdown. This trade-off has inspired decades of research on efficient branch prediction strategies and led to the development of numerous branch predictors with varying degrees of complexities and efficiency [55][59][83][96].

An interesting evolution in branch prediction research aimed at improving the prediction accuracy of branch predictors, has been the proposal of multi-component branch predictors [26] which work by synergistic orchestration of dynamic single component branch predictors at run-time. The main motivation driving this has been the observation [35][45] that different dynamic predictors fare differently on different branches in terms of the misprediction metric. In other words, branches which are ill-suited for a certain branch prediction policy often have better performance when run with another predictor. A single dynamic predictor, therefore, may not be suited for all branches in a program, thereby necessitating the idea of multi-component hybrid branch prediction. Multi-component predictors have been well-studied in literature, with a number of design strategies, attempting to improve prediction accuracy and power consumption [26][36][35][45][54][83][86].

The most widely explored direction of research in predictor design has been around the objective of prediction accuracy maximization, with an expectation of energy reduction due to lesser mis-predictions. However, it has often been the case that the energy gain expected has been nullified by the energy footprint of the history tables that these schemes need to store and manipulate. An additional parameter of interest is the latency of program execution, which varies as well across different predictor designs. Another important

parameter to evaluate the performance of a branch predictor is the size of the predictor history table. Branch predictors typically function based on the history information stored in the corresponding prediction tables. Evidently, accuracy of branch predictors is sensitive to the storage they are allowed to use, more history information usually has been seen to generate more accuracy of prediction. The issue of storage for predictor tables is even more important for a resource constrained embedded environment with low storage budgets, since there is a more acute trade-off that needs to be worked out. On one hand, increased storage budget for the predictor tables, can lead to better accuracy of prediction. However, the cost and other overheads associated with storage structures increase. On the other hand, having a lower storage for predictors reduces cost, however, increases misprediction and manifests in terms of wasted instructions, latency and energy. Selecting an appropriate branch predictor structure for an embedded environment is therefore, quite a crucial task. A branch prediction unit charges a significant amount of power consumption in modern processor designs, and consumes a significant amount of storage and becomes a major issue for relatively small embedded processors.

The main objective of this research is to work on different aspects of the branch prediction methodology. On one side, we investigate how program analysis methods can be used to improve classical prediction techniques to improve system performance. On the other side, we propose some improvements in predictor design suitable for low resource and low power processors. We evaluate our methodologies using architectural simulators on large scale public domain benchmarks.

1.1 Motivation of this dissertation

In the field of computer architecture and processor design, the problem of efficient branch predictor implementation has attracted significant research across several decades. It is

reported that upto 15% of processor energy [83] can be consumed for branch prediction. An inefficient predictor can thus in turn lead to performance degradation as well. The primary motivation of this thesis is to improve the performance of the predictor in terms of different metrics like prediction accuracy, energy consumption, execution latency and predictor size. Another major challenge is to design branch predictors for an embedded processor with low resource budget and low power constraints. This is quite significant in the context of embedded systems or edge devices, which are being increasingly used today as compute nodes in the context of edge and fog computing. In this work, we also focus to work on the improvements of branch predictors for embedded processors.

1.2 Contributions of this dissertation

The objective of this thesis is to propose improvements in different aspects of branch prediction to improve system performance. The contributions of this thesis described in the different chapters, are briefly summarized below.

An efficient dynamic predictor design using program evolution: Software evolution has been extensively studied in the past decade for various properties and interesting patterns. In this work, we study the effect of evolution on branch prediction techniques. Typically for any program, at the hardware level, all dynamic branch prediction strategies learn branch behaviors at run time and later re-use them to predict the direction of future branches. The duration of the learning curve depends heavily on the kind of technique used and also the complexity of the program at hand. We show that saving the branch outcome profile from an older version and reusing it in a new version can significantly reduce this overhead and improve performance. In this thesis, we discuss the effect of program evolution on the performance of branch prediction, study how the individual branches get affected

during evolution, suggest a new method to reuse the branch behavior information from a previous version, and share our results on various software repositories. Experimental results indicate our intuitions are well justified.

A two-component dynamic predictor design with shared predictor tables: It has been widely acknowledged in literature that a single predictor component is often insufficient for prediction for complex programs comprising of branches with diverse history characteristics. This has led to the proposal of predictors with multiple components. These multi-component predictors have been able to achieve better accuracy, at the cost of storage. In this work, we study the problem of designing two-component branch predictors for low resource budget. We first examine a two-component predictor design where the individual predictor components are made to share the predictor table structure. Our experiments reveal that this sharing leads to a loss in prediction accuracy due to extensive interference between the predictors. To take care of the interfering entries, we add a tiny dealiasing cache to store the interfering entries, while the components still work on the shared prediction history table for the remaining. We present our findings on the CBP-2 traces [33]. Experiments show expected accuracy improvements over the shared design. Synthesis results generated using the Synopsys Design Compiler with TSMC 90nm libraries confirm the area and power benefits of our design.

A multi-component dynamic predictor design with interference control: We study the problem of designing multi-component branch predictors for low resource budget. We first examine a multi-component predictor design where the individual predictor components are made to share the predictor table structures. Further, we propose a simple modification to contemporary multi-component predictor designs to improve accuracy. We present our findings on the SPEC 2006 benchmarks.

Branch predictor selection with aggregation on multiple parameters: Selecting a branch predictor for a program for prediction is a challenging task. The performance of a branch predictor is measured not only by the prediction accuracy - parameters like predictor size, energy expenditure, latency of execution play a key role in predictor selection. For a specific program, a predictor which provides the best results based on one of these parameters, may not be the best when some other parameter is considered. The task to select the best predictor considering all the different parameters, is therefore, a non-trivial one, and is considered one of the foremost challenges. In this thesis, we propose a framework to systematically address this important challenge using the concept of aggregation and unification. For a given program, our framework considers the performance of the different predictors, with respect to the different parameters, and makes a predictor selection based on all of them. On one side, our framework can be an important aid for deciding on the best predictor to use at runtime. On the other side, the proposal of a new predictor can be systematically evaluated and placed in purview of existing ones, considering the parameters of choice. We present experimental results on the Siemens and SPEC 2006 benchmarks.

Performance attacks on branch predictors in Simultaneous Multithreading Processors: In an embedded processor with support for multi-threaded execution, with multiple different applications executing in different threads, and managed by a single predictor, significant inter-application interference due to sharing of predictor data structures has been acknowledged to be a serious concern. In this work, we show an attack methodology which exploits these shared structures for performance attacks on a benign application. In particular, we propose a methodology for creating a variant of a benign application, which when dispatched in a concurrently executing thread, can definitively slow down the performance of the benign one. We report the effect of such attacks on the Siemens benchmarks.

An empirical study on predictor storage and fault sensitivity: We first examine the common branch predictor designs available in literature, and characterize their accuracy versus storage performance. As expected, many of the predictors which are known to have high accuracy in general, lose out on performance when exercised in low storage scenarios. This work presents an empirical evaluation of different branch predictors at various storage points and the resulting effect on processor performance in terms of prediction accuracy and latency. We present our findings using contemporary branch predictors and the traces of the Championship Branch Predictor-2 benchmarks. We believe that our study will be extremely beneficial for choosing a branch predictor design for embedded processors working in resource constrained environments. In addition to this, we also present an empirical exploration of different state-of-the-art predictors with respect to faults on their storage structures (e.g. history tables, registers) and the resulting effect on processor performance.

1.3 Organization of the dissertation

This dissertation is organized into 9 chapters. A summary of the contents of the chapters is as follows:

Chapter 1 : This chapter contains an introduction and a summary of the major contributions of this work.

Chapter 2 : A detailed study of branch prediction methodologies and relative research is presented here.

Chapter 3 : This chapter studies the effect of program evolution on the performance of branch prediction techniques.

Chapter 4 : This chapter presents our two-component branch predictor design for low storage environments.

Chapter 5 : This chapter discusses the multi-component branch predictor design for low storage environments.

Chapter 6 : An evaluation framework for branch predictor selection on multiple parameters is proposed here.

Chapter 7 : This chapter discusses a performance slowdown technique on predictors.

Chapter 8 : An empirical study on storage sensitivity of branch predictors and their robustness against faults in prediction information is discussed in this chapter.

Chapter 9 : We summarize with conclusions on the contributions of this dissertation.

Chapter 2

Background and Related Work

In this chapter, we first present a few background concepts related to branch predictor design. We then present a brief overview of different methodologies proposed in literature for branch prediction.

2.1 Preliminary Concepts

In a pipelined architecture, instructions typically go through an assembly line while a program executes, as shown in Figure 2.1. Figure 2.1 shows a simplified view of a pipelined processor, wherein each instruction goes through different stages to reach the end of execution. Thus in every clock cycle, a new instruction can be fetched, while other instructions transit through the different stages inside the pipeline. This leads to improved execution by overlapping instruction latencies and different instructions can be in flight, resulting in improved performance. Branch instructions in this pipeline typically get resolved in the later stages (second / third stage in a 5-stage pipeline), thus it is a tricky task to decide which instruction to fetch next, when a branch instruction is encountered as the current in-

struction. The processor may still go ahead and load instructions from the immediate next instruction, but if the branch gets resolved in the else-path, the entire instruction stream loaded has to be flushed off. To improve the performance of program execution, efficient predictors are employed inside a modern processor which guide the processor to predict the direction of a branch and start fetching, decoding and executing instructions before the result of the branch condition is evaluated, as shown in Figure 2.1.

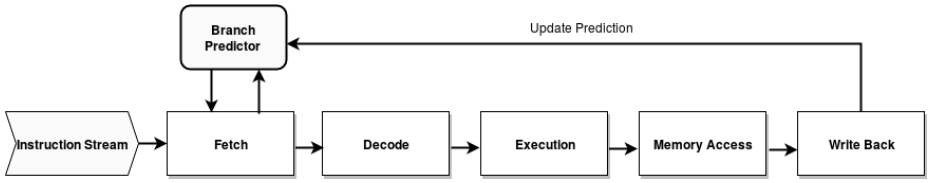


Figure 2.1: Architecture of a Pipelined Processor

To illustrate the importance of branch prediction, we consider the performance of a 5-wide super-scalar (fetches 5 instructions per cycle) 30-stage pipeline with a steady state performance of 1 instruction per cycle and a 20-cycle branch resolution latency (the cycle at which the exact direction of a branch is known / resolved). Hence, 1 mis-prediction causes wastage of 19 instruction fetches and new ones need to be brought in, thereby increasing the total number of clock cycles by 20. Consider a program with 500 instructions where 1 out of 5 instructions is a branch (uniformly distributed). Therefore, we have 100 branch instructions. We now discuss the accuracy versus overhead for processing these 500 instructions for different cases, as explained below.

- All instructions are fetched from the correct path (predictor predicts correctly for all 100 branches) - we have 100% accuracy and it takes 100 cycles.
- For 1 mis-prediction - it gives 99% accuracy and takes 120 clock cycles (100 cycles for correct path and 20 cycles for the wrong path).
- For 2 mis-predictions, it gives 98% accuracy and takes 140 clock cycles (100 for the

Processor	Minimum mis-prediction Penalty(cycle)
PPro,P2,P3	10
Intel Atom and Silvermont	11
Intel Nehalem	17
P4, P4-E4	24
Sandy Bridge, Ivy Bridge	15
Intel Haswell	15
VIA Nano	16
AMD K8 and K10	12
AMD Bobcat and Jaguar	13

Table 2.1: Mis-prediction penalties for some example processors

correct path + 20 * 2 for the wrong path).

- For 5 mis-predictions, it gives 95% accuracy and takes 200 clock cycles (100 for the correct path + 20 * 5 for the wrong path).

It is seen that only 5 mis-predictions out of 100 branches can double the number of processor cycles and cause 100% extra instruction fetch. The example above illustrates the crucial role of the efficiency of a branch predictor as a metric for processor performance. An inefficient predictor can lead to significant performance degradation. Designing efficient branch predictors has therefore been one of the top priority tasks in computer architecture.

Table 2.1 shows the mis-prediction penalties in terms of the cycles wasted for some representative processors. It can be observed that in all the cases, incorrect predictions can cause significant performance degradation due to high mis-prediction penalty. For example, P4 and P4E processors take around 24 clock cycles and 45 micro-operations as mis-prediction penalty [49]. Apparently, the processor cannot cancel an unnecessary micro-operation before it has reached the retirement stage. This implies that if we have a lot of micro-operations with long latency or poor throughput, then the mis-prediction penalty may be substantial.

Before we present an overview of branch predictor research, we discuss in the following some preliminary concepts related to branch instructions.

```
int main () {
   int     i, j;
   for (i=0; i<1000; i++) {
       for (j=0; j<4; j++) {
           c++;
       }
   }
   return c;
}
```

```
main:
      leal    4(%esp), %ec          ; function overhead
      andl    $-16, %esp
      pushl   -4(%ecx)
      pushl   %ecx
      xorl    %ecx, %ecx                ; i = 0 (%ecx)
.L2:
      xorl    %edx, %edx                ; j = 0 (%edx)
.L3:
      movl    c, %eax                   ; %eax = c
      addl    $1, %edx                  ; j++
      addl    $1, %eax                  ; %eax++
      movl    %eax, c                   ; c = %eax
      cmpl    $4, %edx                  ; if j < 4 then goto L3
      jne     .L3                       ; INNER LOOP BRANCH
      addl    $1, %ecx                  ; i++
      cmpl    $1000, %ecx               ; if i < 1000 then goto L2
      jne     .L2                       ; OUTER LOOP BRANCH
      movl    c, %eax                   ; return c
      popl    %ecx                      ; function overhead
      leal    -4(%ecx), %esp
      ret
```

Figure 2.2: Source code of a C program and compiler generated assembly

2.2 Preliminary terms and concepts

Some fundamental concepts related to branch instructions are discussed below. Consider the program fragment shown in Figure 2.2. The set of instructions shown in the X86 assembly language on the right are generated by the compiler as a result of the compilation of the C code on the left. Consider a pipelined processor on which these instructions get executed. The first 10 instructions do not pose any problem in smooth pipeline execution since the successor instruction is uniquely defined (unless these are exceptions or interrupts). However, for the instruction "jne .L3", the decision on which instruction to fetch next, is difficult to take, unless the condition gets evaluated. This is where the branch predictor steps in and attempts to predict the direction of a branch based on *history* of outcomes of earlier branch instances. We discuss below some related concepts in this direction.

Definition 2.1 Local and global history

Some branches use only their own outcome history for prediction. This history is known as local history. However, some branches use outcome history of other preceding branch

instances, this history is called as global history. ■

Example 2.1 *Consider the inner loop branch of the program shown in Figure 2.2. If the loop test is performed at the end of the loop, it shows the history pattern as 1110 since the loop branch is taken for the first 3 iterations, and not-taken the last time when the loop exists. Hence, by knowing the outcome of this branch for last 3 iterations, the next outcome is predicted. Hence, 1110 is the local history of this branch. When this loop is encountered again in a subsequent iteration of the outer loop, the 1110 local history is used for the prediction. The global history for this branch would include the outcomes of the other loop instance as well, in addition to its own.* □

We now present some classification metrics on branches.

Definition 2.2 Taken / not-taken branch

A branch is considered as a not-taken *branch if it follows the normal sequence of program execution, otherwise it is considered as a* taken *branch.* ■

As an example, if an instruction of the form if-$else$ evaluates to the $else$ path, it is considered as *taken*, while it is considered as *not-taken* if evaluated on the *then* path.

Example 2.2 *Consider the 2 branches (inner and outer loop) of the program shown in Figure 2.2. The inner loop branch is considered as* taken *for the first 3 times and as* not-taken *for the 4th time in each execution of the outer loop branch.* □

Definition 2.3 Biased / unbiased branch

Branches mostly evaluated towards one direction (taken / not taken) are called biased branches. The preferred direction for such a branch is its bias. ■

Example 2.3 *The inner loop branch of the program shown in Figure 2.2 is biased towards the taken direction, hence is considered as a biased branch.* □

Definition 2.4 Easy to predict / hard to predict branch

If a branch is predicted correctly most of the times by a branch predictor, then the branch is considered as an easy to predict *branch for the given predictor. A branch is considered as a* hard to predict *branch, if most of the times the branch predictor predicts it incorrectly.* ■

2.3 Research overview on branch predictor design

In this section, we first present an overview of branch predictors in literature. We begin with a classification of these predictors, depending on the state at which they operate. A broad classification of branch predictor in shown in Figure 2.3 and discussed below.

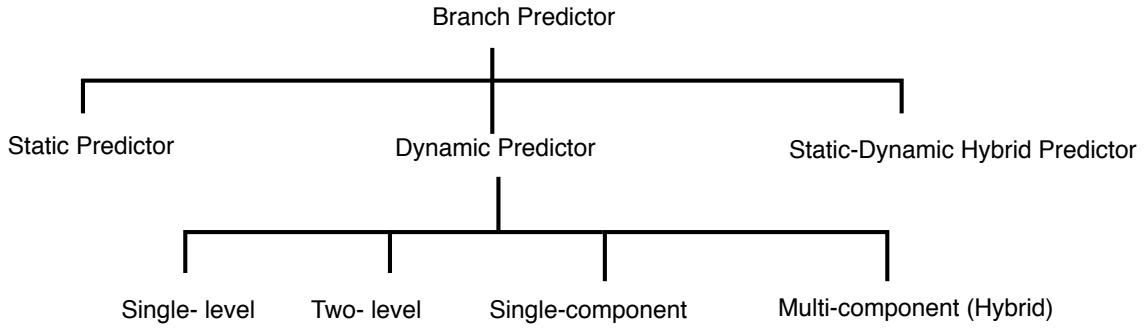


Figure 2.3: Predictor classification

2.3.1 Static Branch Prediction

In static branch prediction, the direction of a branch is assumed to be known at compile time. The earliest static branch predictors use always taken, BTFN (backward taken, forward not taken) strategies [105]. Another important static branch prediction policy uses

profile information for prediction [122]. This technique runs a program with different input sets, records the branch direction of every branch, and uses this information for further prediction. Static prediction with value range propagation is also widely used [87]. In [31], neural networks are used to perform static branch prediction. In this case, the likely direction of a branch is predicted by extracting program features such as control-flow and opcode information and supplying these features as input to a trained neural network. This approach achieves almost 80% correct prediction in comparison to 75% for other static heuristics [21][31].

2.3.2 Dynamic Branch Prediction

Modern processors mostly rely on dynamic branch prediction strategies, since they work during execution by collecting run time information. These predictors accumulate data during run-time or execution of an instruction. Since they consider the execution history of branches, they are more likely to adapt to dynamic changes in branch behaviour. History information is stored in special registers (Branch History Registers), and history tables. This history can be either the previous outcomes of a particular branch instance (local history) or the outcomes of preceding branch instances (global history). Branch History Register or BHR is a n bit shift register which shifts in bits to represent the branch outcomes of the most recent n branches (or the last n occurrences of the same branch). So in BHR a *1* is recorded if a branch is taken, otherwise a *0* is recorded. For a BHR of size n bits, at most 2^n different patterns can appear in this history register. For each of these 2^n patterns, there is a corresponding entry in the Pattern History Table (PHT) which is achieved by using an indexing function as discussed later. Each PHT entry contains the branch result for the last k times the same output of the indexing function appears. A two bit saturating counter [75] is used to perform this operation. Local history is stored in the Local Branch History Register (LBHR) and global history is stored in the Global Branch History Register

(GBHR). A branch predictor works in the following two stages :

Fetch stage During the fetch stage, the branch predictor provides its prediction based on the prediction information stored in its prediction tables. Generally, it uses history information to access the prediction table. Each entry of the prediction table stores a saturating counter whose Most Significant Bit (MSB) gives the final prediction. If the MSB is 1, prediction for the branch direction is taken, otherwise it is considered as not-taken.

For prediction, branch prediction typically keeps track of the outcome history of each branch based on the program paths that it encounters earlier. This is known as history information, and is encoded and stored in some form in history tables for lookup when the same branch is encountered again.

Retire stage The actual outcome of a branch is known when it retires. The branch predictor updates the history register and the contents of the prediction tables, more specifically the state of the counter used for the prediction according to the actual outcome. If actual outcome is taken, history register is left shifted and a 1 is inserted at the Least Significant Bit (LSB) position, otherwise a 0 is inserted at LSB. Similarly, the counter value is incremented by 1 if outcome is taken, otherwise it is decremented by 1.

However these history based predictions require hardware storage to save some past data of branches and utilize that data for prediction of the branches [112][117]. Early predictors used in dynamic branch prediction include a single level predictor consisting of a branch history table that keeps a record for each branch whether the previous branches were taken or not [105]. It comprises a table of saturating counters, where every counter tracks the past directions. The concept of two level prediction technique that can improve the prediction accuracy of a single level predictor is proposed in [119]. An active research in branch prediction [42][78][82][84] done during 1990 to 2000 resulted in the design of complex branch

predictors. After 2001, some sophisticated predictors are proposed to improve prediction accuracy. Research on neural inspired prediction is introduced in [107][64][59][61]. The GEometric History Length predictors proposed in [95] and the TAGE predictor and its variants are proposed in [57][96][97][98][100]. A recent work involves the number of taken branch instructions (NTB) branch predictor [37]. The NTB branch predictor utilizes two-bit saturating counters in the pattern history table based on the information about the number of taken-branches in the global branch history. It achieves high accuracy at no additional hardware cost. In the discussion below, we present a little more detail on some representative branch predictors.

2.3.2.1 Last Time Predictor

This predictor stores only the last branch outcome for every branch instruction and performs prediction using that single bit of information [105]. It is actually a 1-bit state machine with two different states 0 (not-taken) and 1 (taken) as shown in Figure 2.4. The state transitions occur in response to a taken (T) or not-taken (NT) outcome resulting from the execution of one or more branch instructions. When a branch instruction arrives, this predictor predicts the branch outcome as T or NT depending on the current state of the state machine. If a branch instruction generates branch outcomes like TTTTTTTTTTNNNNNNNNNN, this predictor gives the correct prediction every time except the transition from T to N. This predictor always mis-predicts the last iteration and the first iteration of a loop that can be resolved with a bimodal branch predictor, which is explained in the following.

2.3.2.2 Bimodal Predictor

A bimodal predictor (also referred in literature as a saturating counter) is essentially a state machine as shown in Figure 2.5. It has four different states 00 (strongly not-taken),

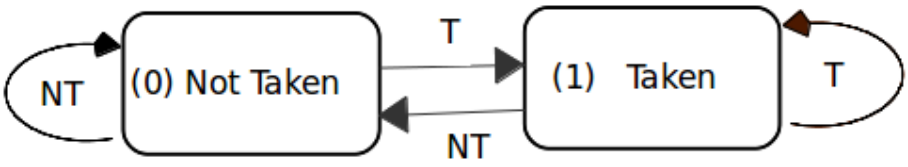


Figure 2.4: Last Time branch predictor

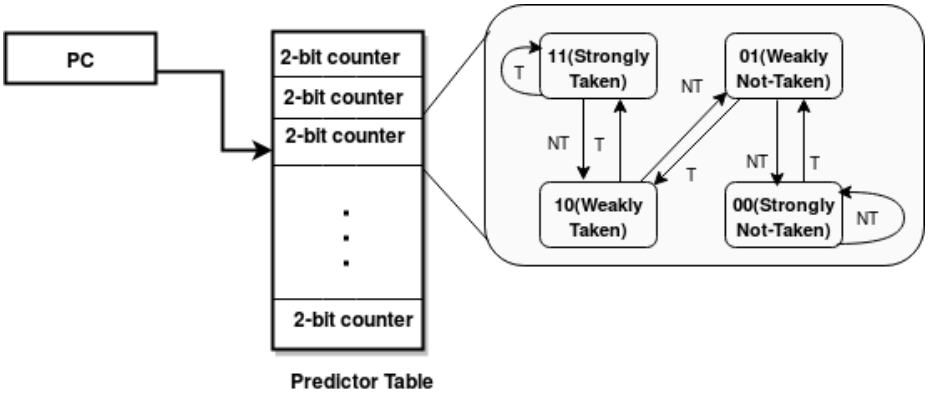


Figure 2.5: Bimodal branch predictor

01 (weakly not-taken), 10 (weakly taken) and 11 (strongly taken) defined by the 2 bits as shown in Figure 2.5. The counter transitions from one state to another in response to a taken (T) or not-taken (NT) outcome resulting from the execution of one or more branch instructions. Each bit of the two-bit counter plays a different role. The most significant bit, called the direction bit is used to track the direction of branches. If the counter is in state 01 or 00, the branch is predicted as NT. When it is in state 10 or 11, the prediction is T. The least significant bit provides a hysteresis which prevents the direction bit from immediately changing when a mis-prediction occurs.

2.3.3 Two-Level Adaptive Branch Predictor

The Two-Level Adaptive Branch Predictor was proposed by Yeh and Patt in 1991 [119][118] to make the predictions using two levels of branch history information. The first level stores the history of the last n branches encountered. The second level is the branch behavior for the last k occurrences of the same pattern of these n branches. Prediction is done on the basis of branch behavior for the last k occurrences of the pattern. This predictor uses two main data structures, the Branch History Register (BHR) and the Pattern History Table (PHT), as shown in Figure 2.6. The information required for branch prediction is collected at run-time by updating the contents of the history registers and the pattern history bits in the entries of the pattern history table depending on the outcomes of the branches.

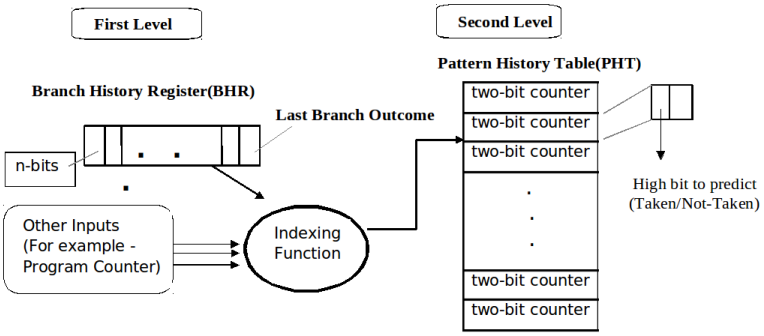


Figure 2.6: Architecture of the Two-Level Adaptive Branch Predictor

Variations of the Two-Level Adaptive Branch Predictor [34][46][78][119] have been proposed in literature. These are based on different parameters as discussed in the following:

- In first level, the last n branches can mean any of the following :
  - The actual last n branches encountered (GShare [78], GAg, GAp, GAs [119][120])
  - The last n occurrences of the same branch (PAp, PAg, PAs [119][120])
  - The last n occurrences of the branch from the same set (SAp, SAg, SAs [120])

- Different indexing functions are used to index an entry of the PHT. For example the indexing function can use any of the following :
  - Only the content of BHR (GAg, PAg, PAp)
  - The XOR of the branch address (available from the program counter PC) and the BHR content (GShare)
  - The concatenation of the low address bit and BHR (GAp) etc.

When a conditional branch is predicted, the content of the BHR is used as one of the inputs of the indexing function to address the pattern history table. The pattern history bits at that particular address in the PHT are then used for predicting the branch. After a conditional branch is resolved (at a later stage in the pipeline when the actual branch direction gets known exactly), the BHR is left shifted by one bit and the branch outcome is recorded at the least significant bit position in the BHR. This new updated BHR is also used to update the corresponding pattern history table entry by changing the current state of the saturating counter. Some of the popular two-level predictors are discussed below.

GShare and GAg: GShare and GAg are the global history based branch predictors. The GShare predictor [78] uses two main data structures - a Pattern History Table (PHT) and a Branch History Register (BHR), as shown in Figure 2.7. In this predictor, BHR is an n bit shift register that contains the branch outcomes of the most recent n branches. For each of these BHR patterns, a corresponding PHT entry is selected by XOR-ing the Program Counter (PC) value and the BHR content. Each PHT entry contains a two bit saturating counter and the MSB of that counter gives the final prediction. After resolving the branch condition, the states of the two bit counter and the BHR content are updated according to the actual branch outcome. The GAg predictor [120] is almost similar to the GShare predictor, the only difference is that it keeps the PHT indexed by the BHR value, as shown in Figure 2.8.

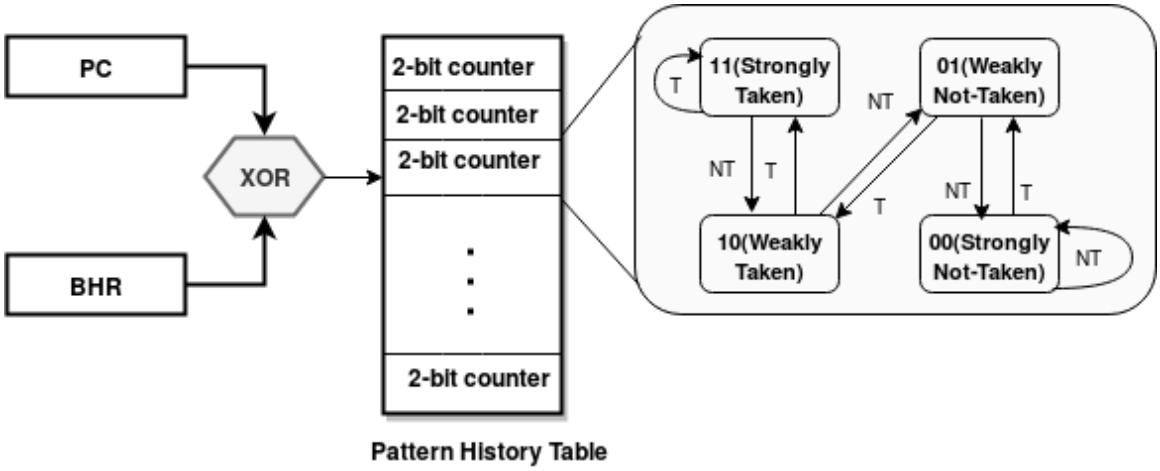


Figure 2.7: GShare Predictor

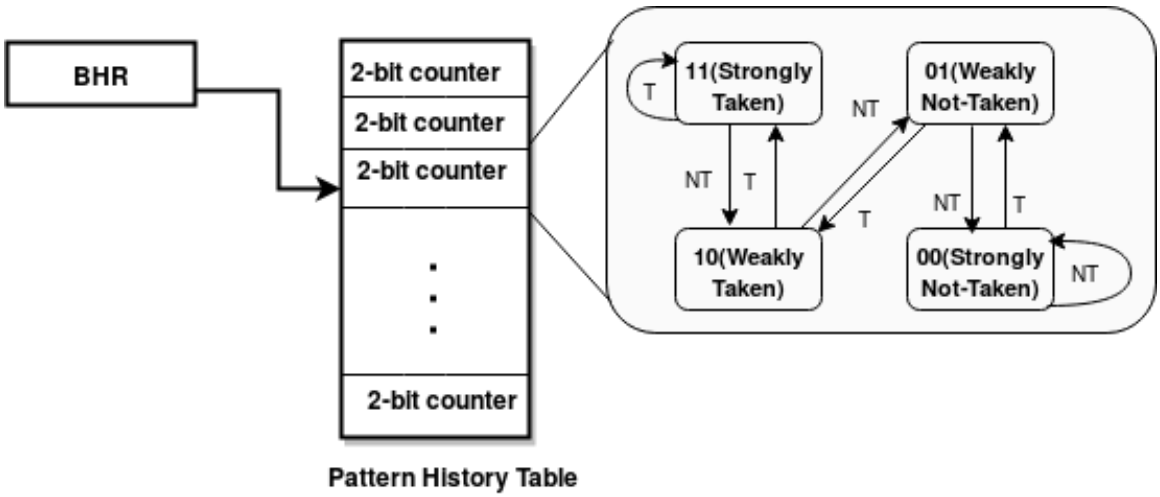


Figure 2.8: GAg Predictor

GAp: GAp [35] is a combination of global BHRs with multiple per-address PHTs. In GAp, each branch has its own PHT. The global BHR content is used to select the index in a PHT whereas a PHT is selected by the branch instruction address.

PAp: PAp [35] is a popular local history based branch predictor. It is a combination of multiple per-address BHRs with multiple per-address PHTs as shown in Figure 2.9. In

PAp, each branch has its own BHR as well as its own PHT. The BHR content is used to select the index in a PHT whereas a PHT is selected by the branch instruction address. For this predictor, storage size depends on BHR size and per address PHT size.

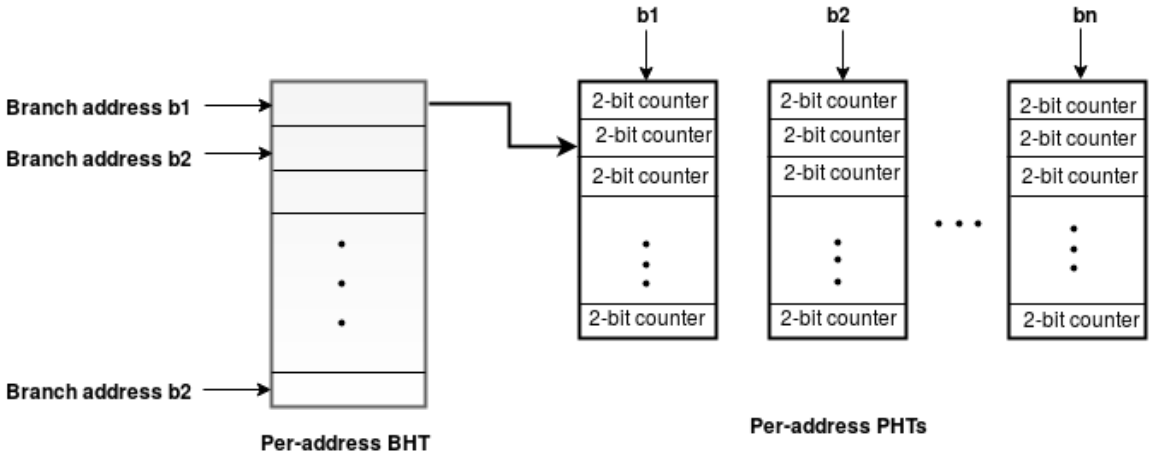


Figure 2.9: PAp Predictor

2.3.4 Perceptron Predictor

It is well-understood, that the prediction accuracy can be increased with the preservation of longer branch history. However, this is found to be an impossible proposition for large programs with many branches. To address this issue, neural dynamic branch predictor is introduced in [115]. This works proposes the use of learning vector quantization, which is soon usurped by a series of works based on Perceptrons [65]. The perceptron predictor [64] uses a simple neural network, the perceptron instead of the two-bit counters. Figure 2.10 shows a diagram of the internals of a perceptron predictor. It learns a target Boolean function $f(x_1, x_2, ..., x_n)$ over the inputs that predicts whether a particular branch will be *taken* or not. Here $x_i s$ are the bits of a global branch history shift register. $x_i s$ are bipolar, -1 represents *not taken* and 1 represents the *taken* outcome. A perceptron is represented as a vector of weights $w_0, w_1, ..., w_n$. Here, the weights are signed integers. The output

is calculated as the dot product of the weight vector $w_0, w_1, ..., w_n$ and the input vector $x_0, x_1, ..., x_n$ (x_0 provides the bias input and is always set to 1). The output y of a perceptron is as below:

$$y = w_0 + \sum_{i=1}^{n} w_i * x_i$$

The prediction algorithm requires computation of the sum of weights, which also results in a serious disadvantage for Perceptron-based branch predictors. Even after using high-speed arithmetic circuits, more than 4 cycles of latency is reported in [65]. To address this concern, path-based neural branch predictor is introduced in [60]. In path-based neural branch predictors, the branch scenario is considered to a greater extent, by accounting for the path that leads up to the branch, instead of considering the branch address alone. By considering the paths, the weight summation could be done before the actual branch instruction is encountered. As a result, both high accuracy and lower latency compared to [65] could be achieved.

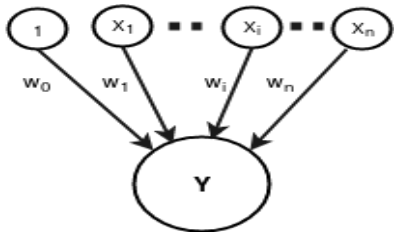


Figure 2.10: Perceptron Model

A major weakness of perceptrons is their increased computational complexity when compared with two-bit counters. However, it provides better prediction accuracy compared to other popular predictors even at lower resource budget. In a perceptron predictor, the best performance can be achieved by tuning the history length, the number of bits used to represent the weights, and the threshold. A smaller predictor size affects the number of bits of these three parameters, thereby degrading the prediction accuracy in turn.

2.3.5 Neural-Inspired Geometric History Length Table-based Predictors

In parallel with the rapid development of neural branch predictors, a set of prediction techniques that rely on separately tagged predictor tables showed slightly better accuracy for the same hardware budget. A comparative study in [108] reports that a tag-based predictor performs better when compared to an analog neural predictor.

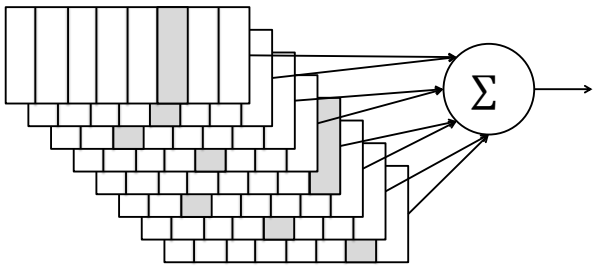


Figure 2.11: GEHL: GEometric History Length Predictor

The idea of separate predictor tables combined for branch prediction is in-depth analyzed in [95] (represented graphically in Figure 2.11). The predictor features M distinct tables T_i, where $0 \leq i < M$. The tables are indexed with hash functions of the branch address and the global branch/path history and contains predictions as signed saturating counters. The prediction is computed by reading a single counter $C(i)$ from each predictor table T_i and adding those like Neural predictors. The following equation provides the sum S.

$$S = \frac{M}{2} + \sum_{0 \leq i < M} C(i) \tag{2.1}$$

When S is positive or null, the branch is taken, otherwise not. The predictor update policy is also inspired from the Neural predictor [95]. A key innovation in the table-based predictor

is to index different tables using different history lengths, so as to capture the correlation with long branch/path history. For example, a predictor with 8 tables will be indexed with the lengths $L(i) = \alpha^{i-1} \times L(1)$. Considering table T_0 to be indexed with branch address, and with $L(1) = 1$ and $\alpha = 2$, the rest of the tables are indexed with $\{2, 4, 8, 16, 32, 64, 128\}$. This approach addresses the shortcoming due to less history lengths of earlier adaptive predictor schemes. T_0 could be a simple bimodal predictor, indexed by branch address.

2.3.6 TAgged GEometric (TAGE) History Length Branch Predictor

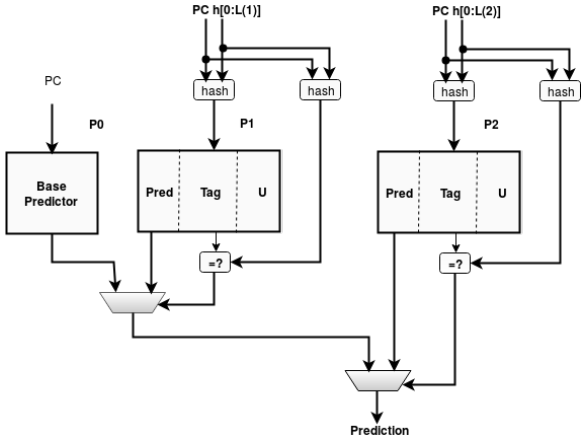


Figure 2.12: Architecture of the Two-Component TAGE Predictor

The GEHL scheme together with Perceptron-inspired update and prediction policy formed this efficient branch prediction technique [97]. This technique has introduced additional tagging in the predictor tables. The tagging is extremely useful to remove the aliasing effect (when two paths are identified as the same) in tables indexed with short history lengths. For tables indexed with long history lengths, tagging allows to keep a check on aliases and thereby dynamically switches to the prediction with shorter/longer history lengths. This TAGE predictor uses a base predictor P0 and a set of (partially) tagged components Pi, as shown in Figure 2.12. The base predictor is a simple PC-indexed 2-bit counter which is

used to provide the default prediction. The tagged predictor components Pi, $1 \leq i \leq M$ are indexed using different history lengths that form a geometric series [95]. Each entry of a tagged component contains a partial tag, an unsigned counter u and a signed counter ctr with the sign providing the prediction. At prediction time, the base predictor and the tagged components are accessed simultaneously. If no matching in the tagged component is found, the base predictor is used to provide the default prediction.

2.3.7 L-TAGE Branch Predictor

L-TAGE was introduced in the realistic track of the 2nd Championship Branch Prediction [97]. This is a combination of a TAGE predictor and a loop predictor. Default prediction is provided by the TAGE predictor when the loop prediction has low confidence. The TAGE predictor is storage-effective for realistic storage budgets whereas, with limited storage, the loop predictor can capture some branch behaviors that are not captured by TAGE. This predictor can be directly implemented in hardware as a multi-cycle overriding predictor (discussed later) backing a fast single cycle predictor and achieves higher accuracy.

2.3.8 Multi-component branch Predictors

An important observation [35] [45] in predictor design literature has been the fact that different predictors perform differently on different branches in terms of the misprediction metric, in other words, branches which are ill-suited for a certain branch prediction policy often have better performance when run with another predictor. A single predictor component, therefore, may not be well suited for all branches in a program, thereby necessitating the idea of multi-component branch prediction. *Multicomponent hybrid predictors* with multiple predictor components, with varying algorithms, have been well studied in the literature, with a number of design strategies, attempting to improve prediction accuracy and power

consumption [26] [29] [36] [53] [54] [83] [86]. Hybrid (multiple-scheme) branch predictors are introduced in [36]. In this research, according to the branch mis-prediction rates, authors classify branches into two different classes: single-directional branches and mixed directional branches. Once that is done, they identify which class provides better prediction accuracy for a given predictor data structure. According to this, predictors for different classes of branches are selected. Some popular dynamically selected multi-component branch predictors are discussed in the following subsections.

2.3.8.1 Tournament predictor

This predictor [69], maintains multiple predictor components at run-time, queries each component for their prediction for each branch, and finally selects the best predictor to use, based on past performance of these. Internally, these predictors use a *choice predictor* component that maintains a running counter to keep track of each predictor's performance for each branch and overall, till that point, based on which, it decides whose prediction to go forward with. Architectural details of a tournament predictor that combines local and global predictor together is shown in Figure 2.13. The different components are described in the following.

Predictor components We have two components here, a local and a global predictor operating independently. The local predictor uses a Branch History Table (BHT) to store the branch specific history patterns (branch outcomes). The Local Branch History register (LBHR) corresponding to a branch stores a *n-bit* pattern corresponding to the last n outcomes of the branch. The BHT is of size 2^pXn. For a given branch, the BHT is indexed by the last p bits of the branch address or the Program Counter (PC) value and the *n-bit* LBHR value stored at that index is used to index the PHT. Each entry in the PHT stores a 2-bit saturating counter value, the most significant bit of this counter value is taken as

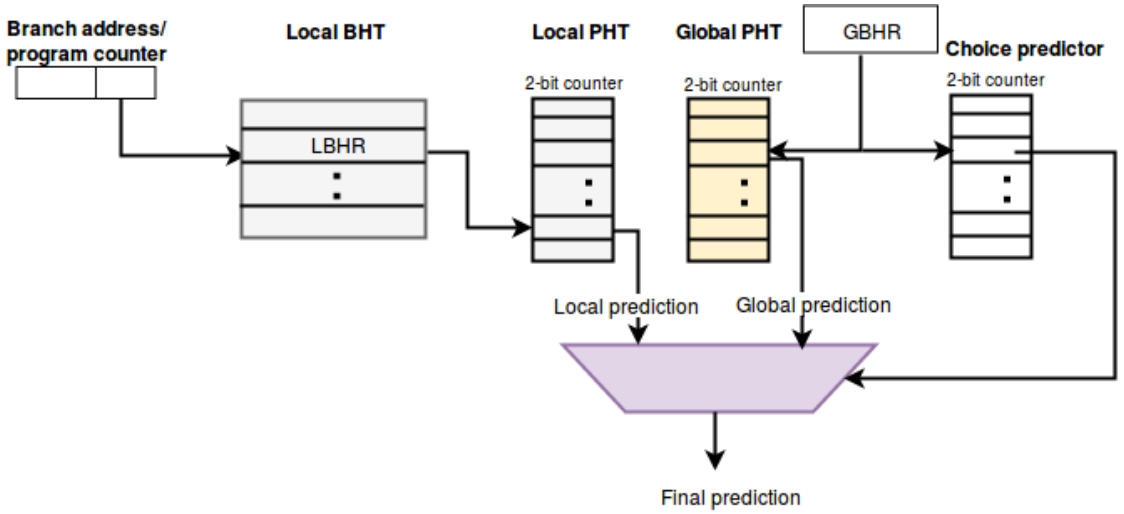


Figure 2.13: Tournament branch predictor

the predicted direction of a branch instance. Therefore, the size of the PHT is $2^n X2$ bits. An identical PHT is used by the global predictor component, as shown in Figure 2.13. The global predictor uses a *m-bit* Global Branch History Register (GBHR) to index its PHT. The GBHR stores a running outcome history of the m previous branch instances. As earlier, each PHT entry is a 2-bit saturating counter. When a branch is issued, the predictions of the local and the global predictors are read from the corresponding indices of the PHT. When a branch retires, the corresponding LBHR, GBHR and the 2-bit counter are updated with the actual outcome to train the predictors for future prediction.

Choice Predictor To select between the local and the global predictions for a branch, a choice predictor is used. The choice predictor is a choice pattern table of size $2^m X2$ bits, indexed by the m bit GBHR. Each entry of this table stores a 2-bit counter whose MSB determines the best predictor (either global or local) for every branch. If the counter state is 00 or 01, the prediction from the global predictor is considered as the final prediction. If the counter state is 10 or 11, the local prediction is taken as the final prediction. When the

branch outcome is known, and the predictions from the two predictors are different, this counter is updated [69]. The counter state is decremented if the branch outcome matches with the global prediction, otherwise it is incremented.

Alpha 21264 [69] is a two-component predictor that uses the tournament prediction policy as discussed before. An architectural overview of Alpha 21264 is shown in Figure 2.14.

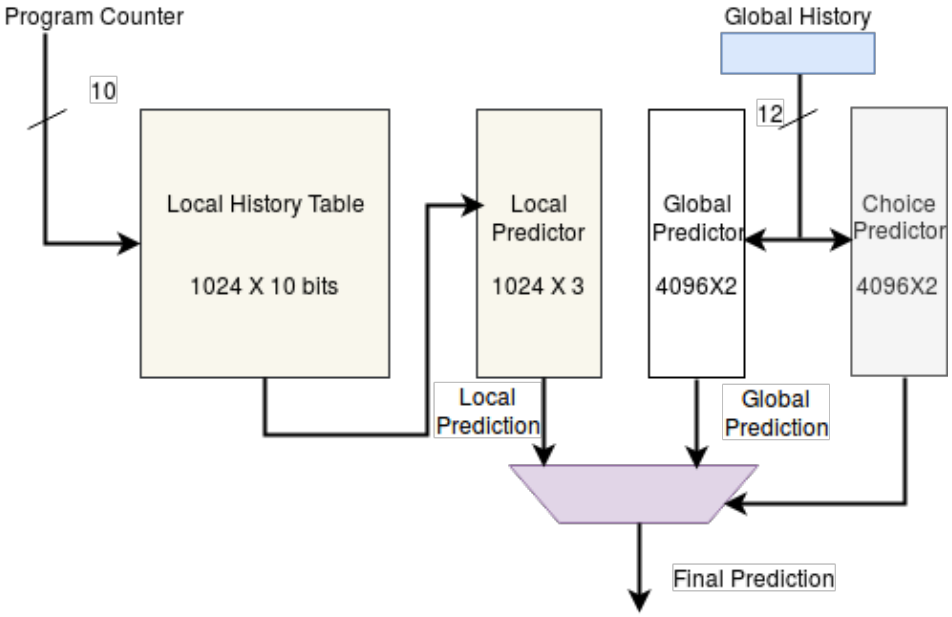


Figure 2.14: Architecture of Alpha 21264 Predictor

2.3.8.2 Overriding predictor

This predictor [67] [103] usually comprises of a combination of low response time (and usually, less accurate) and high response time (and possibly more accurate) predictors. These predictors typically start off with the prediction of a low response time predictor, and override the decision, in case the prediction from the high response time and expected to be more accurate component, differs from the former. This is done at execution time.

2.3.8.3 Static-Dynamic Hybrid Predictor

An alternative implementation of this hybrid predictor using a two level selection for dynamic predictors is introduced in [35]. The work in [36] proposes another popular hybrid prediction technique that combines static and dynamic predictors together. Static predictors are selected to perform prediction for mostly single-directional biased branches and dynamic predictors for the mixed directional branches. This predictor selection technique is done statically based on previous simulation output. In addition to this static selection, authors have also applied a dynamic selection technique to select a dynamic predictor among multiple dynamic predictors. They show that this static-dynamic predictor not only improves the prediction accuracy but processor energy expenditure as well, since branch predictors are not invoked at runtime corresponding to these static predictions. A variation of static-dynamic hybrid predictor is implemented in [86] to reduce the effects of destructive aliasing caused in a dynamic predictor when two branches with different branch behaviors share the same entry. Authors propose to identify branches for which this aliasing does not occur and use static prediction for them. Along with this, they suggest another new selection method which selects the static prediction policy for all easy to predict and tough to predict branches, since involvement of any dynamic predictor is redundant for these branches as the nature of these branches are well known beforehand. Their work shows the improvements gained in processor performance in terms of accuracy and energy expenditure.

2.4 Metrics for evaluating branch predictors

From the experiments reported in literature, we observe that different metrics have been considered for efficient predictor design. In some domains, metrics like prediction accuracy, mis-prediction per kilo instruction (MPKI), latency of execution, number of cycles etc. are of extreme importance, while energy is a crucial element of concern for some other

domains. Evidently, branch predictor design, in most of the cases, has tried to address one of these metrics as the primary design goal, with suitable constraints, if any, on the rest. An overview of related literature on branch prediction with respect to the objectives considered in the design is summarized in Table 2.2. In [67], researchers have shown that a branch predictor design optimized for accuracy can have a negative impact on overall instructions per clock (IPC) when the wire delays or the clock rates increase. A cost effectiveness study of different dynamic branch predictors is discussed in [120].

Researchers have extensively explored the role of a branch predictor considering the energy / performance tradeoff for processor design [85]. In [23], researchers have introduced a power-aware branch prediction technique for high performance processors. They selectively turn on and off some tables used in the combined branch predictor to reduce branch prediction power dissipation. In [56], authors have shown that customized branch predictors can be implemented by exploiting program behaviour repetition. Their approach can achieve a significant amount of energy saving, while causing a minimal degradation of prediction accuracy and performance. Along with prediction accuracy and energy consumption of a branch predictor, latency and storage requirement are two additional important performance parameters for branch predictor designs. Branch predictors are designed for fixed storage budgets in all Championship Branch Prediction programs [32][33][51][58][62][63][66][80][89][102], while latency aware branch predictors are designed in [59][67][103].

In [56], researchers propose the general principle of on-demand resource allocation for hybrid branch prediction using software to customize the predictor according to its resource demands. This on-demand branch prediction, which they call adaptive branch prediction, incurs two types of overhead: reconfiguration overhead and energy waste incurred by increased mis-speculation due to weaker predictor configurations. To reduce such overhead, they adopt a feedback-based approach. They divide an application into smaller units called modules, characterize their branch prediction demand through profiling, and then instru-

ment the application to dynamically reconfigure the predictor. The benefit they achieve is twofold: first, since characterization and decision making take place off line, this approach incurs little runtime reconfiguration overhead and adds little to the processor design complexity. Secondly, because of the module behaviors being repeated often, the hardware receives accurate demand information, which leads to a highly efficient reduction of operational complexity.

Another energy efficient branch prediction scheme has been introduced in a recent article in [54]. They use profiling, bias measurement and delay region scheduling to find the branches well suited for static prediction and perform static prediction instead of dynamic prediction for them. They show that, with the combined use of these two methods, the number of dynamic branch predictor accesses / updates are reduced by up to 62% which ultimately results in an average global processor power saving of 6.22% in a high-performance embedded architecture. To save processor energy in hybrid prediction, an interesting research is proposed in [83]. They introduce the idea of on-demand dynamic branch prediction that uses compiler generated hints to identify those instructions that can be more accurately predicted statically to eliminate unnecessary branch prediction unit lookups. They avoid branch prediction unit table lookups and use static prediction for mostly biased branches with high accuracy. This on-demand prediction technique enables 36% average energy saving in the fetch unit and 7% average energy saving in the core by combining dynamic and static branch prediction together. The compiler, using profiling information, provides static hints that enables the program to avoid dynamic predictions on highly-biased branches (where the bias exceeds the accuracy of the dynamic predictor). They show that this leads to 80% reduction in lookup events, and an average gain of 9 to 12% in energy-delay product. [26] introduces another efficient static-dynamic hybrid predictor to save more energy and die space. In their work, a new bias parameter is introduced as a mechanism for trading off small amount of performance for saving die-space and energy.

Another variation of the hybrid predictor named as poTAGE-SC is introduced in [81]. It combines several TAGE based predictors using different forms of histories (local, global, and frequency), a COLT inspired [74] prediction combiner and a statistical corrector (SC) predictor [98][99] fed with various forms of branch histories together. In [101], MTAGE-SC is proposed to improve the prediction accuracy of this poTAGE-SC. This improvement is done in two ways, first, through incorporating new forms of branch histories, adding a new TAGE component and incorporating other forms of histories in the statistical corrector predictor and second, in conveying more information from the TAGE predictors stage to the statistical corrector and to the final prediction computation stage. Results show, this MTAGE-SC predictor achieves 2.575 MPKI, 4.7% lower than the poTAGE-SC predictor. A new hybrid prediction approach that combines a static profile-based branch correlation analyzer and a dynamic branch predictor is proposed in recent literature [110]. The static profile-based branch correlation analyzer finds precise correlations between branches using profiling and data dependency analysis and identifies all the branches that do not have any impact on the worst case execution time of the task and then the dynamic predictor uses the correlation information to make online predictions. Experimental results show that this approach outperforms some of the state-of-the-art approaches. An overview of related literature on branch prediction with respect to the objectives considered in the design is summarized in Table 2.2.

In contrast to the existing literature on branch predictor design, the objective of this thesis is to study predictors for low resource architectures. We explore some branch prediction schemes existing in literature and attempt to tailor them to make them suitable at low storage points. This gives us a unique novelty and standpoint that we illustrate in the following chapters.

Existing Work	**References**	Metrics
Static prediction	[105] [87] [122] [21] [31]	Prediction accuracy
Dynamic prediction	[112] [117] [42] [82] [78] [84]	Prediction accuracy energy consumption
State of the art prediction	[57] [96] [97] [98] [100]	Prediction accuracy
Neural branch prediction	[64] [59] [61] [109]	Prediction accuracy Energy consumption
Single-level dynamic prediction	[105]	Prediction accuracy
Two-level dynamic prediction	[118] [119]	Prediction accuracy
Single-component dynamic prediction	[118] [119] [64] [59] [61] [109] [57] [96] [97] [98] [100]	Prediction accuracy
Multi-component dynamic prediction	[67] [103] [69] [98] [99]	Prediction accuracy
Power aware branch prediction technique	[23] [55]	Energy consumption
Customized branch predictor	[56]	Prediction accuracy/energy tradeoff
Static-dynamic hybrid predictor	[36] [26] [54] [86] [119]	Prediction accuracy/energy tradeoff
On demand branch prediction	[83]	Energy-delay product
IPC focused Branch predictor design	[67] [59] [103]	Instruction per clock(IPC)
Branch predictor attacks	[17] [16] [18] [47] [48] [111]	Prediction accuracy

Table 2.2: Summary of metrics considered in branch predictor design

2.5 Overview of Architectural simulators used

Architectural simulators used in this work are discussed below.

2.5.1 Tejas Simulator

Tejas [11] is an open source, Java based multicore architectural simulator. It is highly configurable i.e. number of cores, cache configurations, memory, prediction policy, etc. can be configured in an XML file, which the simulator reads before execution. It translates the emulated instructions to Virtual Instruction Set Architecture instructions and simulates these unlike other simulators like SimpleScalar [9] and SESC [8]. This simulator can run any software without the hassles of cross compilation. Firstly it emulates the x86 executable with the PIN Tool [76]. At the end of each execution, it reports various statistics related to cache utilization, branch prediction accuracy, energy expenditure, etc. Tejas provides two types of pipeline: In-Order and Out-of-Order. Here, the In-Order pipeline is a standard five-stage (Fetch, Decode, Execute, Memory, Write-back) pipeline architecture where branch

prediction is performed in the decode stage and a mis-prediction results in a penalty of further instruction fetches being stalled for a pre-determined number of cycles. We use the In-Order pipeline architecture of this simulator in our experiments.

2.5.2 Gem5 Simulator

Gem5 [27] is a modular discrete event driven simulation platform. It can simulate user specified programs where system services are provided by the simulator in the *syscall* emulation mode and can also simulate a complete system i.e. an operating system in full system mode. This simulator supports simultaneous multi-threading (SMT) mode where multiple user specific programs can run simultaneously. In this SMT mode, each program is provided a unique thread id with a round-robin scheduler as the default scheduler.

2.5.3 Championship Branch Prediction-2 (CBP-2)

CBP-2 provides an evaluation framework that includes a set of traces, and a driver that reads traces and simulates the behavior of a branch predictor. The processor has a 14-stage, 4-wide pipeline. The trace set given in CBP-2 includes 40 traces which are classified into 5 categories as - CLIENT, INT (Integer), MM (Multimedia), SERVER and WS (Workstation). The driver reads a trace and calls the branch predictor through a standard interface. The predictor can decide when and what predictions to provide to the driver. The driver records whether the predictor is correct when the prediction is provided. For each branch, a mis-prediction penalty value is calculated by the number of cycles that the fetch unit was on the wrong path. After execution, it reports the performance statistics in terms of Misprediction Penalty per Kilo Instructions (MPKI) for conditional and indirect branches.

In the following chapters, we present the contributions of this thesis in more detail.

Chapter 3

An efficient dynamic predictor design using program evolution

The main contribution of this chapter is an energy efficient dynamic predictor design, utilizing inputs and outcome histories from software version repositories. Dynamic predictors make predictions based on dynamic information about the program under execution collected at runtime. Since they consider the execution histories of branches, they are more likely to adapt to dynamic changes in branch behaviour. However this improvement in branch prediction accuracy comes at a cost. For most sophisticated processors, dynamic predictors account for a significant share of the processor's energy consumption [54]. A number of research articles have reported a nice handshake of the predictors, whereby a static predictor (working with the compiler), based on its analysis and profiling, can insert taken / not-taken hint bits into branch instructions, which the dynamic branch predictor at runtime can take help from, to decide the branch direction. In [54], Hicks et al. proposed a mechanism to use these hint bits in improving the energy utilization of a processor. The energy expenditure for the branch predictor is directly proportional to the number of

accesses to the branch predictor. They used profiling and local delay regions to reduce the number of calls to the branch predictor. This information was encoded as taken / not-taken hint bits for the branch instructions and some calls to the branch predictor were avoided for such branches, resulting in improved energy utilization.

The main observation driving our research presented in this work is the fact that in many cases, the direction of a branch tends to be heavily biased to either the taken or not-taken path resulting in a skewed distribution in contrast to being bimodal. In such cases, it is better to prevent calls to the dynamic predictor and rather make fixed choices (taken / not-taken) each time it is encountered. This choice has to be made maintaining the accuracy of branch prediction. Hence, only those branches can be chosen, for which the accuracy of the predictor is almost equal to or less than bias based prediction. We use this idea here and extend this to different program versions, where we collect the profiling information from an earlier program version from software repositories and explore how it can be used for expediting the branch prediction for later versions.

Software archives and bug databases provide researchers with a wealth of information. Over the past decade, researchers have come up with various interesting patterns and use cases related to source code development and bug scenarios. Software evolution involves addition, deletion and modification of functionality that the application provides. The change may range from a simple bug fix patch to addition of a new feature. Software evolution was introduced as early as in 1980 [73]. This work classified programs according to their relationship with the environment in which they are executed and proposed laws on program evolution based on quantitative studies on various systems. In a more recent study, researchers [25] exploited graph topologies to better model software evolution and constructed predictors for enhancing software development and maintenance by estimating bug severity, prioritizing refactoring efforts, and predicting defect-prone releases. They tested their methodology on various open source programs like Firefox, Eclipse, etc. In

[22], researchers showed how a functionally correct software version, referred by them as the *"Golden Implementation"*, can serve as a reference model in software debugging. They used dynamic slicing and weakest precondition methods to produce bug reports which point to the location of fault precisely. In our work, we use weakest preconditions to bring out differences in behaviour of program branches across two versions of a software. In [90] a new approach, DARWIN, was proposed for debugging evolving programs. DARWIN performs a dynamic symbolic execution along the execution of a given test input in two programs. Thus, it is applicable for debugging a buggy implementation with respect to a golden implementation. A study of change effect analysis over software versions, in an ideal setting, computes all the differences that a change of a code in a program causes on every element or statement of that program. This difference includes which program elements are affected by that change and also how exactly their behavior in terms of frequency and states have been affected. The work in [93] elaborates on this idea. This work describes change-impact analysis and change testing. Building from the foundation of research in software evolution, in this chapter, we analyze how branch behaviour changes across program versions and apply our methodology to demonstrate the re-usability of predictions across versions.

In this work, we explore how the features of software evolution can possibly help us improve branch prediction performance. As we witness from a number of software repositories, the behaviour of a significant number of branches remains unchanged across program versions. This motivates us to explore the idea of using branch outcome profiles from previous versions of the software for enhancing static branch prediction for future versions. Given two versions of a software, our analysis collects information about the unchanged parts of program code and possibly easy to predict branches. This information is later used to aid branch prediction for branches for the modified program version. This idea forms the core highlight of this work and we provide experimental evidences to support our claims as well.

Static branch prediction through profiling has often been found to be quite useful in practice,

given that the test suite for a production software is quite exhaustive and consists of test inputs that can explore the different paths of the program. The process of profiling is time consuming and as we observe here, can often be dispensed with, at least partially for later program versions exploiting version change information. One can consider an incremental process of accumulating branch profiling information. The efforts involved in generating a branch profile through rigorous testing for the first version can be reused in the successive versions, thereby eliminating the profiling overhead, at least partially for branches which are expected to behave identically. This is the main motivation behind our work here.

Contributions of this work

- We study the effects of software evolution on different dynamic prediction strategies.

- We propose a selective dynamic prediction call strategy for a given program, based on the branch profile information recorded from a previous version of the program.

- We perform experiments on open source benchmarks to show the benefits in all the metrics for 4 different predictors used using our selective predictor call strategy.

The rest of the chapter is organized as follows: Section 3.1 demonstrates an example which discusses the problem at hand. Section 3.2 elaborates the main methodology behind our framework. Section 3.3 discusses the building blocks of our implementation. In Section 3.4, we show the results obtained on benchmark code versions. Section 3.5 discusses about some of the limitations of this work, while Section 3.6 concludes the chapter.

```
1 void dodash(char delim, char *src, int i, char *dest,
int *j, int maxset)
2 {
3     while ((src[*i] != delim) && (src[*i] != ENDSTR))
4     {
5         if (src[*i] == ESCAPE) {     /* Branch 1 */
6           ...
7         } else
8            if (src[*i] != DASH)      /* Branch 2 */
9               ...
10           else if (*j <= 1 || src[*i + 1] == ENDSTR)
11                ...                  /* Branch 3 */
12           else if ((isalnum(src[*i  1])) &&
13                    (isalnum(src[*i + 1])) &&
14                    (src[*i  1] <= src[*i + 1]))
15              ...                    /* Branch 4 */
16           else                       /* Branch 5 */
17              ...
18    }
19 }

20 bool getccl(char *arg, int *i, char *pat, int *j)
21 {
22     if (arg[*i] == NEGATE) {       /* Branch 6 */
23        ...
24     } else {                       /* Branch 7 */
25        ...
26     }
27     ...
28     dodash(CCLEND, arg, i, pat, j, MAXPAT);
29     ...
30     return (arg[*i] == CCLEND);
31 }
```

(a)

```
1 void dodash(char delim, char *src, int i, char *dest,
int *j, int maxset)
2 {
3     while ((src[*i] != delim) && (src[*i] != ENDSTR))
4     {



5         if (src[*i] != DASH)         /* Branch 2 */
6            ...
7         else if (*j <= 1 || src[*i + 1] == ENDSTR)
8            ...                      /* Branch 3 */
9         else if ((isalnum(src[*i  1])) &&
10                (isalnum(src[*i + 1])) &&
11                (src[*i  1] <= src[*i + 1]))
12           ...                      /* Branch 4 */
13        else                        /* Branch 5 */
14           ...
15    }
16 }

17 bool getccl(char *arg, int *i, char *pat, int *j)
18 {
19     if (arg[*i] == NEGATE) {     /* Branch 6 */
20        ...
21     } else {                     /* Branch 7 */
22        ...
23     }
24     ...
25     dodash(CCLEND, arg, i, pat, j, MAXPAT);
26     ...
27     return (arg[*i] == CCLEND);
28 }
```

(b)

Figure 3.1: Source code of *replace* program, program version V1 (a) and V2 (b)

3.1 Motivating Example

In this section, we present an overview of our approach on a simple fragment based on the replace code piece (replace.c) of the SIR benchmarks [10]. The replace program performs string substitution. We consider two versions of the replace program, versions *v1* and *v2* from the distribution [10]. We refer to these two versions as V1 and V2 in the following sections. Figure 3.1(a) and (b) show two functions, `dodash` and `getccl`, from the versions V1 and V2. Across these two versions, the only change is the removal of *Branch 1* in the `dodash` function. Let us analyze the effect of this change on the following branches and examine the branch conditions. To understand the effect of this change, it is not only sufficient to examine the branch conditions, but also the preceding code logic that may influence the evaluation of the branch condition. As we see here, as a result of this change,

the other branches, previously in the else part of *Branch 1*, now are evaluated by default. It can be observed that the consequence of this change is that branches 2, 3, 4 and 5 in the `dodash` function in V2 now have different conditions on the preceding logic under which they are evaluated in V1. This is formalized in the discussion later.

Additionally, in the `getccl` function, which invokes `dodash`, the statements following the function call may be affected by the change in `dodash`. For example, the statement `return (arg[*i] == CCLEND)` in `getccl` (line number : 27 of Figure 3.1(b)) is dependent on `dodash` as the variable `i` may be modified inside `dodash`. This implies that the other functions, which are dependent on this return statement, may also be affected. On the other hand, in `getccl`, branches 6 and 7 are not affected by this code change.

A thorough analysis of these two versions of the replace program reveals that a considerable number of branches are not affected by the modifications done within `dodash`. Hence, the branch behaviour of all the unaffected branches of V2 will be the same as in V1 for the same set of inputs. Therefore, it is possible to reuse the branch profile information from V1 as the predictions for the corresponding branches in V2. Additionally, the direction of some branches tend to be heavily biased to either *taken* or *not taken* direction. The objective of our work is to relinquish the branch predictor for all such unaffected biased branches in V2 by inheriting the branch profile information from V1. In the next section we propose a formal methodology to this end.

We now explain the working of our method with the help of V1 and V2. We run V1 and record the branch outcome information for all the branches which are biased towards a fixed direction (*taken/not taken*). We identify all the branches in V2 corresponding to the biased branches in V1 and isolate the ones which are unaffected or affected less than a threshold. For V2, we load that recorded branch profile information for all these branches and during branch prediction of these branches, we relinquish the predictor by reusing this branch profile information.

3.2 Detailed Methodology

We now describe the overall methodology behind our framework. Our approach takes two versions of a software and proceeds in three main steps.

- Step 1 : Branch profiling and biased branch identification.
- Step 2 : Branch mapping and candidate branch identification.
- Step 3 : Selective dynamic predictor invocation.

Below, we discuss each of these steps in detail.

3.2.1 Branch profiling and biased branch identification:

We run an architectural simulator on the first version and record the execution and predictor statistics for each branch. From this information, we isolate branches which are biased towards a particular direction - *taken/ not-taken*. Recall that a *taken* branch is one which takes the *else* path. These branches are good candidates for our work. We present a couple of definitions to formalize this.

```
1   int i1, i2, var, var2;   /* input variables */
2   if (i1 > 0)               /* Branch 1 */
3      x = f1(i1);
4    else                     /* Branch 2 */
5       x =  f2(i1);
6   y = f1(i2);
7   if (x > y) {             /* Branch 3 */
8     ...
9   }

10   var2 = var + 1;
11   if (var2 != var1) {     /* Branch 4 */
12    ...
13  }
```

Figure 3.2: Example program code snippet

Definition 3.1 Biased Branches: *Branches mostly evaluated towards one direction (taken / not taken) are called biased branches. The preferred direction for such a branch is its bias.* ■

Example 3.1 *Consider the code snippet in Figure 3.2.* Branch 4 *is always evaluated as* not taken *for any input value of* var *and is biased towards one direction.* □

Definition 3.2 Bias Hit Ratio: *This is the ratio of a branch's bias to the total number of times it is executed.* ■

Example 3.2 *In Figure 3.2, bias hit ratio for* Branch 4 *is 1, since it is executed as taken every time. However, for Branches 1, 2 and 3, it depends on the executions.* □

Definition 3.3 Prediction Hit Ratio: *This is the ratio of the total number of correct predictions for a branch to the total number of times the branch is executed.* ■

Example 3.3 *Consider a scenario where the number of correct predictions for* Branch 4 *in Figure 3.2 is 900 and it is executed 1000 times. Hence the prediction Hit Ratio is 0.9.* □

In our method, if the bias hit ratio of a branch in the first program version is more than the prediction hit ratio, we select the branch as a candidate for the next version and follow the steps discussed below. *Branch 4* in Figure 3.2 is such a candidate since its bias hit ratio (1) is more than its prediction hit ratio (0.9) when executed 1000 times with 900 correct predictions. Bias Hit Ratio serves as a hint for the branch in the next version. All such identified branches are passed on to the next step.

3.2.2 Branch mapping and candidate branch identification:

We identify the extent to which the branches received from the first step have changed in the evolved version, this is formally characterized in terms of the weakest precondition with respect to the branch condition. If the change is below a certain threshold, we use that branch and its outcome from the first version as the prediction for the evolved version. We expect the dynamic predictor will utilize this hint while executing the evolved version and dispense the dynamic predictor call on these. Our motivation in this work is to explore if an evolution aware branch prediction strategy can be designed by comparing the weakest preconditions of the corresponding branches across the two program versions. The predictor tracks changes in the weakest preconditions and benefits from the evolution information.

We assume as earlier two versions of a given program, say V1 and V2. Between these two versions, a lot may have changed at the source code level. Hence directly correlating the branches in the two versions might be difficult. Again, even if a branch remains unchanged i.e. its condition is unaltered, some conditions in the influencing cone of the branch condition along the paths leading to this branch might have changed. Hence, we require a formal way to characterize the influencing logic and map the branches in V1 to those in V2 without compromising the accuracy of branch prediction. We formalize the characterization in terms of the weakest precondition [14] of a branch condition. Before presenting our approach, we present some formal definitions below.

Definition 3.4 Weakest Precondition (WP): *For a given branch $\mathcal{B}$ with a branch condition $\mathcal{C}$, it is a disjunction of the path conditions [20] of paths leading to $\mathcal{B}$.* ■

Essentially, this expression consists of program statements which either directly or indirectly influence the evaluation of $\mathcal{C}$. It may be noted that any set of inputs which satisfy the WP leads to the branch condition being evaluated to true.

Example 3.4 *In Figure 3.2, WP for* Branch 3 is computed as :

$$(x > y) \wedge (y = f1(i2)) \wedge (((i1 > 0) \wedge (x = f1(i1)) \bigvee (\neg(i1 > 0) \wedge (x = f2(i1)))$$

It is evident that the condition evaluation for *Branch 3* is influenced by the program statements at line numbers 2, 3, 4, 5, 6 and 7. □

We now present an approach for computing the weakest precondition.

3.2.2.1 Weakest precondition (WP) computation

Weakest precondition of a given branch with branch condition c can be defined as follows. Let π be a set of instructions $\langle i_1, \ldots i_n \rangle$ in program P, where i_n is our branch where c should hold true. Inductively we calculate $wp(i_n, c) = cond_{n-1}$, then $wp(i_{n-1}, cond_{n-1}) = cond_{n-2}$ and so on. The weakest precondition of c along π is then the formula $cond_0$ obtained when we reach the beginning of the program.

We now elaborate on the WP computation method for each branch of a given program. It is done recursively by computing the weakest precondition for every statement.

WP computation rules: Our WP computation algorithm is simple. We start from the first branch of the program. For a branch, to compute the WP, we need to first set a post-condition c, with respect to which the weakest precondition is to be computed. This is straightforward in our case, we use the branch condition for which WP is being calculated as the post-condition. For each statement instance *stmt* encountered during the backward traversal, the algorithm updates the current WP $\mathcal{Q}$ as follows. Initially, $\mathcal{Q} = c$.

1. *Data dependency (Assignment statement):* For a statement *stmt* of the form $x = e$, we use the following rule:

   $$wp(x = e, \mathcal{Q}) : \mathcal{Q}[e/x]$$

   This rule essentially substitutes e for all occurrences of variable x in the currently computed WP $\mathcal{Q}$.

2. *Control dependency (branch statement):* For a control statement *stmt* involving the condition $\mathcal{R}$, we use the rule:

   $$wp(\mathcal{R}, \mathcal{Q}) : \mathcal{R} \wedge \mathcal{Q}$$

   This rule essentially conjoins the control condition with the currently maintained WP.

This algorithm terminates when we reach the beginning of the program. The resultant WP, $\mathcal{Q}$, is reported as the final WP.

Example 3.5 *The WPs are computed for both the programs* $V1$ *and* $V2$ *in Figure 3.1 (a) and (b). Let us examine the application of the rules given above in computing the WP for* Branch 2 *in example program V1 given in Figure 3.1(a), line number 8. Initially, the WP is set as the post-condition* $src[*i]! = DASH$ *(branch condition of* Branch 2*) . While traversing backward, we encounter statement 5 (*$\neg(src[*i] ==$ `ESCAPE`*) as a control dependency. Therefore, Rule 2 above applies, and we have the updated WP as (*$src[*i] \neq$ `DASH`*)*$\wedge\neg$ *(*$src[*i] ==$ `ESCAPE`*). Proceeding in this fashion, we compute the WP as :*

$(src[*i] \neq$ `DASH` $) \wedge \neg(src[*i] ==$ `ESCAPE` $) \wedge (src[*i] \neq$ `delim)` $\wedge(src[*i] \neq$ `ENDSTR`$)$ $\square$

3.2.2.2 Comparing program branches across program versions

We now proceed to discuss the methodology to identify the candidate branches in the evolved version. We assume that the mapping of the branches (either at the source or the assembly level) of version V1 to those in V2 is already available. A mapper performs a more detailed analysis of these branches before the final mapping. The discussion below provides the intelligence behind choosing a subset of those addresses in V1 which can be safely mapped to their counterparts in V2 without compromising on branch prediction accuracy. To do so, we first identify the possible scenarios to be considered for the two branch counterparts in the two program versions. For each branch pair B1 in V1 and corresponding B2 in V2, we consider the the following cases.

Case I: Branch Condition and WP unchanged:

This is the simplest and most common case, as seen in our experiments, for most branches. The change across two program versions is quite small in comparison to the whole code base. Hence for many branches there might not be any change in the branch condition as well as the WP. In such a case, we can safely inherit the prediction data from the older version, if available, and prevent the call to the branch predictor. For example, *Branch 6* at line number 19 in Figure 3.1(b) is not affected by the change between the replace versions. As mentioned earlier, the only change between the versions is inside the dodash function. This change does not influence the WP of *Branch 6* since the dodash function is called after *Branch 6*. In this case, we can reuse the profile of *Branch 6* from V1 in V2.

Case II: Branch Condition unchanged, WP changes:

This is a more complex situation. Here we have to determine whether the branch behaviour will be affected and if it is, then to what extent. In this scenario, we compute the WP for each branch condition first. Let us assume the WP for the branch in V1 is ϕ_1 and that for the corresponding branch in V2 is ϕ_2. Then we consider the following cases:

- $\phi_1 \Rightarrow \phi_2$: This means that ϕ_2 is a weaker condition than ϕ_1. In this case we can reuse the predictions from V1 as whenever ϕ_1 is true, ϕ_2 will be true.

  Example 3.6 *The WP for* Branch 2 *in V1 is :*

  $\phi_1 : (src[*i] \neq \texttt{DASH}\ ) \wedge \neg(src[*i] == \texttt{ESCAPE}\ ) \wedge (src[*i] \neq \texttt{delim)}\ \wedge (src[*i] \neq \texttt{ENDSTR})$

  On the other hand, for V2, the WP for Branch 2 *is :*

  $\phi_2 : (src[*i] \neq \texttt{DASH}\ ) \wedge (src[*i] \neq \texttt{delim)}\ \wedge (src[*i] \neq \texttt{ENDSTR})$

  Branch 2 *is now missing one clause and hence has been weakened. Intuitively, this means that* Branch 2 *will now evaluate to true on more input scenarios than before. This also means that whenever ϕ_1 is true, ϕ_2 will be true. So the predictions for* Branch 2 *in V1 will also be valid in V2, although the hit ratio might increase.* □

- $\phi_2 \Rightarrow \phi_1$: If ϕ_2 is a stronger condition than ϕ_1, for some cases when ϕ_1 is true, ϕ_2 will be false. Hence we may not be able to reuse the prediction information for this as easily as in the previous case. In such a scenario, we might consider inheriting the branch information based on some similarity index. In this work, we fix a threshold, say T, which quantifies a percentage of similarity. If ϕ_1 and ϕ_2 have a similarity above T, then we pass on the bias based branch prediction to the next level. This technique

```
1 void dodash(char delim, char *src, int i, char *dest,
int *j, int maxset)
2 {
3     while ((src[*i] != delim) && (src[*i] != ENDSTR))
4     {
5         if (src[*i] == ESCAPE) {      /* Branch 1* /
6          ...
7         }
8     }
9     while (src[*i] != DASH)           /* Branch 2* /
10    {
11        if (*j <= 1 ) {
12           if (src[*i + 1] == ENDSTR) /* Branch 3 */
13              ...
14           else                       /* Branch 4 */
15              ...
16        }
17         ...
18     }
19  }
```

(a)

```
1 void dodash(char delim, char *src, int i, char *dest,
int *j, int maxset)
2 {
3     while ((src[*i] != delim) && (src[*i] != ENDSTR))
4     {
5         if (src[*i] != DASH) {         /* Branch 1 * /
6             if (src[*i] == ESCAPE) {   /* Branch 2 * /
7          ...
8         }
9     }
10    while (src[*i] != delim)           /* Branch 3* /
11    {
12        if (maxset <= 1 ) {
13           if (src[*i + 1] == ENDSTR)  /* Branch 4 */
14              ...
15           else                        /* Branch 5 */
16              ...
17        }
18         ...
19    }
20  }
```

(b)

Figure 3.3: Modified program code snippet of *replace* program

is based on the intuition that if the two weakest preconditions are mostly similar, then it is highly probable that the branch in consideration still has the same bias and in V2 it will have a behaviour that is similar to that in V1. We now need to decide on how to measure the similarity of ϕ_1 and ϕ_2. Let

$$\phi_1 = Collection\ of\ clauses : (\alpha_1, \alpha_2, ..., \alpha_n) \text{ and}$$
$$\phi_2 = Collection\ of\ clauses : (\beta_1, \beta_2, ..., \beta_m)$$

Definition 3.5 Similarity Index: *We define the similarity index of ϕ_2 with respect to ϕ_1 as :*

$$\frac{Number\ of\ satisfying\ valuations\ common\ to\ \phi_1\ and\ \phi_2}{Total\ number\ of\ satisfying\ valuations\ for\ \phi_2}$$

■

Example 3.7 *We calculate the similarity index for ϕ_2 for the affected branch in V2. If this is more than the chosen threshold T, then we allow the bias based prediction to be re-applied in V2. In our experimentation a threshold of* 70% *worked well for us. Consider the program fragments shown in Figure 3.3.* Branch 1 *in V1 corresponds to*

Branch 2 *in V2. Weakest precondition of* Branch 1 *of V1 is :*

$\phi_1 : (src[*i] == \texttt{ESCAPE}\ ) \wedge (src[*i] \neq \texttt{delim)}\ \wedge (src[*i] \neq \texttt{ENDSTR})$

On the other hand, for V2, the WP for Branch 2 *is :*

$\phi_2 : (src[*i] == \texttt{ESCAPE}\ ) \wedge (src[*i] \neq \texttt{DASH}\ ) \wedge (src[*i] \neq \texttt{delim}) \wedge (src[*i] \neq \texttt{ENDSTR})$

So in ϕ_1, one clause is missing and hence becomes weakened. This implies for some cases when ϕ_1 is true, ϕ_2 may not be true. In this case we calculate the similarity index for ϕ_2 and decide accordingly. □

- $(\phi_1 \nRightarrow \phi_2) \wedge (\phi_2 \nRightarrow \phi_1) \wedge (\phi_1 \cap \phi_2 \neq \Phi)$: This implies that except the branch condition no other similarity or relationship (weak/strong) exists between ϕ_1 and ϕ_2. This scenario is much more difficult to analyze. Typically, this means there have been a lot of changes in the logic preceding the branch which led to significant restructuring and modifications. In this case as well, we fall back to the Similarity Index based comparison, as discussed in the previous case.

Apart from the two cases discussed above, we do not reuse the prediction profile across versions and allow the normal calls to the dynamic branch predictor. Specially, this corresponds to cases where the branch condition changes (with the remaining WP changed or unchanged). For these cases as well, we can adopt a similarity indexed based analysis. At the end of this step, we isolate a subset from the set of branches identified in Step 1, as candidate branches that have not undergone any/much changes. These are passed to the the third step.

3.2.3 Selective dynamic predictor invocation :

From the previous two steps, we identify the candidate branches in the evolved program version and their bias branch direction that can be used as hint bits. We propose to relinquish the branch predictor for these identified branches and reuse the outcome profile from the previous version as the prediction. This saves a lot of dynamic predictor calls and possibly, compute cycles and energy. However, in case of a mis-prediction, we allow the normal rollback to happen as usual. We compare our performance in terms of prediction accuracy, time and energy consumption between the runs with and without relinquishing the branch predictor for the candidate branches, and observe considerable benefits as explained in Section 3.4.

3.3 Implementation

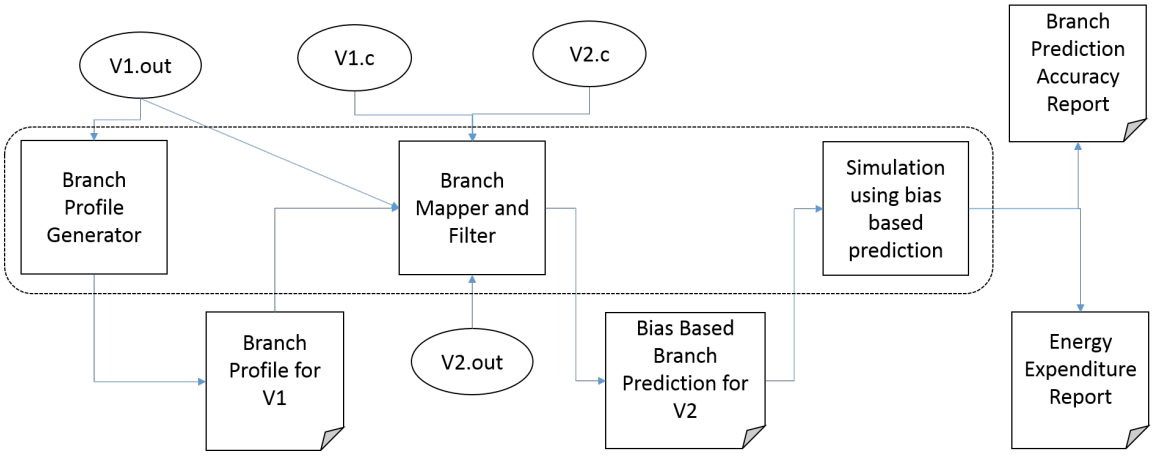


Figure 3.4: System Architecture

We now describe the implementation of our framework. The framework performs three major steps described below:

1. **Branch Profile Generation:** An architectural simulator, *Tejas* [77] is used to record the branch behaviors for the software version **V1** across multiple testcases.

2. **Branch Mapper and Filter:** Identifies the branches in **V2** for which one can actually use the bias based prediction as explained in the previous section and prevents calls to the branch predictor.

3. **Simulation using the bias based prediction:** The software version **V2** is run using the results from the previous stage for four branch predictors - GShare, PAp, TAGE and Tournament. For each, the prediction accuracy and energy improvement is recorded and contrasted against a normal run (without bias information) for **V2**.

Figure 3.4 shows an architectural view of our proposed framework. In this framework, we use the Tejas Simulator [77] and Frama-C [2] as discussed in Section2. The steps are discussed in detail below.

3.3.1 Branch Profile Generation

This is the first stage of the framework. The older program version **V1** is compiled into an executable and run using Tejas. The simulator is modified to record the behavior of each branch for different prediction strategies. In this work we consider 4 different branch prediction strategies i.e. GShare [78], PAp[120], TAGE [97] and Tournament[70]. In each run, for every individual branch we record the following : a) frequency of taken (condition of the branch is evaluated as false), b) frequency of not-taken (condition of the branch is evaluated as true) and c) bias hit ratio. We record these statistics for all the four predictors used. Thus, for each branch, we have 3 tuples. Only for those branches for which the bias hit ratio is more than or equal to the predictor hit ratio, across all test cases for the program, the branch address and bias based prediction are saved for future stages. The intuition behind this is that we do not intend to compromise on the accuracy of branch

prediction of the processor and wish to save energy as increase in mis-predictions will result in penalties and incur more delay and energy loss. It is to be noted that if for some branch, the bias based prediction accuracy is more than or equal to the predictor accuracy for some test cases and it is less for some other test cases, then those branches are ignored. Hence, only branches having a consistent bias are carried over to the next stages.

3.3.2 Branch Mapper and Filter

The Branch Mapper takes the source code of the two program versions, the executables for each and branch profile information from the previous stage. It then does the following:

Code Change Tracker: This module takes in the source code of the two software versions and maps the unchanged portions of the code. Basically it computes the difference for the two files and uses it to map the original source code lines to their location in the new version. It is to be noted that we do not map the changed or added or deleted lines in the new version. The reason behind this is that the behavior of the changed portion of the code, especially the branches, might have changed and hence reusing the bias based prediction for such branches might not be advantageous. At the end of this step we have a SourceMap which contains the mapped lines from **V1** to **V2**. This is done using a Python [7] script.

DumpReader: Next we use the executables for mapping each source line to its corresponding assembly level branch address. A Linux utility `objdump` [6] can dump the assembly code from the executable annotated with source code line numbers. A simple parser can therefore read such a file and store the source line to branch address map. This has to be done for **V1** as well as **V2**. We create a Python script to perform this task as well.

Branch Mapper: The Branch Mapper forms the core of this framework. Its task is to map the branch addresses from **V1** to corresponding branch addresses in **V2**. A simple mapper takes the SourceMap generated by the code change tracker module and the source

to branch address map for both versions from the previous module and using it maps all the branches in **V1** to **V2**. Though simple, this method might suffer from a lot of excess mis-predictions and hence extra energy expense. For instance, in the replace program in the previous section, *branch 2* in **V1** (Figure 3.1(a), line number 8) will be mapped to *branch 2* in **V2** (Figure 3.1(b), line number 5) and hence the bias based prediction, if available, will be reused for this. However, as discussed previously, the path condition for *branch 2* in **V1** has now been altered in **V2**. To augment the simple syntactic mapper as implemented in the code change tracker, we have a new mapper module in Python. This mapper uses a more formal methodology, as discussed in the previous section, to map the branches across two versions, using WP change analysis. Since computing the number of satisfying valuations for an arbitrary WP expression may be infeasible in practice, we approximated this with the number of clauses common in the WPs between the program versions. For Boolean and integer programs, this may be achieved using model counting [5], however, it is hard in general. Our approximation does not hurt as well since we still allow rollback in case of mis-prediction as usual.

3.3.3 Simulation using bias based prediction

The Branch Mapper gives as output the addresses and static predictions for those branches of **V2** which have not been affected by the software evolution or the extent of change is below a given threshold, and making such predictions would not harm the accuracy of the underlying branch prediction strategy. Next we need to reuse this information and use *Tejas* to run **V2** with these updates. We modify the source code of *Tejas* to be able to do this. The simulator updates this information in an internal data structure during initialization and calls the branch predictor only in case no hints are provided for a certain branch. This decision is made in the decode unit of the pipeline. The mechanism discussed here puts minimum load on the processor and tries to improve the energy efficiency and accuracy of

branch prediction with minimum hardware support. Simulation is done for different branch predictors as mentioned earlier. Each time *Tejas* is run, a simulation report is produced. We obtain the energy and branch prediction statistics from this report.

3.4 Evaluation

We now report our experience in using our method on various benchmarks.

3.4.1 Experience with Siemens benchmarks

The Siemens benchmark is a set of C programs [10] widely used in the program analysis community. Each program is associated with a set of faults and a set of test cases. We use our framework on these benchmarks and record the results. We provide the results on 6 Siemens benchmarks : replace, schedule, schedule2, printtokens, printtokens2 and totinfo. Two versions for each program are obtained from V1 and V2 sub-folders in the distribution. Lines of codes (LOC) and number of branches in V1 and V2 are shown in Table 3.1. Percentage of unaffected branches in V2 are also listed in Table 3.1, which are found in the Branch Mapper and Filter stages as described in Section 3.3. The branch profile is obtained by running the V1 executable in the Tejas Simulator for 50 test cases for each benchmark. As mentioned before, we only consider those branches for which the bias based prediction accuracy is equal or more than the predictor accuracy for all the runs. Branches which show any variation in their bias are removed from consideration. For these benchmarks, we find that a majority of branches are heavily biased. This allows us to apply our technique to most branches. For many branches, Frama-C is unable to compute the weakest preconditions due to various limitations pointed out in Section 3.5. In spite of these shortcomings, our results are quite encouraging as seen in Tables 3.2 and 3.3 for all the different predictors. Table 3.2 shows the results for 6 benchmarks for the PAp and

Programs	LOC		Number of branches		% of unaffected branches
	V1	V2	V1	V2	
replace	565	561	94	93	20
schedule	415	415	30	30	63
schedule2	311	311	41	42	47
totinfo	407	407	45	45	66
printtokens	727	726	40	39	50
printtokens2	564	571	78	80	20
Gzip	59547	60941	1636	1641	4

Table 3.1: Benchmark Detail

Programs	LOC	GShare				PAp			
		Energy		Prediction Accuracy(%)		Energy		Prediction Accuracy(%)	
		V2	V2'	V2	V2'	V2	V2'	V2	V2'
replace	565	55.9884	26.1664	61.1684	69.0722	55.9884	26.1664	69.0722	78.6942
schedule	415	43.4824	10.1972	46.0177	81.8584	43.4824	10.1972	59.292	72.5664
schedule2	311	119.6728	47.138	51.1254	60.9325	119.6728	47.138	60.1286	68.9711
totinfo	565	214.7184	45.214	51.2545	67.9211	214.7184	45.214	56.4516	72.8495
printtokens	727	801.5384	52.91	61.4978	94.4791	801.5384	52.91	85.6457	94.5271
printtokens2	564	962	395.1896	80.42	85.16	962	395.1896	82.28	83.48

Table 3.2: Energy Statistics and Prediction Accuracy for GShare and PAp

GShare predictors and Table 3.3 shows the same for the TAGE and Tournament Predictors. V2 represents the second version when run solely with the predictor and V2' represents the same version when run with the biased based prediction. The column labeled as Energy shows the *dynamic energy expenditure* measured by Tejas, given in Nanojoules. In all cases, we obtain a significant reduction in energy and the prediction accuracy is enhanced as well. This indicates that our Branch Mapping strategy improves system performance.

Programs	LOC	TAGE				Tournament			
		Energy		Prediction Accuracy(%)		Energy		Prediction Accuracy(%)	
		V2	V2'	V2	V2'	V2	V2'	V2	V2'
replace	565	55.9884	26.1664	50.1718	76.2887	55.9884	26.1664	68.0412	80.0687
schedule	415	43.4824	10.1972	50.885	78.7611	43.4824	10.1972	59.7345	72.1239
schedule2	311	119.6728	47.138	56.2701	61.8971	119.6728	47.138	60.4502	68.8103
totinfo	565	214.7184	45.214	65.233	73.2079	214.7184	45.214	60.8423	73.6559
printtokens	727	801.5384	52.91	88.4061	94.7672	801.5384	52.91	85.7657	94.5511
printtokens2	564	962	395.1896	81.14	85.16	962	395.1896	83.58	85.28

Table 3.3: Energy Statistics and Prediction Accuracy for TAGE and Tournament

Predictor	Energy		Prediction Accuracy(%)	
	V2	V2'	V2	V2'
PAp	1426.646	646.2716	87.5927	87.7815
Gshare	1426.646	646.2716	78.8672	89.8314
TAGE	1426.646	646.2716	83.1962	88.6042
Tournament	1426.646	646.2716	87.8085	88.0243

Table 3.4: Results for Gzip

3.4.2 Experience with Gzip

We use our framework on Gzip versions 1.5 and 1.6 [3] for our experiments and apply our methodology over it to test the scalability of our framework. Gzip version 1.5 contains 59547 LOC and 1.6 contains 60941 LOC. Since this is a multi file source repository, we apply the code change tracker over all the files in the two repositories pairwise and recursively. Each version has around 1600 branches as shown in Table 3.1. Due to limitations of Frama-C, we are able to compute the weakest preconditions for 600 of these. Table 3.1 shows the percentage of unaffected branches in Gzip-1.6. Table 3.4 shows that our method works quite well in this case as well. The reduction in energy (nanojoules) is significant for Gzip using our strategy. The branch profile obtained from Gzip-1.5 is applied to the simulation of Gzip-1.6 using Tejas.

3.5 Discussion and limitations

Observing from our experiments, we can conclude that our method works quite well in practice. We even manage to improve the accuracy of the branch predictors by our method and energy expenditure is reduced as well. The Branch Mapper maps the branch addresses very efficiently and in reasonable time. In this phase it is assumed that the source to assembly mapping can be obtained from the executable itself. This requires the programs to be compiled with the "-g" option (debugging option). In large softwares, this might not

be the default case and hence their build configurations may need to be changed.

Some branches (in the two versions) can be semantically same but their syntax may have changed somehow. For example, the order of the conditions might have changed or multiple conditions may have been split into multiple lines. Such changes make it more challenging to map these branches at the source code level. Currently the code change tracker ignores them and leaves them from consideration of the developer.

For computing the weakest precondition at the source code level, Frama-C poses some problems while handling some specific programming controls which forced us to leave them. For example, disjunction and function constraints had to be discarded.

3.6 Summary

In this work, we demonstrate how the branch profile of an older (preferably preceding) software version can be reused for branch prediction of a future version. This reduces the burden on the developer as now he does not need to profile the software from scratch every time a new version is released, which is both time consuming and inefficient. We present a formal methodology that can easily inherit the information from a previous software version and make it usable for the next version. Experiments show that our technique produces promising results. We also observe that in all the cases, the prediction accuracy is also improved. This shows that there are indeed many branches which the predictor finds very hard to predict and which are heavily biased as well. This phenomenon gives us as an extra bonus in improving the accuracy of the predictor.

Chapter 4

A two-component dynamic predictor design with shared predictor tables

4.1 Introduction

In the previous chapter, we proposed a strategy to improve energy consumption by relinquishing the invocation to dynamic predictors for some branches of a given program, utilizing outcome histories from previous software versions. The motivation of this chapter is to design a storage and energy efficient multi-component branch predictor, more specifically, a two-component branch predictor, to achieve better performance than a single component, at comparable storage. It is widely acknowledged that a single component predictor can cater to branches of a specific type, however, for a program, in general, branches exhibit widely varying characteristics, and history patterns, which may be difficult for a single predictor to keep track of and learn from. Multi-component predictors, with predictor components,

individually catering to history patterns of specific types, and employing suitable learning schemes, have therefore been proposed in literature and adopted in the pipeline of most modern commercial processors. The objective of this chapter is to explore the design of a 2-component predictor, with a focus of limiting their combined storage footprint, while enjoying the benefits of improved accuracy, due to the presence of the two components.

As discussed in Chapter 2, a classification of dynamic predictors is based on the information they use about other branches for predicting a specific branch at run-time. A *local* predictor uses history information only about the branch under consideration for its current prediction, while a *global* history-based predictor takes into account the direction histories of the preceding branches in addition to the present one while making a prediction for a specific branch. Combination of a local and a global predictor have been shown to have more prediction accuracy in contrast to any single component predictor [69]. However, these predictors have a major drawback, they require more space than a single predictor. These predictors have therefore not been able to make way into low storage processors.

In this work, we aim to propose a storage-efficient design for a two-component hybrid predictor. We base our study on the design of one of the most popular two-component hybrid predictors, namely, the Alpha 21264 [69], described in Chapter 2. This predictor consists of a local and a global predictor component and uses a sophisticated tournament prediction scheme to choose the final prediction between these predictors at run time. The local and global components maintain separate Pattern History Tables (PHT) to store prediction information, which is later used for predicting directions of future instances of branch instructions. This ensures the predictors can operate in their individual spaces, without interfering in predictions of each other.

Our first attempt towards a storage-efficient two-component predictor design is to examine if these local and global components can operate on a single prediction table. Indeed, the structures of the tables are similar, hence sharing of a single table is possible. We

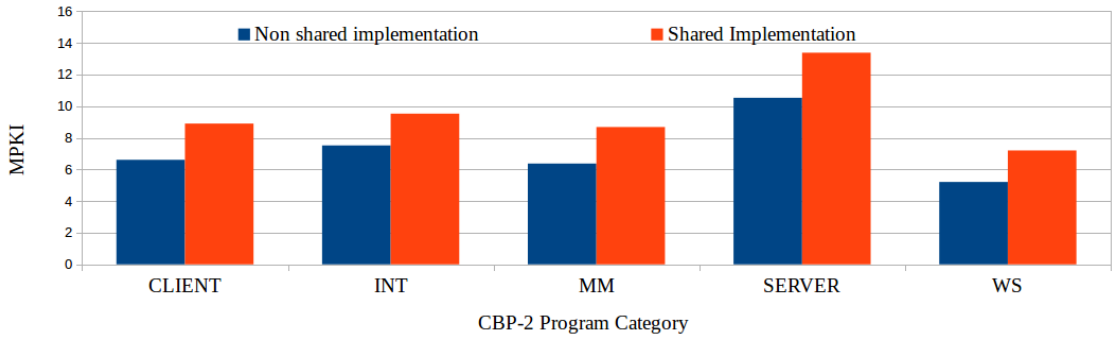


Figure 4.1: Average MPKI of Non shared and Shared implementation

implement and simulate this shared table multi-component design on top of the popular Championship Branch Prediction-2 (CBP-2) traces [33] to examine the performance difference in terms of prediction accuracy. Figure 4.1 presents the average Misprediction per kilo instruction (MPKI) of the shared implementation and the non-shared one for some of the CBP2 benchmark program categories [33]. In general, there is a decline in prediction accuracy and increase in mis-prediction, when we move to a shared implementation from the non-shared one. An in-depth analysis reveals that the cause is extensive interference on the shared PHT on which both the local and global predictors work. The prediction information of the global predictor stored in the PHT is at times, modified by the local predictor, and vice-versa, leading to the interference, and therefore, accuracy degradation.

CBP-2 Program category	Maximum number of PHT entries interfered
CLIENT	256
INT	205
MM	251
SERVER	254
WS	253

Table 4.1: Interference Statistics on CBP2

The main motivation behind our design proposed in this work is to resolve this interference between the predictor components with minimum additional storage as compared to the shared single-table implementation. The essential idea behind our design is to use an

additional small storage to handle only the interfering entries. To fix the size of this, we examine the indices of the PHT being interfered upon by both the components, and as expected, the number of entries suffering from interference is quite less in comparison to the total PHT size. Table 4.1 shows the maximum number of PHT entries that are victims for some of the CBP-2 program categories. Our contributions are summarized below.

Contributions of this work

- We propose a two-component dynamic predictor design with a single shared PHT and discuss the interference therein.
- As a solution to handling interference, we propose an additional de-aliasing cache to specifically cater to the interfering indices, while the operation of the PHT remains shared between predictors otherwise.
- We present a complete implementation of our proposal on top of CBP-2. Experimental results in Section 4.5 show that our proposed implementation can indeed boost performance when compared to the shared table implementation. More importantly, for some of the benchmarks, our design achieves comparable performance in terms of prediction accuracy in comparison to the non shared implementation.
- Further, we present synthesis results produced using the Synopsys DC compiler and TSMC 90 nm libraries to show the energy and area benefits of our design.

The rest of the chapter is organized as follows. Section 4.2 presents our two-component predictor design with a shared PHT. Section 4.3 elaborates our architecture, while Section 4.4 discusses the overall principle. Section 4.5 presents the implementation details of our proposed design, while in Section 4.6, we show results on the CBP-2 traces. Synthesis report of the proposed design is discussed in Section 4.7 and Section 4.8 concludes this discussion.

4.2 A two-component predictor with shared PHT

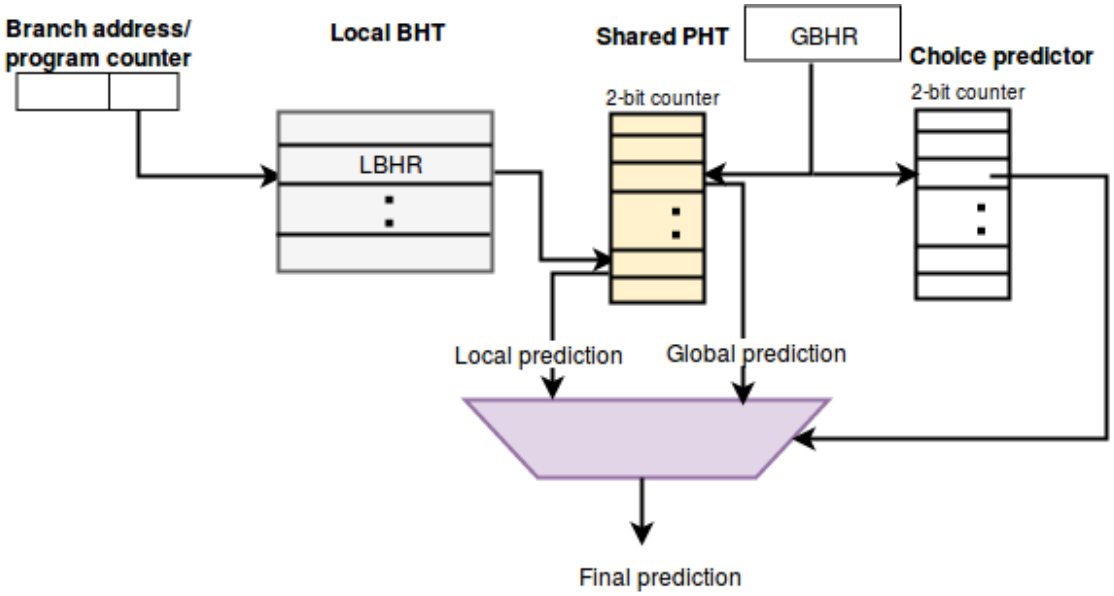


Figure 4.2: Two-component branch predictor with Shared PHT

The main motivation of this work is to design a two-component predictor with a shared PHT. This predictor combines a local and a global predictor component together as shown in Figure 4.2. The different components are described below.

4.2.1 Predictor components :

In this case, the local and the global predictor components operate independently. We consider a design with a single PHT table, shared by both the local and the global predictors as shown in Figure 4.2. The local predictor uses a Branch History Table (BHT) to store the branch specific history patterns (branch outcomes). The Local Branch History register (LBHR) corresponding to a branch stores a *n-bit* pattern corresponding to the last n outcomes of the branch. The BHT is of size $2^p X n$. For a given branch, the BHT is indexed by the last p bits of the branch address or the Program Counter (PC) value and the *n-bit*

LBHR value stored at that index is used to index the shared PHT. Each entry in the shared PHT stores a 2-bit saturating counter value, the most significant bit of this counter value is taken as the predicted direction of a branch instance as discussed below. Therefore, the size of the PHT is $2^n X 2$ bits. The global predictor uses a *n-bit* Global Branch History Register (GBHR) to index its PHT. The number of bits in LBHR should be equal to the number of bits in GBHR. The GBHR stores a running outcome history of the n previous branch instances. When a branch is issued, the predictions of the local and the global predictors are read from the corresponding indices of the shared PHT. When a branch retires, the corresponding LBHR, GBHR and the 2-bit counter are updated with the actual outcome to train the predictors for future prediction. This shared PHT architecture results in interference between the predictors, with both predictors sometimes accessing the same PHT entry for different branch instances, and thereby modifying the predictions stored by one another, leading to accuracy degradation. We discuss the interference in Example 4.1 to motivate the addition of the cache.

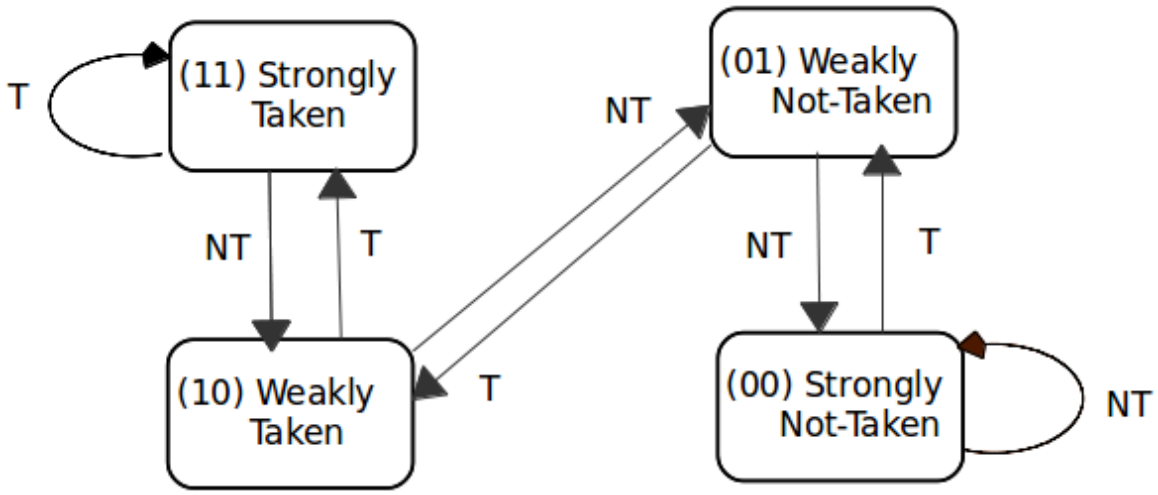


Figure 4.3: 2-bit Saturating Counter

2-bit Saturating Counter : The 2-bit saturating counter is considered as a finite state machine for most of the dynamic branch predictor designs [119][118]. It has four different states (00, 01, 10 and 11) defined by the 2 bits of it as shown in Figure 4.3. The counter transitions from one state to another in response to a taken (T) or not-taken (NT) outcome resulting from the execution of one or more branch instructions that are assigned to the index value of the predictor. Each bit of the two-bit counter plays a different role. The most significant bit, called the direction bit is used to track the direction of branches. If the counter is in states 01 or 00, the branch is predicted as NT. When it is in states 10 or 11, the prediction is T. The least significant bit provides a hysteresis which prevents the direction bit from immediately changing when a mis-prediction occurs.

4.2.2 Choice Predictor :

To select between the local and global predictions for a branch, a choice predictor is used. The choice predictor is a choice pattern table of size $2^m X2$ bits, indexed by the m bit GBHR. Each entry of this table stores a 2-bit counter whose MSB stores the decision of the current best predictor (either global or local) for every branch. If the counter state is 00 or 01, the prediction from the global predictor is considered as the final prediction. If the counter state is 10 or 11, the local prediction is taken as the final prediction. When the branch outcome is known, and the predictions from the two predictors are different, this counter is updated [69]. The counter state is decremented if the branch outcome matches with the global prediction, otherwise it is incremented.

Example 4.1 *Consider a design with a local BHT of size* $2^{10}X12$ *and a shared PHT of size* $2^{12}X2$*. Here, BHT is indexed by the lower 10 bits of the branch address and the shared PHT is indexed by a 12 bit GBHR and LBHR value. Figure 4.4 shows the assembly code of a C program fragment. Let us assume for Branch 1, the local predictor is selected for the*

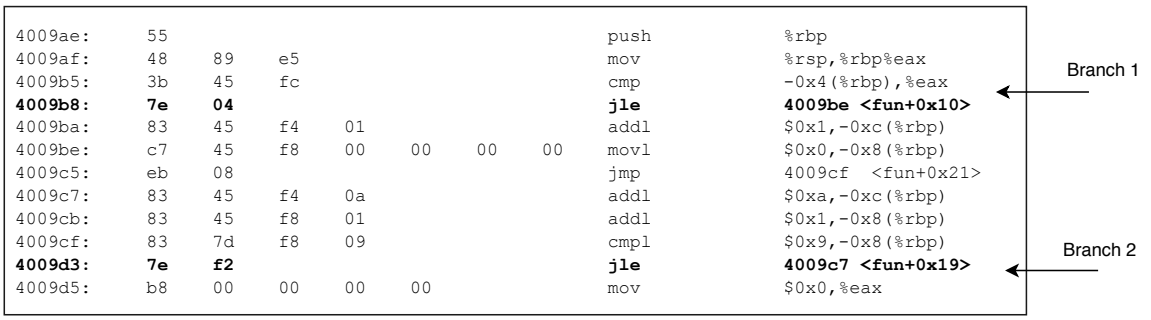


Figure 4.4: Assembly code snippet for branch instructions

final prediction, at retire stage it keep 01 as a 2-bit counter value in the shared PHT index with a LBHR value- 100010111100. During the prediction of Branch 2, the global predictor accesses the same PHT index with a GBHR value - 100010111100. In this case, the global predictor will give the prediction according to the prediction information stored by the local one (for Branch 1) which is not-taken. Now the current state of the 2-bit counter in the choice predictor table at index 100010111100 is 01. The MSB of this counter selects the global predictor for the final prediction for Branch 2. At retire stage, branch outcome is known as taken, the global predictor increments the 2-bit counter value and it becomes 10. Now the 2-bit counter state stored by the local predictor for Branch 1 is updated/changed by the global predictor for another branch instance Branch 2. This is an inter predictor interference caused due to sharing of the same PHT table by the two different predictors.□

In the following, we present our modified designs to resolve these interferences.

4.3 An improved design with PHT ownership information and a dealiasing cache

For interference resolution, we propose to augment our design with the following.

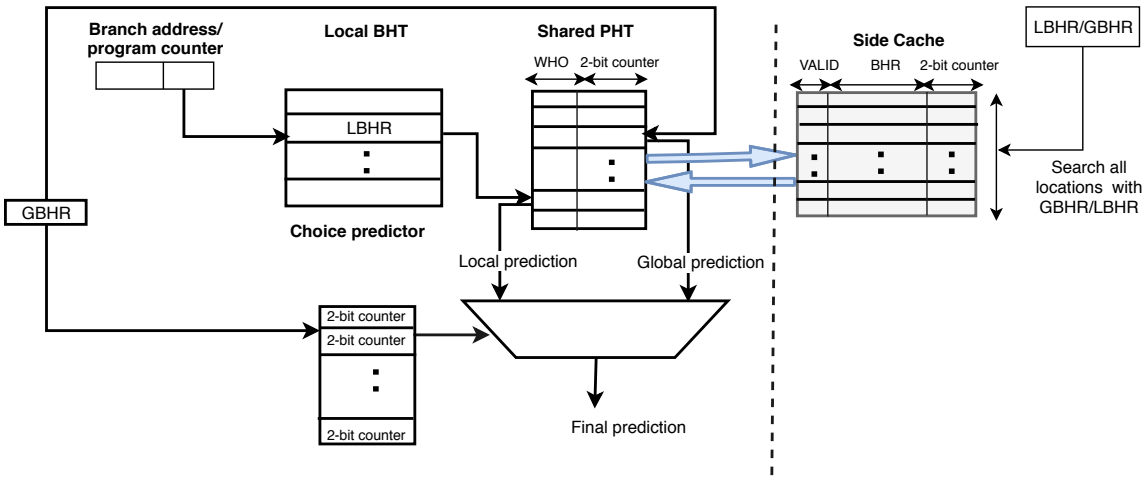


Figure 4.5: Our proposed architecture

- Ownership information inside PHT entries
- A dealiasing cache

We explain the detail in the following.

4.3.1 Shared PHT modification

The first task in enabling a shared PHT structure without interference between predictors is to come up with an interference detection mechanism. To achieve this in our design, each entry in the shared PHT is associated with an *ownership* bit, which holds the current owner of the prediction. A predictor can access the prediction information at a particular PHT entry only if it owns the prediction from a previous retire stage or nobody owns it. Ownerships for all PHT entries are initially set to 0, thus the global predictor has the ownership by default. However, this ownership can change as discussed later, our design tries to ensure that whoever accesses the entry first, becomes the owner and that disallows the other component from accessing the entry. This is in spirit of the *first touch* allocation

policy [72] commonly used in Operating System Kernels [116]. If an owned entry is accessed by a non-owning component, a side cache entry is created to serve the latter, so that both the components can operate independently. This ensures all interferences are nullified. To this effect, we modify the shared PHT structure to include an ownership bit (called WHO bit as shown in Figure 4.5) for each PHT entry. Since we have 2 predictor components, a single bit is sufficient to indicate which predictor stored the prediction information into this entry last time. We adopt the following convention. If the WHO bit of a PHT entry is 0, it indicates that the global predictor owns this entry, otherwise the local predictor is the owner. The size of the shared PHT becomes $2^n X 3$ bits (2^n entries, each storing the WHO bit and the 2-bit counter value) as shown in Figure 4.5.

We now describe the structure of the side cache, as shown in Figure 4.5.

4.3.2 A dealiasing side cache

The role of the side cache is to hold the prediction information of all PHT entries for which interferences occur. In the side cache, each entry is associated with some PHT entry on which an interference occurs. To this effect, whenever an entry is created in the side cache (as discussed below), the corresponding PHT index (GBHR or LBHR) is stored as well. To store the prediction information, each entry contains a 2-bit counter. Additionally, each entry contains a valid bit to indicate whether this entry contains a valid PHT entry or not. Initially, all entries are invalidated. In this case, if the width of GBHR and LBHR is n-bits each, the size of each entry in this side cache is (n+3) bits (1 valid bit + n bits for GBHR/LBHR + 2 bits for counter). For our purpose, we keep a small fully associative side cache. In a fully associative cache, no index is needed since a cache block can go anywhere in the cache. Each entry stores a tag value which is an unique identifier for each cache entry. In our design, GBHR/LBHR is considered as the tag-bit. Every tag must be compared when finding a block in the cache. If the tag matches, the MSB of the 2-bit counter stored

in the side cache index gives the prediction information. We use the Least Recently Used (LRU) [79] replacement policy to replace an entry in the side cache. In our experiments, we present results with different cache sizes.

4.4 Overall working principle of our design

We now explain the overall working of the proposed design. In a pipeline, branch prediction usually relates to two different pipeline stages, in addition to the stage where the condition is evaluated:

- The *fetch* stage, when a predictor predicts the branch direction based on the history information available, and
- The *update* stage when a branch retires, the actual branch outcome is known and all history information are updated to train for future prediction.

In this work, we first propose a methodology that works on our proposed side cache implementation and eliminates any interference that occurs on the shared PHT. A detailed discussion on this methodology is as follows.

4.4.1 Elimination of all interferences (Methodology-1)

We describe the modifications in the fetch and retire stages.

Fetch Stage

When a branch instance is fetched, the two predictors are active simultaneously and access the shared PHT to retrieve the predictions for a branch. Based on the current state of the

choice counter stored in the Choice predictor (indexed by global BHR), the final prediction is taken from one of them as discussed before (shown in Algorithm 1). The Choice predictor is updated at the retire stage and discussed later.

ALGORITHM 1: Prediction at fetch cycle

```
1  begin
2      cindex = an index of the Choice predictor, calculated using GBHR
3      chCtr = 2-bit counter stored at cindex in the Choice predictor
4      gpred = prediction from the global predictor
5      lpred = prediction from the local predictor
6      pred = final prediction
7      if chCtr = 00 or chCtr = 01 then
8          pred = gPred
9      else
10         /*chCtr = 10 or chCtr = 11*/
11         pred = lPred
```

Accessing the prediction values for the predictors

We explain first the working of the global component. As discussed earlier, the shared PHT stores a WHO bit for each entry to record which predictor updated this entry most recently. Based on the current value of the WHO bit, the global prediction is taken. Algorithm 2 shows the detailed steps of extracting a global prediction. Initially all WHO bits are 0 and all 2-bit counters are initialized to 00. Hence, 0 as the WHO bit for a PHT entry is interpreted to be one of the following situations:

- No predictor has accessed this PHT entry as yet
- The global predictor owns this entry, i.e. it recorded a value here in its last access

If a predictor finds an entry there with WHO bit 0, it takes the MSB of the current 2-bit counter value as the global prediction. Since the 2-bit counter is initialized as 00, its

MSB also gives the default prediction as *not-taken*. The detailed steps of how the predictor components generate their predictions are shown in Figure 4.6 . In this flow diagram, steps for the global predictor are shown with dashed lines that follow each step of Algorithm 2 as discussed below. The step numbers of Algorithm 2 are written within brackets.

- For a branch instance, the global predictor looks up the corresponding *gindex* (gindex = GBHR) entry of the shared PHT (Steps 5 and 6).

- If it finds an entry there with WHO bit 0, it takes the MSB of the current 2-bit counter value as the global prediction gPred (Steps 7 and 8).

- If the WHO bit of this particular PHT entry is 1, it indicates that this is owned by the local component (Step 9). In such a case, the global component looks up all the entries of the side cache whose VALID bits are 1 with the same *gindex* (Steps 9 and 10) to check for a valid match. The side cache being content addressable, this search is quite efficient.

- If an entry is found in the side cache, the MSB of the 2-bit counter stored in that entry is considered as the global prediction (Steps 11 and 12),

- If no entry is found in the side cache, 00 is taken as the prediction (Step 14) for the global component.

A similar activity is carried out for the local predictor as well, except that in this case, the shared PHT is accessed with the LBHR and the side cache is looked up when the WHO bit is 0. The working methodology of the local predictor is described in Algorithm 3 and the overall flow is described in Figure 4.6. The solid lines of this flow diagram represent the detailed steps of Algorithm 3.

ALGORITHM 2: global Prediction at fetch cycle

```
1  begin
2      T : Shared PHT
3      Side cache : S
4      GBHR : Global BHR
5      gindex = an index of the shared PHT table (T)/* gindex is calculated using the
         indexing function of the global predictor (GBHR) */
6      Access the T[gindex] entry and find the WHO bit stored in that entry.
7      if WHO = 0 then
8          gpred = MSB of the T[gindex].2BitCounter
9      else
10         Search the side cache (S) for GBHR
11         if Side cache entry available for that GBHR with VALID = 1 at sindex then
12             gpred = MSB of S[sindex].2BitCounter
13         else
14             gPred = 0. /*side cache entry not present, take default as not-taken */
```

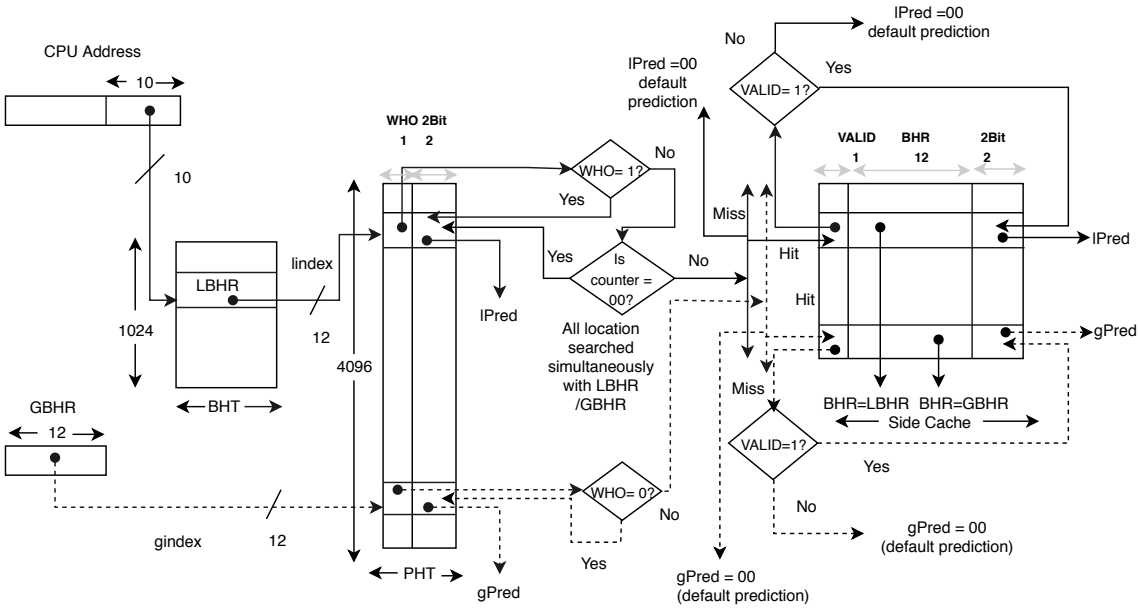


Figure 4.6: Steps for predictions at fetch stage

Branch retire and update stage

Creation of an entry in the side cache and updation of prediction information occur at the branch retire stage, when the outcome of the branch is known. The predictions from the

ALGORITHM 3: Local Prediction at fetch cycle

```
1  begin
2      L1 : local BHT
3      T : Shared PHT
4      Side cache : S
5      Addr = Lower 10 bits of PC
6      LBHR = L1[Addr] /*Local BHR */
7      lindex = LBHR
8      Access T[lindex] entry and find the WHO bit stored in that entry.
9      if WHO = 1 then
10         lpred = MSB of the T[lindex].2BitCounter
11     else
12         /* WHO = 0*/
13         if T[lindex].2BitCounter = 00 then
14             lpred = MSB of T[lindex].2BitCounter /* No predictor has accessed this
                entry before */
15         else
16             Search S for LBHR
17             if Side cache entry available for that LBHR at sindex with VALID = 1
                then
18                 lpred = MSB of S[sindex].2BitCounter
19             else
20                 lPred = 0. /*side cache entry not present, take default as not-taken*/
```

global prediction *gpred* and local predictor *lpred* are known previously from the fetch stage. The state of the choice counter stored in the Choice predictor (indexed by global BHR) indicates the predictor chosen for the final prediction in the corresponding fetch stage and the corresponding update method is called. The working steps of the Choice predictor are described in Algorithm 4, the corresponding flow is shown in Figure 4.7 and discussed below. The numbers written within brackets represent the line numbers of Algorithm 4.

- The state of the choice counter is checked. If that state is 00 or 01 (Step 7), the global predictor gave the final prediction and the corresponding update method (Algorithm 5) is invoked (Step 8).
- If this state is 10 or 11, the local predictor gave the final prediction for the branch

and the corresponding update method (Algorithm 6) is invoked (Step 10).

- The state of the choice counter is updated if and only if the predictions provided by the two predictors are different (Step 11).
- The state of the choice counter is decremented if the actual branch outcome is same as *gpred* (Steps 12 and 13), and incremented if the actual branch outcome is same as *lpred* (Steps 14 and 15).

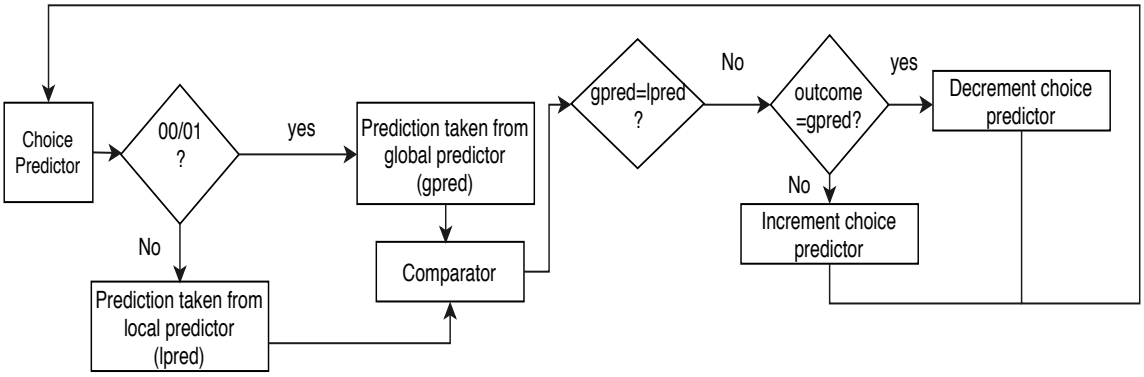


Figure 4.7: Steps for the Choice predictor

Cache entry creation and PHT / cache update

Algorithm 5 describes the steps to update the shared PHT entry / cache entry associated with the global predictor. The overall flow of this predictor corresponding to line numbers shown in Algorithm 5 is shown in Figure 4.8 using dashed lines and discussed below. Consider *gindex* as the index of the shared PHT for the global predictor. The numbers inside brackets represent line numbers of Algorithm 5.

- With the *gindex*, the shared PHT is accessed (Step 2).
- If the WHO bit of the accessed PHT entry is 0 (Step 3), the 2-bit counter of the entry is now updated according to the branch outcome (Step 4). If the branch outcome is

ALGORITHM 4: Updation at retire stage

```
1  begin
2      cindex = an index of the Choice predictor, calculated using GBHR
3      chCtr = 2-bit counter stored at cindex in the Choice predictor
4      gpred = prediction from the global predictor from fetch stage
5      lpred = prediction from the local predictor from fetch stage
6      actual = actual branch outcome resolved at retire stage
7      if chCtr = 00 or chCtr = 01 then
8          update using Algorithm 5 - "Side cache entry creation and global Update
           retire stage"
9      else
10         update using Algorithm 6 - "Side cache entry creation and local Update retire
           stage"
11     if gpred ≠ lpred then
12         if gpred = actual then
13             Decrement chCtr
14         else
15             Increment chCtr
```

taken, the 2-bit counter value is incremented, otherwise it is decremented as shown in Figure 4.8 (c).

- If the WHO bit at this entry is 1 (Step 5), the side cache is searched with the GBHR pattern (Step 6).

- If a valid entry is found for that GBHR, we update the 2-bit counter value stored in that entry according to the branch outcome (Step 8).

- If no valid entry is found in the cache for that GBHR, an empty location is searched for in the cache to store the prediction information for the global predictor (since local predictor stored the prediction information in the shared PHT) (Step 10). The overall flow of this step is shown in Figure 4.8 (b) and discussed below.

  – If an empty entry is found (Step 11), we first mark the VALID bit of that entry as 1 (Step 12), insert the GBHR (Step 13), initialize the 2-bit counter stored as

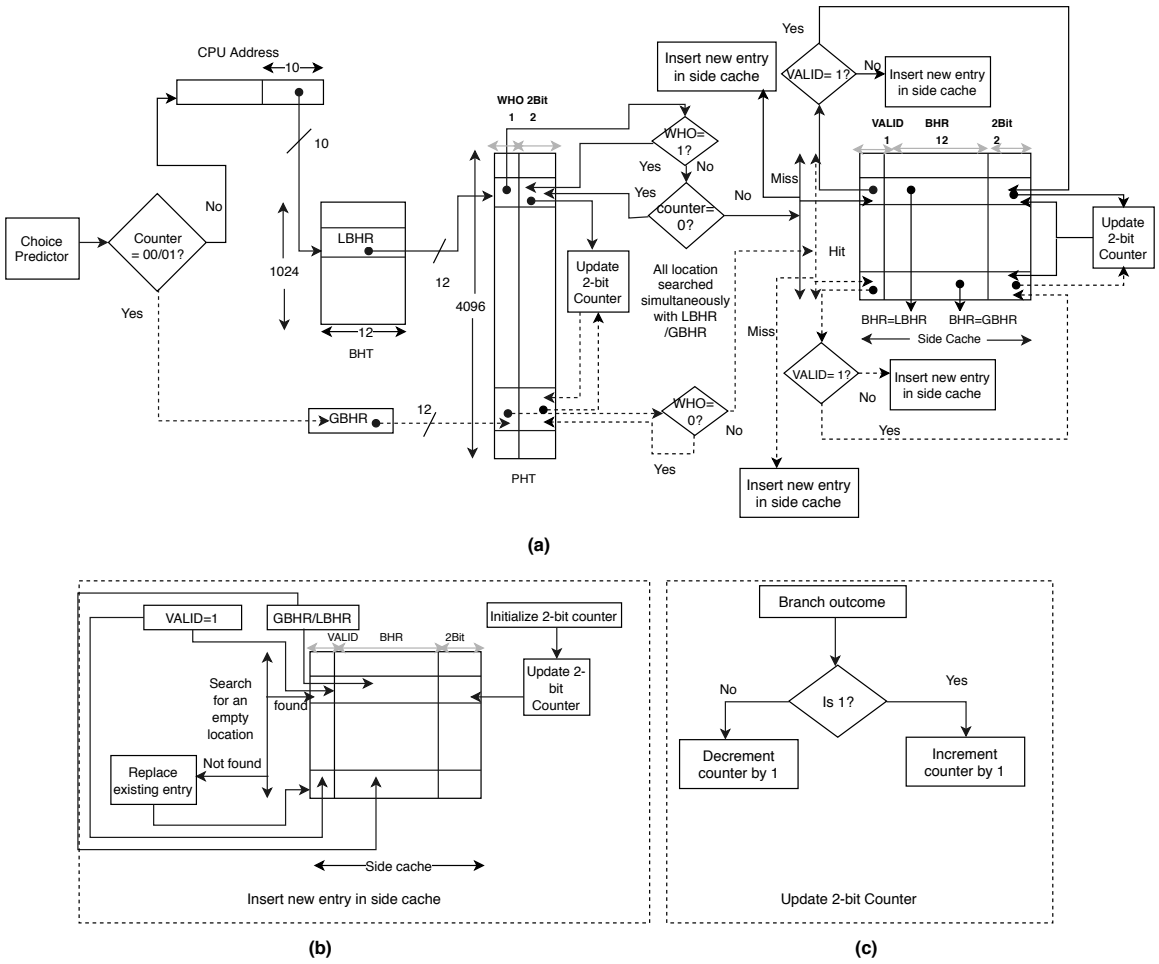


Figure 4.8: Steps for Cache entry creation and PHT / cache update

00 there (Step 14) and finally update the 2-bit counter value according to the branch outcome (Step 15).

– If no empty entry is found indicating that the side cache is full, replacement of an existing valid entry is needed. Replacement can be done using any replacement policies like LRU, FIFO etc (Step 17). After replacement of an entry, the GBHR (Step 18) and the default value 00 of the 2-bit counter are inserted into that entry (Step 19). After that, the 2-bit counter is updated according to the branch outcome (Step 20).

- Finally, the GBHR is updated according to the actual branch outcome (Step 21).

ALGORITHM 5: Side cache entry creation and global updation at retire stage

```
1  begin
2      Find PHT entry T with GBHR
3      if WHO bit of T is 0 then
4          Update 2-bit counter of PHT entry with actual outcome
5      else
6          Search the side cache (S) for GBHR
7          if side cache entry S available for that GBHR with VALID = 1 then
8              Update 2-bit counter of S with the actual outcome
9          else
10             Search for an empty location /*side cache entry for that GBHR is not
                present, Insert new entry in side cache*/
11             if An empty entry E is found then
12                 Set VALID bit of E as 1
13                 Insert GBHR at E
14                 Initialize 2-bit counter of E as 00
15                 Update 2-bit counter of E with the actual outcome
16             else
17                 Replace an entry R
18                 Insert GBHR at R
19                 Initialize 2-bit counter of R as 00
20                 Update 2-bit counter of R with the actual outcome
21     GBHR is updated according to the actual branch outcome
```

A similar activity is carried out for the local component, as described in Algorithm 6 except for the following: a) the PHT is indexed with the LBHR, b) the side cache creation or update is done when the WHO bit of the particular PHT entry is 0 and c) both the GBHR and the LBHR are updated with the actual outcome when a branch retires. In this case, GBHR update is needed since the Choice predictor works on the GBHR. Additionally, if the local component finds that the WHO bit of the PHT entry is 0 and the prediction stored therein is 00 (either due to initialization or as stored by the global component), we transfer ownership to the local component by setting the WHO bit of the PHT entry as 1. This does not harm the global component even though it might have stored 00 in this

PHT entry, since when it returns with the same PHT entry and finds the WHO bit as set, it does not get a side cache entry and hence, restarts with the default prediction as 00. The detailed flow diagram is shown in Figure 4.8 (solid lines), the numbers correspond to the line numbers shown in Algorithm 6. Example 4.2 shows how the *first touch* allocation policy works on our shared PHT design.

ALGORITHM 6: Side cache entry creation and local updation at retire stage

```
1  begin
2      Find PHT entry T with LBHR
3      if WHO bit of T is 1 then
4          Update 2-bit counter of PHT entry with actual outcome
5      else
6          if T[lindex].2BitCounter = 00 then
7              /* No predictor uses this location, transfer ownership to Local */
8              Update 2-bit counter of PHT entry with actual outcome
9              T[lindex].WHO = 1
10         else
11             Search the side cache (S) for LBHR
12             if Side cache entry S available for that LBHR with VALID = 1 then
13                 Update 2-bit counter of S with the actual outcome
14             else
15                 Search for an empty location /*side cache entry for that LBHR is not
                    present, Insert new entry in side cache*/
16                 if An empty entry E is found then
17                     Set VALID bit of E as 1
18                     Insert LBHR at E
19                     Initialize 2-bit counter of E as 00
20                     Update 2-bit counter of E with the actual outcome
21                 else
22                     Replace an entry R
23                     Insert LBHR at R
24                     Initialize 2-bit counter of R as 00
25     LBHR is updated according to the actual branch outcome
```

Example 4.2 *We revisit Example 4.1 where interference occurred since the local and the global predictor accessed the same shared PHT index for Branch 1 and 2 respectively. In*

our methodology, an extra WHO bit is stored in each entry of the shared PHT to indicate the predictor ownership. We now explain the working of our method. □

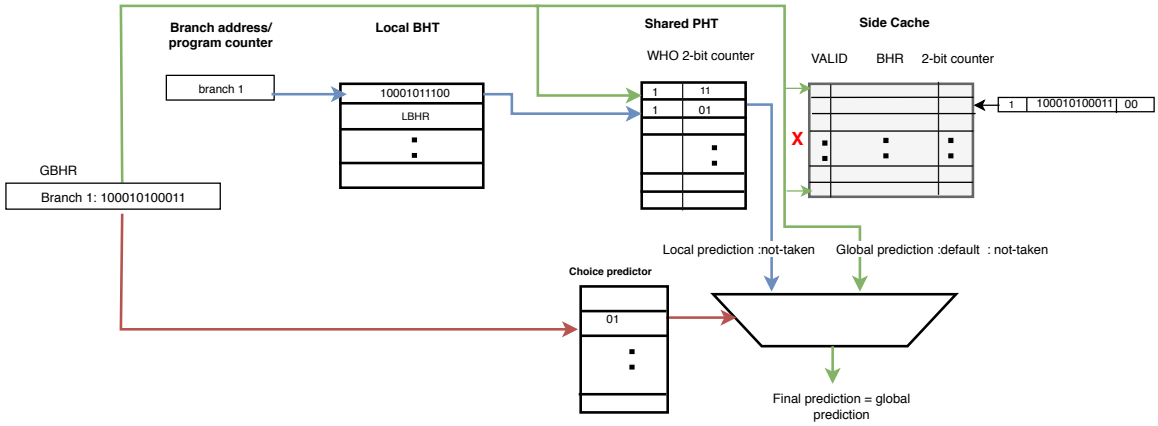


Figure 4.9: Fetch and Retire stage of Branch 1

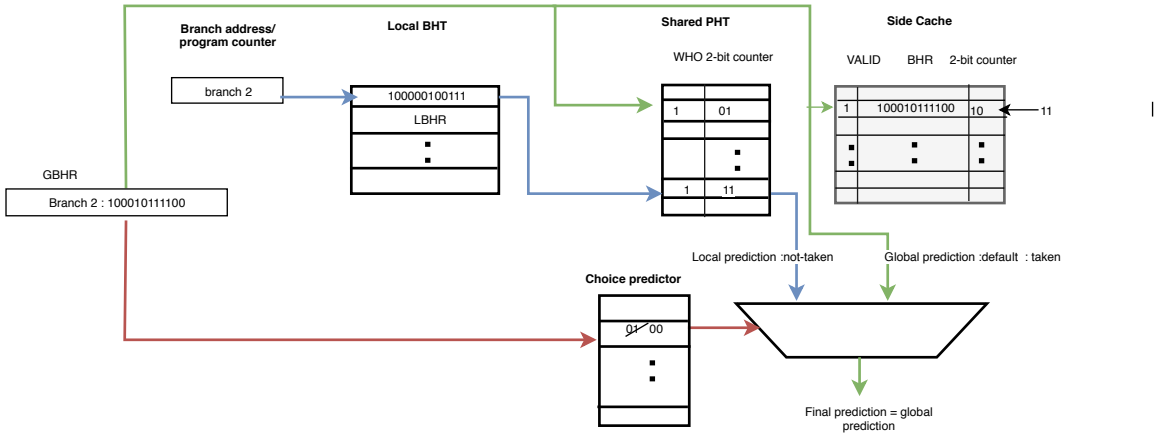


Figure 4.10: Fetch and Retire stage of Branch 2

Fetch stage of Branch 1 : *We present a detailed walk through of the fetch stage. We begin with Branch 1. Let us assume the local predictor searches the shared PHT with a LBHR 100010111100. The WHO bit stored at this entry is 1, hence the local prediction will*

be taken from the MSB of the 2-bit counter stored in it. Now the 2-bit counter value stored in it is 01, hence the local prediction for Branch 1 is not-taken. Further assume for the same branch, the global predictor is called with the GBHR value - 100010100011. The WHO bit stored at the shared PHT index - 100010100011 is 1, hence the side cache entry is searched with the GBHR. In this design, the side cache is fully associative and each entry stores a VALID bit, a BHR (GBHR/LBHR) as tag bit and a 2-bit counter. But no valid side cache entry is found whose BHR field matches with 100010100011. Hence, the default prediction (00) is taken as the global prediction. MSB of the 2-bit counter stored in the choice predictor table at index 100010100011 selects the global predictor for the final prediction. Hence, the final prediction for Branch 1 is not-taken. Figure 4.9 shows the fetch stage of Branch 1.

Fetch stage of Branch 2 : *For Branch 2, the local predictor accesses the BHT with the lower 10 bits of the branch address. Let us assume that the LBHR stored at this entry is 100000100111 and the shared PHT at index 100000100111 is accessed by the local predictor. The WHO bit at this shared PHT entry is 1, hence the local prediction is taken from this entry as shown in Figure 4.10. The global predictor accesses the shared PHT index with the GBHR value 100010111100 and finds WHO bit at this entry as 1 which indicates that this entry is owned by the local predictor. Hence, the global predictor searches the side cache entry with the GBHR. Let us assume that it finds a valid entry stored earlier, as shown in Figure 4.10. The MSB of the counter value gives the prediction for the Branch 2 as taken. The state of the choice predictor at index 100000100111 is 01, which implies that the final prediction will be taken from the global predictor.*

Retire and update stage of Branch 1 : *Let us assume that the branch outcome of Branch 1 is resolved as not-taken. For this branch, the global predictor was selected for the final prediction. During prediction, no side cache entry was available for the GBHR value - 100010100011, hence default prediction was taken. Now an empty entry is searched in*

the side cache to store the prediction information for the global predictor and it is found as shown in Figure 4.10. In that entry, VALID bit is kept as 1, GBHR value - 100010100011 is kept as BHR and 00 is kept as the 2-bit counter value since branch outcome is not-taken. In this case, choice predictor updation is not needed since both the local and the global predictions have the same prediction, as explained earlier.

Retire and update stage of Branch 2 : *The global predictor was selected for the final prediction and the final prediction was taken from a valid side cache entry as in Figure 4.10. Now the branch outcome is resolved as taken, hence the 2-bit counter value stored at this cache entry is incremented by 1. Since the local and global predictions are different and branch outcome is same as the global prediction, the 2-bit counter value stored at the choice predictor table index 100010111100 is decremented by 1 as shown in Figure 4.10.* □

The methodology described in this section creates a side cache entry whenever an interference occurs on a PHT entry, so that the two predictors can keep their own data in separate locations (PHT or cache). Thus, all instances of PHT interference are eliminated and as expected, there is an improvement in prediction accuracy, over the shared PHT two-component design. In the following, we describe two optimizations on top of this.

4.4.2 Eliminating negative interferences only and allowing positive ones (Methodology-2)

We analyse the interference a little more closely. We classify inter-predictor interference into two different types: *positive interference* and *negative interference* as defined below.

Definition 4.1 Positive and Negative Interference

An instance of positive interference *for a shared PHT entry occurs when the prediction*

given by the current predictor and the state of the 2-bit counter stored at that shared PHT entry by the other component owning the entry are aligned in the same direction (either both taken or both not taken). A negative interference *occurs when they are not aligned in the same direction.* ■

Example 4.3 *Consider that the branch direction predicted by the current predictor is* taken *and the 2-bit counter stored by the other predictor owning the entry at a particular PHT entry is 10 or 11. In this case, a* positive interference *occurs. A* negative interference *occurs when the branch direction predicted by the other component is* taken *and the 2-bit counter stored by the owning component at that particular PHT entry is 00 or 01.* □

In this methodology, we avoid creation of an additional side cache entry for instance of positive interference and allow both the components to operate on the same PHT entry. In the previous method outlined in Subsection 4.4.1, we disallow any interference, hence the benefit of *positive interference* is lost as well. We now propose another methodology that eliminates only negative interferences and allows the components to work on the same PHT entry as long as their predictions match. In such a case, a lesser number of cache entries are expected to be created. We now describe the modifications needed in the fetch and retire stages to incorporate this enhancement.

Fetch Stage

For a branch instance, in the fetch stage, when the global predictor is invoked for prediction and the WHO bit of the corresponding PHT entry is 1, the side cache is looked up. If a valid side cache entry is available for that PHT index, the MSB of the 2-bit counter stored in the entry gives the prediction. If no valid side cache entry is available for that PHT index, it indicates either no interference occurred till that point or only instances

of positive interference occurred between the components for that entry. At the time of positive interference, the 2-bit counter of the PHT index was updated. Hence, to get the benefit of the positive interference, the MSB of the 2-bit counter stored at that PHT index is taken as the global prediction. Similarly, the 2-bit counter of the shared PHT gives the local prediction if the WHO bit of that entry is 0 and no side cache entry is available for that entry.

Branch retire and update stage

The methodologies for creating an entry in the side cache and corresponding prediction information update for a branch instance are different as compared to the previous methodology outlined in Subsection 4.4.1 for both the global and the local predictor components. If the current state of the choice counter stored in the Choice predictor (indexed by global BHR) is 00 or 01, the global predictor was selected for final prediction and the corresponding update method is invoked as shown in Algorithm 7. If this state is 10 or 11, the local predictor was selected for final prediction for that branch and the corresponding update method is invoked. The Choice Predictor update is similar as in the earlier case.

Global predictor and side cache updation

Let us first examine the action taken for the global component in the retire stage. As earlier, if the WHO bit of the corresponding entry is 0, the shared PHT is looked up with the GBHR, and the action is exactly as outlined in the first case. However, if the WHO bit is 1, it indicates that the global component does not own the entry and there is an interference. This interference can be positive or negative. We first check for a valid side cache entry for the GBHR. If found, we update the 2-bit counter of that entry according to the actual branch outcome. If the side cache entry for this GBHR is not available, we

check for the interference type. We create a new side cache entry only if the interference is negative. To create a new side cache entry, as earlier, we first check for an available side cache entry. If the cache is full, we use a replacement policy as discussed. The new side cache entry is updated with the GBHR, default value of the 2-bit counter (00) and 1 as the VALID bit. Finally, the GBHR is updated according to the actual branch outcome. The detailed steps are shown in Algorithm 7. For the local component, the methodology is similar as in Algorithm 7) except for the following: a) the PHT is indexed with the LBHR, and b) the side cache creation / update is done when the WHO bit of the particular PHT entry is 0 and c) both the GBHR and LBHR entries are updated with the actual outcome when a branch retires.

Example 4.4 *In the previous example, for Branch 1, the global predictor searched the side cache since the WHO bit stored at that shared PHT index was 1 and a default prediction was taken as the global prediction since the side cache entry was not available. Now in Methodology 2, side cache lookup is not needed for the global predictor. The MSB of the 2-bit counter stored at that entry gives the global prediction. At the retire stage, branch outcome is not-taken and it is aligned to the 2-bit counter value stored at this entry, hence no side cache entry is created to store the prediction information for the global predictor.* □

4.4.3 A further optimization (Methodology-3)

We now propose another optimization that can lead to further consolidation of the side cache entries, and allow the predictors to benefit from each other, even though their predictions are in separate locations. In our previous scheme, at the retire stage, the 2-bit counter of only one entry (either the PHT or the cache) corresponding to the component that was chosen for final prediction is updated according to the actual branch outcome. As an example, if the global predictor was chosen for the final prediction from the side cache entry in the fetch

ALGORITHM 7: Side cache entry creation and global updation at the retire stage

```
1  begin
2      Find PHT entry T with GBHR
3      if WHO bit of T is 0 then
4          Update 2-bit counter of PHT entry with actual outcome
5      if WHO bit of T is 1 then
6          /* search the side cache (S) for GBHR */
7          if side cache entry S available for that GBHR then
8              Update 2-bit counter of S with the actual outcome
9          else
10             /*side cache entry for that GBHR is not present */
11             if Positive interference then
12                 Update 2-bit counter of S with the actual outcome
13             else
14                 /* Negative interference, create separate entry */
15                 if An empty entry E is found in side cache then
16                     Insert GBHR at E
17                     Set VALID bit of E as 1
18                     Initialize 2-bit counter of E as 00 and then update it with the
                         actual outcome
19                 else
20                     Replace an entry R
21                     Keep GBHR at R
22                     Set VALID bit of R as 1
23                     Initialize 2-bit counter of R as 00 and then update it with the
                         actual outcome
```

stage, only the 2-bit counter of that side cache entry is updated according to the actual branch outcome in the retire stage. We propose a further optimization over the single entry update method. In this proposal, we strengthen the state of the other three 2-bit counters. We suggest to update the states of the other three entries if the actual branch outcome is aligned with the states of the two-bit counters stored in these entries. Results shown in Section 4.6 present the effect of this optimization.

Example 4.5 *According to Example 4.2, for Branch 2, the global predictor accesses the side cache to get the prediction since the WHO bit of the corresponding shared PHT entry*

is 1 which signifies that the local predictor owns that particular entry. Now consider that the global predictor was chosen for the final prediction from the side cache entry in the fetch stage and branch outcome is not-taken. As obvious, the side cache entry is updated according to the actual branch outcome. Now, if the two bit counter stored in the shared PHT entry is 01 (kept by the local predictor), it indicates that it is aligned with the actual branch outcome (MSB of 01 is 0, hence branch direction is not-taken). In this case, we strengthen the state of this counter and make it as 00. Similarly, we check the states of the 2-bit counters stored at the shared PHT and the side cache entry of the local predictor and update their states if they are 01. □

4.5 Implementation

To compare the prediction accuracy, we implement our predictor design along with 2 others, on top of the CBP-2 [33] predictor evaluation infrastructure. After execution, it reports the performance statistics in terms of Mispredictions Per Kilo Instructions (MPKI). We implement the following for evaluating the efficiency of our design.

- Classical multi-component predictor with separate PHT: local one with a BHT of size 1024 X 12 bits, accessed by a 10 bit PC value and a 4096 X 2 bit PHT, accessed by a 12 bit LBHR. The global keeps a PHT of 4096 X 2 bits, accessed by the 12 bit GBHR, and a 4096 X 2 bit choice predictor. We refer to this as *(a)*.

- Shared PHT multi-component predictor: local predictor with a BHT of size 1024 X 12 bits, a shared PHT of 4096 X 2 bits is used for both the local and global components. The choice predictor size is as earlier. We refer to this as *(*b).

- The enhanced multi-component shared-PHT predictor with side cache: BHT size is same as before. PHT size is increased to 4096 X 3 bits, since we keep an extra

WHO bit along with the 2-bit counter. In addition to this, we keep a fully associative cache, each entry stores a 12-bit tag value, 2-bit saturating counter and a valid bit. To replace an entry in the side cache, we use the Least Recently Used (LRU) replacement policy [79]. In this work, we use the side cache at different size points of 32-entry, 64-entry, 128-entry and 256-entry. We refer to this as *(c)*.

To replace an entry in the side cache, we use the Least Recently Used (LRU) replacement policy [79]. We perform our experiments on all the CBP-2 program trace categories and record the MPKI and hit ratio for all sizes discussed. For each program category in CBP-2, we record the MPKI and hit ratio for each individual program in that category, and compute the average for each category for each cache size, as depicted in our results below.

4.6 Experimental Results

Table 4.2 presents the detail of positive and negative interferences with respect to the total number of PHT accesses, recorded in the shared PHT implementation that we started with. The negative interferences reduce to 0 in our implementation scheme, since we eliminate all instances of negative interference.

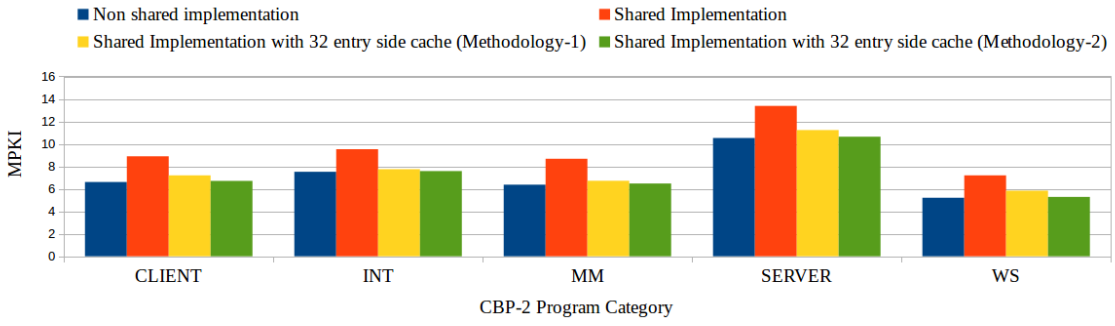


Figure 4.11: Average MPKI for different methodologies

CBP-2 Programs	Positive Interferences(%)	Negative Interferences(%)
CLIENT01.bz2	2.99	0.35
CLIENT03.bz2	4.66	0.86
CLIENT04.bz2	48.19	0.24
CLIENT05.bz2	3.89	0.53
CLIENT06.bz2	1.04	0.08
CLIENT07.bz2	6.61	0.39
CLIENT08.bz2	6.45	0.50
CLIENT10.bz2	4.71	0.70
CLIENT11.bz2	14.13	0.44
CLIENT12.bz2	4.75	0.67
CLIENT13.bz2	15.22	1.26
CLIENT14.bz2	5.84	0.42
CLIENT15.bz2	9.66	0.70
CLIENT16.bz2	16.51	0.86
INT01.bz2	12.61	1.10
INT05.bz2	8.013	1.22
INT06.bz2	10.93	1.18
MM01.bz2	5.90	0.33
MM02.bz2	11.70	0.57
MM03.bz2	13.13	0.55
MM04.bz2	29.17	0.90
SERVER02.bz2	2.72	0.47
SERVER03.bz2	3.16	0.58
SERVER04.bz2	20.77	1.81
SERVER05.bz2	19.25	1.74
WS02.bz2	3.78	0.40
WS05.bz2	24.31	1.32
WS06.bz2	21.06	1.05

Table 4.2: Interference statistics for CBP-2 program traces

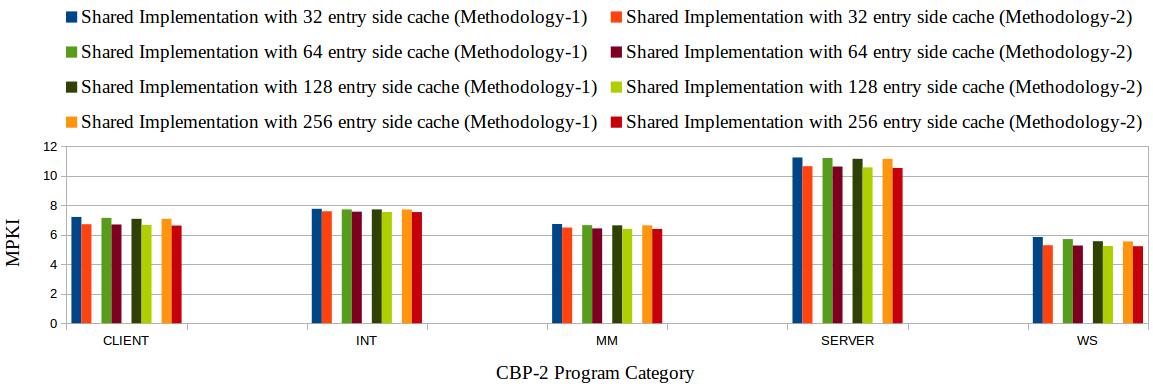


Figure 4.12: Average MPKI for different cache size with Methodology-1 & Methodology-2

4.6.1 MPKI comparison between shared / non-shared designs

We first present results on both Methodology-1 and Methodology-2. Recall that lower the value of MPKI, better is the performance, since MPKI records the mispredictions. Figure

4.11 shows the average MPKIs of (a), (b) and (c) with a 32-entry side cache. Evidently, MPKIs increase in the shared PHT over the non-shared implementation. However, the MPKIs reduce when a side cache is used along with the shared PHT, when compared to the MPKI of the shared PHT design without the side cache. Even with a 32-entry side cache, the MPKIs reduce for both Methodology-1 and Methodology-2 as shown in these Figures. Additionally, it may be noted that for some of the cases, our side cache implementation achieves comparable average MPKI when compared with the non-shared implementation. Hence, we conclude that our proposal helps to get almost similar prediction accuracy for a less storage budget (since we do away with one PHT and replace it with a small side cache) over the baseline multi-component predictor implementation, which supports our objective.

4.6.2 MPKI with different cache sizes

Figure 4.12 reports the average MPKIs for a 32-entry, 64-entry, 128-entry and 256-entry side cache for both the methodologies. It can be observed that the MPKIs decrease with increase in size of the side caches for all the traces. This is as expected since the number of replacements is less.

4.6.3 Comparing Methodologies 1 and 2

We now compare the MPKI values achieved using the two methodologies. As discussed in Section 4.3, in Methodology-1, we create a side cache entry for each interference and eliminate all interferences. However, in Methodology-2, we eliminate only the negative interferences to preserve the benefits of positive interferences. Hence the obvious expectation is that the number of cache entries created is usually the highest in Methodology-1 without the optimization proposed in Methodology-3, while it is the lowest in the second with optimization. The other two schemes (Methodology 1 with optimization and Methodology

2 with optimization) lie in between, as shown in Table 4.3.

CBP-2 Programs	Methodology-1 without optimization	Methodology-2 without optimization	Methodology-1 with optimization	Methodology-2 with optimization
CLIENT07.bz2	21417	19183	20789	6127
CLIENT08.bz2	5783	3834	5666	1662
INT01.bz2	53767	33252	52116	19293
INT06.bz2	11683	10790	9888	4006
MM01.bz2	17012	14825	16483	5211
SERVER04.bz2	53387	29065	50538	9073
SERVER05.bz2	58143	30567	50662	8509
WS02.bz2	19982	25373	19774	4882
WS06.bz2	50679	29850	44210	9885

Table 4.3: Cache entry creation statistics

Figures 4.13 and 4.14 show the average hit ratios of different side cache sizes for both the methodologies. It is noticed that the hit ratio increases with increase in the size of the side cache, since a lesser number of replacements are needed.

Finally, we present the results of the optimization proposed in Subsection 4.4.3, on all side cache sizes for Methodology-1 and Methodology-2. Figure 4.15 compares the MPKIs achieved in the side cache implementation with and without the proposed optimization for Methodology-1. Figure 4.16 reports the same for Methodology-2.

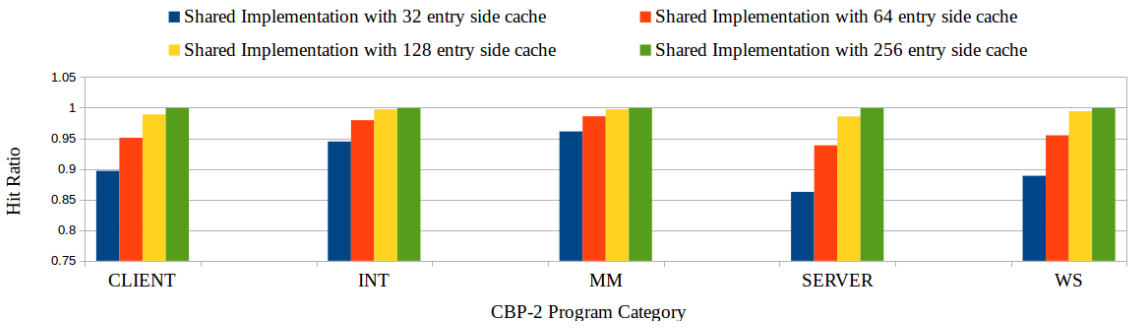


Figure 4.13: Average hit ratio for different side cache implementations using Methodology-1

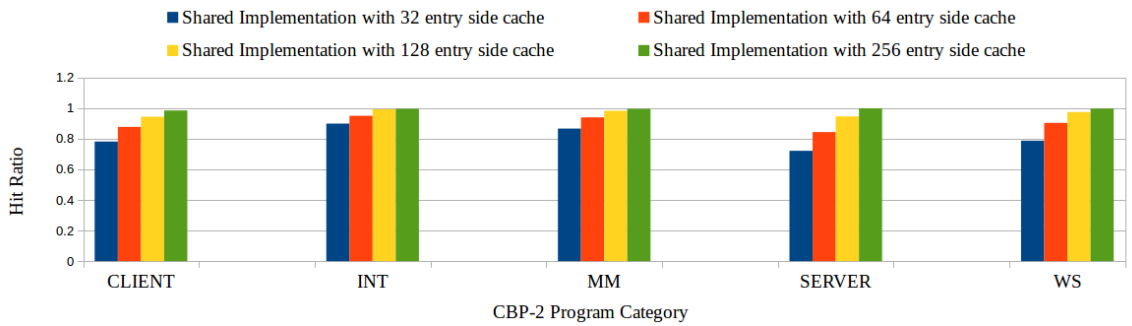


Figure 4.14: Average hit ratio for different side cache implementations using Methodology-2

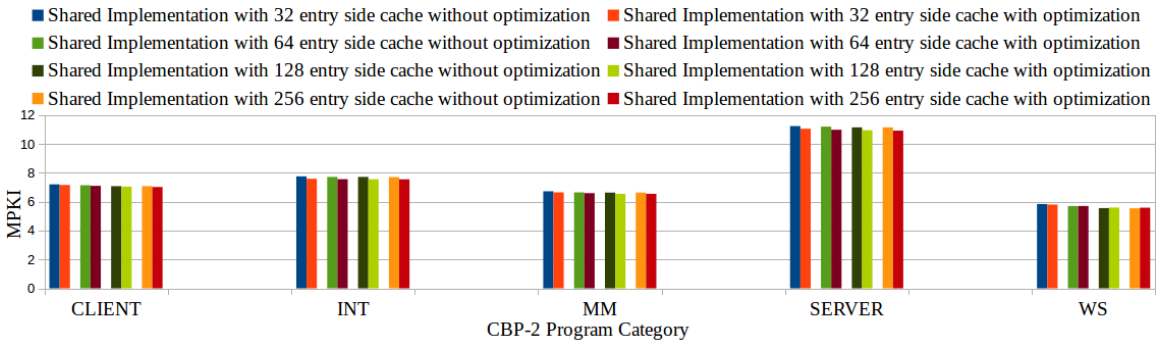


Figure 4.15: Average MPKI for different side cache implementations (with and without optimization) using Methodology-1

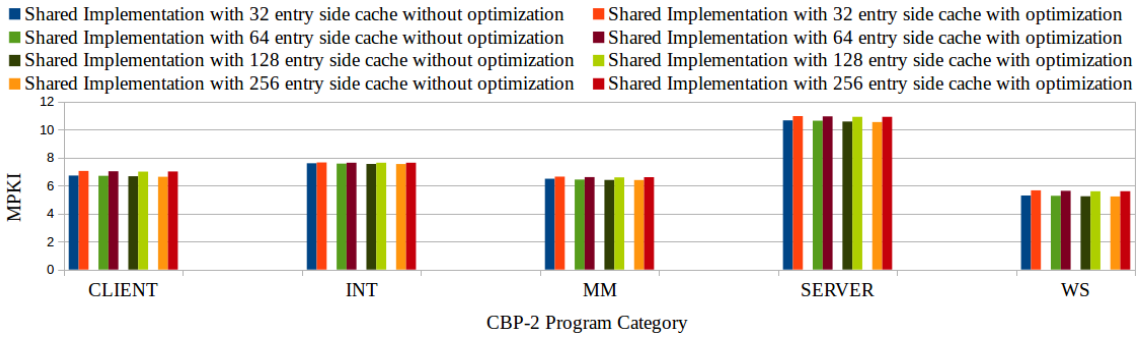


Figure 4.16: Average MPKI for different side cache implementations (with and without optimization) using Methodology-2

4.6.4 Evaluating our *First-Touch* allocation method

In this work, we use a *first touch* allocation policy for PHT ownership, wherein the ownership of the PHT, though assigned to the global by default, can be overridden by the local

component if it accesses a PHT entry and finds the pattern 000 (0 for WHO bit and 00 for 2-bit counter value). This requires us to keep an additional WHO bit to record and arbitrate the ownership of each PHT entry. To get more storage benefits by doing away with the WHO bit, we also experiment with a static PHT ownership assignment method where every even PHT index is assigned to the global predictor and every odd index is assigned to the local one. A loop branch's local predictor entry almost always maps to powers of two minus 1 - this leads to the motivation for giving only odd entries to local predictors. To index the PHT, the global and local predictor use GBHR and LBHR respectively, as in the *first touch* allocation policy. When the global predictor needs to retrieve / store prediction information with an odd GBHR, no PHT index is available for that. In this case, the side cache is used. Similarly, for the local predictor, the side cache is accessed to retrieve / store the prediction information when LBHR is even.

Figure 4.17 compares the average MPKIs obtained from this method with the *First-Touch* allocation method for different storage budgets of the side cache. It can be seen that the *First-Touch* allocation policy performs better (less MPKI) than the *Static* method for all the cases since in general it is hard to foresee the PHT entry pattern that the predictors will map to. An in depth analysis reveals that MPKIs in the *Static* method increase since the number of different indices accessed by the two predictors become half and a fixed PHT index assignment is done for these two. However, for an arbitrary program, equal number of PHT indices may not be created by these two predictors and it is also quite rare that all GBHRs are even and LBHRs are odd (or vice versa). Hence, this static assignment causes more side cache accesses. Since the side cache size is small, more side cache replacements occur. Replacement of a side cache entry loses the prediction information stored at the entry and this leads to decrease in prediction accuracy (increase in MPKI). As a result of this, MPKIs of the *Static* method tend to decrease with increase in the size of the side cache as shown in Figure 4.17.

In the previous static method, we store the prediction information for the local predictor in the shared PHT if LBHR is odd and for the global predictor in the shared PHT if GBHR is even. It is obvious that both can ask for odd and even entries depending on the LSB of the history register. For loop branches, LBHR is always odd, however it is not valid for other types of branches such as alternating branches. Hence, we consider another static method where two alternate consecutive PHT entries are assigned to global and local predictors. For example, the first two consecutive PHT entries (even and odd indices) are assigned to the global predictor, the next two consecutive shared PHT entries are assigned to the local predictor, again the next two consecutive entries are assigned to the global predictor and so on. In this case, the second least significant bit (LSB) determines which predictor will store the prediction information in the shared PHT. If the second LSB is 0, the global predictor will store the prediction information into that PHT entry, otherwise the local predictor will store the prediction information there. This makes the WHO bit redundant since the ownerships are fixed apriori, as opposed to our first touch method. Results (Figure 4.17) show MPKI increases for this static method as well when compared to the first touch allocation policy.

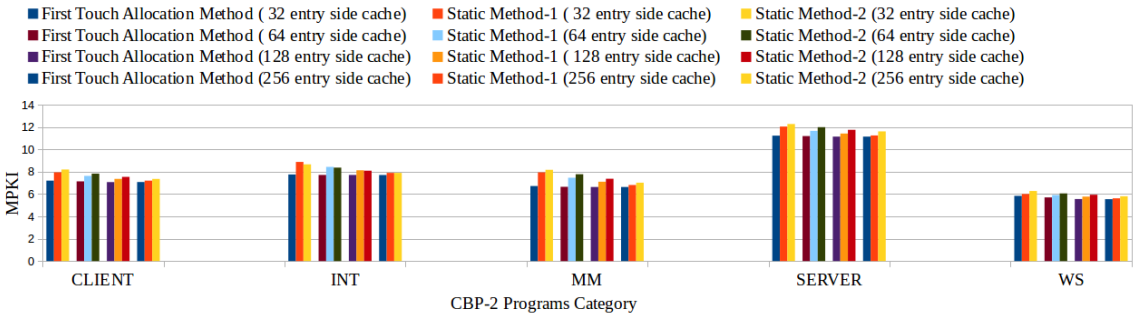


Figure 4.17: Average MPKI for First Touch versus Static Methods

4.7 Synthesis Results

To evaluate the area and power benefits, we develop RTL designs in Verilog [4] for both the non-shared design and the shared design with a 32 entry side cache. We use the Veriwell simulator [13] to verify all the designs. The Synopsys Design Compiler (DC) [40] is used to synthesize our designs with TSMC 90nm libraries [12]. Table 4.4 shows the area results of our shared PHT implementation over the non-shared implementation. The 2 PHTs in the classical design (a) are synthesized as 2 single-port SRAMs, while in our proposals (b) and (c), the shared PHT is synthesized as a dual-port SRAM [1] to enable simultaneous access by the 2 components in the fetch and retire stages. The synthesis of the BHT and the choice tables are similar in all the 3 designs, while the cache in (c) leads to a fully associative SRAM. Results indicate a *8.27%* improvement in total area with our design. This is as per our expectation since we do away with one of the PHT tables and replace it with a small side cache. The PHT table is synthesised as a SRAM while the side cache is a fully associative one. The PHT and the side cache are looked up in parallel.

Design	Total cell area	Total net area	Total area
Non-shared PHT	797207	197987	995194.2
Shared PHT with side cache	763244	149635	912879.3

Table 4.4: Area report of the design compiler

We also present our analysis on using a Verilog module to measure per access read energy for both the SRAM and the side cache. As earlier, in our implementation, the synthesised netlist contains a fully associative memory as the side cache and a direct mapped memory for the PHT SRAM table. We collect the total number of read and write accesses for all the designs from the CBP-2 program traces and multiply these numbers with the per access energy obtained from DC to obtain the total power consumption for each CBP-2 program. Figure 4.18 shows the average % power improvement for our design over the non-shared implementation for each CBP-2 program category. It can be seen that there is

an improvement in power as well.

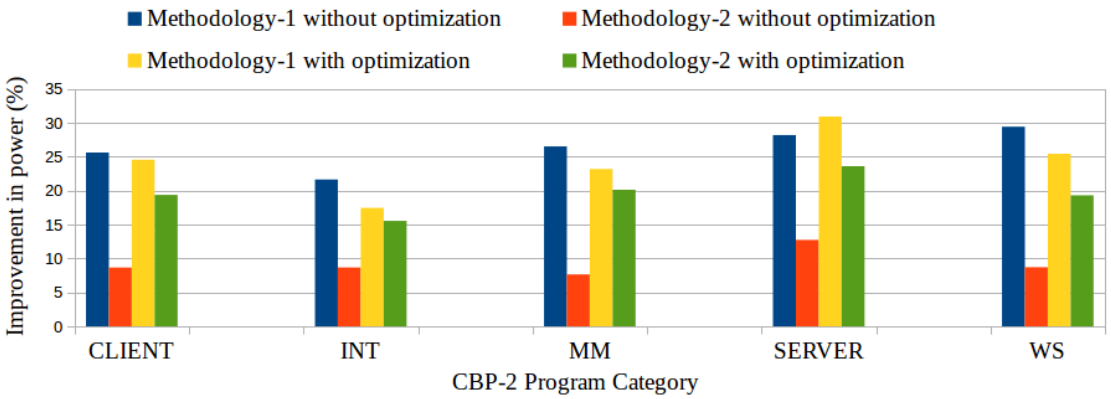


Figure 4.18: Average improvement in power over non-shared implementation

4.8 Summary

In this work, we propose a storage efficient design for two-component branch predictors, that have been the defacto choice in modern processors. We show that it is possible to attain significant prediction accuracy, without investing too much on storage space. The main highlight of our design is to enable the predictor components work on a shared prediction table, while maintaining a tiny dealiasing cache to resolve entries on which the predictors collide. Experimental results are quite encouraging and we believe that our proposal will have important consequences going forward to harness the full power of sophisticated branch prediction techniques for resource constrained embedded compute devices.

Chapter 5

A multi-component dynamic predictor design with interference control

5.1 Introduction

In the previous chapter, we proposed a two-component predictor design suitable for resource constrained environments. It is interesting to note that different predictors that are used in practice today, have quite varying prediction strategies, and often designed to cater to branches of a particular type [45]. A single predictor or even a two-component predictor, therefore, may not just be enough for all branches in a program, thereby necessitating the idea of multi-component branch prediction. It is interesting to note that different predictors that are used in practice today have quite varying prediction strategies, often specifically designed to branches of a specific type. A multi-component predictor uses multiple components. For each branch, it uses the best predictor that provides maximum

prediction accuracy and leads to minimum mis-prediction for that branch. Similar to the two-component predictor discussed in the last chapter, multi-component hybrid predictors also consume more space in comparison to a single predictor since separate predictor tables are kept by individual predictor components. Therefore a multi-component predictor design is not suitable for low resource budget processors. The objective of this chapter is to study the problem of designing multi-component hybrid prediction techniques for resource constrained environments.

Our experimental findings on employing a multi-component hybrid predictor with individual predictor components sharing the same predictor table on the SPEC 2006 benchmarks [52] reveal a similar finding as in case of our initial shared-table 2-component predictor design - the accuracy gain expected with the hybrid scheme is often not achieved, as in the case of our initial observations for the two-component design. This happens due to extensive interference on the shared predictor table, due to frequent switching between the predictors. The prediction information of one predictor is overwritten by another predictor during its prediction. This negative interference often leads to an incorrect direction for prediction, and therefore, accuracy degradation. In this chapter, we revisit the multi-component hybrid predictor design and study the accuracy versus storage perspective.

Contributions of this work

- We revisit the multi-component hybrid predictor design that combines more than one dynamic branch predictor for prediction at run-time based on the best predictor information. We study this multi-component predictor design for low resource budget processors where the individual predictor components are made to share the predictor table structures. Our experiments reveal that this sharing often leads to a loss of prediction accuracy due to the extensive interference between the predictors on the shared data structures they operate on.

- We propose a heuristic that attempts to improve a classical hybrid multi-component prediction mechanism by minimizing the number of context switches to different predictor components, with an objective of interference reduction.

- We show results of using our heuristic on top of the shared table implementation. Our heuristic attempts to control the amount of predictor interference by controlling the number of instances of new predictors being employed for prediction. We show an average prediction accuracy improvement of 3-4% on the SPEC 2006 benchmarks.

This chapter is organized as follows. Section 5.2 presents a baseline architecture of a multi-component predictor. We present our shared table architecture for multi-component predictors in Section 5.3. Section 5.4 presents a detailed analysis of the complete solution space of multi-component predictor designs, along with our proposal. Section 5.5 discusses the proposed architectural modification to handle the interference. Section 5.6 presents experimental results. Section 5.7 concludes this discussion.

5.2 A baseline multi-component branch predictor

Architectural details of a baseline multi-component predictor that combines multiple dynamic predictors together is shown in Figure 5.1. In this design, all the predictors are available to the program. However, for a branch only one predictor which is the best according to prediction accuracy, is invoked to give the prediction. For every branch of a program, the best predictor information is collected and recorded using static analysis and is kept as part of the corresponding branch instruction using some additional bits. During execution of a branch, these additional bits of the branch instruction are checked to invoke the corresponding predictor. This is done using the predictor enable signal. Here, each individual predictor keeps its own predictor table and therefore the total storage re-

quired increases with the number of predictors included. Such a multi-component design is therefore, not suitable for low resource processors.

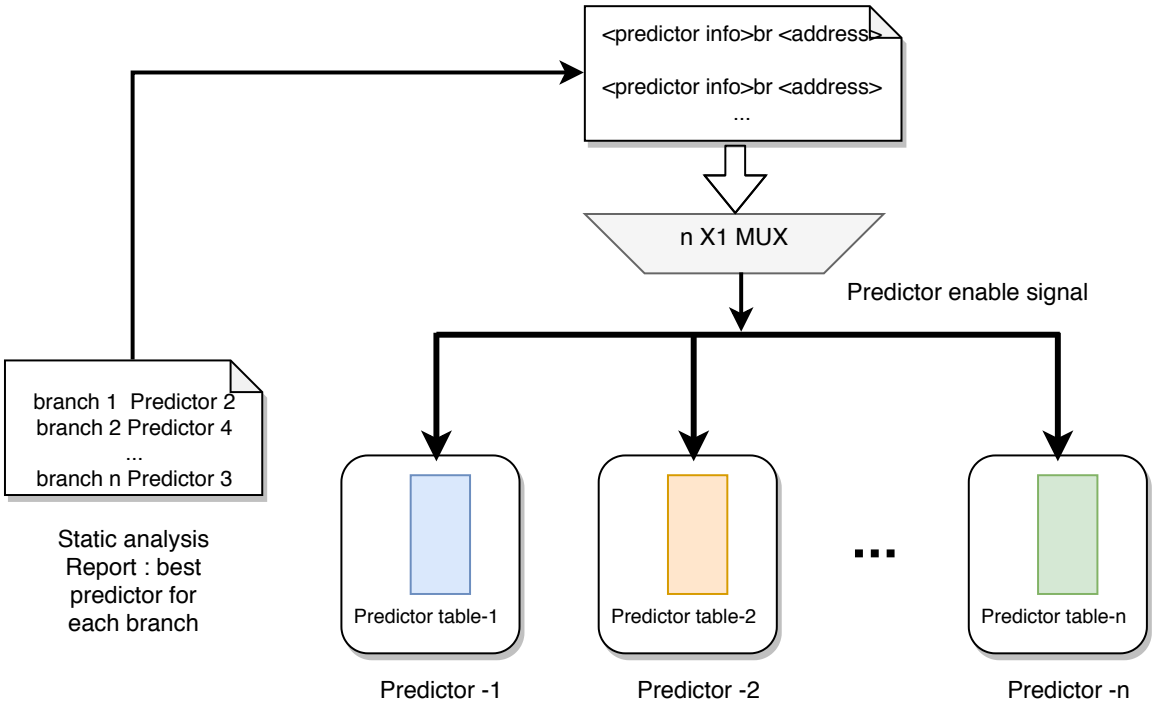


Figure 5.1: A baseline multi-component branch predictor

5.3 Shared table multi-component predictor predictor design

Multi-component prediction improves the prediction accuracy over single and two-component prediction strategies. The only drawback is the size of the predictor that increases with the number of predictors combined. It is observed most of the times that the predictors that are included in the multi-component design can use the same predictor table structure, hence motivating the idea of sharing the same predictor tables as shown in Figure 5.2. This shows our proposal of a shared predictor table design for multi-component predictor realization.

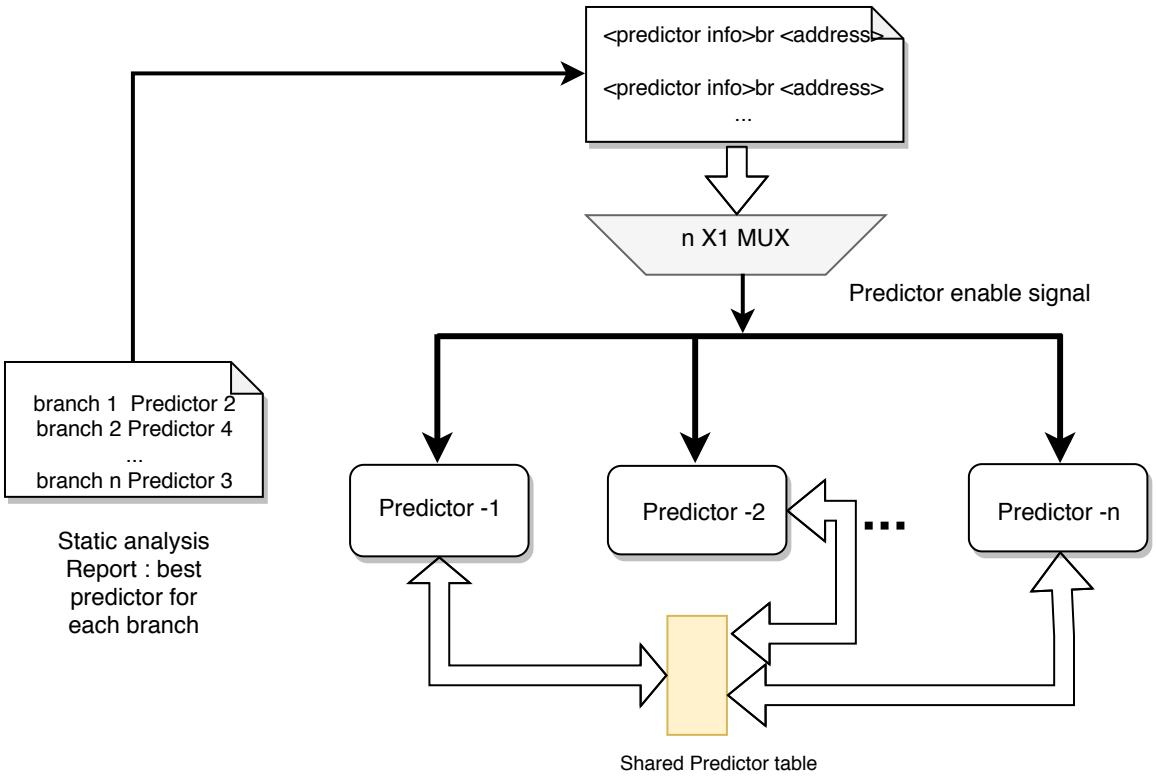


Figure 5.2: Multi-component branch predictor with Shared PHT

In this work, we use only those dynamic predictors that can share the predictor tables. However, this shared predictor table design has a major drawback, it greatly reduces the prediction accuracy of a multi-component predictor in comparison to any single component predictor. As noted earlier, this occurs due to extensive interference among the predictors involved when they try to access the same PHT entry for different branch instances and modify the prediction information stored by one another.

In this section, we illustrate an overview of the interference problem on a fragment of the *mcf* program of the SPEC2006 [52] benchmark, as shown in Figure 5.3. All figures and tables used here are generated by running *mcf* with different predictors on a small framework designed on top of the Tejas [94] architectural simulator. We first explain the performance

of GAg and GShare [78].

```
 long read_min( network_t *net )
{
  ...
  if(( in = fopen( net->inputfile, "r")) == NULL )
                                          //branch 1
       return -1;
  ...             /*Non-branch statements*/
  if( sscanf( instring, "%ld %ld", &t, &h ) != 2 )
                                         //branch 2
       return -1;
  ...              /*Non-branch statements*/
  if( net->n_trips <= MAX_NB_TRIPS_FOR_SMALL_NET )
                                         //branch 3
    {
      net->max_m = net->m;
      net->max_new_m = MAX_NEW_ARCS_SMALL_NET;
    }
  ...            /*Non-branch statements*/
  if( !( net->nodes && net->arcs && net->dummy_arcs ) )
                                      //branch 4, 5,6
    {
      printf( "read_min(): not enough memory\n" );
      getfree( net );
      return -1;
   }
  ...           /*Non-branch statements*/
  if( sscanf( instring, "%ld %ld", &t, &h ) != 2 || t > h )
                                       //branch 7
           return -1;

  ...         /*Non-branch statements*/

}
```

Figure 5.3: A fragment of the *mcf program*

Table 5.1 presents the misprediction rate obtained, averaged over multiple simulation runs on the standard simulation data provided as part of the benchmarks, for 7 branches of the *mcf* program [52] shown in Figure 5.3 when a single dynamic predictor is used for branch prediction. GAg provides lowest misprediction rate for branches 1, 2, 4 and 6, whereas GShare provides the same for branches 3, 5 and 7. Quite evidently, the misprediction rate / prediction accuracy for a particular branch varies for different predictors which substantiates the fact that no single predictor performs best for all the branches. This motivates use of a hybrid predictor (HP). For each branch, HP selects the predictor with the lowest misprediction rate / maximum prediction accuracy (we call it the best predictor) for that

branch, as obtained from Table 5.1. The last column of this table presents the predictor selected for each branch. HP switches to a different predictor when the current active predictor is not the best for that branch. As an example, HP selects GAg for branch 6 and switches to GShare for branch 7. Figure 5.5 presents the prediction accuracy degradation for shared PHT table implementation with respect to GAg, GShare and Bimodal predictors.

Branch	Branch Address	BHR Pattern	Misprediction rate(%)			Predictor Invoked (HP)
			GShare	GAg	Hybrid	
Branch1	401999	110101010	12.0	10.0	10.0	GAg
Branch2	4019d6	101010101	7.0	6.0	6.0	GAg
Branch3	401a17	010101010	3.6	4.0	10.0	GShare
Branch4	401a91	101010101	11.0	10.0	15.0	GAg
Branch5	401a9a	010101011	0.10	3.0	0.10	GShare
Branch6	401aa3	101010111	10.3	10.0	10.0	GAg
Branch7	401cd9	010101110	2.0	5.0	7.0	GShare

Table 5.1: Branch profile for *mcf* program branches

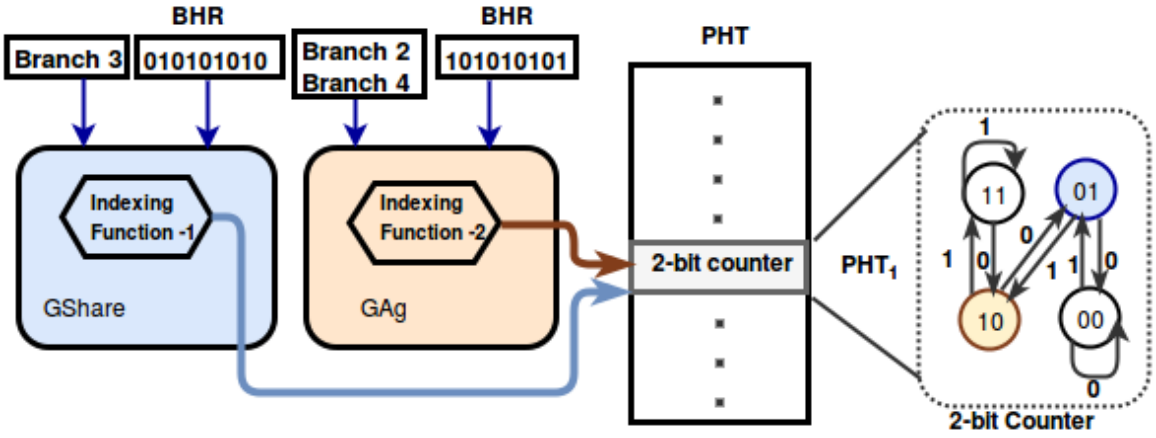


Figure 5.4: Predictor table interference between two predictors

We further experiment with a single shared PHT to be used by both GShare and GAg, as summarized in Figure 5.4. Table 5.1 Column 6 shows the result for the 7 branches run with a hybrid predictor that combines GShare and GAg together. It is observed, there is a loss in prediction accuracy in some cases due to the interference between predictors.

We present interference statistics on some SPEC 2006 benchmarks in Table 5.2. For a hybrid

predictor with a shared 32KB PHT, Column 2 shows the percentage of entries suffering from interference (access and modification by the different components) with respect to all PHT accesses. Figure 5.5 shows the difference in average prediction accuracy between three single stream predictors (GAg, GShare and Bimodal) and the shared table hybrid predictor implementation. It is seen, for most of the cases the bars are in the positive direction, i.e. there is an accuracy fall with the shared implementation. These observations motivate us to address this interference.

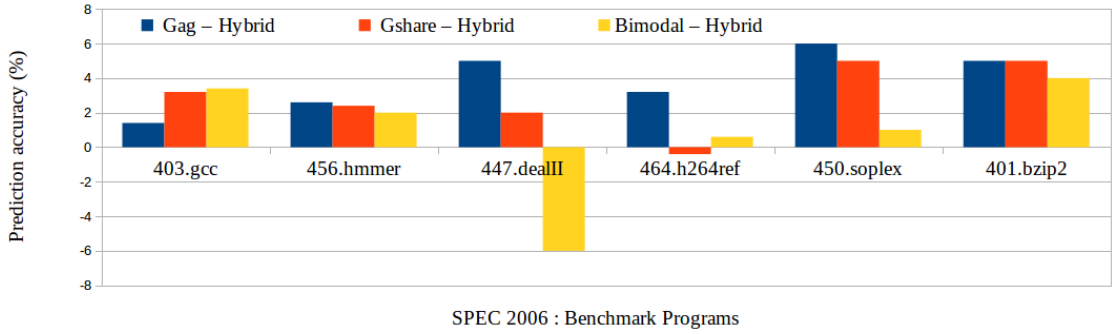


Figure 5.5: Accuracy comparison: hybrid and single predictors

5.4 Selective Switching based Interference Control

In this section, we present our proposal in greater detail. We begin by illustrating the solution complexity of the problem space, from the perspective of designing a hybrid predictor scheme that can maximize the overall end-to-end prediction accuracy for a given program. For ease of illustration and simplicity of explanation, we consider a hybrid predictor that combines 4 individual predictors: GShare (P1), GAg (P2), TAGE (P3) and PAp (P4) together and a program P with 4 branches: Branch_1, Branch_2, Branch_3 and Branch_4 (in this order of their occurrence in P), any of which can be predicted using any of the 4 predictors as shown in Figure 5.6.

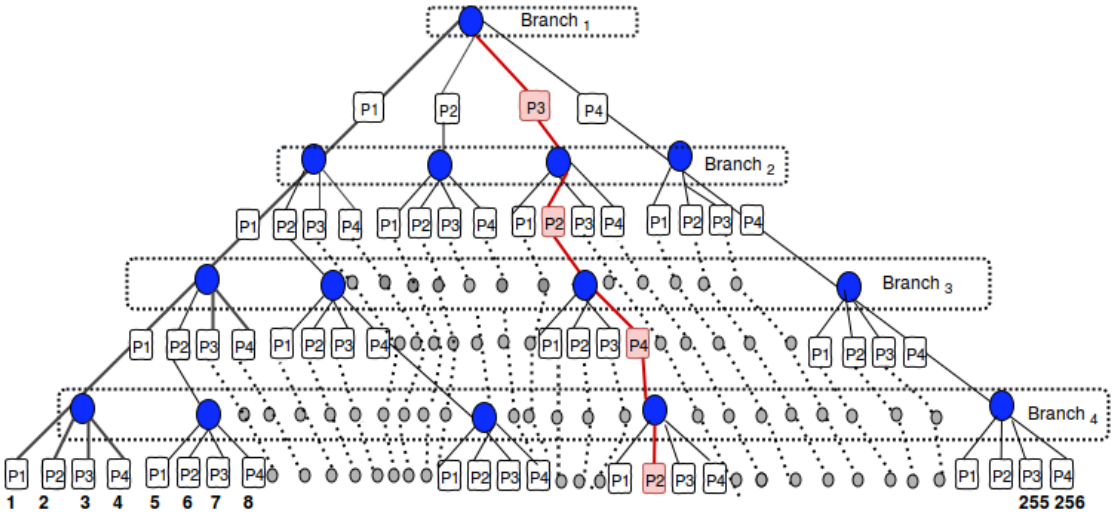


Figure 5.6: Predictor Sequence Tree

Figure 5.6 shows a level-ordered 4-ary tree that shows all possible predictor combinations for all the four branches in the way the branches are ordered in the original program. In this case, this tree has 4 levels, where each level represents an individual branch of this program (and the different predictor choices for that branch) and 256 different paths to represent 256 different predictor sequences. Intuitively, this tree captures all the possible ways a hybrid predictor scheme can be designed using individual predictors. The path marked in red is the predictor sequence generated by the traditional hybrid predictor which works by selecting the best predictor for every branch based on prediction accuracy for that branch. As illustrated in the previous section, a hybrid scheme with shared PHT table, that picks the best predictor for each branch is not necessarily the best choice in terms of overall prediction accuracy for the entire program, and hence, the need for this exhaustive enumeration. A hybrid predictor should ideally explore all the paths of this tree to get all possible predictor sequences, compute the average prediction accuracy of each path and choose the best that maximizes it. Thus, all possible interleavings of the predictors (and the resulting interferences) would be automatically considered, and a choice can be made that

minimizes the negative interference as much as possible, while also maximizing prediction accuracy. The exhaustive enumeration described above is however, beyond scope for any realistic implementation. The following subsection presents our proposal for interference reduction, which is a different approach to address this problem.

We propose to reduce predictor interference using selective switching as described in Algorithm 8. The philosophy of Algorithm8 is to find the best predictor $\mathcal{R}$ that has the best overall prediction accuracy for a program and the best predictor P_i for each branch i, obtained from previous simulation runs or earlier phases of execution. We start with $\mathcal{R}$ as the current predictor $\mathcal{C}$. Let $A_{\mathcal{C}}$ denote the prediction accuracy of $\mathcal{C}$, and A_{P_i} likewise denote the prediction accuracy of predictor P_i. We elaborate on our choice of these in the following section. For each branch, if the best predictor is not $\mathcal{C}$, it checks whether the prediction accuracies of the current predictor and P_i differ beyond a threshold value (say θ). If this condition is true, it selects that best predictor for prediction and updates the current predictor accordingly. Otherwise, it continues with the predictor $\mathcal{C}$ for prediction.

ALGORITHM 8: Selective Switching

```
1  begin
2      Find the predictor R with maximum prediction accuracy for a program Q
3      Initialize currentPredictor C = R
4      for each branch i in Q do
5          Let P_i be its best predictor if available, or else P_i = R.
6          if |A_C − A_{P_i}| < θ then
7              Select C for prediction of i
8          else
9              Select P_i for prediction of i
10             Update C = P_i
```

Example 5.1 *Consider the* mcf *program (Figure 5.3) and a threshold θ of 0.5. Let us assume GAg is the overall best predictor to start with. From the information given in Table 5.1, for branch 1, the best predictor is the same, and we continue with GAg as*

the current best predictor and use it for prediction of this branch. A similar thing happens for branch 2 as well. For branch 3, the best predictor is GShare, however the gain in prediction accuracy is less than our threshold. Hence, we continue with GAg for prediction. For branch 4, the best predictor is GAg as well. For branch 5, the best predictor is GShare and the difference in accuracy gain is beyond 0.5, hence we change the current best predictor and switch to GShare. For branch 6, the best predictor is GAg, however, we do not switch to GAg since the accuracy gain is less than θ. We continue using the same predictor for branch 7 as well, since GShare is the best for it. The number of predictor switches (GAg→GAg→GAg→GAg→GShare→GShare→GShare) in our proposal is 1, while in a classical hybrid scheme (as shown in Column 7 of Table5.1), it is 5 (GAg→GAg→GShare→GAg→GShare→GAg→GShare). Thus the total number of inter predictor interferences is less than the classical scheme. This also helps in reducing the number of interferences on a shared-table implementation, which is the main motivation behind this work. It may be noted that the worst case number of predictor switches in our case is upper bounded by the number of switches in a classical scheme. □

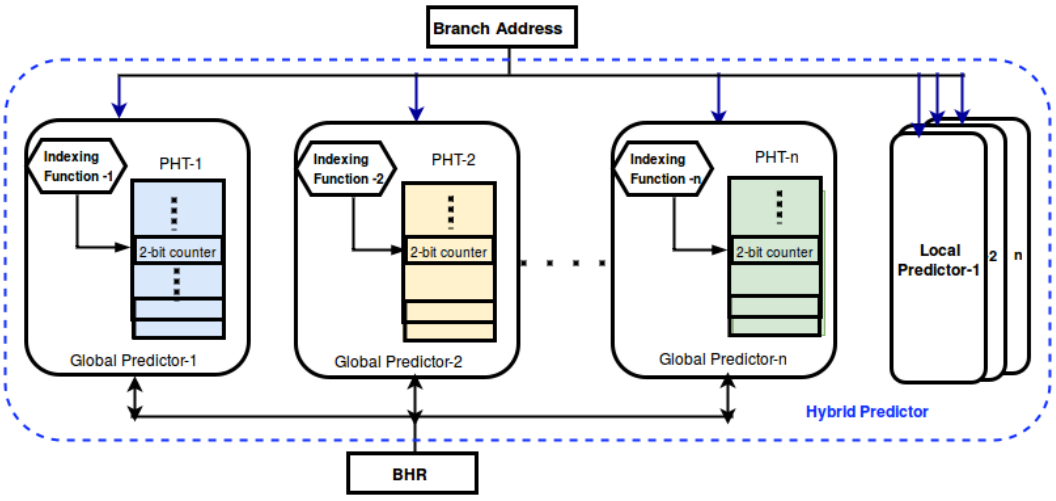


Figure 5.7: Split table Architecture for Multi-component Predictor

5.5 Architectural Modifications to reduce interference

We define an instance of PHT interference when two unrelated branches of a program are mapped to the same PHT entry by the predictor's indexing function. This phenomenon, as discussed earlier, is known as PHT interference, since the outcome of one branch is interfering with the subsequent prediction of another completely unrelated branch. Three different types of interference can occur as explained below.

In the first case, the interference does not change the direction of the prediction and we term this as a neutral interference. In the second case, this interference causes a correct prediction where there would be a mis-prediction otherwise and is termed a positive interference. In the third case, this interference causes a mis-prediction, which would not have been a mis-prediction otherwise and is referred as negative interference, since it has a negative impact on performance. It has been acknowledged that negative interference is a substantial contributor to the number of branch mis-predictions [106] and it is worthwhile to attempt to reduce it.

In a two-level branch predictor, this interference is unavoidable since the number of PHT entries is finite. A global predictor that shares the same data structure across all branches suffers from the interference problem, since it has only one global PHT which is shared by all branches. Hybrid branch prediction introduces an additional PHT interference along with this. In hybrid prediction, multiple global predictors are appointed for prediction and they use the same PHT table as well as the BHR. The negative effect resulting out of this scheme has already been illustrated earlier.

To avoid this type of interference, we propose to keep a separate PHT for each individual global predictor appointed for prediction. Figure 5.7 shows an architectural overview of our proposed architecture for a hybrid predictor. Here, *PHT-1* is kept for *Global Predictor-1*, *PHT-2* is kept for *Global Predictor-2*, and so on. Hence, *PHT-n* will be accessed and

updated only when *Global Predictor-n* is active for prediction. This implies that branch information collected by one predictor for a specific branch history pattern is not interfered with, by a different predictor for a different branch history pattern. However, this implementation loses the full flavor of branch correlation, since a PHT table is not updated always for all branches, it is updated only when its corresponding predictor is on. Our architecture increases the number of PHT tables according to the number of different global predictors appointed for prediction. The estimated hardware cost of our proposed hybrid predictor is $S + (L-1) * 2^n * 2$, where L is the number of mutually interfering predictors, S is the size of a traditional hybrid predictor and n is the BHR size. Since, one PHT is already kept in the hybrid predictor, an additional (L-1) PHTs are needed for this. Size of each PHT is 2^n and size of each PHT entry is 2 bits (contains a two bit saturating counter). In our experiments, we implement our selective switching mechanisms on top of this split PHT architecture and the accuracy improvements are encouraging. We explain the idea through an example below.

Example 5.2 *Consider the* mcf *program (refer Table 5.1). The same PHT entry* PHT_1 *is accessed three times: first time, it is accessed by the GAg predictor for branch 2 and pattern* 101010101*, and updated according to the actual outcome of this branch by this predictor; the second time for branch 3, it is accessed by GShare for a completely different pattern* 010101010 *and updated according to the actual outcome of this branch, and finally, the third time for branch 4, it is again accessed by GAg for a pattern same as in branch 2. However, this time, GAg cannot find the actual prediction information which was stored by it last time for this pattern, since this information was incorrectly interfered and updated by GShare for another pattern. In our architecture, GShare keeps a separate PHT to store the entries for all its BHR patterns and GAg also keeps another separate PHT to store the entries for the same. Hence, when branch 2 comes with a pattern* 101010101*, the PHT of GAg is accessed to get the prediction the and state of its corresponding two-bit saturating counter*

gets updated accordingly for future reference. This PHT is accessed only when both the GAg predictor is active and pattern 101010101 *occurs. This PHT can not be accessed or updated for any different pattern for any other predictor. Hence, GShare cannot access or interfere the PHT of GAg.* □

5.6 Implementation and Results

We now present details of our experiments. All simulations are run on top of the Tejas [94] architectural simulator. To mimic an embedded resource constrained environment, we take a pipeline depth of 5, with a PHT table size of 32 KB. Additionally, we disable the out-of-order-execution, VLIW features. We modify the Tejas simulator source code to implement a hybrid predictor that includes a combination of three predictors (GShare, GAg and Bimodal). We use 2-bits (00 for GShare, 01 for GAg, 10 for Bimodal) to mask the predictor to be chosen for each branch based on our algorithm. Thus, each branch instruction in the generated code is prefixed with 2 additional bits, corresponding to the predictor choice. In this work, execution driven simulation is done for the SPEC 2006 benchmarks [52]. Tejas is used to record the branch behaviors for the benchmark programs across all test cases provided and we record the average performance numbers. Each program is run with different branch predictors and the prediction accuracy for all the branches of the program on every predictor is recorded. For each branch, we mark a best predictor and for the overall program in terms of prediction accuracy. Our predictor selection algorithms are implemented on top of the hybrid prediction method that shares the PHT table among the predictors to improve the overall performance of a processor. As input, these methods take the branch profile information for all branches from the profile generation stage as discussed above. The simulation of the new selection mechanism is done using Tejas. For each execution, prediction accuracy, energy expenditure and latency are recorded for comparing against

other hybrid prediction techniques (without selective switching) that either share the PHT table among interfering predictors or split the PHT table for them.

Benchmark	Interference (in shared implementation) (%)	Interference (using switching algorithm) (%)
403.gcc	3.7	2.2
400.perlbench	4	3.3
429.mcf	1.2	0.5
458.sjeng	1	0.2
456.hmmr	1.1	0.05
447.dealII	1	0.1
464.h264ref	2	0.7
450.soplex	2.5	1.2
401.bzip2	3.4	0.4

Table 5.2: PHT interference statistics with our method

We now report the results of our methods obtained with a threshold of 0.5. Column 3 of Table 5.2 presents the percentage interference counts with our proposed switching method, with respect to the original shared table implementation. It can be seen that there is a reduction in interferences for almost every program used here. Figures 5.8, 5.9 and 5.10 respectively present the differences in average prediction accuracy, energy and execution time with respect to the original 32KB shared PHT table implementation for :

- A shared 32KB PHT hybrid predictor implementation with selective switching.
- A split PHT table of size 16KB for every individual predictor.

It can be observed from Figure 5.8 that the original shared PHT table implementation performs much worse than each of the above schemes, hence the differences are positive. For most of the programs, the prediction accuracy obtained from the shared PHT table implementation is even lower than the accuracy of any single predictor. Our experiments show that our proposed switching algorithm that works on a shared PHT table can improve the prediction accuracies as expected. Although the average accuracy improvement is not so high, around 2%-3%, however, a single misprediction can increase a significant amount

of processor cycles as well as extra instruction fetches. Figure 5.9 shows that there is an increase in energy expenditure in the shared implementation and our switching method can reduce the energy expenditure for all the benchmark programs. Figure 5.10 presents the same detail with respect to execution time. It may be noted that a lower value of execution time is more desirable, and our heuristic indeed achieves comparable or marginally less at times. From the experiments, it can also be seen that our selective switching method when employed on top of a shared-table implementation produces better accuracy than the split table implementation for all the performance metrics discussed here.

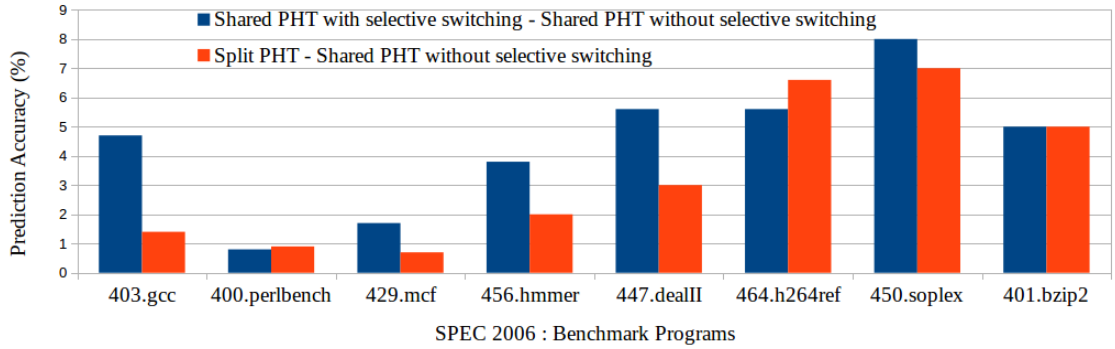


Figure 5.8: Prediction accuracy comparison

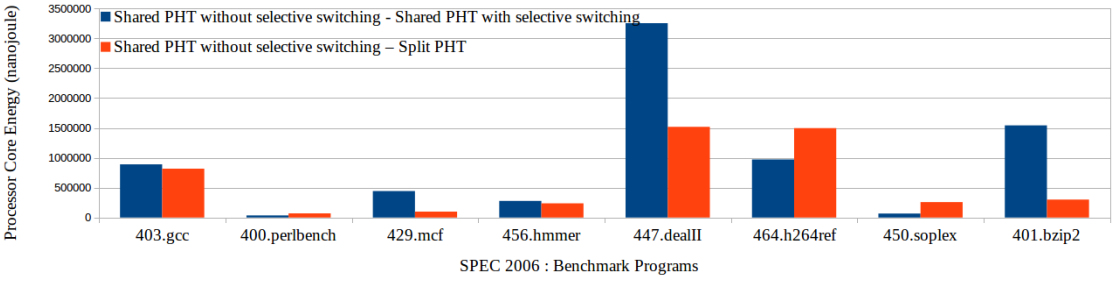


Figure 5.9: Processor core energy comparison

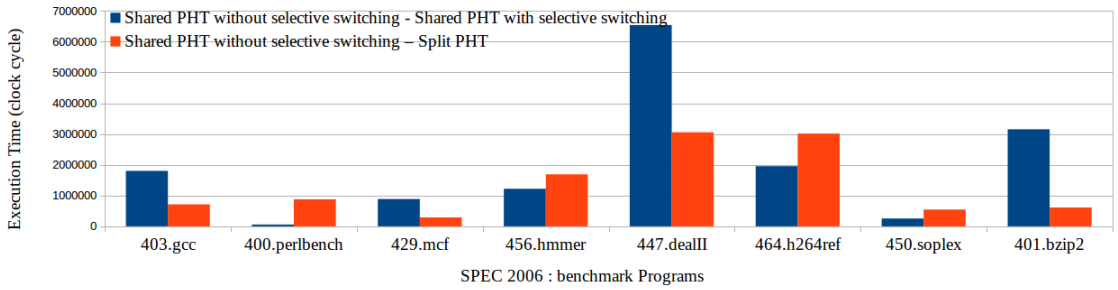


Figure 5.10: Execution time comparison

5.7 Summary

In this work, we examine the effect of predictor table interference on prediction accuracy of a hybrid predictor for resource constrained environments. We propose a predictor selection method to improve prediction accuracy by controlling the interference. Experimental results show the improvement.

We utilize a static profile based selection scheme to select the overall best predictor with highest average prediction accuracy for a program and also for each branch. We use this profile information to control the number of predictor switches. This requires a good set of representative test-cases that can exercise the different branches with different conditions, and help us collect the required information about the predictors. Also, since this is an offline step, we need to somehow store this information for use at run-time, which can be considered as a significant overhead. In addition to this, we need to track during execution, whether there is a better choice than the current predictor, and whether that choice is better by a sufficiently high amount to cross the threshold. This extra overhead may lead to higher energy cost per prediction. Another obvious limitation of our proposed switching algorithm is the fact that along with the negative interferences, it also reduces the instances of positive interference (since we limit the number of uses of different predictors) and thereby, lose the benefit of it. However, as evident from our results, the gain in being able to reduce negative

interference outweighs the benefit that we may have received from positive interference between predictors, since the number of instances of negative interference for a program is much more than their positive counterparts.

Chapter 6

Branch predictor selection with aggregation on multiple parameters

6.1 Introduction

In the last few chapters, we have discussed our work on new techniques for branch predictor design, mostly considering a single parameter as the primary objective. The objective of this chapter is to present a framework for branch predictor selection, that takes into account multiple performance parameters. The most widely explored direction of research in predictor design has been around the objective of prediction accuracy maximization, with an expectation of energy reduction due to lesser mis-predictions. However, it has often been the case that the energy gain expected has been nullified by the energy footprint of the sophisticated runtime data structures that these schemes need to store and manipulate. An additional parameter of interest is the latency of program execution, which varies as well across different predictor designs. As evident from our experiments, a predictor design that performs the best in terms of prediction accuracy, does not always excel in terms of

latency and energy. A similar trend is often observed for the other parameters as well. This necessitates the development of a framework for branch predictor design and evaluation that can simultaneously optimize multiple parameters. To the best of our knowledge, the trade-off between accuracy, latency and energy has been mostly studied empirically through workload simulations. This motivated us to consider a systematic optimization framework that examines all the parameters in a mathematical setting for branch predictor designs.

Contributions of this work

- We present a rank aggregation based predictor selection framework that considers the different parameters of concern (e.g. latency, energy, accuracy) and determines the aggregate rank of predictors such that individual parameters are well captured.
- We propose an extended rank aggregation method for predictor selection with parameters associated with user specific priorities [50] considering the fact that the parameters of consideration in our problem context are incomparable and uncorrelated, and therefore, not suitable candidates for averaging-based aggregation.
- We perform experiments on the Siemens [38] and the SPEC 2006 [52] benchmark programs, with multiple state-of-the-art predictor designs and a variety of parameters to show the benefits of our framework. We also report comparative performance details on the aggregation methods used, along with their aggregation outcomes.

The framework proposed in this work may have multiple applications in the branch predictor design context. On one hand, our framework can be used to systematically evaluate and compare the performance of a new predictor design with respect to existing ones, considering different parameters of interest. In the current process, designers typically carry out this benchmarking by examining the performance of the new design individually with respect to the different parameters, along with a comparison with other off-the-shelf predictors. An

important use of our framework can be in the choice of the components to be used in a hybrid predictor design. Our framework can be used to decide the predictors to use, based on all parameters.

This chapter is organized as follows. Section 6.2 presents an overview of this work. In Section 6.3, we discuss the predictor ranking framework as well as our methodology behind predictor selection. Section 6.4 describes the implementation details, experimental set-up used as well as the results of our experiments, while Section 6.5 concludes the chapter.

6.2 Motivating Example

To establish the motivation of this work, we adopt a similar philosophy as used in the popular Championship Branch Prediction (CBP) [33] configuration. We keep the total storage available to the predictors as constant, and carry out experiments on accuracy, energy and latency on the Siemens benchmarks, with different predictors being used for prediction. While TAGE [96][97] and its variants dominate when accuracy is the sole standpoint of comparison, our findings reveal widely varying predictor performance when the other parameters are considered, as shown in Table 6.1.

A close analysis of the results show that it is often the case that the predictor with the best prediction accuracy, consumes more energy than the one which is ranked second according to prediction accuracy or a predictor which ensures lowest latency. Table 6.1 shows 3 different parameter values, namely, the average prediction accuracy, processor core energy, and the time taken for program execution with six contemporary predictors (GShare, GAg, TAGE, PAp, GAp and Bimodal) [107][64][78][97] on 3 Siemens benchmark programs [38]. As is evident from Table 6.1, the performances vary widely across predictors for every program. As an example, for the tcas, schedule2 and printtokens programs, though TAGE ranks first according to prediction accuracy, it is not the best according to the total time and

Benchmark Programs	**Predictor**	**Average Prediction Accuracy** (%)	**Core Energy** (nanojoule)	**Time Taken** (micro second)
tcas	Bimodal	86	95922	208
	GAg	83	96046	200
	GAp	79	95899	205
	GShare	84	95933	212
	PAp	78	95829	203
	TAGE	87	95956	203
schedule2	Bimodal	85	146136	320
	GAg	84	145949	354
	GAp	82	146088	322
	GShare	83	146088	320
	PAp	79	145899	339
	TAGE	87	145906	333
printtokens	Bimodal	85	149627	344
	GAg	83	149678	361
	GAp	81	149690	351
	GShare	83	149728	337
	PAp	78	149721	339
	TAGE	87	149740	322

Table 6.1: Performance variation on Siemens benchmarks

energy expenditure and falls behind compared to the other predictors. It is therefore quite a non-trivial task to select the best predictor for a given application, since the performances often vary widely, considering the different parameters. This motivated us to examine the problem of parameter-driven predictor selection in a multi-objective optimization setting.

6.3 The multi-parameter predictor selection framework

In this section, we describe our predictor ranking and selection framework. Our framework takes in the following inputs:

- A set of predictors $\mathcal{P}_1, \mathcal{P}_2, \ldots, \mathcal{P}_n$

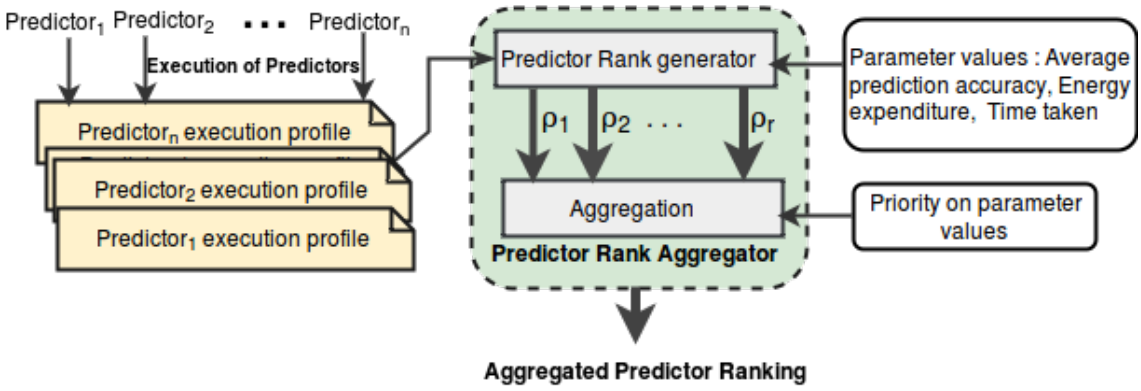


Figure 6.1: The overall system architecture

- A set of performance parameters $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_r$ (accuracy, energy, latency etc.).

- The predictor profiles $\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_n$, where each $\mathcal{R}_i$ corresponds to a predictor $\mathcal{P}_i$ for $i = 1, \ldots, n$. Each $\mathcal{R}_i$ represents a tuple $< \mathcal{K}_1^{(i)}, \mathcal{K}_2^{(i)}, \ldots, \mathcal{K}_r^{(i)} >$ containing the values of the parameters for the predictor $\mathcal{P}_i$, for a given application program.

- For each predictor $\mathcal{P}_i$, each parameter may as well be optionally associated with a weight / priority value $\mathcal{W}_1^{(i)}, \mathcal{W}_2^{(i)}, \ldots, \mathcal{W}_r^{(i)}$. It may be the case that while prediction accuracy has the highest priority for one program, energy consumption may have the highest priority for another. This can be tuned by setting priority values appropriately.

Figure 6.1 shows the overall architecture of our framework. For a given program, the average performance outputs (the values of the parameters of interest) are monitored and stored in a profile database when run with different predictors on a set of designated test inputs.

Once the execution profiles are generated through simulation and the parameter values recorded, they are passed on to the aggregation block. We begin by generating the individual parameter based rankings $\rho_1, \rho_2, \ldots, \rho_r$ of the predictors, where ρ_i corresponds to the rank of the predictor based on the parameter $\mathcal{K}_i$ for $i = 1, 2, \ldots, r$. This is achieved by a

Parameters	Rank list for predictors				
Parameter-1	TAGE (P_4)	GAp (P_3)	GAg (P_2)	GShare (P_1)	PAp (P_5)
Parameter-2	PAp (P_5)	GAg (P_2)	GShare (P_1)	GAp (P_3)	TAGE (P_4)
Parameter-3	PAp (P_5)	TAGE (P_4)	GShare (P_1)	GAp (P_3)	GAg (P_2)
Parameter-4	GShare (P_1)	GAg (P_2)	TAGE (P_4)	PAp (P_5)	GAp (P_3)

Table 6.2: Rank lists on 4 parameters for 5 predictors

sorted ordering of the predictor set based on each parameter. The next step generates an aggregate ranking ρ_A of the predictors considering the individual parameter based rankings $\rho_1, \rho_2, \ldots, \rho_r$ obtained above with the parameters associated with priority values if any. We discuss the two major building blocks of the aggregator and the analysis carried out therein.

6.3.1 Predictor Rank Generator

This is the first building block of our framework. This block takes as input the parameter values (e.g., prediction accuracy, energy expenditure, latency etc.) of the predictors. The ranking of the predictors with respect to a parameter $\mathcal{K}_i$ is generated by sorting the predictors based on their values of $\mathcal{K}_i$ either in ascending order or in descending order depending on the ordering objective of $\mathcal{K}_i$. As an example, we sort the predictors based on the values of the prediction accuracy in descending order. In other words, a predictor with higher prediction accuracy should come earlier in the ranking (and therefore, have a lower rank value) than a predictor with comparatively lower prediction accuracy. On the other hand, predictors are sorted in ascending order with respect to the energy expenditure. The output of this block is a set of ranked lists $\rho_1, \rho_2, \ldots, \rho_r$ for each parameter $\mathcal{K}_1, \mathcal{K}_2, \ldots, \mathcal{K}_r$, as shown in Table 6.2 for an example benchmark program. For ease of illustration in the following discussion, we have used P_1 for GShare, P_2 for GAg, P_3 for GAp, P_4 for TAGE and P_5 for PAp, as shown in Table 6.2.

6.3.2 Aggregation

This is the next stage of our framework. The objective of this block is to analyze and select the best predictor from the individual parameter based rankings. The number of parameters may be many in number, and hence, generate many ranked lists. To add to this, a predictor may have widely varying rank positions in the different rank lists, thus making the aggregation task even harder. A predictor $\mathcal{P}_l$ for example, may have a very high rank position in some rank list ρ_i, but a very low rank position in another rank list ρ_j. The objective of this stage is to come up with an aggregate ranking considering the individual predictor based rankings. In this work, we propose two different formulations to achieve this aggregation, as explained below.

6.3.2.1 Predictor ranking based on the Kemeny method

This method based on [68] generates an aggregate rank list that minimizes the number of pair-wise disagreements between predictor pairs between the individual rank lists. Intuitively, if a predictor P_i is ranked before a predictor P_j in most of the individual rank lists, the aggregate list should reflect this. Consider two distinct predictors P_i and P_j. A binary variable $Z_{P_iP_j}$ is defined as:

$$Z_{P_iP_j} = \begin{cases} 1, & \text{if the aggregate list positions } P_i \text{ before } P_j \\ 0, & \text{otherwise.} \end{cases}$$

Consider another variable $n_{P_iP_j}$ which denotes the number of individual parameter rank lists that rank P_i ahead of P_j and $n_{P_jP_i}$ has a similar interpretation. Such variables are defined for each predictor pair. The objective of aggregation is to come up with an aggregate list that minimizes the number of disagreements with the individual parameter rank lists.

This is expressed by the following optimization formulation.

$$\text{Minimize} \sum_{P_i \neq P_j} (n_{P_i P_j} \times Z_{P_j P_i}), \text{ subject to}$$

$$Z_{P_i P_j} \in \{0, 1\} \tag{6.1}$$

$$Z_{P_i P_j} + Z_{P_j P_i} = 1 \tag{6.2}$$

$$(\forall P_i, P_j, P_k : P_i \neq P_j, P_i \neq P_k, P_j \neq P_k),$$

$$Z_{P_i P_j} + Z_{P_j P_k} + Z_{P_k P_i} \leq 2 \tag{6.3}$$

The first constraint states that the $Z_{P_i P_j}$ variables are binary. Constraint 2 expresses for any predictor pair P_i, P_j, one of them has to be ranked ahead of the other, thus both the binary variables cannot be 0 or 1. The third constraint is the transitivity constraint between predictor triplets. Without this, the aggregate ranking may assign values to the binary variables with a cyclic majority: P_i ahead of P_j, P_j ahead of P_k, and P_k ahead of P_i. The output of the optimization is a value (0 / 1) for each binary variable $Z_{P_i P_j}$, that leads to the minimum value of the objective, subject to the constraints, and a final aggregate rank list. The top position in the list is the predictor our framework finds to be most suitable, considering all the parameters. In general, predictors in the lower rank values of this list are more suitable candidates than ones in the higher rank values, when all the parameters are considered.

Example 6.1 Consider the example in Table 6.2, with 4 rank lists. We apply rank aggregation on these with all parameters using Equation 6.1 to Equation 6.3. This creates an aggregate rank list that minimizes the number of pair-wise disagreements between the

individual rank lists. The $n_{P_iP_j}$ values are:

$$n_{P_1P_2} = 2,\ n_{P_2P_1} = 2,\ n_{P_1P_3} = 3,\ n_{P_3P_1} = 1,\ n_{P_1P_4} = 2,\ n_{P_4P_1} = 2,\ n_{P_1P_5} = 2,\ n_{P_5P_1} = 2,$$
$$n_{P_2P_3} = 2,\ n_{P_3P_2} = 2,\ n_{P_2P_4} = 2,\ n_{P_4P_2} = 2,\ n_{P_2P_5} = 2,\ n_{P_5P_2} = 2,\ n_{P_3P_4} = 1,\ n_{P_4P_3} = 3,$$
$$n_{P_3P_5} = 1,\ n_{P_5P_3} = 3,\ n_{P_4P_5} = 2,\ n_{P_5P_4} = 2.$$

Using Equations 6.1 to 6.3, the final aggregate list is P_2, P_5, P_4, P_1, P_3. This shows that P_2 is the best predictor, when all the parameters are taken into consideration. □

6.3.2.2 Predictor ranking based on weighted Kemeny

We now discuss the aggregate ranking computation of predictors with the individual parameters having priority values. In this case, each parameter has a relative normalized priority value associated with each rank list. To compute the aggregate ranking, we incorporate the priority on each parameter ranking. We modify the definition of $n_{\mathcal{P}_i\mathcal{P}_j}$ as the priority induced sum of the parameter values for which $\mathcal{P}_i$ comes earlier than $\mathcal{P}_j$ in the ranking based on those parameters. To formalize, we define a variable $L^{(p)}_{\mathcal{P}_i\mathcal{P}_j}$ as:

$$L^{(p)}_{\mathcal{P}_i\mathcal{P}_j} = \begin{cases} 1 & \text{, if } \mathcal{P}_i \text{ is positioned ahead of } \mathcal{P}_j \\ & \text{in parameter p} \\ 0 & \text{, Otherwise} \end{cases} \tag{6.4}$$

The variable $n_{\mathcal{P}_i\mathcal{P}_j}$ is defined as:

$$n_{\mathcal{P}_i\mathcal{P}_j} = \sum_{p=1}^{r} \mathcal{W}_p L^{(p)}_{\mathcal{P}_i\mathcal{P}_j} \tag{6.5}$$

where, $\mathcal{W}_p$ is the priority on the parameter $\mathcal{K}_p$. The remaining constraints and the objective remain same as earlier.

Our formulation for aggregation with priorities is generic in nature. If we need to consider only a single parameter, we can put the other priorities to 0. Similarly, we can use our framework to focus only on a subset of the parameters as well. In summary, our framework provides the flexibility to evaluate multiple different parameters in a systematic way with appropriate priorities assigned. As an example, if we wish to perform an analysis with accuracy only as the factor for consideration, we can assign it the maximum priority of 1, and the rest to 0. However, if we wish to consider the other parameters as well, while treating accuracy as most important, we can adjust the weights to assign maximum value to accuracy and distribute the remaining weights among the other parameters (e.g. energy, latency etc.). Our framework can thus take care of different priorities and analysis objectives that a designer may have.

Example 6.2 Consider the same example as above with normalized priority values of Parameter-1, Parameter-2, Parameter-3 and Parameter-4 as 0.4, 0.3, 0.2 and 0.1 respectively. The values of the $n_{P_iP_j}$ variables are computed as:

$$n_{P_1P_2} = 0.2 + 0.1 = 0.3, n_{P_2P_1} = 0.4 + 0.3 = 0.7,$$
$$n_{P_1P_3} = 0.3 + 0.2 + 0.1 = 0.6, n_{P_3P_1} = 0.4, n_{P_1P_4} = 0.3 + 0.1 = 0.4,$$
$$n_{P_4P_1} = 0.4 + 0.2 = 0.6, n_{P_1P_5} = 0.4 + 0.1 = 0.5,$$
$$n_{P_5P_1} = 0.3 + 0.2 = 0.5, n_{P_2P_3} = 0.3 + 0.1 = 0.4,$$
$$n_{P_3P_2} = 0.4 + 0.2 = 0.6, n_{P_2P_4} = 0.3 + 0.1 = 0.4,$$
$$n_{P_4P_2} = 0.4 + 0.2 = 0.6, n_{P_2P_5} = 0.4 + 0.1 = 0.5, n_{P_5P_2} = 0.3 + 0.2 = 0.5,$$
$$n_{P_3P_4} = 0.3, n_{P_4P_3} = 0.4 + 0.2 + 0.1 = 0.7, n_{P_3P_5} = 0.4,$$
$$n_{P_5P_3} = 0.3 + 0.2 + 0.1 = 0.7, n_{P_4P_5} = 0.4 + 0.1 = 0.5, n_{P_5P_4} = 0.3 + 0.2 = 0.5$$

In this case, the final aggregate list is P_5, P_4, P_3, P_2, P_1.

Consider a different set of priority values of Parameter-1, Parameter-2, Parameter-3 and

Parameter-4 as 0.1, 0.2, 0.3 and 0.4 respectively. The $n_{P_iP_j}$ values are as below.

$$n_{P_1P_2} = 0.4 + 0.3 = 0.7, n_{P_2P_1} = 0.1 + 0.2 = 0.3,$$
$$n_{P_1P_3} = 0.4 + 0.3 + 0.2 = 0.9, n_{P_3P_1} = 0.1, n_{P_1P_4} = 0.2 + 0.4 = 0.6,$$
$$n_{P_4P_1} = 0.1 + 0.3 = 0.4, n_{P_1P_5} = 0.1 + 0.4 = 0.5,$$
$$n_{P_5P_1} = 0.3 + 0.2 = 0.5, n_{P_2P_3} = 0.2 + 0.4 = 0.6,$$
$$n_{P_3P_2} = 0.3 + 0.5 = 0.8, n_{P_2P_4} = 0.4 + 0.2 = 0.6,$$
$$n_{P_4P_2} = 0.3 + 0.1 = 0.4, n_{P_2P_5} = 0.1 + 0.4 = 0.5, n_{P_5P_2} = 0.3 + 0.2 = 0.5,$$
$$n_{P_3P_4} = 0.2, n_{P_4P_3} = 0.1 + 0.4 + 0.3 = 0.8, n_{P_3P_5} = 0.1,$$
$$n_{P_5P_3} = 0.3 + 0.2 + 0.4 = 0.9, n_{P_4P_5} = 0.4 + 0.1 = 0.5, n_{P_5P_4} = 0.3 + 0.2 = 0.5$$

In this case, the final aggregate list becomes P_1, P_2, P_4, P_5, P_3. ■

The above creates an aggregate rank list that reflects the different priorities assigned to the different parameters, based on the classical Kemeny aggregation.

6.3.2.3 Computational Hardness of Aggregation:

The disagreement minimization problem is known to be NP-hard [24] [41] in the case of 4 or more complete lists. The reduction can be shown from the feedback edge set problem [104]. Further, the Kemeny Method is expected to be computationally inefficient for a large number of parameters, due to its reliance on the optimization. We therefore, discuss a lightweight aggregation method inspired by the Borda count [28] [114]. The Borda count method can be implemented in linear time [92]. In Subsection 6.3.2.4 below, we show how the Borda count method improves the computation time with respect to the Kemeny method. The aggregation methods are different in the aggregation definition as well. On one side, the Kemeny method is concerned more about creating the aggregation list as close as possible to the ordering in the rank lists, however, in doing so, it does not give

importance to the relative positions of the predictors in the constituent lists in which a predictor defeats the other. The Borda count ranks the predictors based on the position in each list. In a generic setting, based on user priorities on latency or accuracy, any of the aggregation methods can be adopted. We thus build both the methods in our predictor ranking framework.

6.3.2.4 Predictor ranking based on Borda count

Borda's method [39] [44] is a positional method, it assigns a score corresponding to the positions in which a predictor appears within each individual ranked list, and the predictors are sorted by their total score. A primary advantage of positional methods is that they are computationally lightweight and can be implemented in linear time. This method first finds the score of every predictor P_i for every individual rank list R_j as:

$$S_{P_iR_j} = \text{number of predictors ranked below } P_i \text{ in } R_j$$

Further, it calculates the total score of P_i by summing up all its scores from all rank lists as:

$$Score_{P_i} = \sum_{j=1}^{k} S_{P_iR_j} \tag{6.6}$$

The final aggregated rank list is created by sorting predictors in decreasing order of total Borda score.

Example 6.3 Consider the example in Table 6.2, with 4 rank lists R_1, R_2, R_3 and R_4, based on 4 parameters. We apply rank aggregation on these with all parameters using Equation 6.6. The values of the $S_{P_iR_j}$ and $Score_{P_i}$ variables are shown below.

$$S_{P_1R_1} : 1,\ S_{P_1R_2} : 2,\ S_{P_1R_3}: 2,\ S_{P_1R_4} : 4$$

$$S_{P_2R_1}\text{: } 2,\ S_{P_2R_2}\text{: } 3,\ S_{P_2R_3}\text{: } 0,\ S_{P_2R_4} : 3$$

$$S_{P_3R_1} : 3,\ S_{P_3R_2} : 1,\ S_{P_3R_3} : 1,\ S_{P_3R_4} : 0$$

$$S_{P_4R_1} : 4,\ S_{P_4R_2} : 0,\ S_{P_4R_3} : 3,\ S_{P_4R_4} : 2$$

$$S_{P_5R_1} : 0,\ S_{P_5R_2}\text{: } 4,\ S_{P_5R_3} : 4,\ S_{P_5R_4} : 1$$

$$\text{Hence, } Score_{P_1} = 9,\ Score_{P_2} = 8,\ Score_{P_5} = 9,\ Score_{P_4} = 9,\ Score_{P_3} = 5$$

This creates an aggregate rank list that sorts the predictors according to their scores in decreasing order. The final aggregated list for the above has 4 choices, resolving ties arbitrarily:

P_1, P_4, P_5, P_2, P_3 / P_4, P_1, P_5, P_2, P_3 / P_1, P_5, P_4, P_2, P_3 / P_5, P_4, P_1, P_2, P_3 / P_5, P_1, P_4, P_2, P_3 / P_4, P_5, P_1, P_2, P_3. □

6.3.3 Predictor ranking on weighted Borda count

We now calculate the aggregate ranking of predictors using the Borda count method with the individual parameters having priority values [71]. In this case, as earlier, each parameter has a relative normalized priority value associated with each rank list. Considering priorities for each rank list, Eqn. 6.6 can be rewritten as :

$$Score_{P_i} = \sum_{j=1}^{k} \mathcal{W}_{R_j} S_{P_iR_j} \tag{6.7}$$

$\mathcal{W}_{R_j}$ is the priority on the parameter $\mathcal{K}_j$ for rank list R_j.

Example 6.4 Consider the same example as above with normalized priority values of Parameter-1, Parameter-2, Parameter-3 and Parameter-4 as 0.4, 0.3, 0.2 and 0.1 respectively. The values of the $S_{P_iR_j}$ and $Score_{P_i}$ variables are computed as shown below. We

apply rank aggregation on these with all parameters using Equations 6.6 and 6.7. The values of the $S_{P_iR_j}$ and $Score_{P_i}$ variables are computed as shown in the following.

$$S_{P_1R_1} : 1,\ S_{P_1R_2} : 2,\ S_{P_1R_3}: 2,\ S_{P_1R_4} : 4$$

$$S_{P_2R_1}: 2,\ S_{P_2R_2}: 3,\ S_{P_2R_3}: 0,\ S_{P_2R_4} : 3$$

$$S_{P_3R_1} : 3,\ S_{P_3R_2} : 1,\ S_{P_3R_3} : 1,\ S_{P_3R_4} : 0$$

$$S_{P_4R_1} : 4,\ S_{P_4R_2} : 0,\ S_{P_4R_3} : 3,\ S_{P_4R_4} : 2$$

$$S_{P_5R_1} : 0,\ S_{P_5R_2}: 4,\ S_{P_5R_3} : 4,\ S_{P_5R_4} : 1$$

$$Score_{P_1} = 1*0.4 + 2*0.3 + 2*0.2 + 4*0.1 = 1.8,$$

$$Score_{P_2} = 2*0.4 + 3*0.3 + 0*0.2 + 3*0.1 = 2.0,$$

$$Score_{P_3} = 3*0.4 + 1*0.3 + 1*0.2 + 0*0.1 = 1.7,$$

$$Score_{P_4} = 4*0.4 + 0*0.3 + 3*0.2 + 2*0.1 = 2.4,$$

$$Score_{P_5} = 0*0.4 + 4*0.3 + 4*0.2 + 1*0.1 = 2.1$$

Hence, the final aggregated list for the above is: P_4, P_5, P_2, P_1, P_3. □

6.4 Implementation and Results

The proposed predictor ranking framework has been implemented in Python [7] using the lpsolve [15] optimization library. Figure 6.2 shows the internal architecture of our predictor aggregation framework. For each program, predictor rank lists are first created for each performance parameter in stage 1. Each performance parameter has user defined priority / weight values that may be same or may vary across the programs. In the second stage, we take the predictor rank lists and weight vectors as inputs and produce the aggregated rank lists, using Borda, Kemeny, weighted Borda and weighted Kemeny methods. The Kemeny and weighted Kemeny methods internally call lpsolve to produce the aggregated lists. The output from lpsolve is used to produce the final aggregated list by our framework.

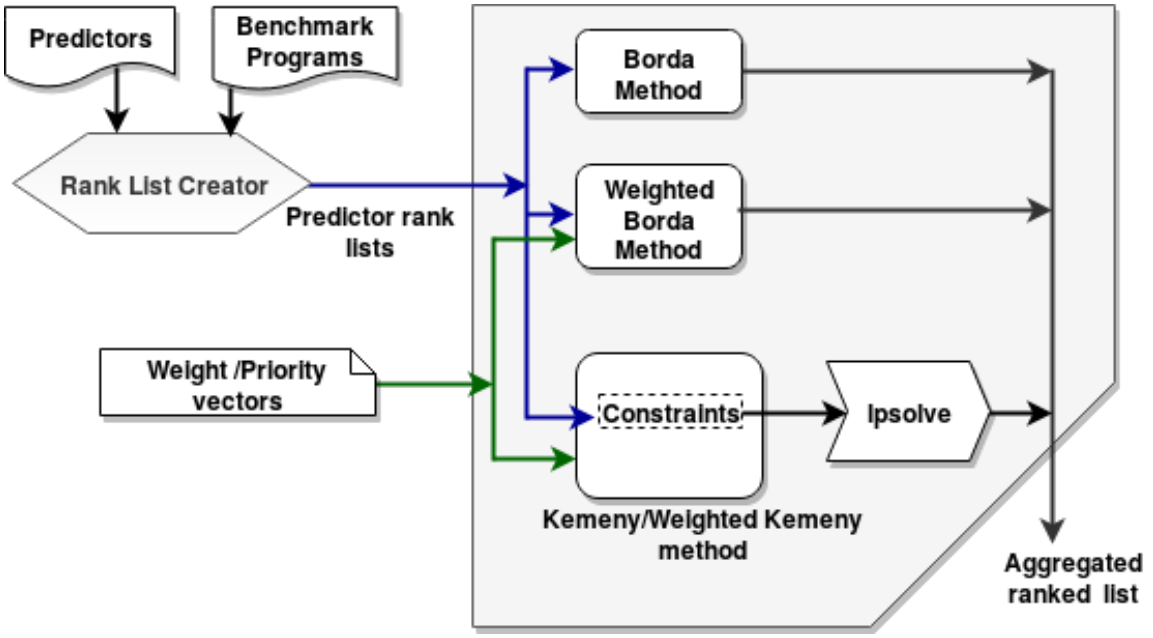


Figure 6.2: Predictor rank aggregator

6.4.1 Experimental setup

In this work, we use 6 dynamic branch predictors: GShare, GAg, GAp, PAp, TAGE and Bimodal implemented on top of the Tejas architectural simulator [11]. We fix the storage budget for all these predictors as given in the Championship Branch Prediction competitions (CBP) [33]. We use a maximum storage budget of 32KB and modify the predictor designs present inside the simulator codebase to perform with this constraint. We report our experiments of running Tejas simulations on the Siemens benchmark programs [38] and SPEC 2006 [52] benchmarks. In this work, we use only C/C++ programs of these benchmarks since Tejas can simulate only C/C++ executables.

6.4.2 Results on Siemens benchmarks

We now discuss our results on the Siemens benchmark programs. Table 6.3 shows the 4 different rank lists generated by the Predictor Rank Generator component for 6 predictors. For every individual predictor, these rank lists are prepared based on 4 different parameter values - the average prediction accuracy, processor core energy expenditure, latency / time taken and branch predictor energy expenditure. It is interesting to note that the rank lists are sometimes quite different. The aggregated rank list for each program is generated by the aggregator component using Kemeny and Borda count methods based on all the 4 parameters, both with and without priorities as shown in Tables 6.3 - 6.5.

It is observed that the TAGE predictor performs best according to prediction accuracy across all the programs of this benchmark. However, TAGE is not the best every time while considering the other three parameters, namely core energy, branch predictor energy and execution time. Hence, TAGE can not be the best predictor for all the programs when all the parameters are taken care of. For this reason, TAGE does not appear on top of all the aggregated lists generated by the Kemeny or the Borda methods. Table 6.3 also shows that the aggregated list for each program computed using the two aggregation methods, differ significantly as well, even when priorities are not considered. For example, in case of the Tcas program, the predictor GAg is on the top of the aggregated list that is computed using the Kemeny method and TAGE is behind GAg. However, TAGE is on top of the aggregated list that is computed using the Borda count and GAg is behind. If we compare these lists, we can see that GAg is behind TAGE in the two rank lists based on accuracy and core energy, and ahead of TAGE in other rank lists (time taken and branch predictor energy). This shows the variation in predictor performance when different metrics are considered for selection and the results also vary across benchmarks. In case of the Replace program, GAg is not the best in any rank list, however it is selected as the best predictor by both the Kemeny and Borda methods. For our experiments, we use

seven combinations of normalized relative priority values to see the change in the aggregate lists, and the results are shown in Table 6.4 and Table 6.5. These results show, if we set the priority such that only the average prediction accuracy is considered (assigned weight as 1.0), it generates the same aggregated list as the rank list generated by considering the average prediction accuracy. Similarly, for the other three parameters as well, we have the same results as shown in Table 6.3. In another case, when equal priority is assigned to processor energy expenditure and branch predictor energy and no priority is assigned to average prediction accuracy and execution time, we do not have TAGE on the top of the aggregated list for all programs though TAGE gives the maximum average prediction accuracy for all programs compared to all the predictors used in this work. For the rest of the experiments, we modify the priority ordering for all parameters in different ways. For one case, equal priority is assigned to average prediction accuracy and execution time and no priority is assigned to the processor energy expenditure and branch predictor energy expenditure. For another case, by assigning equal priority to average prediction accuracy and execution time with no priority value assigned to processor energy and branch predictor energy expenditure, we obtain a different set of predictor orderings. We also perform an experiment with the weight assigned to average prediction accuracy as 0.4 (dominating parameter), 0.3 each to the core energy and branch predictor energy and no priority to the execution time parameter. It can be observed that the aggregate lists generated vary quite widely for each benchmark program across the experiments. This shows the utility of our framework in helping an architect decide and evaluate the crucial components in a predictor setup. For each benchmark program, the top position in the aggregated list in each table is the predictor our framework recommends to be most suitable, considering all the parameters. In all the tables shown in the following, Pred. Acc. stands for Prediction Accuracy, Acc. stands for Average Prediction Accuracy (%), CE stands for Core Energy in nanojoule, TT stands for Time taken in seconds, and bPred stands for branch predictor energy in nanojoule.

6.4.3 Results on SPEC 2006 benchmarks

For the SPEC 2006 benchmark programs, the first 1 billion instructions from each benchmark are simulated. Tables 6.6 and 6.7 show different rank lists generated by the Predictor Rank Generator component of our framework for the six predictors for the SPEC 2006 integer and floating point benchmark programs respectively. These tables also show the aggregated lists generated by the Kemeny and Borda count methods without considering the priorities on the parameters. For these benchmark programs, no predictor consistently performs best in any of the rank lists and the programs have different rank lists as well, as can be seen from the tables. The same combination of normalized relative priority values as used for the Siemens simulations are also used here to see the change in the aggregate lists, and the results are shown in Tables 6.8, 6.9, 6.10 and 6.11.

It is interesting to observe that with respect to our experiments, the findings of [88] is not supported. In [88], the authors have examined all the different testcases in the SPEC 2006 benchmarks and came to a conclusion that a subset of 6 integer and 8 floating pont tests are sufficiently representative of the entire test-suite. This is important since an architecture research now no longer needs to run experiments on the entire suite, but can conclude the findings based on only the subsets. However, for our case, this does not hold since all the benchmarks show distinct variation in performance. In this case as well, the top position in the aggregated list is the predictor our framework finds to be most suitable, considering all the parameters.

To show the relative computational efficiency of the two methods for aggregation, we show in Figure 6.3, a comparative plot of the run-times of these two procedures. Computation time taken by all the methods are in milliseconds. As expected, Borda fares better in

terms of run-time, this being a lightweight procedure. In our experiments, the number of parameters is restricted to 4, hence the Kemeny-based method often achieves a comparable run-time performance. These aggregated lists are generated statically, hence do not impact the overall system performance.

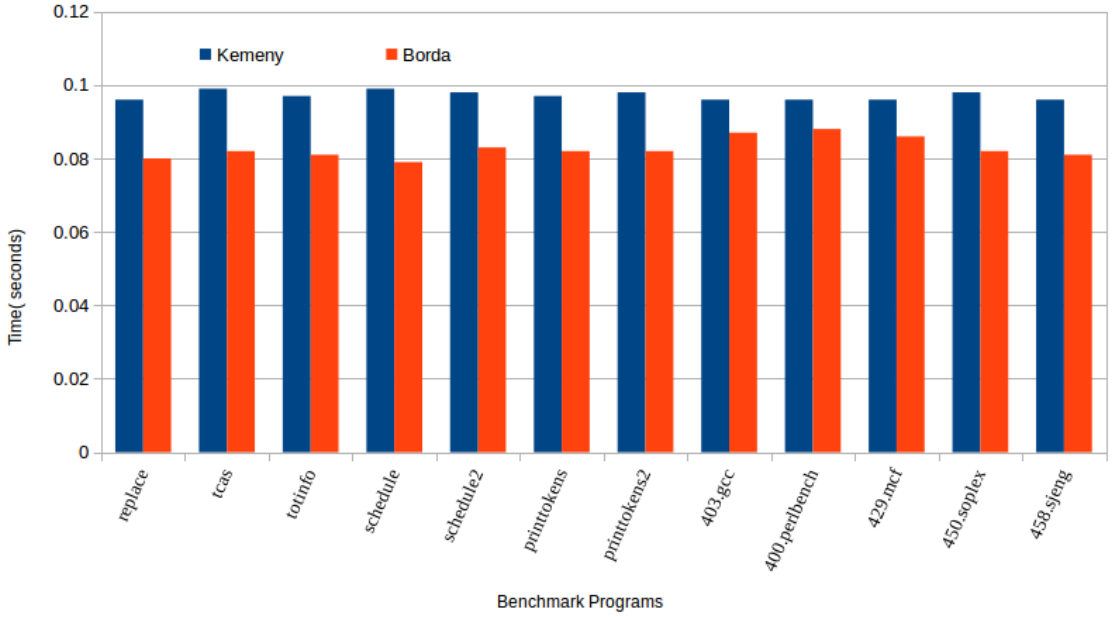


Figure 6.3: Comparative analysis of time taken by Kemeny and Borda methods

Benchmark Programs	Predictor Rank Lists				Aggregated List	
	Average Pred. Accu. (%)	Core Energy (nanojoule)	Execution time (second)	bPred Energy (nanojoule)	Kemeny Method	Borda Method
replace	TAGE	TAGE	GAp	GAp	GAg	GAg
	Bimodal	GAg	GAg	GAg	GAp	GAp
	GAg	GAp	Bimodal	PAp	Bimodal	TAGE
	GShare	PAp	PAp	Bimodal	PAp	Bimodal
	GAp	GShare	TAGE	TAGE	TAGE	PAp
	PAp	Bimodal	GShare	GShare	GShare	GShare
tcas	TAGE	PAp	GAg	GAg	GAg	TAGE
	Bimodal	GAp	TAGE	PAp	PAp	GAg
	GShare	Bimodal	PAp	TAGE	TAGE	PAp
	GAg	GShare	GAp	GAp	GAp	GAp
	GAp	TAGE	Bimodal	Bimodal	Bimodal	Bimodal
	PAp	GAg	GShare	GShare	GShare	GShare
totinfo	TAGE	Bimodal	TAGE	TAGE	TAGE	TAGE
	GShare	GAp	PAp	PAp	Bimodal	Bimodal
	GAg	GAg	GAp	Bimodal	GAp	GAp
	Bimodal	GShare	Bimodal	GAp	GShare	PAp
	GAp	PAp	GShare	GShare	GAg	GShare
	PAp	TAGE	GAg	GAg	PAp	GAg
schedule	TAGE	TAGE	PAp	PAp	PAp	TAGE
	Bimodal	GAp	GShare	GShare	GShare	GShare
	GShare	PAp	TAGE	TAGE	TAGE	PAp
	GAg	GShare	GAp	Bimodal	Bimodal	GAp
	GAp	GAg	Bimodal	GAp	GAp	Bimodal
	PAp	Bimodal	GAg	GAg	GAg	GAg
schedule2	TAGE	PAp	Bimodal	Bimodal	Bimodal	Bimodal
	Bimodal	TAGE	GShare	GAp	GShare	TAGE
	GAg	GAg	GAp	GShare	GAp	GAp
	GShare	GAp	TAGE	TAGE	TAGE	GShare
	GAp	GShare	PAp	PAp	PAp	PAp
	PAp	Bimodal	GAg	GAg	GAg	GAg
printtokens	TAGE	Bimodal	TAGE	TAGE	TAGE	TAGE
	Bimodal	GAg	GShare	GShare	GShare	Bimodal
	GAg	GAp	PAp	PAp	PAp	GShare
	GShare	PAp	Bimodal	GAp	Bimodal	PAp
	GAp	GShare	GAp	Bimodal	GAp	GAp
	PAp	TAGE	GAg	GAg	GAg	GAg
printtokens2	TAGE	Bimodal	TAGE	TAGE	TAGE	Bimodal
	Bimodal	GAg	PAp	PAp	Bimodal	TAGE
	GAg	GAp	Bimodal	Bimodal	GAg	GAg
	GShare	GShare	GAg	GAg	GShare	PAp
	GAp	TAGE	GAp	GShare	GAp	GAp
	PAp	PAp	GShare	GAp	PAp	GShare

Table 6.3: Aggregated rank lists for Siemens benchmarks

Benchmark Programs	Aggregated List						
	Weighted Kemeny Method						
	Acc:0.5 CE:0.0 TT:0.5 bPred:0.0	Acc:0.0 CE:0.5 TT:0.0 bPred:0.5	Acc:0.4 CE:0.3 TT:0.0 bPred:0.3	Acc:1.0 CE:0.0 TT:0.0 bPred:0.0	Acc:0.0 CE:1.0 TT:0.0 bPred:0.0	Acc:0.0 CE:0.0 TT:1.0 bPred:0.0	Acc:0.0 CE:0.0 TT:0.0 bPred:1.0
replace	TAGE	GAp	TAGE	TAGE	TAGE	GAp	GAp
	GAg	GAg	GAg	Bimodal	GAg	GAg	GAg
	Bimodal	PAp	GAp	GAg	GAp	Bimodal	PAp
	GShare	Bimodal	PAp	GShare	PAp	PAp	Bimodal
	GAp	TAGE	Bimodal	GAp	GShare	TAGE	TAGE
	PAp	GShare	GShare	PAp	Bimodal	GShare	GShare
tcas	TAGE	GAg	PAp	TAGE	PAp	GAg	GAg
	Bimodal	PAp	TAGE	Bimodal	GAp	TAGE	PAp
	GShare	GAp	Bimodal	GShare	Bimodal	PAp	TAGE
	GAg	Bimodal	GShare	GAg	GShare	GAp	GAp
	GAp	GShare	GAg	GAp	TAGE	Bimodal	Bimodal
	PAp	TAGE	GAp	PAp	GAg	GShare	GShare
totinfo	TAGE	PAp	TAGE	TAGE	Bimodal	TAGE	TAGE
	GAp	Bimodal	Bimodal	GShare	GAp	PAp	PAp
	PAp	GAp	GAp	GAg	GAg	GAp	Bimodal
	GShare	GAg	GShare	Bimodal	GShare	Bimodal	GAp
	GAg	GShare	GAg	GAp	PAp	GShare	GShare
	Bimodal	TAGE	PAp	PAp	TAGE	GAg	GAg
schedule	GShare	PAp	TAGE	TAGE	TAGE	PAp	PAp
	TAGE	GShare	GShare	Bimodal	GAp	GShare	GShare
	Bimodal	TAGE	Bimodal	GShare	PAp	TAGE	TAGE
	GAg	Bimodal	GAp	GAg	GShare	GAp	Bimodal
	GAp	GAp	PAp	GAp	GAg	Bimodal	GAp
	PAp	GAg	GAg	PAp	Bimodal	GAg	GAg
schedule2	Bimodal	TAGE	TAGE	TAGE	PAp	Bimodal	Bimodal
	GShare	PAp	Bimodal	Bimodal	TAGE	GShare	GAp
	GAp	GAg	GAg	GAg	GAg	GAp	GShare
	TAGE	Bimodal	GAp	GShare	GAp	TAGE	TAGE
	PAp	GAp	GShare	GAp	GShare	PAp	PAp
	GAg	GShare	PAp	PAp	Bimodal	GAg	GAg
printtokens	TAGE	TAGE	TAGE	TAGE	Bimodal	TAGE	TAGE
	Bimodal	PAp	Bimodal	Bimodal	GAg	GShare	GShare
	GAg	Bimodal	GAg	GAg	GAp	PAp	PAp
	GShare	GAg	GShare	GShare	PAp	Bimodal	GAp
	GAp	GAp	GAp	GAp	GShare	GAp	Bimodal
	PAp	GShare	PAp	PAp	TAGE	GAg	GAg
printtokens2	TAGE	Bimodal	TAGE	TAGE	Bimodal	TAGE	TAGE
	Bimodal	GAg	Bimodal	Bimodal	GAg	PAp	PAp
	GAg	GAp	GAg	GAg	GAp	Bimodal	Bimodal
	GAp	GShare	GShare	GShare	GShare	GAg	GAg
	GShare	TAGE	GAp	GAp	TAGE	GAp	GShare
	PAp	PAp	PAp	PAp	PAp	GShare	GAp

Table 6.4: Aggregated rank lists for Siemens benchmarks with weighted Kemeny

Benchmark Programs	Aggregated List Weighted Borda Method						
	Acc:0.5 CE:0.0 TT:0.5 bPred:0.0	Acc:0.0 CE:0.5 TT:0.0 bPred:0.5	Acc:0.4 CE:0.3 TT:0.0 bPred:0.3	Acc:1.0 CE:0.0 TT:0.0 bPred:0.0	Acc:0.0 CE:1.0 TT:0.0 bPred:0.0	Acc:0.0 CE:0.0 TT:1.0 bPred:0.0	Acc:0.0 CE:0.0 TT:0.0 bPred:1.0
replace	GAg	GAp	TAGE	TAGE	TAGE	GAp	GAp
	Bimodal	GAg	GAg	Bimodal	GAg	GAg	GAg
	GAp	PAp	GAp	GAg	GAp	Bimodal	PAp
	TAGE	Bimodal	Bimodal	GShare	PAp	PAp	Bimodal
	GShare	TAGE	PAp	GAp	GShare	TAGE	TAGE
	PAp	GShare	GShare	PAp	Bimodal	GShare	GShare
tcas	TAGE	PAp	TAGE	TAGE	PAp	GAg	GAg
	GAg	GAp	Bimodal	Bimodal	GAp	TAGE	PAp
	Bimodal	GAg	PAp	GShare	Bimodal	PAp	TAGE
	GAp	TAGE	GAg	GAg	GShare	GAp	GAp
	GShare	Bimodal	GAp	GAp	TAGE	Bimodal	Bimodal
	PAp	GShare	GShare	PAp	GAg	GShare	GShare
totinfo	TAGE	Bimodal	TAGE	TAGE	Bimodal	TAGE	TAGE
	GShare	GAp	Bimodal	GShare	GAp	PAp	PAp
	GAp	TAGE	GShare	GAg	GAg	GAp	Bimodal
	Bimodal	PAp	GAp	Bimodal	GShare	Bimodal	GAp
	PAp	GAg	GAg	GAp	PAp	GShare	GShare
	GAg	TAGE	PAp	PAp	TAGE	GAg	GAg
schedule	TAGE	TAGE	TAGE	TAGE	TAGE	PAp	PAp
	GShare	PAp	GShare	Bimodal	GAp	GShare	GShare
	Bimodal	GShare	PAp	GShare	PAp	TAGE	TAGE
	PAp	GAp	Bimodal	GAg	GShare	GAp	Bimodal
	GAp	Bimodal	GAp	GAp	GAg	Bimodal	GAp
	GAg	GAg	GAg	PAp	Bimodal	GAg	GAg
schedule2	Bimodal	GAp	TAGE	TAGE	PAp	Bimodal	Bimodal
	AGE	TAGE	Bimodal	Bimodal	TAGE	GShare	GAp
	GShare	PAp	GAp	GAg	GAg	GAp	GShare
	GAp	Bimodal	GAg	GShare	GAp	TAGE	TAGE
	GAg	GShare	GShare	GAp	GShare	PAp	PAp
	PAp	GAg	PAp	PAp	Bimodal	GAg	GAg
printtokens	TAGE	Bimodal	TAGE	TAGE	Bimodal	TAGE	TAGE
	GShare	GAp	Bimodal	Bimodal	GAg	GShare	GShare
	Bimodal	GShare	GAg	GAg	GAp	PAp	PAp
	GAg	TAGE	GShare	GShare	PAp	Bimodal	GAp
	PAp	PAp	GAp	GAp	GShare	GAp	Bimodal
	GAp	GAg	PAp	PAp	TAGE	GAg	GAg
printtokens2	TAGE	Bimodal	Bimodal	TAGE	Bimodal	TAGE	TAGE
	Bimodal	GAg	TAGE	Bimodal	GAg	PAp	PAp
	GAg	TAGE	GAg	GAg	GAp	Bimodal	Bimodal
	PAp	PAp	GShare	GShare	GShare	GAg	GAg
	GAp	GAp	GAp	GAp	TAGE	GAp	GShare
	GShare	GShare	PAp	PAp	PAp	GShare	GAp

Table 6.5: Aggregated rank lists for Siemens benchmarks with weighted Borda

Benchmark Programs	Predictor Rank Lists				Aggregated List	
	Average Pred. Accu. (%)	Core Energy (nanojoule)	Execution time (seconds)	bPred Energy (nanojoule)	Kemeny Method	Borda Method
403.gcc	TAGE	PAp	PAp	PAp	PAp	PAp
	PAp	TAGE	TAGE	TAGE	TAGE	TAGE
	Bimodal	GAp	GAp	GAp	GAp	GAp
	GAp	Bimodal	Bimodal	Bimodal	Bimodal	Bimodal
	GShare	GShare	GAg	GAg	GAg	GAg
	GAg	GAg	GShare	GShare	GShare	GShare
400.perlbench	TAGE	PAp	TAGE	GAp	GAp	TAGE
	Bimodal	GAp	GAp	TAGE	TAGE	GAp
	GAp	TAGE	Bimodal	PAp	GShare	PAp
	GShare	GAg	GShare	GShare	GAg	GShare
	GAg	GShare	GAg	GAg	PAp	Bimodal
	PAp	Bimodal	PAp	Bimodal	Bimodal	GAg
429.mcf	TAGE	GAp	TAGE	GAp	GAp	GAp
	Bimodal	GShare	GAp	GShare	Bimodal	TAGE
	GAp	Bimodal	Bimodal	Bimodal	TAGE	Bimodal
	GShare	TAGE	GShare	TAGE	GShare	GShare
	PAp	PAp	GAg	PAp	PAp	PAp
	GAg	GAg	PAp	GAg	GAg	GAg
456.hmmer	GShare	GShare	GShare	Bimodal	GShare	GShare
	GAg	GAg	GAg	PAp	GAg	GAg
	TAGE	GAp	PAp	GShare	Bimodal	Bimodal
	Bimodal	Bimodal	Bimodal	TAGE	PAp	PAp
	PAp	PAp	TAGE	GAp	TAGE	TAGE
	GAp	TAGE	GAp	GAg	GAp	GAp
458.sjeng	TAGE	GAp	GAp	Bimodal	GAp	Bimodal
	GAg	GShare	GShare	PAp	GShare	GAp
	PAp	Bimodal	Bimodal	GAg	Bimodal	PAp
	Bimodal	PAp	PAp	TAGE	PAp	GAg
	GAp	GAg	GAg	GShare	GAg	GShare
	GShare	TAGE	TAGE	GAp	TAGE	TAGE
401.bzip2	GAp	PAp	GAg	Bimodal	Bimodal	PAp
	TAGE	GAg	Bimodal	PAp	PAp	Bimodal
	Bimodal	GShare	GShare	GShare	GShare	GAg
	PAp	GAp	PAp	TAGE	GAg	GShare
	GShare	TAGE	TAGE	GAg	GAp	TAGE
	GAg	Bimodal	GAp	GAp	TAGE	GAp
401.libquantum2	GAg	GShare	Bimodal	PAp	GAg	GShare
	GShare	GAg	GAg	TAGE	GShare	GAg
	PAp	PAp	GShare	GShare	PAp	PAp
	Bimodal	TAGE	GAp	GAp	TAGE	Bimodal
	TAGE	GAp	PAp	Bimodal	GAp	TAGE
	GAp	Bimodal	TAGE	GAg	Bimodal	GAp

Table 6.6: Aggregated rank lists for SPEC 2006 Integer benchmarks

Benchmark Programs	Predictor Rank Lists				Aggregated List	
	Average Pred. Accu. (%)	Core Energy (nanojoule)	Execution time (seconds)	bPred Energy (nanojoule)	Kemeny Method	Borda Method
447.dealII	Bimodal	GShare	Bimodal	Bimodal	Bimodal	Bimodal
	TAGE	TAGE	GAg	TAGE	TAGE	TAGE
	GAp	GAp	TAGE	GShare	GAp	GAp
	PAp	Bimodal	GAp	GAp	GAg	GShare
	GAg	GAg	GShare	GAg	GShare	GAg
	GShare	PAp	PAp	PAp	PAp	PAp
444.namd	GShare	GAp	GShare	PAp	GShare	GShare
	GAg	PAp	GAg	GAp	GAg	Bimodal
	Bimodal	TAGE	Bimodal	Bimodal	PAp	PAp
	TAGE	Bimodal	TAGE	TAGE	GAp	GAp
	PAp	GShare	PAp	GAg	Bimodal	GAg
	GAp	GAg	GAp	GShare	TAGE	TAGE
453.povray	GShare	GAp	GShare	GAg	GShare	GAg
	GAg	Bimodal	GAg	TAGE	GAg	TAGE
	TAGE	PAp	TAGE	Bimodal	TAGE	GShare
	Bimodal	TAGE	Bimodal	PAp	Bimodal	Bimodal
	PAp	GShare	PAp	GAp	PAp	PAp
	GAp	GAg	GAp	GShare	GAp	GAp
470.lbm	PAp	TAGE	PAp	TAGE	TAGE	PAp
	GAg	PAp	GAg	PAp	PAp	GAg
	GShare	Bimodal	GShare	GAg	GAg	TAGE
	Bimodal	GAp	GAp	Bimodal	GShare	Bimodal
	GAp	GAg	Bimodal	GShare	Bimodal	GShare
	TAGE	GShare	TAGE	GAp	GAp	GAp
450.soplex	TAGE	GAp	GAp	TAGE	PAp	TAGE
	PAp	GAg	PAp	PAp	TAGE	PAp
	Bimodal	GShare	TAGE	Bimodal	Bimodal	GAp
	GAp	Bimodal	Bimodal	GAg	GAp	Bimodal
	GAg	PAp	GAg	GShare	GAg	GAg
	GShare	TAGE	GShare	GAp	GShare	GShare
482.sphinx3	GShare	PAp	GShare	TAGE	GShare	TAGE
	GAg	GAp	GAg	Bimodal	TAGE	GAg
	TAGE	Bimodal	TAGE	GAp	Bimodal	Bimodal
	Bimodal	TAGE	Bimodal	GAg	GAp	GShare
	GAp	GAg	GAp	PAp	GAg	GAp
	PAp	GShare	PAp	GShare	PAp	PAp
433.milc	PAp	Bimodal	TAGE	TAGE	TAGE	TAGE
	Bimodal	GAg	Bimodal	GAg	Bimodal	Bimodal
	TAGE	TAGE	GShar	Bimodal	GAg	PAp
	GAg	GShare	PAp	PAp	GShare	GAg
	GAg	PAp	GAp	GAp	PAp	GShare
	GShare	GAp	GAg	GShare	GAp	GAp

Table 6.7: Aggregated rank lists for SPEC 2006 Floating Point benchmarks

Benchmark Programs	**Aggregated List**						
	Weighted Kemeny Method						
	Acc:0.5 CE:0.0 TT:0.5 bPred:0.0	Acc:0.0 CE:0.5 TT:0.0 bPred:0.5	Acc:0.4 CE:0.3 TT:0.0 bPred:0.3	Acc:1.0 CE:0.0 TT:0.0 bPred:0.0	Acc:0.0 CE:1.0 TT:0.0 bPred:0.0	Acc:0.0 CE:0.0 TT:1.0 bPred:0.0	Acc:0.0 CE:0.0 TT:0.0 bPred:1.0
403.gcc	TAGE	PAp	PAp	TAGE	PAp	PAp	PAp
	PAp	TAGE	TAGE	PAp	TAGE	TAGE	TAGE
	GAp	GAp	GAp	Bimodal	GAp	GAp	GAp
	Bimodal	Bimodal	Bimodal	GAp	Bimodal	Bimodal	Bimodal
	GAg	GAg	GShare	GShare	GShare	GAg	GAg
	GShare	GShare	GAg	GAg	GAg	GShare	GShare
400.perlbench	TAGE	PAp	GAp	TAGE	PAp	TAGE	GAp
	GAp	GAp	TAGE	Bimodal	GAp	GAp	TAGE
	Bimodal	TAGE	PAp	GAp	TAGE	Bimodal	PAp
	GShare	GShare	GShare	GShare	GAg	GShare	GShare
	GAg	GAg	GAg	GAg	GShare	GAg	GAg
	PAp	Bimodal	Bimodal	PAp	Bimodal	PAp	Bimodal
429.mcf	TAGE	GAp	GAp	TAGE	GAp	TAGE	GAp
	Bimodal	GShare	GShare	Bimodal	GShare	GAp	GShare
	GAp	Bimodal	Bimodal	GAp	Bimodal	Bimodal	Bimodal
	GShare	TAGE	TAGE	GShare	TAGE	GShare	TAGE
	PAp	PAp	PAp	PAp	PAp	GAg	PAp
	GAg	GAg	GAg	GAg	GAg	PAp	GAg
458.sjeng	GAp	GAp	Bimodal	TAGE	GAp	GAp	Bimodal
	GShare	Bimodal	PAp	GAg	GShare	GShare	PAp
	Bimodal	PAp	GAg	PAp	Bimodal	Bimodal	GAg
	GAg	GAg	TAGE	Bimodal	PAp	PAp	TAGE
	PAp	TAGE	GAp	GAp	GAg	GAg	GShare
	TAGE	GShare	GShare	GShare	TAGE	TAGE	GAp
401.bzip2	GAp	PAp	GAp	GAp	PAp	GAg	Bimodal
	Bimodal	GAg	TAGE	TAGE	GAg	Bimodal	PAp
	GShare	GShare	Bimodal	Bimodal	GShare	GShare	GShare
	PAp	GAp	PAp	PAp	GAp	PAp	TAGE
	GAg	TAGE	GShare	GShare	TAGE	TAGE	GAg
	TAGE	Bimodal	GAg	GAg	Bimodal	GAp	GAp
401.libquantum	Bimodal	GShare	GShare	GAg	GShare	Bimodal	PAp
	GAg	GAg	GAg	GShare	GAg	GAg	TAGE
	GShare	PAp	PAp	PAp	PAp	GShare	GShare
	PAp	TAGE	TAGE	Bimodal	TAGE	GAp	GAp
	TAGE	GAp	GAp	TAGE	GAp	PAp	Bimodal
	GAp	Bimodal	Bimodal	GAp	Bimodal	TAGE	GAg
462.h264ref	GShare	GAp	GShare	GShare	Bimodal	GShare	GAp
	GAg	GShare	GAg	GAg	TAGE	GAg	PAp
	PAp	TAGE	PAp	PAp	GShare	Bimodal	GShare
	Bimodal	Bimodal	Bimodal	Bimodal	GAg	TAGE	TAGE
	TAGE	GAg	TAGE	TAGE	PAp	PAp	Bimodal
	GAp	PAp	GAp	GAp	GAp	GAp	GAg

Table 6.8: Aggregated rank lists for SPEC 2006 Integer benchmarks with weighted Kemeny

Benchmark Programs	Aggregated List						
	Weighted Kemeny Method						
	Acc:0.5 CE:0.0 TT:0.5 bPred:0.0	Acc:0.0 CE:0.5 TT:0.0 bPred:0.5	Acc:0.4 CE:0.3 TT:0.0 bPred:0.3	Acc:1.0 CE:0.0 TT:0.0 bPred:0.0	Acc:0.0 CE:1.0 TT:0.0 bPred:0.0	Acc:0.0 CE:0.0 TT:1.0 bPred:0.0	Acc:0.0 CE:0.0 TT:0.0 bPred:1.0
447.dealII	Bimodal	GShare	Bimodal	Bimodal	GShare	Bimodal	Bimodal
	GAg	TAGE	TAGE	TAGE	TAGE	GAg	TAGE
	TAGE	GAp	GShare	GAp	GAp	TAGE	GShare
	GAp	Bimodal	GAp	PAp	Bimodal	GAp	GAp
	GShare	GAg	GAg	GAg	GAg	GShare	GAg
	PAp	PAp	PAp	GShare	PAp	PAp	PAp
444.namd	GShare	PAp	PAp	GShare	GAp	GShare	PAp
	GAg	GAp	GAp	GAg	PAp	GAg	GAp
	Bimodal	TAGE	Bimodal	Bimodal	TAGE	Bimodal	Bimodal
	TAGE	Bimodal	TAGE	TAGE	Bimodal	TAGE	TAGE
	PAp	GShare	GShare	PAp	GShare	PAp	GAg
	GAp	GAg	GAg	GAp	GAg	GAp	GShare
453.povray	GShare	GAp	GAg	GShare	GAp	GShare	GAg
	GAg	Bimodal	TAGE	GAg	Bimodal	GAg	TAGE
	TAGE	PAp	Bimodal	TAGE	PAp	TAGE	Bimodal
	Bimodal	TAGE	PAp	Bimodal	TAGE	Bimodal	PAp
	PAp	GShare	GAp	PAp	GShare	PAp	GAp
	GAp	GAg	GShare	GAp	GAg	GAp	GShare
470.lbm	PAp	TAGE	TAGE	PAp	TAGE	PAp	TAGE
	GAg	PAp	PAp	GAg	PAp	GAg	PAp
	GShare	GAg	GAg	GShare	Bimodal	GShare	GAg
	GAp	Bimodal	Bimodal	Bimodal	GAp	GAp	Bimodal
	Bimodal	GShare	GShare	GAp	GAg	Bimodal	GShare
	TAGE	GAp	GAp	TAGE	GShare	TAGE	GAp
450.soplex	TAGE	TAGE	TAGE	TAGE	GAp	GAp	TAGE
	PAp	GAp	PAp	PAp	GAg	PAp	PAp
	Bimodal	PAp	Bimodal	Bimodal	GShare	TAGE	Bimodal
	GAp	GAg	GAp	GAp	Bimodal	Bimodal	GAg
	GAg	GShare	GAg	GAg	PAp	GAg	GShare
	GShare	TAGE	GShare	GShare	TAGE	GShare	GAp
482.sphinx3	GShare	GAp	TAGE	GShare	PAp	GShare	TAGE
	GAg	TAGE	Bimodal	GAg	GAp	GAg	Bimodal
	TAGE	Bimodal	GAp	TAGE	Bimodal	TAGE	GAp
	Bimodal	GAg	GAg	Bimodal	TAGE	Bimodal	GAg
	GAp	PAp	PAp	GAp	GAg	GAp	PAp
	PAp	GShare	GShare	PAp	GShare	PAp	GShare
433.milc	TAGE	Bimodal	Bimodal	PAp	Bimodal	TAGE	TAGE
	Bimodal	GAg	TAGE	Bimodal	GAg	Bimodal	GAg
	GShare	TAGE	GAg	TAGE	TAGE	GShare	Bimodal
	PAp	GShare	PAp	GAp	GShare	PAp	PAp
	GAp	PAp	GAp	GAg	PAp	GAp	GAp
	GAg	GAp	GShare	GShare	GAp	GAg	GShare

Table 6.9: Aggregated lists for SPEC 2006 Floating Point benchmarks with weighted Kemeny

Benchmark Programs	Aggregated List						
	Weighted Borda Method						
	Acc:0.5 CE:0.0 TT:0.5 bPred:0.0	Acc:0.0 CE:0.5 TT:0.0 bPred:0.5	Acc:0.4 CE:0.3 TT:0.0 bPred:0.3	Acc:1.0 CE:0.0 TT:0.0 bPred:0.0	Acc:0.0 CE:1.0 TT:0.0 bPred:0.0	Acc:0.0 CE:0.0 TT:1.0 bPred:0.0	Acc:0.0 CE:0.0 TT:0.0 bPred:1.0
403.gcc	TAGE	PAp	PAp	TAGE	PAp	PAp	PAp
	PAp	TAGE	TAGE	PAp	TAGE	TAGE	TAGE
	GAp	GAp	GAp	Bimodal	GAp	GAp	GAp
	Bimodal	Bimodal	Bimodal	GAp	Bimodal	Bimodal	Bimodal
	GAg	GAg	GShare	GShare	GShare	GAg	GAg
	GShare	GShare	GAg	GAg	GAg	GShare	GShare
400.perlbench	TAGE	GAp	TAGE	TAGE	PAp	TAGE	GAp
	GAp	PAp	GAp	Bimodal	GAp	GAp	TAGE
	Bimodal	TAGE	PAp	GAp	TAGE	Bimodal	PAp
	GShare	GAg	GShare	GShare	GAg	GShare	GShare
	GAg	GShare	Bimodal	GAg	GShare	GAg	GAg
	PAp	Bimodal	GAg	PAp	Bimodal	PAp	Bimodal
429.mcf	TAGE	GAp	GAp	TAGE	GAp	TAGE	GAp
	GAp	GShare	Bimodal	Bimodal	GShare	GAp	GShare
	Bimodal	Bimodal	GShare	GAp	Bimodal	Bimodal	Bimodal
	GShare	TAGE	TAGE	GShare	TAGE	GShare	TAGE
	GAg	PAp	PAp	PAp	PAp	GAg	PAp
	PAp	GAg	GAg	GAg	GAg	PAp	GAg
458.sjeng	GAp	Bimodal	Bimodal	TAGE	GAp	GAp	Bimodal
	GAg	PAp	PAp	GAg	GShare	GShare	PAp
	TAGE	GAp	GAg	PAp	Bimodal	Bimodal	GAg
	Bimodal	GShare	TAGE	Bimodal	PAp	PAp	TAGE
	PAp	GAg	GAp	GAp	GAg	GAg	GShare
	GShare	TAGE	GShare	GShare	TAGE	TAGE	GAp
401.bzip2	Bimodal	PAp	PAp	GAp	PAp	GAg	Bimodal
	GAp	GShare	Bimodal	TAGE	GAg	Bimodal	PAp
	GAg	GAg	GAp	Bimodal	GShare	GShare	GShare
	TAGE	Bimodal	TAGE	PAp	GAp	PAp	TAGE
	GShare	TAGE	GShare	GShare	TAGE	TAGE	GAg
	PAp	GAp	GAg	GAg	Bimodal	GAp	GAp
libquantum	GAg	GShare	GShare	GAg	GShare	Bimodal	PAp
	GShare	PAp	PAp	GShare	GAg	GAg	TAGE
	Bimodal	TAGE	GAg	PAp	PAp	GShare	GShare
	PAp	GAg	TAGE	Bimodal	TAGE	GAp	GAp
	GAp	GAP	Bimodal	TAGE	GAp	PAp	Bimodal
	TAGE	Bimodal	GAp	GAp	Bimodal	TAGE	GAg
462.h264ref	GShare	GShare	GShare	GShare	Bimodal	GShare	GAp
	GAg	TAGE	PAp	GAg	TAGE	GAg	PAp
	Bimodal	Bimodal	Bimodal	PAp	GShare	Bimodal	GShare
	PAp	GAp	GAg	Bimodal	GAg	TAGE	TAGE
	TAGE	PAp	TAGE	TAGE	PAp	PAp	Bimodal
	GAp	GAg	GAp	GAp	GAp	GAp	GAg

Table 6.10: Aggregated rank lists for SPEC 2006 Integer benchmarks with weighted Borda

Benchmark Programs	Aggregated List						
	Weighted Borda Method						
	Acc:0.5 CE:0.0 TT:0.5 bPred:0.0	Acc:0.0 CE:0.5 TT:0.0 bPred:0.5	Acc:0.4 CE:0.3 TT:0.0 bPred:0.3	Acc:1.0 CE:0.0 TT:0.0 bPred:0.0	Acc:0.0 CE:1.0 TT:0.0 bPred:0.0	Acc:0.0 CE:0.0 TT:1.0 bPred:0.0	Acc:0.0 CE:0.0 TT:0.0 bPred:1.0
447.dealII	Bimodal	GShare	Bimodal	Bimodal	GShare	Bimodal	Bimodal
	TAGE	TAGE	TAGE	TAGE	TAGE	GAg	TAGE
	GAp	Bimodal	GAp	GAp	GAp	TAGE	GShare
	GAg	GAp	GShare	PAp	PAp	GAp	GAp
	PAp	GAg	GAg	GAp	GAg	GShare	GAg
	GShare	PAp	PAp	GShare	GShare	PAp	PAp
444.namd	GShare	GAp	PAp	GShare	GAp	GShare	PAp
	GAg	PAp	GAp	GAg	PAp	GAg	GAp
	Bimodal	TAGE	Bimodal	Bimodal	TAGE	Bimodal	Bimodal
	TAGE	Bimodal	GShare	TAGE	Bimodal	TAGE	TAGE
	PAp	GAg	TAGE	PAp	GShare	PAp	GAg
	GAp	GShare	GAg	GAp	GAg	GAp	GShare
453.povray	GShare	Bimodal	GAg	GShare	GAp	GShare	GAg
	GAg	GAp	TAGE	GAg	Bimodal	GAg	TAGE
	TAGE	TAGE	Bimodal	TAGE	PAp	TAGE	Bimodal
	Bimodal	GAg	GShare	Bimodal	TAGE	Bimodal	PAp
	PAp	PAp	PAp	PAp	GShare	PAp	GAp
	GAp	GShare	GAp	GAp	GAg	GAp	GShare
470.lbm	PAp	TAGE	PAp	PAp	TAGE	PAp	TAGE
	GAg	PAp	TAGE	GAg	PAp	GAg	PAp
	GShare	Bimodal	GAg	GShare	Bimodal	GShare	GAg
	GAp	GAg	Bimodal	Bimodal	GAp	GAp	Bimodal
	Bimodal	GAp	GShare	GAp	GAg	Bimodal	GShare
	TAGE	GShare	GAp	TAGE	GShare	TAGE	GAp
450.soplex	TAGE	GAg	TAGE	TAGE	GAp	GAp	TAGE
	PAp	GAp	PAp	PAp	GAg	PAp	PAp
	GAp	TAGE	Bimodal	Bimodal	GShare	TAGE	Bimodal
	Bimodal	Bimodal	GAp	GAp	Bimodal	Bimodal	GAg
	GAg	PAp	GAg	GAg	PAp	GAg	GShare
	GShare	GShare	GShare	GShare	TAGE	GShare	GAp
482.sphinx3	GShare	GAp	TAGE	GShare	PAp	GShare	TAGE
	GAg	TAGE	Bimodal	GAg	GAp	GAg	Bimodal
	TAGE	Bimodal	GAp	TAGE	Bimodal	TAGE	GAp
	Bimodal	PAp	GAg	Bimodal	TAGE	Bimodal	GAg
	GAp	GAg	GShare	GAp	GAg	GAp	PAp
	PAp	GShare	PAp	PAp	GShare	PAp	GShare
433.milc	TAGE	GAg	Bimodal	PAp	Bimodal	TAGE	TAGE
	Bimodal	TAGE	TAGE	Bimodal	GAg	Bimodal	GAg
	PAp	Bimodal	PAp	TAGE	TAGE	GShare	Bimodal
	GAp	PAp	GAg	GAp	GShare	PAp	PAp
	GShare	GShare	GAp	GAg	PAp	GAp	GAp
	GAg	GAp	GShare	GShare	GAp	GAg	GShare

Table 6.11: Aggregated lists for SPEC 2006 Floating Point benchmarks with weighted Borda

6.5 Summary

The motivation behind this work is to develop a systematic framework that can consider multiple performance parameters and find the aggregated rank of each predictor for a program. When a new predictor comes in, it is possible to know the exact position of this predictor with respect to the already existing ones. This knowledge can be of great help to decide whether this new predictor can give any benefit when implemented in real hardware. We believe that our framework can be of great value for evaluating a new prediction policy. Given the fact that our framework is purely a pre-processing step, and can be done prior to program execution, there are no additional performance slowdowns due to our mechanism. We expect our framework to be an important useful piece for designing efficient combinations of predictors, which have been popular in literature.

Chapter 7

Performance attacks on branch predictors in Simultaneous Multithreading Processors

7.1 Introduction

This chapter extends our study on branch predictors to execution environments with concurrency support. While our previous contributions outlined in the preceding chapters have been around efficient strategies for predictor design and evaluation, the objective of this chapter is to present a new direction of research on predictor attacks.

Different side channel attacks on branch prediction buffers have been well studied in literature [17][19][30]. Mitigation of covert channels through branch predictor has also been studied in [47]. A fine-grained attack *Branchscope* is shown on the directional branch predictor in [48]. In this case, the attack targets complex hybrid branch predictors with unknown

organization. This work demonstrates how an attacker can force these predictors to switch to a simple 1-level mode to simplify the direction recovery. In this work. in particular, we study performance slowdown attacks in Simultaneous Multithreading processors, executing multiple concurrent threads, and show how the presence of the predictor can be detrimental to performance, in a low resource budget environment.

In modern processor architectures, Simultaneous Multi-Threading (SMT) [43] is used to improve overall performance of program execution. It allows multiple threads of execution working simultaneously on a processor to better utilize the processor resources. The operating system scheduler and the hardware thread dispatcher underneath seamlessly manage the multiple threads of execution with context switches between the threads at runtime. Indeed, most modern processors today utilize the benefits of SMT with multiple threads working on the different cores with individual predictor components and using separate predictor history tables for individual threads to avoid inter-thread interference[91]. However, in a resource constrained embedded environment (e.g. a mobile processor), managing separate predictor tables is often infeasible, and a single shared table [47] turns out to be the only choice to mitigate concerns on high resource usage and energy dissipation.

The setup we consider in this work is an embedded processor with SMT support, and we study performance attacks on these shared structures. The main motivation behind our attack exploitation is the observation that most embedded processors with shared structures suffer significantly in terms of accuracy of prediction, and in turn, performance slowdown due to the resulting pipeline activity that has to be performed as a mis-prediction penalty, as discussed earlier.

Our analysis of context switching performance for SMT execution on the popular Siemens software benchmarks [52], with multiple independent applications running in different threads mapped to a single core implemented with shared predictor tables, reveals an interesting finding – there is usually a decrease in branch prediction accuracy (and increase in mispredic-

tion) in concurrent execution, in comparison to the accuracy achieved when the applications are run in isolation as shown in Figure 7.1. While pollution effects on the shared cache in multi-threaded execution have been studied in literature [113], effects on branch prediction have been relatively less examined [47] [91], to the best of our knowledge. In our setup, our findings reveal that the expected overall performance gain in program execution is reduced as well, due to the extra overhead of handling branch mis-predictions, often leading to a performance slowdown. An in-depth analysis reveals that this is a result of extensive interference on the shared predictor tables and registers on which the branch predictors operate, resulting out of the frequent switching between the different applications running in multiple threads, leading to performance slowdown. The prediction information of one application running in one thread stored in some shared table is used and overwritten by another application running in a different thread, once context switches during execution. This often leads to a negative interference among the threads, thereby leading to an incorrect direction for prediction, and therefore, accuracy degradation and slowdown. This motivates us to study the interference phenomenon on the predictor tables in a concurrent multithreaded environment. In particular, we work on a strategy to slowdown application performance by carefully crafting threads that can mimic the execution of an ongoing application, and alter its predictor table contents. This main contribution of this work is outlined below.

7.1.1 Contributions of this work

- We show how to exploit the negative interference between threads in a shared-table SMT single-core processor to slow down the performance of a benign application.

- We propose an attack methodology that can automatically create variants of a benign application, which when dispatched in a concurrently executing thread, can definitively induce negative interference on the benign one. Our methodology exploits the

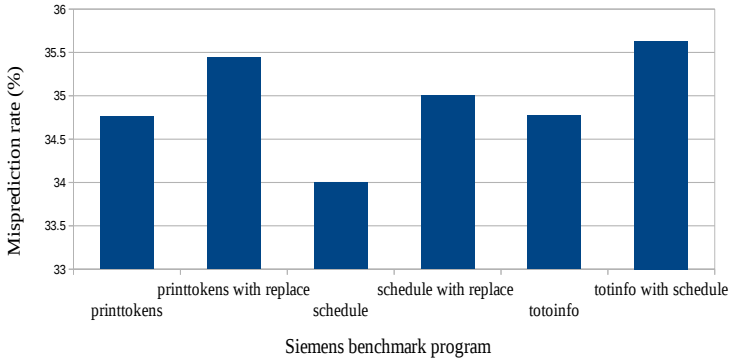


Figure 7.1: Misprediction rate for Siemens benchmark programs

control flow structure of a given benign application to create the variant. Section 7.3 presents the details of the attack methodology.

- We perform experiments on the Siemens benchmarks [38] to demonstrate the effect of such performance attacks. Experimental results show that the prediction accuracy degrades leading to performance slowdowns with our proposed attack policy.

The rest of the chapter is organized as follows: Section 7.2 presents an overview of the interference phenomenon that may occur for a predictor in SMT mode. Section 7.3 elaborates our method of attack construction for negative interference. In Section 7.4, we show the performance effects that result due to such attacks, while Section 7.5 concludes the work.

7.2 Predictor table interference in SMT execution

We use a simple Bimodal predictor to illustrate our attack methodology, on a SMT processor with a shared predictor table configuration, and the same Bimodal components being used for branch prediction for multiple threads. The architectural overview of the bimodal predictor is already discussed in Chapter 2, we revisit the structure once again for better

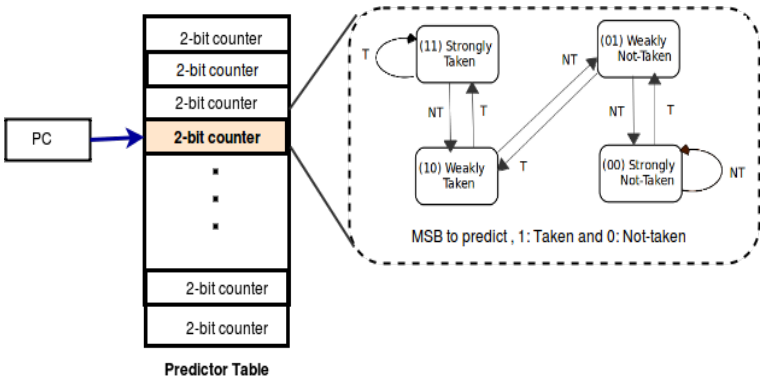


Figure 7.2: Bimodal branch predictor

understanding of our attack methodology. We choose this predictor for simplicity of illustration and ease of demonstration, attacks with other predictors can similarly be worked out. A bimodal predictor is a simple dynamic predictor that maintains a two bit saturating counter for every branch of a program for prediction. The main data structure of this predictor is a prediction table that stores the two-bit counter state for each branch, as shown in Figure 7.2. The counter maintains four different states 00 (strongly not-taken), 01 (weakly not-taken), 10 (weakly taken) and 11 (strongly taken) defined by the 2 bits of it, as shown in Figure 7.2. The counter transitions from one state to another in response to a taken (T) or not-taken (NT) outcome resulting from the execution of one or more branch instructions. Each bit of the two-bit counter plays a different role. The most significant bit, called the direction bit is used to track the direction of branches. If the counter is in states 01 or 00, the branch is predicted as NT. When it is in states 10 or 11, the prediction is T. The least significant bit provides a hysteresis which prevents the direction bit from immediately changing when a mis-prediction occurs. When a branch instruction is encountered during program execution, its address is used to index into the appropriate location in the predictor table, and the state of the 2-bit counter is used for predicting the direction of the branch. We now explain the working of this predictor in the context of SMT execution, and the interference generated.

We consider two programs of the Siemens software benchmarks, *printokens* and *tcas*. Consider a situation where *branch n* of *printokens* comes with an address *101010101* (i.e. the program counter PC value), and a predictor table entry X is accessed with the 5 LSB bits (10101), shown with a dotted line in Figure 7.3. X gives the final prediction according to the current state of the two-bit counter stored in it and updates the state with the actual outcome after the branch gets resolved and the actual outcome is known.

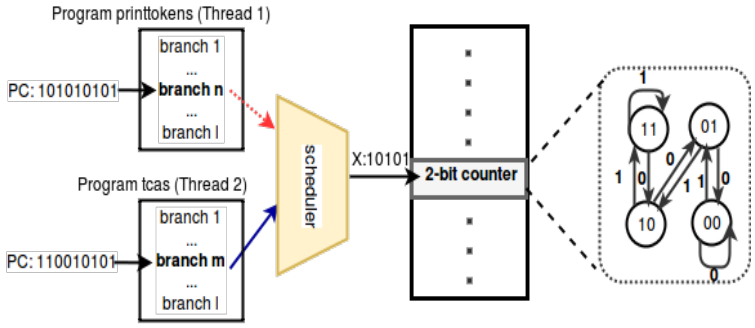


Figure 7.3: Predictor table interference of two programs

Consider the present state of the two bit counter stored in X is 10, hence it predicts the branch direction as taken since the most significant bit (MSB) is 1. If the actual outcome of *branch n* is 1, its state changes to 11 to give the predicted direction as taken, otherwise it changes to 01 to give the predicted direction as not-taken. Consider the actual outcome as 0, which implies that the current state of this two bit counter becomes 01, and when this table entry is accessed in future, it provides the prediction as not-taken. Now, due to SMT, *branch m* from the *tcas* program is selected with the current PC value as 110010101. If we take the least significant five bits from it to index the table, we point to the same entry X for this different program counter (shown with a solid line). Hence, the predicted direction for *branch m* is not-taken, since the current state of the two bit counter is 01. Now, the actual outcome of this *branch m*, which is 1, changes the counter state to 10 to give the

prediction as taken. Hence, the branch behavior stored for *printtokens* for a specific pattern is incorrectly altered with, and updated by *tcas* for another branch. Now, when the same branch of *printtokens* occurs again in future, the same PHT entry X is accessed again for prediction. It predicts the branch direction as taken - which is incorrect and cannot give the expected result, since a not-taken direction was stored for that pattern, when *branch n* of *printtokens* was processed, but overwritten when *branch m* of *tcas* was processed.

Context switching between threads generates significant negative interference that ultimately has an impact on overall prediction accuracy as well as overall performance, since increase in mis-prediction rate has an adverse effect on energy consumption due to increase in the number of wrong path executions. This phenomenon of negative interference is what we utilise to attack a benign application by creating a carefully crafted variant, as explained in the following.

7.3 Attack Methodology

We now present the details of our proposed attack creation and execution. Our proposal is based on some assumptions on the underlying architecture model, as outlined below.

7.3.1 Assumptions

In a context-switched SMT execution model, we assume that the two programs (benign and the variant created by us) run concurrently, with one instruction or a block of instructions from each, taking turns in a round robin schedule. Round Robin schedulers are quite popularly used for handling tasks in real time embedded environments, due to the simplicity of implementation and the property of no starvation. Further, we assume that the programs

are co-located on the same core, this assumption is needed since the branch predictor unit is shared on the same physical core, but not across the different cores of a multi-core processor architecture [47]. In this work, our branch predictor uses a shared configuration that shares the predictor table [91] and branch history register (BHR) among all the threads as shown in Figure 7.4. For the shared configuration, we consider a simple bimodal predictor that takes only the address to index the branch predictor table.

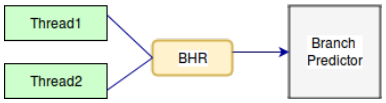


Figure 7.4: Branch Predictor Configuration for SMT

7.3.2 Attack creation

For a program P, we create a variant P' (we also call it a *clone*) that is dispatched in a concurrent thread to spoil the prediction information kept by the predictor for P. This variant program is a replica of the actual one except that all the corresponding branch conditions are flipped. In the illustration below, only conditional branches are considered. The variant creation process has three major steps, as shown in Algorithm 9. We illustrate each step using the printtokens program fragment, shown in Figure 7.5.

- **MakeClone:** It is the main method. It takes the benign program P and for each statement of P, creates the variant P' by calling the following functions.
- **CopyOnClone:** This method takes every statement of P and checks if it is a conditional statement. For any statement other than a branch, it simply copies it into program P'. Hence, statements 1, 2 and 3 of program P are just copied into the variant program P' (corresponding statements 1, 2 and 3 are created), as shown in Figure 7.5. For every branch statement, it calls the function *IfInversion* to generate

Program P(actual program)	Program P'(clone program)

1: static int next_state(state,ch)
2: int state;
3: char ch;
{
4: if(state < 0) //branch 1
5: return(state);
 	
6: if(base[state]+ch >= 0) { //branch 2
7: if(check[base[state]+ch] == state)
//branch 3
8: return(next[base[state]+ch]);
9: else
10 return(next_state(default1[state],ch));
}
11: else
12: return(next_state(default1[state],ch));
}
	1:static int next_state(state,ch)
	2:int state;
	3:char ch;
	{
	4: if(!(state < 0)){} //branch 1
	5: else
	6: return(state);
	7: if(!(base[state]+ch >= 0)) { //branch 2
	8: return(next_state(default1[state],ch));
	}
	9: else{
	10: if(!(check[base[state]+ch] == state))
	//branch 3
	11: return(next_state(default1[state],ch));
	12: else
	13: return(next[base[state]+ch]);
	}
	}

Figure 7.5: Program fragment of printokens and its variant

a corresponding branch with the condition flipped and inserts it into P'.

- **IfInversion:** This is the most important function of this variant creation process. It flips the condition of every branch of P and writes it into P'. Since we do not want to change the control flow of the actual program in P', we interchange the content of the *if* block with the *else* block of each branch of P in P'. Figure 7.5 shows the interchange of the *if* and *else* blocks of branch 2 of program P (statement 6) in program P' (branch 2, statement 7). This interchange also happens for branch 3 of program P and it can be seen that statement 8 which is within the if block of this branch in P is copied into the else block (statement 13) of the same branch in P'. For every branch of P that has no *else* block, it creates a dummy else block in P' and copies all the statements within the if block in P into the corresponding dummy else block in P'. Figure 7.5 shows the *else* created for branch 1 (statement 4) of program P in P' (statement 5). Statement 5 of P, that is within the if block of this branch is copied into this newly created dummy else block in P' (statement 6).

ALGORITHM 9: CloneCreation

```
1  Input: Program P
2  Output: Clone program P'
3  Method MakeClone()
4  begin
5      for each statement S in application P do
6          CopyOnClone(S)
7  Method CopyOnClone(S)
8  begin
9      if S contains a branch then
10         IfInversion(S)
11     else
12         Copy S onto variant
13 Method IfInversion(S)
14 begin
15     flip the condition of S and write it on P'
16     if else block of S is present in P then
17         for each statement S_1 in true path of S do
18             call CopyOnClone(S_1) on corresponding else block of P'
19         for each statement S_2 in else path of S do
20             call CopyOnClone(S_2) on corresponding if block of P'
21     else
22     Create else block of S in P'
23     for each statement S' of S do
24         call CopyOnClone(S') on the created else block
```

The above algorithm recursively goes inside if and $else$ blocks and creates the corresponding statements, with the conditions flipped and the statement blocks interchanged. We now explain how this program P', when dispatched as a concurrent thread, can spoil the prediction information for P when both are running on the same input.

7.3.3 Attack execution

In SMT mode, the two programs P and P' are made to run on the same inputs with a round robin scheduler context switching between the two threads. Since the programs are almost identical to each other except in the branch conditions (flipped in the clone program), the same address value (program counter PC) will be generated for both the programs. Note that PC is a virtual address and the same is used for indexing the predictor tables. Now we illustrate how our clone program causes negative interference for every branch of the benign program for this branch predictor configuration. In this configuration, the branch predictor table and the branch history register are shared by the running threads. The bimodal predictor uses only the PC value to index the predictor table entry for prediction. The prediction information stored by the actual program is flipped by the clone program since the branch condition of the two programs are opposite to each other. Thus, when the two programs are made to run on the same input, the outcome in P will be opposite to the one in P'.

Example 7.1 *Figure 7.6 shows that for the branch at line 2 of program P, PHT entry* PHT_1 *is accessed (marked as 1 in Figure 7.6) and the current state of the two bit counter that is stored in that entry gives the prediction (2). If the current state is 10, the prediction is taken. Now, this state is modified according to the actual outcome of the branch. Consider the actual outcome is* not-taken *and the current state becomes 01 to give the prediction as not-taken. In future, for the same branch, this state is stored in that entry (3). Now in SMT mode, for the same branch (branch 2) of our variant program* P'*, this entry is accessed since the PC value for both are same. As before, its current value is used for prediction and this value is modified and stored according to the actual outcome (steps are marked as 1', 2' and 3'). The outcome of this branch is* taken *for program P' (as opposite to P) and so the state of the counter is changed to 10. Hence, a mis-prediction occurs for branch 2 of P*

in future since its prediction information is destructively modified by its variant. □

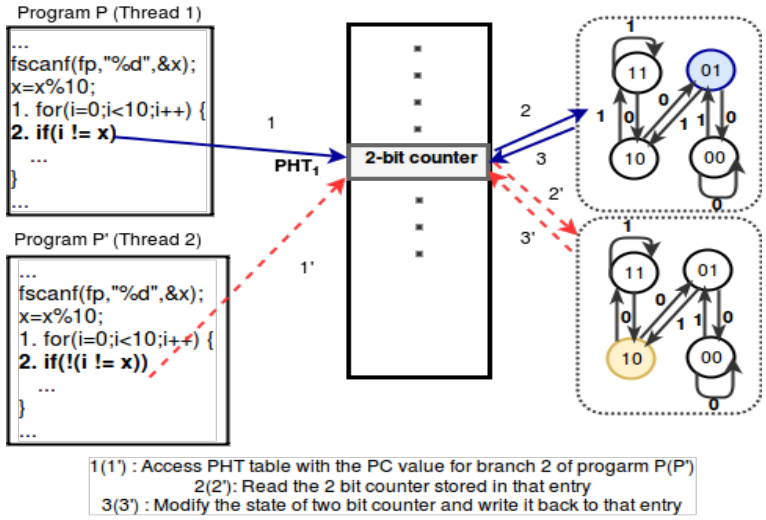


Figure 7.6: Negative interference caused by the variant on a benign application

7.4 Experimental setup and Evaluation

To demonstrate the effect on branch prediction accuracy for a benign application with its variant, we run our experiment on the GEM5 architectural simulator [27]. In Gem5, multiple user specific programs can run simultaneously in simultaneous multi-threading (SMT) mode where each program is provided a unique thread id with a round-robin scheduler as the default scheduler. After execution, it generates statistics that records data like branch prediction accuracy, energy expenditure, time taken etc. We modify the source code of this simulator to support SMT mode for two threads where the same predictor table is shared by them for prediction. We now report our experience in using our methods on the Siemens software programs. Table 7.1 presents the benchmark details. The third column of this table presents the number of branches in each program that are flipped for creating the variant program.

Programs	Lines of code	Number of branches
replace	565	94
schedule	415	30
schedule2	311	41
totinfo	407	45
printtokens	727	40
printtokens2	564	78

Table 7.1: Benchmark Detail

Figure 7.7 presents the mis-prediction rate for each benchmark program when it is run alone and run with our created variant program for attack. It can be seen that in each case, the presence of the variant program increases the mis-prediction rate. It may be noted that the accuracy degradation shown in Figure 7.7 is around 2%. This is quite a significant percentage in modern processors with deep pipelines.

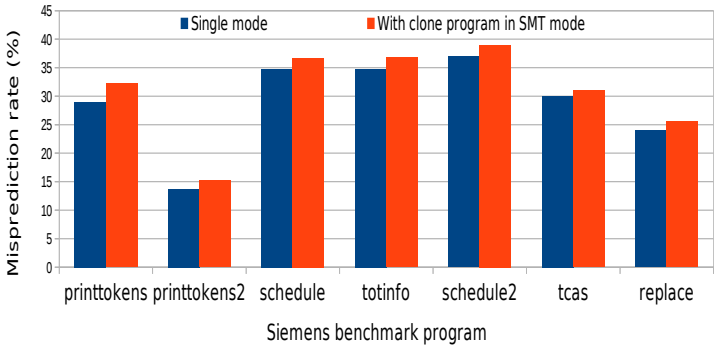


Figure 7.7: Mis-prediction Rate of Siemens benchmark programs

We perform another experiment adopting a non-shared predictor table configuration, as is the case with many desktop processors today. However, as mentioned earlier, storage is a concern in the embedded / mobile world, and hence, an alternative solution proposed in literature to curb interference is to split the predictor table into individual compartments and allocate each smaller compartment to each running thread. Thus, the chance of an attack with the variant program goes away. However, due to the fact that each thread now

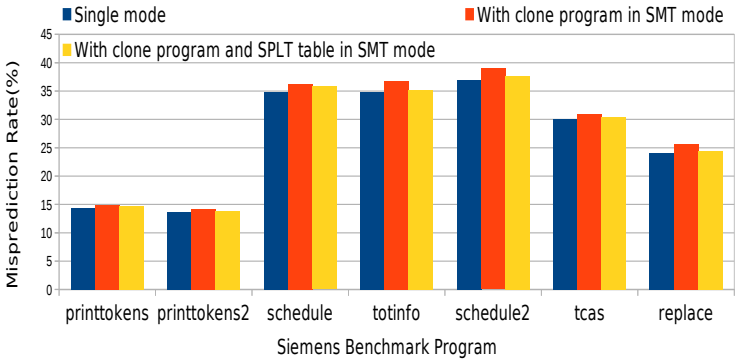


Figure 7.8: Mis-prediction Rate of Siemens benchmark programs with split prediction table

receives a much lesser space to store its working set of branches, a significant intra-thread interference is generated. Figure 7.8 presents the mis-prediction rate recorded for such a split predictor table configuration with the bimodal predictor. We present a comparative plot of the mis-prediction rates of the benchmark program when run a) alone, b) with the variant program in SMT mode with a shared table and c) with the variant in a SPLIT table configuration. It may be observed that even the SPLIT table cannot improve the mis-prediction rate beyond a certain level.

7.5 Summary

In this work, we observe how predictor table interference affects the performance of a branch predictor in SMT mode. We propose an attack method that jeopardises the prediction accuracy by polluting the predictor tables stored for a benign application using a malicious variant. Examining the results generated using our proposed attack method as well as the extent of performance compromised, we can conclude that our approaches can effectively slow down the performance promise expected from a SMT processor.

Chapter 8

An empirical study on predictor storage and fault sensitivity

8.1 Introduction

As discussed in the previous chapters, branch predictors typically function based on the history information stored in the corresponding prediction tables. Evidently, accuracy of branch predictors is sensitive to the storage they are allowed to use, more history information usually has been seen to generate more accuracy of prediction. For a program with even a moderate number of branches, it is not possible to store the history information of all branch outcomes in the predictor table individually, due to constraints on cost and resource size. As a result, each entry of the table has to be used for more than one branch to store history information. This phenomenon, formalized as interference in our earlier discussions, may manifest in either a constructive (positive) or a destructive (negative) way. When the information stored for one branch is accessed and modified for some other branch that shares the same predictor entry for prediction, it is termed as negative interference. However, in

the case of positive interference, the information stored by one branch has often been seen to help the other branches for correct prediction. Prediction accuracy degrades largely due to the negative interference since it has been observed that negative interferences occur much more frequently than their positive counterparts [121]. Evidently, the amount of negative interference is expected to increase if the available predictor table storage is less, and more number of branch instructions access the same address for prediction, leading to performance and accuracy degradation.

The issue of storage for predictor tables is even more important for a resource constrained embedded environment with low storage budgets, since there is a more acute trade-off that needs to be worked out. On one hand, increased storage budget for the predictor tables, can lead to better accuracy of prediction. However, the cost and other overheads associated with predictor tables increase. On the other hand, having a lower sized table for predictors reduces cost, however, increases mis-prediction and manifests in terms of wasted instructions, latency and energy. Selecting an appropriate branch predictor structure for an embedded environment is therefore, quite a crucial task. A branch prediction unit charges a significant amount of power consumption in modern processor designs, and consumes a significant amount of storage and becomes a major issue for relatively small embedded processors. In this chapter, we present an empirical study on the prediction accuracy and latency of different branch predictors for different storage budgets. The motivation behind this empirical exploration is to highlight the storage sensitivity of branch predictors that are available in contemporary literature. The objective of our study is to examine the common predictor designs available in literature, and characterize their accuracy versus storage performance. As our results show, many of the predictors which are known to have high accuracy in general, lose out on performance when exercised in low storage scenarios.

In addition to the storage sensitivity analysis of different branch predictors, in this chapter, we also study through experiments, the resilience of these predictors against faults in the

predictor table and history registers. A fault in the contents of a register or a table entry, which a predictor uses for keeping information can alter the number of correct predictions, and thereby, increase the mis-prediction rate, and affect the overall performance of execution of a given user program.

Contributions of this work

- We present an empirical evaluation of different branch predictors at various storage points and the resulting effect on processor performance in terms of prediction accuracy and latency. We present results of our experiments using the branch predictors and the traces of the Championship Branch Predictor-2 benchmarks [33].
- We perform a detailed empirical study on the effect of a fault in the branch predictor on a processor's performance.
  - We consider a number of contemporary branch predictors and identify the amount of information needed at runtime (which the processor needs to store) for these predictors to work correctly. We change these registers in a controlled fashion, and show the resulting change in prediction and performance.
  - We observe the performance variations for different branch predictors using an architectural simulator. Specifically, we alter the register contents inside the simulator and observe the outcomes. The simulation outputs are measured in terms of performance in respect of prediction accuracy, number of processor cycles needed and power consumption for that processor.

The rest of the chapter is organized as follows: Section 8.2 reports the storage sensitivity analysis of predictors. Section 8.3 discusses different types of faults on different branch predictors. Section 8.4 describes the experimental set-up used as well as the results of our experiments while Section 8.5 concludes the chapter.

8.2 Storage requirement analysis of predictors

In this section, we highlight the elements of storage that are intrinsically used by these prediction units, with a discussion on why the predictor design may lose on accuracy if subjected to lower resources.

Gshare Predictor

The GShare predictor [78] uses two main data structures - a Pattern History Table (PHT) and Branch History Register (BHR) as discussed in Chapter 2. For this predictor, storage size actually depends on the size of the PHT. A small PHT is able to store less number of prediction related information and uses less number of PC and BHR bits. On one hand, lesser PC bits and small sized PHT increases the chances of negative interference. On the other hand, since the BHR gives the flavor of branch correlation, lesser number of BHR bits loses the correlation to some extent. Hence, a smaller storage size increases the chances of negative interference and increases the number of mis-predictions. GShare is thus quite sensitive to storage size, as evident from our experiments as well.

Local history based Two-level predictor

In dynamic branch prediction, two types of histories are mainly used - global branch history and local branch history, as discussed in Chapter 2. A local history based predictor is popularly used in many processors since it can detect the control structures like the loop structure more effectively than the global one. However, these predictors are more resource intensive. In this discussion, we highlight the main features of a popular local history based branch predictor, namely PAp [119]. It is a combination of multiple per-address BHRs with multiple per-address PHT as shown in Chapter 2. In PAp, each branch has its own BHR as

well as its own PHT [119]. The BHR content is used to select the index in a PHT whereas a PHT is selected by the branch instruction address. For this predictor, storage size depends on BHR size and per address PHT size. A small BHR size contains less number of entries that implies lesser number of per address PHT. This increases the chances of inter-branch interference since more number of branches now access the same PHT table. Similarly, smaller sized per address PHT can increase the chances of intra-branch interference since the same PHT entry has to be used for more number of different BHR patterns. It is quite expected that the number of mis-predictions due to negative interferences will increase with smaller storage size.

Perceptron Predictor

The perceptron predictor discussed in Section 2, provides better prediction accuracy compared to other popular predictors even at lower resource budget. In a perceptron predictor, the best performance can be achieved by tuning the history length, the number of bits used to represent the weights, and the threshold. A smaller predictor size affects the number of bits of these three parameters, thereby degrading the prediction accuracy in turn.

LTAGE Predictor

LTAGE [97] is widely recognized as the most popular state-of-the-art branch predictor that exploits several different history lengths to capture correlation between very remote branch outcomes as well as very recent branch history. This predictor combines a TAGE [99] predictor and a loop predictor as presented in Chapter 2. For this predictor, storage size depends on the size of storage for the loop predictor as well as the TAGE predictor. The size of the TAGE predictor is calculated using the size of the base predictor, the number of components and the size of each tagged component. A lesser number of tag components

loses the correlation from very remote branch outcomes. A smaller sized tagged component and the base predictor also increases the chances of negative interferences. Similarly, a smaller sized loop predictor may fail to identify the loop behavior of all branches correctly.

ISL-TAGE Predictor

ISL-TAGE [98] is also considered as a state-of-the-art branch predictor and is extensively used in modern pipelined processors. This predictor combines a TAGE predictor, a loop predictor, a Statistical Corrector predictor and an Immediate Update Mimicker (IUM), discussed in Chapter 2. In this case, storage size depends on all the components associated. Similar to the LTAGE predictor discussed above, smaller sized components as well as lesser number of components causes increased negative interference and increases misprediction.

8.3 Fault sensitivity analysis on tables and history registers

For branch prediction, almost every branch predictor uses the two-bit saturating counter, the BHR, the PC and the PHT, as discussed in Chapter 2. This gives us the motivation to explore how a branch prediction policy gets affected if these structures have faults. We consider permanent faults for ease of analysis. In the following, we discuss the source of the fault and how it affects the prediction structures.

- **Source of a fault: two-bit saturating counter**

  The two bit counter is very crucial for branch prediction since it provides the final prediction for branch direction. Some of the possible ways in which this counter can be affected are as described below.

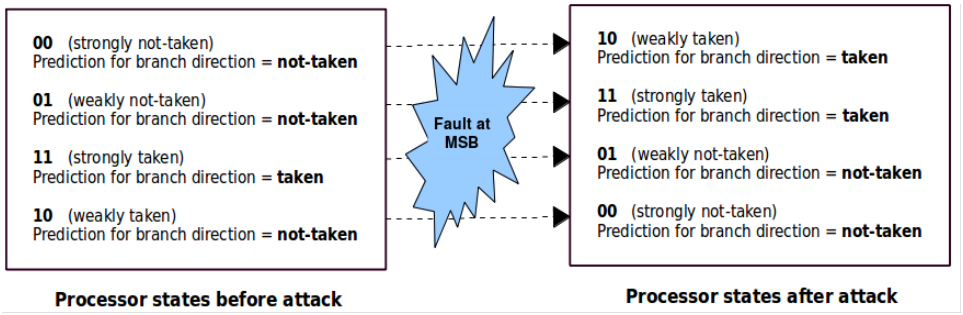


Figure 8.1: Impacts on branch prediction (MSB reversed)

– Most Significant Bit (MSB) of all states flipped :

  The MSB provides the direction for a branch prediction. The bit flip actually inverts the predictions for all branches and as a result, earlier taken predictions for branches become not-taken and vice versa (Figure 8.1). As a result of this, the GShare predictor begins to mis-predict the branches for which it did the correct predictions before. This significantly degrades the performance, as we show through results.

– Least Significant Bit (LSB) of all states flipped :

  The LSB, when paired with the direction bit, provides a hysteresis which prevents the direction bit from immediately changing when a mis-prediction occurs. If we flip this bit, it changes the current state of this two-bit counter and a strongly taken:00 (strongly not-taken:01) state becomes a weakly taken:01 (weakly not-taken:10) state and vice versa. As a result, states which used to take two consecutive mis-predictions to alter the branch direction, now take only one mis-prediction. This again has an impact on prediction accuracy.

– Counter length changed :

  If the two-bit counter is changed to a 1-bit counter due to the fault, the number of states gets reduced and it becomes two (either taken or not-taken). According to the principle of a saturating counter, now only one mis-prediction always changes

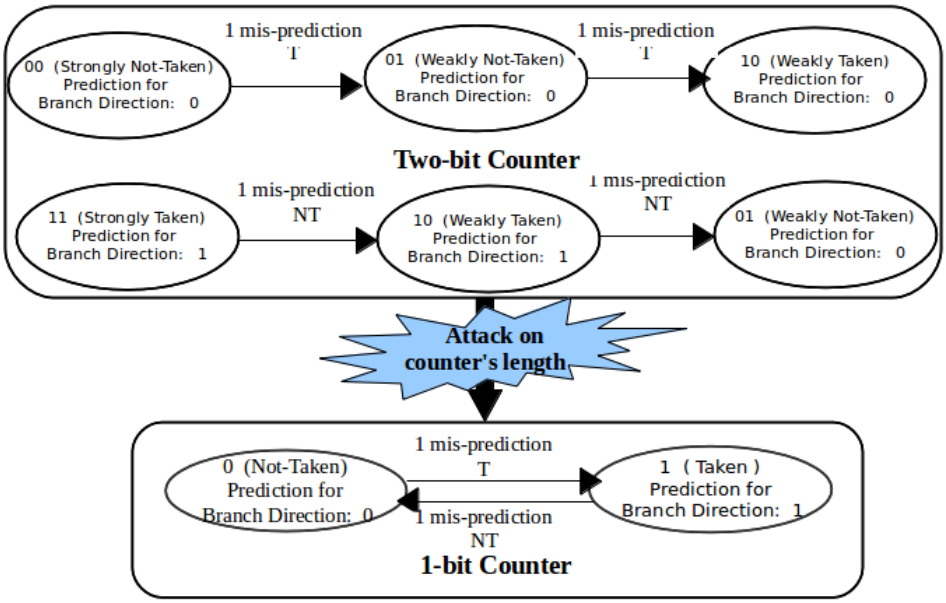


Figure 8.2: Impacts on branch prediction (Two-bit counter becomes 1-bit counter)

the prediction for a branch direction whereas, the two-bit counter usually uses two consecutive wrong predictions to alter the direction for the state:00 (strongly not-taken) and state:11 (strongly taken), refer Figure 8.2. This naturally increases the number of incorrect predictions. Experimental results show the performance degradation due to this fault.

- **Source of fault: Branch History Register (BHR)**

  As noted earlier, BHR is a n-bit shift register which stores the history of the last n branches (global history) or the last n outcomes of the same branch instruction (local history). GShare and TAGE predictors use the global history whereas the PAp predictor uses the local history. The BHR is used to address an entry in a PHT which keeps a two-bit counter to give the final prediction. If this BHR entry has a fault, the branch predictor will index a wrong PHT entry to get the final prediction. This BHR can be affected by its content or its length as discussed below:

- MSB or LSB of BHR is flipped:

  This flip changes the original value of the BHR and a different PHT entry is accessed by the predictor to get the final prediction for branch direction. For example, a 4-bit BHR with a value 1001 becomes 0001, if MSB is flipped. Now in a PAp predictor, instead of an entry at location 1001, the location 0001 will be accessed to get the prediction for a branch, which can vary prediction accuracy.

- Length of BHR is changed:

  If BHR length is changed, the indexing used to address an entry in PHT can differ and as a result a wrong PHT entry will be picked up for the final prediction. For example, consider a 8-bit BHR with a value 11101001. In a PAp predictor, this 8-bit is used to index an entry in a 2^8 entry PHT. If we select only the lower 4 bits of this BHR i.e. 1001 to index a PHT entry, then instead of 11101001, the 00001001 entry will be accessed for prediction. So this can alter the prediction accuracy as well.

- **Source of fault: Program Counter (PC)**

  The program counter contains the branch address and is used by different prediction mechanisms. In PAp, each PC corresponds to an individual BHR as well as individual PHT. If the number of bits of a PC which are taken to select this Branch History Table (BHT) and PHT is changed, a different BHT entry (or BHR) as well as a different PHT entry will be accessed to get the final prediction. However, in case of the GShare predictor, this PC value is used as one input of an indexing function to find an entry in the PHT. So this length change of PC is expected to result in a different output for the same indexing function and as a result, a wrong PHT entry can be chosen to give the prediction. Similarly, for the case of the TAGE predictor, a wrong predictor component or incorrect entry of the base predictor may be accessed for branch prediction.

Along with these, the TAGE predictor also has tagged components whose signed counter (sign bit) is used to make the prediction. Any change on this counter (e.g. flipping of sign bit) is expected to have an impact on the prediction accuracy.

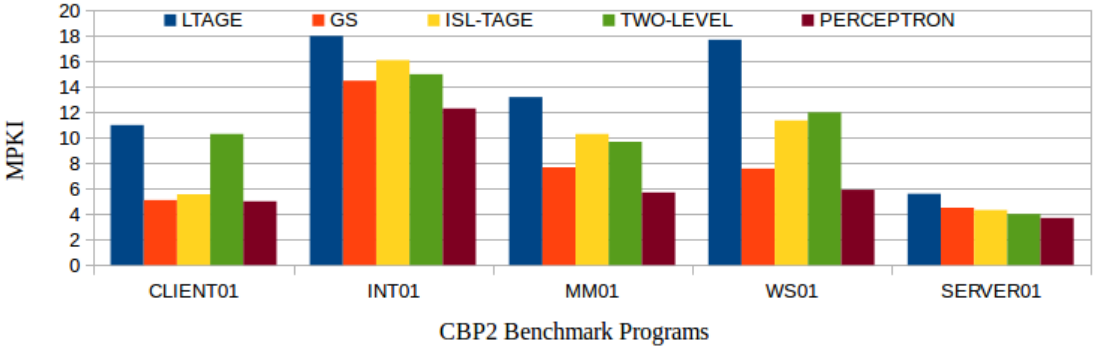
8.4 Implementation and Results

8.4.1 Storage sensitivity analysis of branch predictors

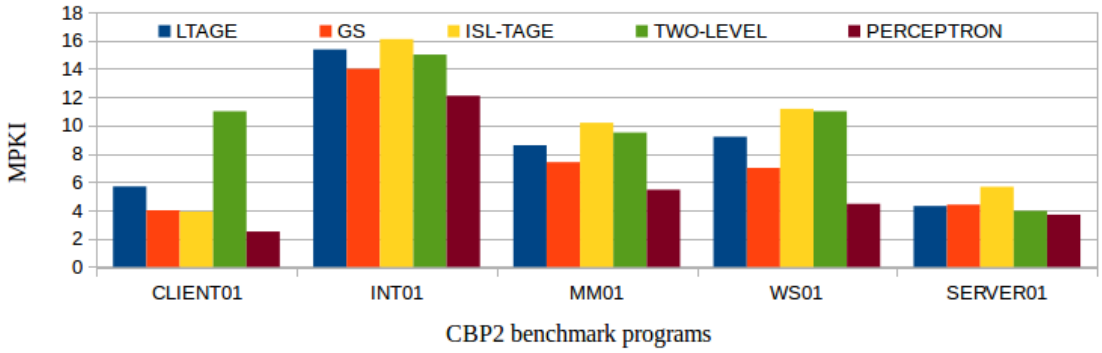
To support our study discussed above, we carry out experiments with different predictor designs available in literature, and record their accuracy and latency performance at different storage points. The objective of this study is to show that indeed many of these predictors are quite sensitive to storage, and often fail to perform as per expectations when subjected to low resources. Our experiments indeed support our intuition, as detailed below.

In this work, we use the five different predictor implementations, namely, GShare, a local history based Two-Level predictor, ISL-TAGE, LTAGE and Perceptron from the Championship Branch Prediction-2 (CBP-2) [32] benchmarks. In CBP-2, all predictors are designed for a fixed storage budget. For our experiments, we modify the predictor codes to work with 6 different storage budgets, namely, 2 KB, 4 KB, 8 KB, 16 KB, 32 KB and 64 KB. The storage value mentioned here is the total storage allocated to a predictor for managing all data structures that it needs internally to store and manipulate history information. We perform our experiments on the CBP-2 traces and record the Mis-prediction Per Kilo Instructions (MPKI) and latency.

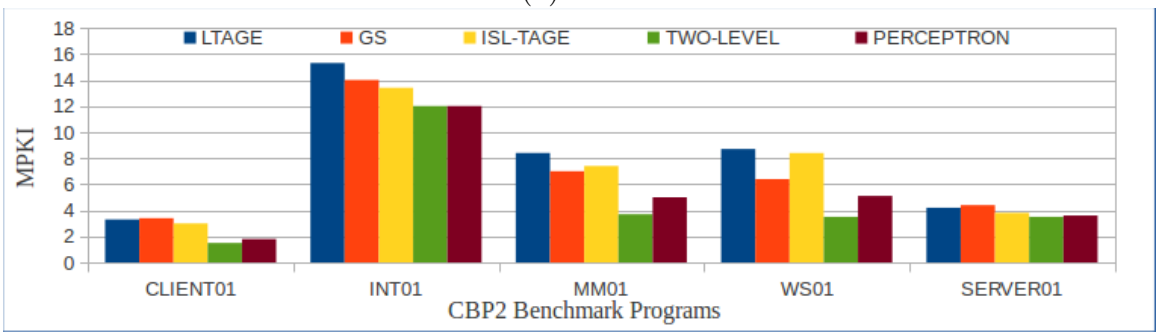
Figure 8.3 and 8.4 present the MPKIs for the five predictors at the different storage points. Results show that the mis-prediction rates for different predictors vary with different storage budget. It is interesting to note that a predictor that has the lowest MPKI at a particular storage point can have the highest MPKI for some other storage point. For example,



(a) 2 KB



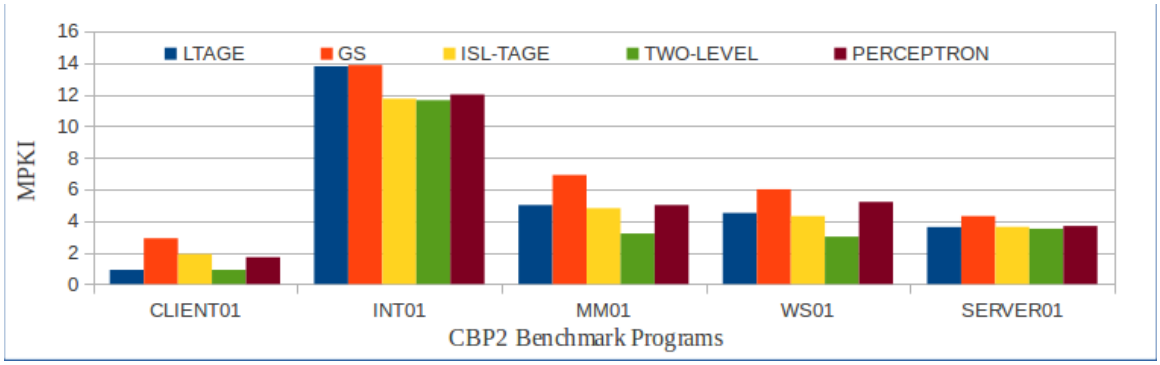
(b) 4 KB



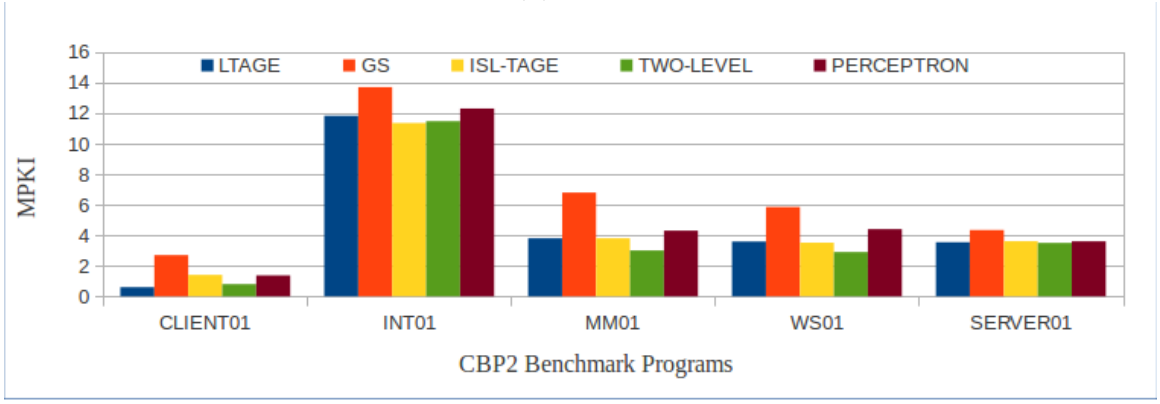
(c) 8 KB

Figure 8.3: MPKI for CBP2 Benchmark Programs

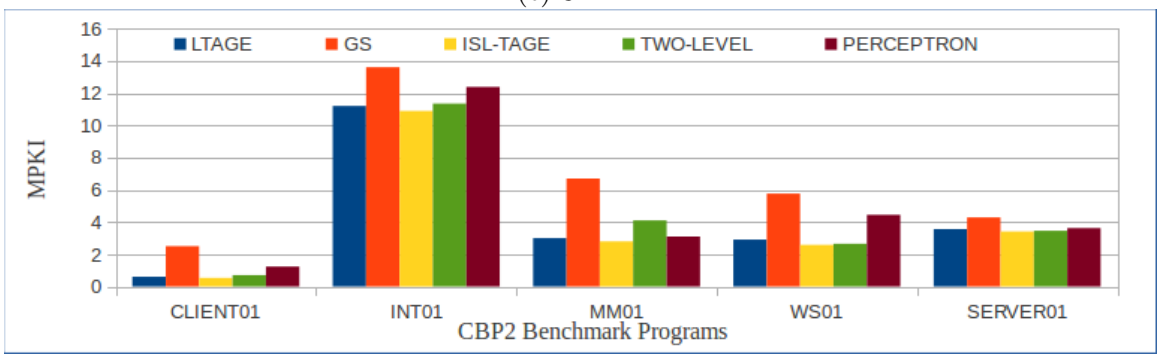
LTAGE has the highest MPKI for all the cbp traces with 2 KB storage size as shown in Figures 8.3 and 8.4, whereas, performance changes for other storage sizes presented here. Indeed, LTAGE has the lowest MPKI for almost all the cbp traces with 64 KB storage size. Additionally, it can be seen that the two state of art predictors, namely, LTAGE



(d) 16 KB



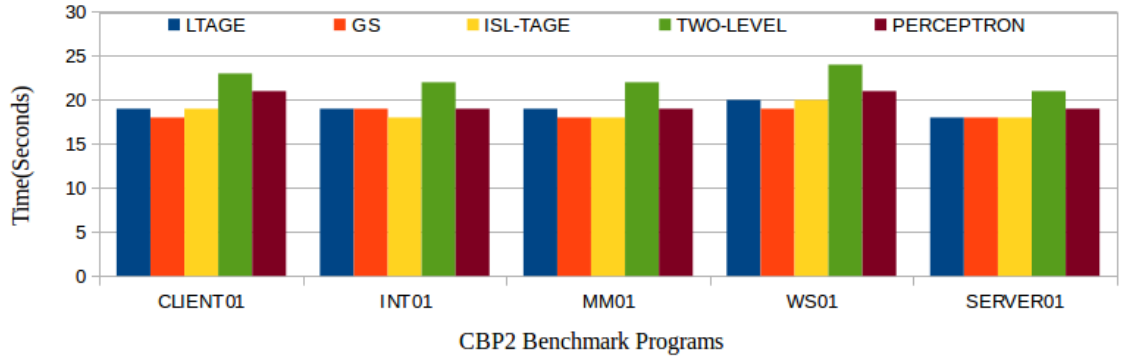
(e) 32 KB



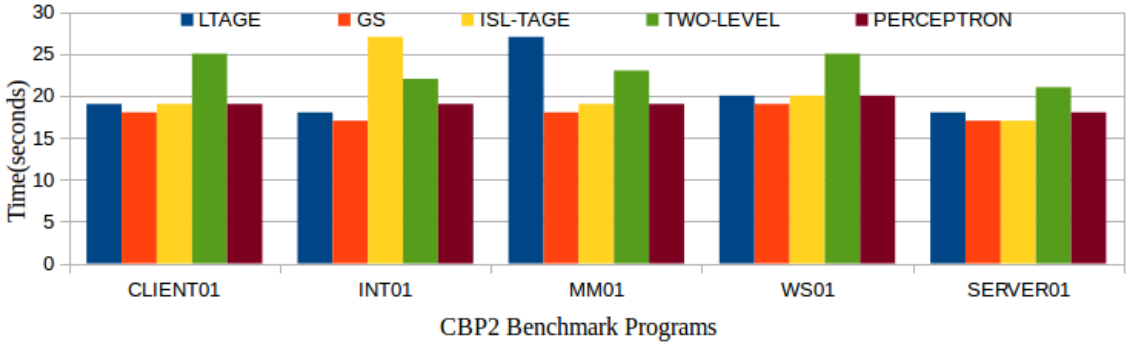
(f) 64 KB

Figure 8.4: MPKI for CBP2 Benchmark Programs

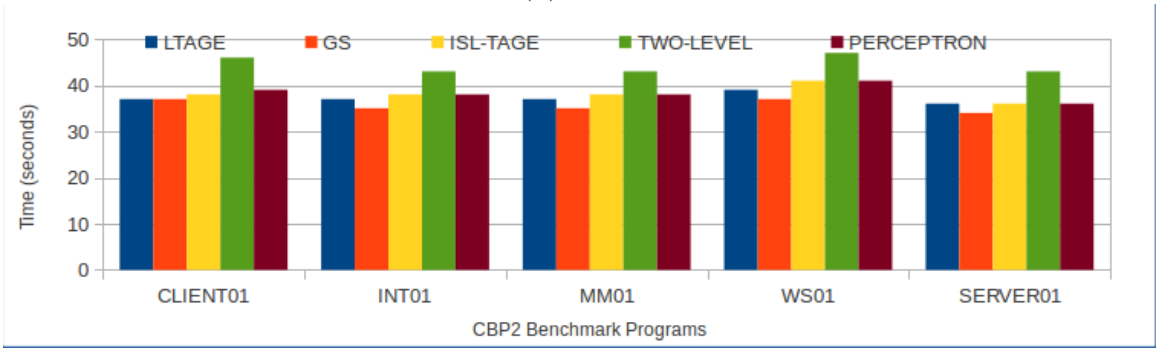
and ISL-TAGE, perform well for storage sizes more than 16 KB. For a low storage budget, performances of the Two level predictor and Perceptron are better, as compared to LTAGE and ISL-TAGE. Thus we see that predictors included in modern processors may not always



(a) 2 KB



(b) 4 KB

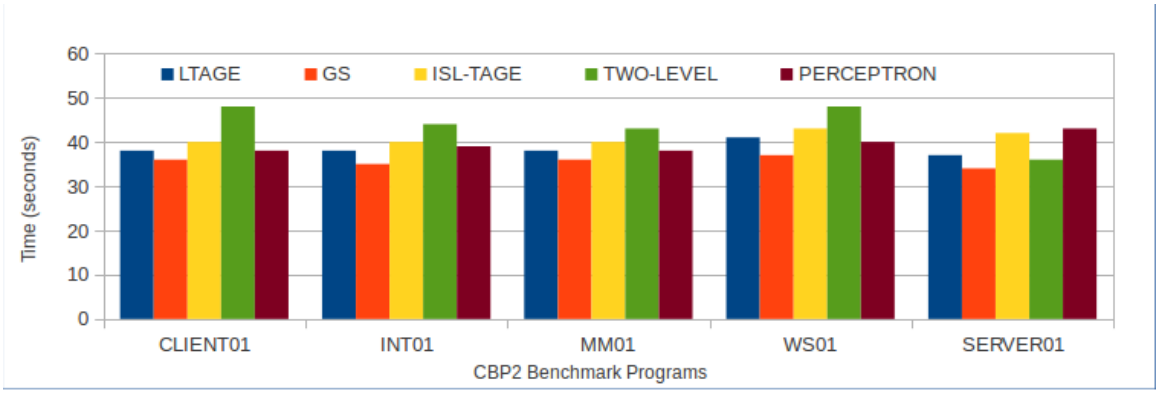


(c) 8 KB

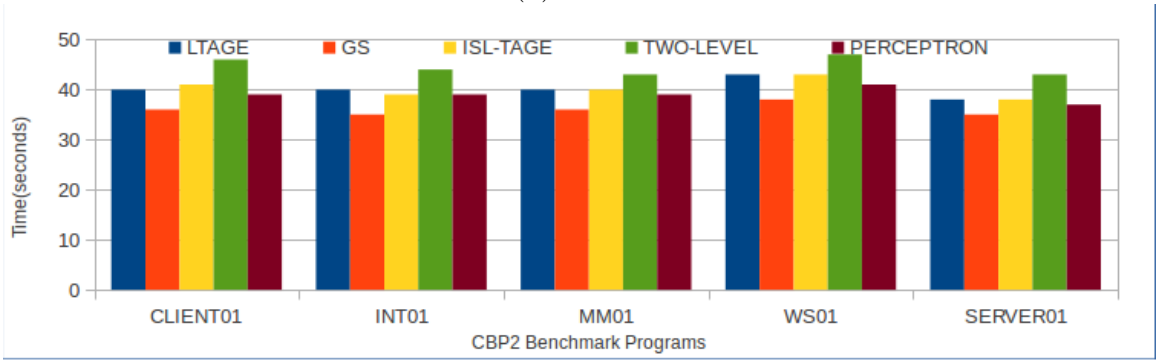
Figure 8.5: Latency plot for CBP2 Benchmark Programs

be the best choice for embedded processors with less storage.

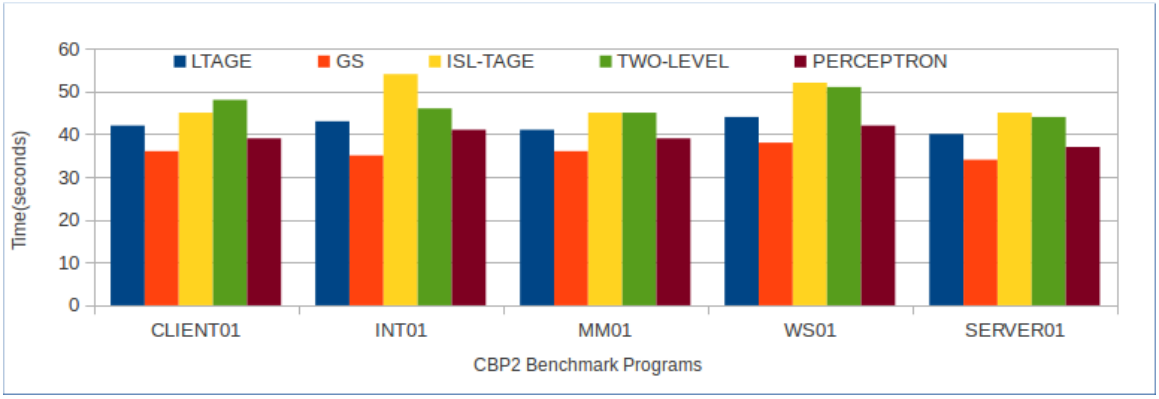
Figures 8.5, 8.6 present the latency plot for different predictors at different storage points. It is interesting to observe that the latency does not vary largely across the points. The Two-



(d) 16 KB



(e) 32 KB



(f) 64 KB

Figure 8.6: Latency plot for CBP2 Benchmark Programs

level predictor has the highest latency compared to the other predictors for all the predictor sizes considered here. From Figures 8.3, 8.4, 8.5 and 8.6, it can be observed that though the Two-level predictor gives better prediction accuracy than LTAGE and ISL-TAGE in a

resource constrained environment, it does not fare well in terms of latency. In this case, the perceptron predictor has the lowest MPKI as well as latency among other predictors.

8.4.2 Fault sensitivity analysis

We now report our findings on fault effect analysis on the GShare, PAp and TAGE predictors on the SPEC 2006 benchmark programs [52]. Performances are shown in terms of prediction accuracy, number of processor cycles and energy consumption. In all the tables, the column labeled as energy shows the dynamic energy expenditure in nanojoules measured by Tejas [11]. We present below the results of our experiments for the different predictors using the different fault sources discussed on different registers. We inject faults in these structures to study the resulting effect.

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original MSB	Reversed MSB	Original MSB	Reversed MSB	Original MSB	Reversed MSB
403.gcc	66.27	33.6931	86754099	96932134	2454452.3728	2635708.1682
400.perlbench	63.75	36.2364	11766325	12867838	339602.8584	359208.8278
458.sjeng	99.92	0.0706	283822066	497448050	5526430.3942	9328972.9094
470.lbm	79.86	20.1389	62229116	144180	2800.2948	2898.889
456.hmmer	74.75	25.1446	56861420	70659713	1567938.3322	1716351.0124
429.mcf	64.98	35.0133	138641	69470610	2146057.1238	2370500.5134
482.sphinx23	62.51	37.4833	633511	684469	16608.3024	17515.1624
462.libquantum	79.64	20.3525	144808	151364	2916.0318	3032.7286
444.namd	78.7	21.2907	157919	164554	3160.5668	3277.9002
464.h264ref	77.31	22.6808	271338	278370	5300.637	5425.8066

Table 8.1: Performance Effects for GShare Predictor (MSB Reversed)

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original MSB	Reversed MSB	Original MSB	Reversed MSB	Original MSB	Reversed MSB
403.gcc	75.346	24.6208	83385248	99911660	2394376.5798	2688633.8706
400.perlbench	69.8519	30.1486	11389779	13426984	332899.9548	369160.4722
458.sjeng	99.947	0.053	283502111	497505905	5520735.1952	9330002.7284
470.lbm	76.5046	23.4954	138832	142940	2803.6946	2876.817
456.hmmer	92.1533	7.8592	60383424	73268920	1534589.5846	1764230.9706
429.mcf	86.4782	13.6234	47692804	78868239	1982855.5666	2537777.9248
482.sphinx23	81.372	18.6287	599968	722759	16011.0446	18196.532
462.libquantum	76.9756	23.0244	146579	149118	2947.5556	2992.7498
444.namd	75.8411	24.4505	159852	165329	3194.2046	3293.042
464.h264ref	74.7855	25.3573	270799	279657	5291.0428	5449.1

Table 8.2: Performance Effects for PAp Predictor (MSB of Counter Reversed)

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original MSB	Reversed MSB	Original MSB	Reversed MSB	Original MSB	Reversed MSB
403.gcc	85.93	88.0023	80209030	79442840	2337946.489	2324347.3642
400.perlbench	83.23	86.4732	10589517	10720538	318653.5596	320985.7334
458.sjeng	99.93	99.9307	283196974	283122778	5515303.7566	5513983.0678
470.lbm	77.31	84.838	140637	138519	2835.8236	2798.1232
456.hmmer	93.08	93.1131	59931808	59113058	1525384.6834	1511650.3746
429.mcf	82.95	91.5021	49226385	45601483	2010152.9236	1945629.668
482.sphinx23	84.25	79.7979	591041	603252	15851.9516	16069.3074
462.libquantum	80.5	85.0483	144364	142846	2908.1286	2881.1082
444.namd	79.42	84.011	156810	157717	3141.4038	3157.5484
464.h264ref	78.87	83.0543	273337	268542	5336.604	5251.253

Table 8.3: Performance Effects for TAGE Predictor (MSB of Counter Reversed)

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original LSB	Reversed LSB	Original LSB	Reversed LSB	Original LSB	Reversed LSB
403.gcc	66.27	64.0966	86754099	87023273	2454452.3728	2459165.9404
400.perlbench	63.75	63.0858	11766325	11796034	339602.8584	340131.2938
458.sjeng	99.92	0.2276	283822066	497415305	5526430.3942	9328390.0484
470.lbm	79.86	53.2407	62229116	140879	2800.2948	2840.1312
456.hmmer	74.75	46	56861420	67806143	1567938.3322	1665543.0364
429.mcf	64.98	70.6935	138641	54436258	2146057.1238	2102889.0478
482.sphinx23	62.51	52.1742	633511	654797	16608.3024	16987.0008
462.libquantum	79.64	53.0984	144808	146188	2916.0318	2940.5958
444.namd	78.7	52.9509	157919	157431	3160.5668	3151.1108
464.h264ref	77.31	53.4941	271338	274159	5300.637	5350.8508

Table 8.4: Performance Effects for GShare Predictor (LSB Reversed)

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original LSB	Reversed LSB	Original LSB	Reversed LSB	Original LSB	Reversed LSB
403.gcc	75.346	64.0929	83385248	87046670	2394376.5798	2459550.2762
400.perlbench	69.8519	63.1424	11389779	11781463	332899.9548	339871.3528
458.sjeng	99.947	0.2631	283502111	497379990	5520735.1952	9327761.4414
470.lbm	76.5046	56.4236	138832	140684	2803.6946	2836.6602
456.hmmer	92.1533	46.0458	60383424	67441662	1534589.5846	1658839.2094
429.mcf	86.4782	71.0449	47692804	54179141	1982855.5666	2098312.3652
482.sphinx23	81.372	53.8234	599968	654465	16011.0446	16981.0912
462.libquantum	76.9756	56.1114	146579	147273	2947.5556	2959.9088
444.namd	75.8411	56.6464	159852	161126	3194.2046	3216.8818
464.h264ref	74.7855	57.5398	270799	273818	5291.0428	5344.781

Table 8.5: Performance Effects for PAp Predictor (LSB of Counter Reversed)

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original LSB	Reversed LSB	Original LSB	Reversed LSB	Original LSB	Reversed LSB
403.gcc	85.93	86.3175	80209030	79715714	2337946.489	2329132.7562
400.perlbench	83.23	84.1727	10589517	10636708	318653.5596	319495.6758
458.sjeng	99.93	99.9299	283196974	283338969	5515303.7566	5517831.2676
470.lbm	77.31	79.6875	140637	139479	2835.8236	2815.2112
456.hmmer	93.08	93.0844	59931808	60526201	1525384.6834	1538361.4132
429.mcf	82.95	84.0822	49226385	48799739	2010152.9236	2002559.202
482.sphinx23	84.25	75.9527	591041	609505	15851.9516	16180.9956
462.libquantum	80.5	80.1592	144364	144169	2908.1286	2904.6576
444.namd	79.42	78.7012	156810	156423	3141.4038	3133.938
464.h264ref	78.87	78.586	273337	272572	5336.604	5322.6022

Table 8.6: Performance Effects for TAGE Predictor (LSB of Counter Reversed)

Benchmark	Prediction Accuracy(%)				Clock Cycles			
	1-bit	2-bit	3-bit	4-bit	1-bit	2-bit	3-bit	4-bit
403.gcc	61.96	66.27	67.92	68.62	87517041	86754099	86322007	85356433
400.perlbench	58.8	63.75	66.27	66.4	11660939	11766325	11357031	11593328
458.sjeng	99.89	99.92	99.92	99.92	283320350	283822066	283552210	283606701
470.lbm	78.58	79.86	79.22	76.04	137526	138641	138090	137120
456.hmmer	59.09	74.75	75.64	76.19	65918003	62229116	63056889	63761834
429.mcf	56.39	64.98	69.75	70.73	60473999	56861420	54817421	54247867
482.sphinx23	48.37	62.51	64.91	65.41	663205	633511	629030	629257
462.libquantum	78.28	79.64	78.68	76.23	144729	144808	143712	143936
444.namd	76.72	78.7	77.22	75.06	156908	157919	158193	157375
464.h264ref	75.72	77.31	76.46	74.04	273733	271338	273181	270123

Table 8.7: Performance Effects for GShare Predictor (Counter Length Changed)

Benchmark	Energy			
	1-bit	2-bit	3-bit	4-bit
403.gcc	2454452.3728	2446727.0804	2429563.9132	
400.perlbench	337726.9876	339602.8584	332318.0024	336524.089
458.sjeng	5517499.8494	5526430.3942	5521626.9574	5522596.8972
470.lbm	2780.4478	2800.2948	2790.487	2773.221
456.hmmer	1634169.4476	1567938.3322	1581324.16	1595889.1102
429.mcf	2210361.03	2146057.1238	2109673.9416	2099535.8804
482.sphinx23	17136.8556	16608.3024	16528.5406	16532.5812
462.libquantum	2914.6256	2916.0318	2896.523	2900.5102
444.namd	3142.571	3160.5668	3165.444	3150.8836
464.h264ref	5343.268	5300.637	5333.4424	5279.01

Table 8.8: Performance Effects for GShare Predictor (Counter Length Changed)

Benchmark	Prediction Accuracy(%)				Clock Cycles			
	1-bit	2-bit	3-bit	4-bit	1-bit	2-bit	3-bit	4-bit
403.gcc	69.8066	75.346	76.9188	76.85	85849140	83385248	82786622	83092197
400.perlbench	65.244	69.8519	72.3361	72.6049	11326321	11389779	11121986	11252867
458.sjeng	99.9348	99.947	99.9416	99.9338	283460925	283502111	283202229	283268135
470.lbm	77.2569	76.5046	74.4792	71.5856	139735	138832	139343	141121
456.hmmer	85.9031	92.1533	92.3705	92.3344	61361536	60383424	60180651	60039036
429.mcf	77.6357	86.4782	87.449	87.5607	51472783	47692804	47355941	47230854
482.sphinx23	77.5252	81.372	81.8159	80.3623	604026	599968	599247	598155
462.libquantum	77.0324	76.9756	74.4741	71.9159	145244	146579	146189	144458
444.namd	75.7308	75.8411	73.7452	71.3734	158178	159852	155382	155052
464.h264ref	74.5403	74.7855	73.1508	70.5762	270863	270799	269808	274275

Table 8.9: Performance Effects for PAp Predictor(Counter Length Changed)

Benchmark	Energy			
	1-bit	2-bit	3-bit	4-bit
403.gcc	2438333.713	2394376.5798	2383780.681	2389245.1204
400.perlbench	331770.4024	332899.9548	328132.6622	330462.344
458.sjeng	5520002.0844	5520735.1952	5515397.2956	5516570.4224
470.lbm	2819.768	2803.6946	2812.7904	2844.4388
456.hmmer	1552125.6154	1534589.5846	1529994.3676	1528588.9634
429.mcf	2050139.1928	1982855.5666	1976859.4052	1974633.049
482.sphinx23	16083.277	16011.0446	15998.2108	15978.9656
462.libquantum	2923.7926	2947.5556	2940.6136	2909.8018
444.namd	3164.4074	3194.2046	3114.6386	3108.7646
464.h264ref	5292.182	5291.0428	5273.403	5352.9156

Table 8.10: Performance Effects for PAp Predictor(Counter Length Changed)

Benchmark	Prediction Accuracy(%)				Clock Cycles			
	1-bit	2-bit	3-bit	4-bit	1-bit	2-bit	3-bit	4-bit
403.gcc	79.83	85.93	87.44	87.6	82189182	80209030	79472066	79754141
400.perlbench	74.9	83.23	85.97	86.56	11175555	10589517	10754710	10422707
458.sjeng	99.91	99.93	99.93	99.93	283681764	283196974	283581362	283144095
470.lbm	73.84	77.31	78.93	75.63	140633	140637	139459	140076
456.hmmer	92.38	93.08	93.1	93.14	60054884	59931808	59553481	59503474
429.mcf	63.52	82.95	84.25	88.47	57288500	49226385	46952993	48633091
482.sphinx23	55.97	84.25	83.51	83.43	648968	591041	593805	595462
462.libquantum	74.18	80.5	75.09	76.52	143713	144364	146949	145676
444.namd	73.52	79.42	74.84	75.34	157164	156810	157275	155027
464.h264ref	73.39	78.87	76.95	74.98	268874	273337	272225	269070

Table 8.11: Performance Effects for TAGE Predictor(Counter Length Changed)

Benchmark	Energy			
	1-bit	2-bit	3-bit	4-bit
403.gcc	2373177.2254	2337946.489	2324840.8434	2329799.6332
400.perlbench	329085.036	318653.5596	321593.995	315684.3416
458.sjeng	5523933.0186	5515303.7566	5522145.863	5514362.5104
470.lbm	2835.7524	2835.8236	2814.8552	2825.8378
456.hmmer	1529230.2686	1525384.6834	1519055.4648	1518348.8898
429.mcf	2153658.5706	2010152.9236	1969686.546	1999592.2904
482.sphinx23	16883.0522	15851.9516	15901.1508	15930.6454
462.libquantum	2896.5408	2908.1286	2954.1416	2931.4822
444.namd	3147.705	3141.4038	3149.6808	3109.6664
464.h264ref	5257.1626	5336.604	5316.8104	5260.6514

Table 8.12: Performance Effects for TAGE Predictor(Counter Length Changed)

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original LSB	Reversed LSB	Original LSB	Reversed LSB	Original LSB	Reversed LSB
403.gcc	75.346	75.3677	83385248	83209708	2394376.5798	2391358.1726
400.perlbench	69.8519	69.853	11389779	11589839	332899.9548	336459.2912
458.sjeng	99.947	99.9478	283502111	283267955	5520735.1952	5516567.2184
470.lbm	76.5046	77.7199	138832	140458	2803.6946	2832.6374
456.hmmer	92.1533	92.2117	60383424	60085459	1534589.5846	1528565.2696
429.mcf	86.4782	86.3769	47692804	47700080	1982855.5666	1982984.6946
482.sphinx23	81.372	81.4399	599968	599026	16011.0446	15994.0846
462.libquantum	76.9756	77.7146	146579	146870	2947.5556	2952.7354
444.namd	75.8411	76.3736	159852	156677	3194.2046	3139.0364
464.h264ref	74.7855	75.3777	270799	271578	5291.0428	5305.2938

Table 8.13: Performance Effects for PAp Predictor(LSB of BHR is Reversed)

Benchmark	Prediction Accuracy(%)				Clock Cycles			
	N=1	N=4	N=6	N=8	N=1	N=4	N=6	N=8
403.gcc	61.4186	66.2913	71.474	77.2345	87836252	85920228	84338856	82805824
400.perlbench	59.9311	63.7636	68.0441	74.5646	11628160	11707093	11614214	10895872
458.sjeng	99.893	99.9294	99.9446	99.947	283844580	283662570	283586056	283160120
470.lbm	78.4722	79.8611	78.4722	75.4051	138623	137200	138562	138684
456.hmmer	61.9252	74.7882	82.8806	89.4284	64661537	63050504	61814183	60022723
429.mcf	66.5272	64.9867	76.8345	88.178	56036438	56892071	51886792	47053821
482.sphinx23	49.7708	62.5167	77.3303	82.6531	663687	635351	606062	597494
462.libquantum	61.1711	79.6475	79.9886	77.0324	147930	144478	145354	144540
444.namd	75.6205	78.7093	75.6044	75.0689	158166	156920	158512	158252
464.h264ref	63.5063	77.3192	75.7662	73.3551	273652	269361	270481	273043

Table 8.14: Performance Effects for GShare Predictor(Length PC and BHR Changed)

Benchmark	Energy			
	N=1	N=4	N=6	N=8
403.gcc	2473697.9574	2439608.3146	2411368.8878	2384063.4098
400.perlbench	337143.1366	338547.5668	336894.8978	324107.833
458.sjeng	5526831.1434	5523591.3654	5522229.4162	5514647.7554
470.lbm	2799.9744	2774.645	2798.8886	2801.0602
456.hmmer	1610135.4752	1581438.8858	1560079.4132	1527856.8416
429.mcf	2131372.2518	2146602.5192	2057508.553	1971481.6692
482.sphinx23	17145.2428	16640.862	16119.5178	15967.0074
462.libquantum	2971.6034	2910.1578	2925.7506	2911.2614
444.namd	3164.1938	3142.015	3171.6994	3165.7246
464.h264ref	5341.8262	5265.4464	5285.3824	5330.986

Table 8.15: Performance Effects for GShare Predictor(Length PC and BHR Changed)

Benchmark	Prediction Accuracy(%)				Clock Cycles			
	N=1	N=4	N=6	N=8	N=1	N=4	N=6	N=8
403.gcc	64.7491	75.346	84.8665	89.1641	86674276	83385248	80058058	79309951
400.perlbench	61.8533	69.8519	78.422	81.6467	11425903	11389779	11202170	11097707
458.sjeng	99.9186	99.947	99.9448	99.9358	283590215	283502111	283416167	283659748
470.lbm	78.7037	76.5046	72.8009	71.2963	139222	138832	141343	140344
456.hmmer	78.785	92.1533	92.7776	91.4749	63257847	60383424	59909206	60805164
429.mcf	68.2992	86.4782	95.4074	98.9042	55449779	47692804	43777778	42320472
482.sphinx23	54.0616	81.372	84.5477	86.7417	654764	599968	589927	586189
462.libquantum	78.7948	76.9756	72.7118	71.2337	143444	146579	146005	147250
444.namd	77.3304	75.8411	72.7523	70.9873	157825	159852	158601	156834
464.h264ref	76.134	74.7855	70.7397	69.2685	268864	270799	273908	269579

Table 8.16: Performance Effects for PAp Predictor(Length PC and BHR Changed)

Benchmark	Energy			
	N=1	N=4	N=6	N=8
403.gcc	2452949.1762	2394376.5798	2335240.717	2321893.436
400.perlbench	333542.962	332899.9548	329560.5146	327701.0732
458.sjeng	5522303.4464	5520735.1952	5519205.392	5523541.1338
470.lbm	2810.6366	2803.6946	2848.3904	2830.6082
456.hmmer	1585173.4584	1534589.5846	1525869.7166	1543013.7274
429.mcf	2120929.7216	1982855.5666	1913168.1038	1887228.057
482.sphinx23	16986.4134	16011.0446	15832.3148	15765.7784
462.libquantum	2891.7526	2947.5556	2937.3384	2959.4994
444.namd	3158.124	3194.2046	3171.9368	3140.4842
464.h264ref	5256.5998	5291.0428	5346.383	5269.3268

Table 8.17: Performance Effects for PAp Predictor (Length PC and BHR Changed)

Benchmark	Prediction Accuracy(%)				Clock Cycles			
	N=1	N=4	N=6	N=8	N=1	N=4	N=6	N=8
403.gcc	85.5172	85.9951	87.2236	89.0868	80540126	80209030	79766975	79398784
400.perlbench	83.3496	83.4818	85.5783	87.907	10651077	10589517	10557308	10612129
458.sjeng	99.9283	99.9319	99.9462	99.9463	283492972	283196974	283262647	283022446
470.lbm	81.713	77.3148	79.3403	77.0255	138652	140637	140824	139641
456.hmmer	93.0952	93.1	93.096	93.1306	59531475	59931808	60875177	59048582
429.mcf	86.7186	82.9504	91.2512	94.692	47556022	49226385	45789343	44224024
482.sphinx23	68.969	84.262	89.939	90.1519	623613	591041	581248	584310
462.libquantum	81.9215	80.5003	79.9886	79.1927	146694	144364	142359	146793
444.namd	80.695	79.5055	79.9779	77.6613	157711	156810	159720	156401
464.h264ref	79.2808	78.8485	78.4226	75.3984	271309	273337	268291	271122

Table 8.18: Performance Effects for TAGE Predictor (Length PC and BHR Changed)

Benchmark	Energy			
	N=1	N=4	N=6	N=8
403.gcc	2343866.1642	2337946.489	2330080.9884	2323536.6162
400.perlbench	319750.482	318653.5596	318081.971	319057.7848
458.sjeng	5520572.521	5515303.7566	5516472.736	5512197.1582
470.lbm	2800.4906	2835.8236	2839.1522	2818.0948
456.hmmer	1519042.9784	1525384.6834	1542194.9296	1509875.093
429.mcf	1980420.847	2010152.9236	1948973.9608	1921111.2826
482.sphinx23	16431.9256	15851.9516	15677.8286	15732.3322
462.libquantum	2949.6026	2908.1286	2872.4396	2951.3648
444.namd	3156.0948	3141.4038	3191.855	3132.7768
464.h264ref	5300.1208	5336.604	5246.4004	5296.7922

Table 8.19: Performance Effects for TAGE Predictor(Length PC and BHR Changed)

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original Sign-bit	Reversed Sign-bit	Original Sign-bit	Reversed Sign-bit	Original Sign-bit	Reversed Sign-bit
403.gcc	85.93	43.4541	80209030	94350717	2337946.489	2589703.342
400.perlbench	83.23	48.7298	10589517	12224435	318653.5596	347757.7936
458.sjeng	99.93	98.5902	283196974	283495606	5515303.7566	5520619.4062
470.lbm	77.31	31.1343	140637	143603	2835.8236	2888.6184
456.hmmer	93.08	20.3876	59931808	71600194	1525384.6834	1733288.9766
429.mcf	82.95	45.2229	49226385	64970633	2010152.9236	2290401.1152
482.sphinx23	84.25	49.3216	591041	663913	15851.9516	17149.458
462.libquantum	80.5	31.552	144364	150317	2908.1286	3014.092
444.namd	79.42	31.8657	156810	163379	3141.4038	3257.7548
464.h264ref	78.87	37.8831	273337	277100	5336.604	5403.2006

Table 8.20: Performance Effects for TAGE Predictor (Sign Bit for Prediction component is Reversed)

Benchmark	Prediction Accuracy(%)		Clock Cycles		Energy	
	Original LSB	Reversed LSB	Original LSB	Reversed LSB	Original LSB	Reversed LSB
403.gcc	66.27	66.2889	86754099	86186566	2454452.3728	2444335.47
400.perlbench	63.75	63.7644	11766325	11508252	339602.8584	335007.04
458.sjeng	99.92	99.9294	283822066	283616519	5526430.3942	5522771.65
470.lbm	79.86	79.8611	62229116	138790	2800.2948	2802.947
456.hmmer	74.75	74.8021	56861420	63252613	1567938.3322	1584946.19
429.mcf	64.98	64.9865	138641	57036506	2146057.1238	2149173.07
482.sphinx23	62.51	62.5117	633511	634257	16608.3024	16621.19
462.libquantum	79.64	79.6475	144808	143428	2916.0318	2891.46
444.namd	78.7	78.1868	157919	157242	3160.5668	3149.09
464.h264ref	77.31	77.1335	271338	268490	5300.637	5250.32

Table 8.21: Performance Effects for GShare Predictor(LSB of BHR is Reversed)

As can be seen from the experimental records, a significant amount of performance and prediction accuracy changes are observed when these faults are in place.

8.5 Summary

In the first work proposed in this chapter, we empirically study the variation in prediction accuracy at different storage points. We analyze the designs of some popular state-of-the-art predictors that are used in modern pipeline processors and show that they are often not well suited in resource constrained environments. We also present our findings in terms of latency for different predictors for different storage sizes. We believe that our study can help a designer select an appropriate predictor for a specific storage budget. We further examine the resiliency of the predictors against faults on their storage structures.

Chapter 9

Conclusion and Future Directions

The objective of this thesis is to revisit the problem of branch predictor design for low storage processors. On one side, we propose new strategies for storage consolidation that allow us to come up with predictor components that can work on shared predictor tables. On the other side, we combine the benefits of static program analysis to better synthesize strategies for prediction. This gives us a unique advantage, as we demonstrate through extensive experiments on architecture workloads.

In this thesis, we begin our study on using software evolution in predictor design. Software evolution involves addition, deletion and modification of functionality in program source code between different program versions. The change may range from a simple bug fix patch to addition of a new feature. Our objective is to explore if we can use the features of software evolution to improve branch prediction performance. As witnessed from a number of software repositories, the behaviour of a significant number of branches remains unchanged across program versions. This motivates us to explore the idea of using branch outcome profiles from previous versions of the software for enhancing branch prediction for future versions. With our approach, we are able to achieve significant improvement in

predictor energy with limited compromise in prediction accuracy.

Our next study is around designing multi-component predictors. Motivated by the observation that complex programs typically have branches of diverse types, prediction of which require manipulation on history information widely varying in nature, researchers have developed the designs of predictor blocks, which include multiple components. Depending on the nature of the branch under consideration, a single or multiple components may be involved in deriving a prediction. Indeed, these multi-component predictors have proved to be quite successful in the context of server class workloads, and have led to significant accuracy improvements, however, at the cost of more storage required to store the histories of the multiple components. Our objective is to examine these multi-component designs and port them to low storage environments, with appropriate modifications. To this effect, we study an effective 2-component predictor design, and follow it up with a multi-component one, with a static analysis step beforehand. Our contributions are demonstrated on public domain workloads at low storage points, and the results are quite encouraging. This is outlined in Chapters 4 and 5 of this thesis.

The following chapters of the thesis are more around evaluation of branch predictors, a direction that has not been that well studied in architecture literature, to the best of our knowledge. While prediction accuracy has been the metric of paramount concern for branch predictor design, other performance metrics like latency of execution, energy and area footprint have received comparatively less attention. However, in an embedded setting, depending on the context and the application at hand, it may as well be worth exploring the effects of the other metrics, while keeping accuracy losses within a desired limit. Our contribution in Chapter 6 of the thesis outlines our efforts in this direction. In particular, we propose a framework for branch predictor selection, that simultaneously considers multiple performance metrics. Our experiments reveal interesting findings, contrary to expectations since many of the promising predictor designs in terms of accuracy fare badly when the

other metrics are examined. We believe that our framework and mechanism for predictor component selection is an important piece in the landscape of predictor research.

In Chapter 7, we examine the performance of branch predictor designs a little more closely, more specifically, in a concurrent execution environment. In particular, the objective of our study is to explore how the latency of execution can be compromised by forcing the predictor to mis-predict, by crafting clone threads with negated branch conditions that are dispatched in parallel with a given application. Once we move to an embedded environment where storage is of paramount concern, a natural design choice as adopted by others and us as well, is a design with shared predictor tables to save storage. We show that such a design can be easily compromised by clone threads leading to performance slowdowns. Experimental results are quite surprising, given that the amount of performance compromise is significant, while the clone creation is a simple automated activity not requiring much effort. We demonstrate that our study applies readily to any application for which handle to the source is available, thereby allowing one to create the clone application, with a little handle to the system scheduler and the dispatcher to create the concurrent execution environment needed for the exploit.

In the final chapter, we study the issue of storage and fault sensitivity of contemporary predictor designs. In particular, we study through workload simulations at different storage points, how these predictors fare, and the results justify our intuition that sophisticated predictor algorithms require more storage for full performance. In fact, as we witness in our simulations, simple predictor designs often surpass sophisticated predictors significantly at low storage points. Our final study is to study the resilience of predictor designs against faults in the predictor table entries and register cells. In particular, we demonstrate empirically that the correctness and performance of these prediction strategies are not quite resilient to faults, their performance fluctuates considerably.

This thesis opens up a lot of avenues for future exploration around predictor design. We

outline some of the directions below.

9.1 Branch prediction for indirect branches

With increasing use of runtime polymorphism and reliance on runtime type interpretation, the presence and importance of indirect branches has seen a considerable rise in recent workloads. Evidently, accurate target prediction for indirect branches has emerged as an important problem. While direction prediction of direct branches has received considerable research attention leading to efficient prediction policies and hardware structures implemented inside modern processors, proposals for target prediction for indirect branches has been relatively few. The problem of accurate target prediction for indirect branches is significantly tough since these transfer control to an address stored in a register that is known only at runtime. Unlike conditional direct branches, indirect branches can have more than two targets to be resolved at runtime, for which prediction requires a full 32-bit / 64-bit address to be predicted, in contrast to just a taken or not-taken decision as needed for direction prediction of direct branches. Recent research shows indirect branches, being mispredicted more frequently, can start to dominate the overall branch misprediction cost. In modern processors, the only hardware structure available to facilitate target address prediction for indirect branches is a fixed-size Branch Target Buffer (BTB). BTB is often designed as a set associative cache for storing recent target addresses for branch instructions encountered during execution, with a motivation of being able to reuse the same addresses for future instances, thereby saving latency cycles. Evidently, designing efficient indexing schemes and replacement mechanisms for BTB structures is crucial, more so, for indirect branches, since these serve as the only prediction handle. We wish to take up the problem of BTB design and associated mechanisms that are particularly suited for prediction of indirect branches.

9.2 Analyzing predictability of branches

Branch predictors often use different lengths of branch outcome history and path history for prediction. We can think of 1-bit, 2-bit, 3-bit... m-bit histories as random variables and find the entropy of each of these distributions. Using the entropy, we aim to find the randomness of the branch history and thereby, identify the hard to predict or easy to predict branches. Additionally, we can check the dependence of a branch decision on its past history by evaluating the autocorrelation of a N-bit history at different time lags. Auto correlations of a binary sequence provides the DPI (Degree of Pattern Irregularity) values and can be used to evaluate the DPI and EPL (effective pattern length) of the branch history. We plan to use these information to identify the hard to predict and easy to predict branches as well as the effective length of history for prediction.

9.3 Predictor design for multi-core embedded processors

Modern embedded processors today have multiple cores that facilitate concurrent workloads of execution simultaneously, thereby generating performance benefits. For such multi core designs, having individual branch predictors inside each core is the design of choice today, with each predictor having its isolated predictor table. While this helps in maintaining isolation between cores and nullifies interference, the benefit of cross core learning is ruled out. Modern neural workloads today often employ an initial training phase, wherein the same neural code is executed on multiple training tests to train the network. For multi-threaded programs as well, one thread executing in one core can benefit from history information from the peer threads executing in the other cores. We wish to examine a hierarchical BTB design for multicores with total size comparable to what exists today, with a motivation towards possible improvement of prediction accuracy by facilitating collaborative constructive learning between programs executing in different cores that encounter similar histories.

9.4 Branch prediction with approximate computing

Approximate computing is now an accepted model of computation today in a wide variety of application domains, starting from image processing to machine learning. The key idea of approximate computing is to trade off correctness within acceptable tolerance limits, thereby saving on latency, energy and resource. For resource constrained embedded processors, approximate computing has found widespread acceptance since this paves the way for a wide variety of sophisticated algorithms to be realized on these cores, with marginal correctness degradations but often significant performance benefits. Most modern embedded processors today employ speculative execution, by which latency sensitive operations (e.g. memory access, branch resolution) are allowed to execute with predicted values, and later compensated for, when the actual outcome is known. We have made some initial progress on strategies to connect speculative execution with approximate computing. For branch instructions that are known to be approximable without affecting program output beyond a threshold, we have proposed a novel scheme by which in case of a misprediction, a rollback can be avoided. We have designed the hardware interface through which this can be made possible, and shown through extensive simulations the benefits in terms of latency that has been achieved. We believe this will make approximate computing more effective today for modern embedded processors in the market. We believe this idea can be extended to more complex workloads and domains, with suitable modifications across the execution stack.

Bibliography

[1] Dual-port srams. https://www.cypress.com/products/dual-port-srams.

[2] Frama-c. http://frama-c.com/.

[3] Gzip repository. http://ftp.gnu.org/gnu/gzip/.

[4] Ieee standard verilog hardware description language - xilinx forums. https://forums.xilinx.com/xlnx/attachments/xlnx/.../1/Verilog

[5] Model counters. http://beyondnp.org/pages/solvers/model-counters-exact/.

[6] objdump - display information from object filescbp. https://www.systutorials.com/docs/linux/man/1-objdump/.

[7] Python. https://www.python.org/.

[8] Sesc: Superescalar simulator. http://iacoma.cs.uiuc.edu/ paulsack/sescdoc/.

[9] Simplescalar llc. http://www.simplescalar.com/.

[10] Software artifacts repository. http://sir.unl.edu/portal/index.php.

[11] Tejas website. http://www.cse.iitd.ac.in/tejas/index.html.

[12] Tsmc - 90nm. https://www.synopsys.com/dw/.

[13] Veriwell simulator. https://veriwell-verilog-simulator.soft112.com/.

[14] Weakest precondition calculuse. http://en.wikipedia.org/wiki/Predicate_transformer_semantics.

[15] Welcome to lpsolveapi project. http://lpsolve.r-forge.r-project.org/.

[16] O. Aciiçmez. Yet another microarchitectural attack:: exploiting i-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, pages 11–18. ACM, 2007.

[17] O. Acıiçmez, S. Gueron, and J.-P. Seifert. New branch prediction vulnerabilities in openssl and necessary software countermeasures. In *IMA*, pages 185–203, 2007.

[18] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert. On the power of simple branch prediction analysis. In *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 312–320. ACM, 2007.

[19] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert. Predicting secret keys via branch prediction. In *Cryptographers Track at the RSA Conference*, pages 225–242, 2007.

[20] R. Baldoni, E. Coppa, D. C. Delia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):50, 2018.

[21] T. Ball and J. R. Larus. *Branch prediction for free*, volume 28. ACM, 1993.

[22] A. Banerjee, A. Roychoudhury, J. A. Harlie, and Z. Liang. Golden implementation driven software debugging. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 177–186. ACM, 2010.

[23] A. Baniasadi and A. Moshovos. Branch predictor prediction: A power-aware branch predictor for high-performance processors. In *Computer Design: VLSI in Computers and Processors, 2002. Proceedings. 2002 IEEE International Conference on*, pages 458–461. IEEE, 2002.

[24] J. Bartholdi, C. A. Tovey, and M. A. Trick. Voting schemes for which it can be difficult to tell who won the election. *Social Choice and welfare*, 6(2):157–165, 1989.

[25] P. Bhattacharya, M. Iliofotou, I. Neamtiu, and M. Faloutsos. Graph-based analysis and prediction for software evolution. In *Proceedings of the 34th International Conference on Software Engineering*, pages 419–429. IEEE Press, 2012.

[26] M. I. Bielby. *Ultra low power cooperative branch prediction.* The University of Edinburgh, 2015.

[27] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, and S. Sardashti. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[28] D. Black. Partial justification of the borda count. *Public Choice*, 28(1):1–15, 1976.

[29] J. J. Bonanno, N. Nijhawan, and B. R. Prasky. Hybrid branch prediction using a global selection counter and a prediction method comparison table, Aug. 30 2005. US Patent 6,938,151.

[30] S. Burman, D. Mukhopadhyay, and K. Veezhinathan. Lfsr based stream ciphers are vulnerable to power attacks. In *INDOCRYPT 2007*, pages 384–392. Springer, 2007.

[31] B. Calder, D. Grunwald, M. Jones, D. Lindsay, J. Martin, M. Mozer, and B. Zorn. Evidence-based static branch prediction using machine learning. *ACM Trans. Program. Lang. Syst.*, 19(1):188–222, Jan. 1997.

[32] CBP-2. 2nd jilp workshop on computer architecture competitions. https://www.jilp.org/jwac-2/, 2011.

[33] CBP-5. 5th jilp workshop on computer architecture competitions. https://www.jilp.org/cbp2016/, 2016.

[34] P.-Y. Chang, M. Evers, and Y. N. Patt. Improving branch prediction accuracy by reducing pattern history table interference. *International journal of parallel programming*, 25(5):339–362, 1997.

[35] P.-Y. Chang, E. Hao, and Y. N. Patt. Alternative implementations of hybrid branch predictors. In *MICRO*, pages 252–257. IEEE Computer Society Press, 1995.

[36] P.-Y. Chang, E. Hao, T.-Y. Yeh, and Y. Patt. Branch classification: a new mechanism for improving branch predictor performance. In *MICRO*, pages 22–31. ACM, 1994.

[37] C. T. Do, H. J. Choi, D. O. Son, J. M. Kim, and C. H. Kim. Ntb branch predictor: dynamic branch predictor for high-performance embedded processors. *The Journal of Supercomputing*, pages 1–15, 2014.

[38] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10(4):405–435, 2005.

[39] M. Dummett. The borda count and agenda manipulation. *Social Choice and Welfare*, 15(2):289–296, 1998.

[40] G. Dupenloup. Automatic synthesis script generation for synopsys design compiler, Dec. 28 2004. US Patent 6,836,877.

[41] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of the 10th international conference on World Wide Web*, pages 613–622. ACM, 2001.

[42] A. N. Eden and T. Mudge. The yags branch prediction scheme. In *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pages 69–77. IEEE, 1998.

[43] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous multithreading: A platform for next-generation processors. *IEEE micro*, 17(5):12–19, 1997.

[44] P. Emerson. The original borda count and partial voting. *Social Choice and Welfare*, pages 1–6, 2013.

[45] M. Evers, P.-Y. Chang, and Y. N. Patt. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *ACM SIGARCH Computer Architecture News*, volume 24, pages 3–11. ACM, 1996.

[46] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An analysis of correlation and predictability: What makes two-level branch predictors work. In *ACM SIGARCH Computer Architecture News*, volume 26, pages 52–61. IEEE Computer Society, 1998.

[47] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):10, 2016.

[48] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, and D. Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. In *ACM SIGPLAN Notices*, volume 53, pages 693–707. ACM, 2018.

[49] A. Fog. The microarchitecture of intel, amd and via cpus. *An optimization guide for assembly programmers and compiler makers. Copenhagen University College of Engineering*, 2011.

[50] R. Ghosh, A. Gupta, S. Chattopadhyay, A. Banerjee, and K. Dasgupta. Cocoa: A framework for comparing aggregate client operations in bpo services. In *SCC*, pages 539–546. IEEE, 2016.

[51] D. Gope and M. H. Lipasti. Bias-free neural predictor. *The 4th Championship Branch Prediction, http://www. jilp. org/cbp2014*, 2014.

[52] J. L. Henning. Spec cpu2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[53] G. G. Henry and T. Parks. Hybrid branch predictor with improved selector table update mechanism, Apr. 15 2003. US Patent 6,550,004.

[54] M. Hicks, C. Egan, B. Christianson, and P. Quick. Towards an energy efficient branch prediction scheme using profiling, adaptive bias measurement and delay region scheduling. In *Design & Technology of Integrated Systems in Nanoscale Era, 2007. DTIS. International Conference on*, pages 19–24. IEEE, 2007.

[55] M. Huang, D. He, X. Liu, M. Tan, and X. Cheng. An energy-efficient branch prediction with grouped global history. In *Parallel Processing (ICPP), 2015 44th International Conference on*, pages 140–149. IEEE, 2015.

[56] M. C. Huang, D. Chaver, L. Pinuel, M. Prieto, and F. Tirado. Customizing the branch predictor to reduce complexity and energy consumption. *IEEE micro*, 23(5):12–25, 2003.

[57] Y. Ishii. Global-local combined branch history: The alternative way to improve tage branch predictor. *JWAC-4: Championship Branch Prediction. JILP*, 2014.

[58] Y. Ishii, K. Kuroyanagi, T. Sawada, M. Inaba, and K. Hiraki. Revisiting local history to improve the fused two-level branch predictor. *3rd Championship Branch Prediction*, 2010.

[59] D. A. Jiménez. Fast path-based neural branch prediction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 243. IEEE Computer Society, 2003.

[60] D. A. Jiménez. Fast path-based neural branch prediction. In *Microarchitecture, 2003. MICRO-36. Proceedings. 36th Annual IEEE/ACM International Symposium on*, pages 243–252, Dec 2003.

[61] D. A. Jiménez. Piecewise linear branch prediction. In *32nd International Symposium on Computer Architecture (ISCA'05)*, pages 382–393. IEEE, 2005.

[62] D. A. Jiménez. Oh-snap: Optimized hybrid scaled neural analog predictor. *Proceedings of the 3rd Championship on Branch Prediction, http://www. jilp. org/jwac-2*, 2011.

[63] D. A. Jiménez. Multiperspective perceptron predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[64] D. A. Jiménez and C. Lin. Dynamic branch prediction with perceptrons. In *HPCA*, pages 197–206. IEEE, 2001.

[65] D. A. Jiménez and C. Lin. Neural methods for dynamic branch prediction. *ACM Trans. Comput. Syst.*, 20(4):369–397, Nov. 2002.

[66] D. A. Jiménez and E. Teran. Multiperspective reuse prediction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 436–448. ACM, 2017.

[67] R. E. Jimenez, A. L. Warshaw, D. W. Rattner, C. G. Willett, D. McGrath, and C. Fernandez-del Castillo. Impact of laparoscopic staging in the treatment of pancreatic cancer. *Archives of Surgery*, 135(4):409–415, 2000.

[68] J. G. Kemeny. Mathematics without numbers. *Daedalus*, 88(4):577–591, 1959.

[69] R. E. Kessler. The alpha 21264 microprocessor. *IEEE micro*, 19(2):24–36, 1999.

[70] R. E. Kessler. The alpha 21264 microprocessor. *IEEE micro*, 19(2):24–36, 1999.

[71] Z. F. Lansdowne and B. S. Woodward. Applying the borda ranking method. *Air Force Journal of Logistics*, 20(2):27–29, 1996.

[72] A. R. Lebeck, X. Fan, H. Zeng, and C. Ellis. Power aware page allocation. *ACM Sigplan Notices*, 35(11):105–116, 2000.

[73] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[74] G. H. Loh and D. S. Henry. Predicting conditional branches with fusion-based hybrid predictors. In *Parallel Architectures and Compilation Techniques, 2002. Proceedings. 2002 International Conference on*, pages 165–176. IEEE, 2002.

[75] G. H. Loh, D. S. Henry, and A. Krishnamurthy. Exploiting bias in the hysteresis bit of 2-bit saturating counters in branch predictors. *Journal of Instruction-Level Parallelism*, 5:1–32, 2003.

[76] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[77] G. Malhotra, P. Aggarwal, A. Sagar, and S. R. Sarangi. ParTejas: A parallel simulator for multicore processors. In *ISPASS*, 2014.

[78] S. McFarling. Combining branch predictors. Technical report, Technical Report TN-36, Digital Western Research Laboratory, 1993.

[79] N. Megiddo and D. S. Modha. Outperforming lru with an adaptive replacement cache algorithm. *Computer*, 37(4):58–65, 2004.

[80] P. Michaud. An alternative tage-like conditional branch predictor. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(3):30, 2018.

[81] P. Michaud and A. Seznec. Pushing the branch predictability limits with the multi-potage+ sc predictor. In *4th JILP Workshop on Computer Architecture Competitions (JWAC-4): Championship Branch Prediction (CBP-4)*, 2014.

[82] P. Michaud, A. Seznec, and R. Uhlig. Trading conflict and capacity aliasing in conditional branch predictors. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 292–303. ACM, 1997.

[83] M. Mohammadi, S. Han, T. M. Aamodt, and W. J. Dally. On-demand dynamic branch prediction. *Computer Architecture Letters*, 14(1):50–53, 2015.

[84] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *ACM Sigplan Notices*, volume 27, pages 76–84. ACM, 1992.

[85] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. R. Stan. Power issues related to branch prediction. In *HPCA*, pages 233–244. IEEE, 2002.

[86] H. Patil and J. Emer. Combining static and dynamic branch prediction to reduce destructive aliasing. In *HPCA-6*, pages 251–262. IEEE, 2000.

[87] J. R. Patterson. Accurate static branch prediction by value range propagation. In *ACM SIGPLAN Notices*, volume 30, pages 67–78. ACM, 1995.

[88] A. Phansalkar, A. Joshi, and L. K. John. Analysis of redundancy and application balance in the spec cpu2006 benchmark suite. In *ACM SIGARCH Computer Architecture News*, volume 35, pages 412–423. ACM, 2007.

[89] S. Pruett, S. Zangeneh, A. Fakhrzadehgan, B. Lin, and Y. Pat. Dynamically sizing the tage branch predictor, 2016.

[90] D. Qi, A. Roychoudhury, Z. Liang, and K. Vaswani. DARWIN: An approach for debugging evolving programs. In *ESEC-FSE*, 2009.

[91] M. Ramsay, C. Feucht, and M. H. Lipasti. Exploring efficient smt branch predictor design. In *Workshop on Complexity-Effective Design, in conjunction with ISCA*, volume 26, 2003.

[92] M. E. Renda and U. Straccia. Web metasearch: rank vs. score based rank aggregation methods. In *Proceedings of the 2003 ACM symposium on Applied computing*, pages 841–846. ACM, 2003.

[93] R. A. Santelices, Y. Zhang, H. Cai, and S. Jiang. Change-effects analysis for evolving software. *Advances in Computers*, 93:227–285, 2014.

[94] S. R. Sarangi, R. Kalayappan, P. Kallurkar, S. Goel, and E. Peter. Tejas: A java based versatile micro-architectural simulator. In *2015 25th International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 47–54. IEEE, 2015.

[95] A. Seznec. Analysis of the o-geometric history length branch predictor. In *ISCA*, pages 394–405. IEEE, 2005.

[96] A. Seznec. A case for (partially) tagged geometric history length branch prediction. *JILP*, 8:1–23, 2006.

[97] A. Seznec. The l-tage branch predictor. *Journal of Instruction-Level Parallelism*, 9, 2007.

[98] A. Seznec. A 64 kbytes isl-tage branch predictor. In *JWAC-2: Championship Branch Prediction*, 2011.

[99] A. Seznec. A new case for the tage branch predictor. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 117–127. ACM, 2011.

[100] A. Seznec. Tage-sc-l branch predictors. In *JILP-Championship Branch Prediction*, 2014.

[101] A. Seznec. Exploring branch predictability limits with the mtage+ sc predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, page 4, 2016.

[102] A. Seznec. Tage-sc-l branch predictors again. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, 2016.

[103] A. Seznec, S. Felix, V. Krishnan, and Y. Sazeides. Design tradeoffs for the alpha ev8 conditional branch predictor. *ACM SIGARCH Computer Architecture News*, 30(2):295–306, 2002.

[104] N. Simjour. Improved parameterized algorithms for the kemeny aggregation problem. In *International Workshop on Parameterized and Exact Computation*, pages 312–323. Springer, 2009.

[105] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th annual symposium on Computer Architecture*, pages 135–148. IEEE Computer Society Press, 1981.

[106] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt. The agree predictor: A mechanism for reducing negative branch history interference. In *ACM SIGARCH Computer Architecture News*, volume 25, pages 284–291. ACM, 1997.

[107] R. St Amant, D. A. Jiménez, and D. Burger. Low-power, high-performance analog neural branch prediction. In *MICRO*, pages 447–458. IEEE, 2008.

[108] R. St. Amant, D. A. Jimenez, and D. Burger. Low-power, high-performance analog neural branch prediction. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, pages 447–458, 2008.

[109] R. St Amant, D. A. Jiménez, and D. Burger. Low-power, high-performance analog neural branch prediction. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 447–458. IEEE Computer Society, 2008.

[110] X. Su, H. Wu, and Q. Yang. An efficient wcet-aware hybrid global branch prediction approach. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2016 IEEE 22nd International Conference on*, pages 195–201. IEEE, 2016.

[111] N. Tasher, V. Teper, D. C. Cheng, and B. Tabachnik. Protection against side-channel attacks on non-volatile memory, May 17 2016. US Patent 9,343,162.

[112] K. Thangarajan, W. Mahmoud, E. Ososanya, and P. Balaji. Survey of branch prediction schemes for pipelined processors. In *Proceedings of the Thirty-Fourth Southeastern Symposium on System Theory (Cat. No. 02EX540)*, pages 324–328. IEEE, 2002.

[113] D. M. Tullsen, S. J. Eggers, and H. M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 392–403. ACM, 1995.

[114] M. Van Erp and L. Schomaker. Variants of the borda count method for combining ranked classifier hypotheses. In *Seventh International Workshop on Frontiers in Handwriting Recognition*. Citeseer, 2000.

[115] L. Vintan and M. Iridon. Towards a high performance neural branch predictor. In *Neural Networks, 1999. IJCNN '99. International Joint Conference on*, volume 2, pages 868–873, Jul 1999.

[116] C. A. Waldspurger. Memory resource management in vmware esx server. *ACM SIGOPS OSR*, 36(SI):181–194, 2002.

[117] Y. Wang and L. Chen. Dynamic branch prediction using machine learning. *ECS-201A, Fall*, 2015.

[118] T.-Y. Yeh and Y. N. Patt. Two-level adaptive training branch prediction. In *MICRO*, pages 51–61. ACM, 1991.

[119] T.-Y. Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. *ACM SIGARCH Computer Architecture News*, 20(2):124–134, 1992.

[120] T.-Y. Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. *ACM SIGARCH Computer Architecture News*, 21(2):257–266, 1993.

[121] C. Young, N. Gloy, and M. D. Smith. *A comparative analysis of schemes for correlated branch prediction*, volume 23. ACM, 1995.

[122] C. Young and M. D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *ACM Sigplan Notices*, volume 29, pages 232–241. ACM, 1994.