MICROBENCHMARKS AND MECHANISMS FOR REVERSE ENGINEERING OF MODERN BRANCH PREDICTOR UNITS

by

VLADIMIR UZELAC

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Engineering
in
The Department of Electrical & Computer Engineering
to
The School of Graduate Studies
of
The University of Alabama in Huntsville

HUNTSVILLE, ALABAMA

2008

In presenting this thesis in partial fulfillment of the requirements for a master's degree from The University of Alabama in Huntsville, I agree that the Library of this University shall make it freely available for inspection. I further agree that permission for extensive copying for scholarly purposes may be granted by my advisor or, in his/her absence, by the Chair of the Department or the Dean of the School of Graduate Studies. It is also understood that due recognition shall be given to me and to The University of Alabama in Huntsville in any scholarly use which may be made of any material in this thesis.

_____ _____

(student signature) (date)

THESIS APPROVAL FORM

Submitted by Vladimir Uzelac in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering and accepted on behalf of the Faculty of the School of Graduate Studies by the thesis committee.

We, the undersigned members of the Graduate Faculty of The University of Alabama in Huntsville, certify that we have advised and/or supervised the candidate on the work described in this thesis. We further certify that we have reviewed the thesis manuscript and approve it in partial fulfillment of the requirements for the degree of Master of Science in Engineering in Computer Engineering.

_____ Committee Chair
(Date)

_____

_____

_____

_____ Department Chair

_____ College Dean

_____ Graduate Dean

ABSTRACT

The School of Graduate Studies
The University of Alabama in Huntsville

Degree Master of Science in Engineering College/Dept. Engineering/Electrical & Computer Engineering

Name of Candidate Vladimir Uzelac

Title Microbenchmarks and Mechanisms for Reverse Engineering of Modern Processor Branch Predictor Units

Modern mid- and high-end microprocessors employ deeper and wider pipelines. With deeper pipelines, the penalties incurred for branch instruction resolution have become much larger, limiting potential performance gains. To mitigate the negative impacts of increased branch penalties, branch predictor units are employed to speculate on both branch target and branch outcomes. Branch prediction techniques have been a very active area of research in both academia and industry. It has been shown that knowledge of exact branch predictor organization can be used in compiler optimization to improve performance. However, exact predictor organizations of commercial processors are rarely made public. In this thesis we introduce a framework for reverse engineering of modern branch predictor units, encompassing a set of mechanisms and microbenchmarks. We demonstrate the use of this framework in reverse engineering of the Pentium M's branch predictor unit -- one of the most sophisticated commercial branch predictor units ever developed. With this framework we have been able to uncover size, organization, internal functioning, and interactions between various hardware structures used in the Pentium M's branch predictor unit, such as branch target buffer, indirect branch target buffer, loop branch predictor buffer, global predictor, and bimodal predictor.

Abstract Approval: Committee Chair _____

Department Chair _____

Graduate Dean _____

ACKNOWLEDGMENTS

The work presented in this thesis would not have been possible without the assistance of a number of people who deserve special mention.

First, I would like to thank my advisor, Dr. Aleksandar Milenkovic, for introducing a problem of reverse engineering of branch predictors to me and for his continual support and guidance throughout all the stages of the work. Second, I would like to thank my committee members, Dr. Jeffrey Kulick and Dr. Earl Wells, for their invaluable insights and feedback in shaping this work. I am also grateful to Dr. Reza Adhami, Chair of the Electrical and Computer Engineering Department, for his support. I would also like to acknowledge support from my fellow students in the LaCASA Laboratory: Mr. Austin Rogers, Mr. Joel Wilder, and Mr. Richard Tuggle.

Last but not least, I would like to thank Tijana, my wife. Without her love and encouragement this work would have not been possible. Finally, I am grateful to my parents, Vukolaja and Nada Uzelac, who have always encouraged and supported me in my pursuit of education.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1

INTRODUCTION

1.1 Background

Instructions that control program flow encompass branches, jumps, procedure calls and returns, and traps. A successful and timely branch resolution is critical for improving performance of all modern processors with pipelined execution. A conditional branch is resolved once we know a branch outcome (taken or not taken) and the target address (the address of the next instruction if the branch is taken). Branches are typically resolved in late stages of the pipeline; in the meantime all the instructions following a branch instruction must wait for the branch resolution. This degrades performance significantly and the problem is exacerbated in superscalar processors that fetch and execute multiple instructions each clock cycle. A common solution to this problem is speculative execution where the branch outcome and target address are dynamically or statically predicted, so the execution can go on without stalling. If the prediction is correct, no processor clock cycles are wasted. Otherwise, the instructions that entered the pipeline speculatively are flushed, and the program execution resumes from the real

target. While static prediction is effective, it is insufficient in modern processors with deep pipelines and hundreds of instructions on-the-fly in various stages of the pipeline. Hence, modern processors rely on branch predictor units to speculatively predict outcomes and target addresses of incoming branch instructions.

To illustrate the importance of branch predictor units, let us consider a single-issue superscalar processor with ideal Cycles per Instruction (CPI), CPI = 1. Let us assume that the branch penalty is 20 clock cycles. Branch instructions make around 20% of all instructions [1]. With pipeline stalling the CPI = 1 + 0.2×20 = 5, a fivefold increase in program execution time. To remedy this, let us introduce a branch predictor unit with a misprediction rate of 10% (9 out of 10 branches will be predicted correctly and one will be mispredicted resulting in a 20 clock cycle penalty). The new CPI = 1 + 0.2×0.1×20 = 1.4, which is still 40% of slowdown. Let us add an improvement in the branch predictor unit to achieve a misprediction rate of 5%; the CPI = 1.2, a significant improvement in performance. Though based on many simplifications, this example fairly illustrates the importance of having branch predictor units with very low misprediction rates. As branch predictors reside on the critical path of processor front-ends, it is also very important that a correct prediction and a target address are available with minimal latency. Finally, the importance of good branch predictors is further underscored with the proliferation of mobile battery-operated platforms. Mispredictions tend to be costly in terms of energy – pursuing wrong paths in program execution wastes limited energy resources.

1.2 Motivation

This thesis introduces a set of microbenchmarks and mechanisms for reverse engineering of modern branch predictor units. The microbenchmarks and experimental flows are applied for reverse engineering of a branch predictor unit found in Intel's Pentium M processors (Dothan core), one of the most advanced branch predictor units ever developed in commercial processors [2]. Intel does not disclose information about the exact branch predictor organization, but it claims that this predictor significantly outperforms previous generation branch predictor designs by 20%.

This work builds on an initial effort in reverse engineering of branch predictor units done by Milenkovic *et al.* [3] where an experimental flow and a set of microbenchmarks are developed and applied to Intel's P6 and NetBurst architectures. A set of carefully crafted "spy" microbenchmarks is applied on a real machine to verify various hypotheses related to branch predictor unit organization. Processor behavior is measured during execution of the spy microbenchmarks using on-chip performance monitoring registers. Based on these results, we have been able to obtain insights into the complete organization of the branch predictor unit in Pentium M processors.

The developed experimental flows and microbenchmarks have potential to greatly benefit academia and industry by allowing for (a) better code optimization techniques either through manual or automatic optimization using architecture-aware compilers, (b) a systematic approach in architectural analysis that can be used for targeted system verification and design space exploration in early or late design stages, and (c) bridging the gap between academia and industry. Each of these three potential benefits is discussed below.

1.2.1 Architecture-aware Compilers

Processors complexity grows in each generation. Instruction-level parallelism, branch prediction mechanisms, multiple-level cache hierarchy, and hardware prefetchers are just a few of processors complex structures that greatly influence program execution time. Compliers typically offer several optimization parameters for a given architecture. However, these optimizations are often limited and do not exploit specific characteristics of the processor the program is running on. To take advantage of specific processor features, a more architecture-aware compiler is desirable.

It has been shown that making compilers aware of the underlying architecture can result in significant performance improvements [4-6]. Architecture aware compliers will become even more important with proliferation of new multi-core processors. For example, a set of programs for automated extraction of the memory hierarchy [4] can be used to guide a manual or automatic program optimization. Gurumani and Milenkovic show that Intel's C++ compiler outperforms Microsoft's Visual C++ compiler on the SPEC CPU 2000 benchmark suite [5]. Jimenez starts with microbenchmarks for determining branch predictor organization in NetBurst architecture [3] and introduces the Camino C compiler that performs code reorganization in order to reduce the number of branch mispredictions [6]. Unfortunately, the details about the underlying architecture are rarely disclosed. Consequently, we need a systematic approach for extracting these secrets from modern processors.

1.2.2 Hardware Design Verification

Ever growing processor complexity and tightening time-to-market make hardware design verification a critical step during the design process. However, often late in the

design process architects introduce changes to the original design in order to achieve performance improvements, reduce cost and power consumption. Full verification that usually assumes running an operating system on the top of a HDL model of the processor is very expensive in time and resources. Obviously, these changes cannot be fully verified using conventional approaches. In addition, it is often very difficult to assess an impact of the introduced changes as the total performance depends on many different structures with very complex interactions between each other. Hence, there is a great need for microbenchmarks -- small programs that target a single structure or a specific functionality. Their small sizes allow designers to manually trace the microbenchmark execution or to completely predict their behavior. The microbenchmarks hence provide a useful tool in the design-phase for architectural analysis and rapid design-space exploration and verification.

1.2.3 Bridging the Gap between Industry and Academia

Research efforts in academia typically focus more on improving branch predictor accuracy and rarely on area and timing constraints imposed by chip economics and design specifications. On the other hand, branch predictor designs in industry strive to achieve the best prediction accuracy within given design constraints, such as timing constraints and hardware budget. In conditions where manufacturers conceal details about actual branch predictor implementations, these opposite approaches lead to an increase in the gap between branch predictors developed in industry and academia.

The experimental flow discussed in this thesis is hand-crafted for the branch predictor unit found in Intel's Pentium M processor. However, we believe that the systematic approach and methodology can be adapted for reverse engineering of all other

modern branch predictor units. The knowledge gained through this effort can be very useful for future research efforts in academia as it can serve as a solid starting point.

1.3 Thesis Outline

The thesis is organized in nine chapters as follows. Chapter 1 gives an introduction, stressing the importance of the thesis topic and listing the main contributions of this work. Chapter 2 surveys historical development of branch predictor units and discusses requirements and design-space constraints in designing a branch predictor unit. Chapter 3 describes relevant industrial branch predictor unit implementations. Chapter 4 describes the experimental environment, including microbenchmark development, microbenchmark deployment, and actual testing and measurement. Chapter 5 introduces experimental flows and microbenchmarks for determining organization of a regular branch target buffer. Chapter 6 introduces experimental flows and microbenchmarks for determining organization of a loop predictor. Chapter 7 introduces experimental flow and microbenchmarks used in determining organization of an indirect branch predictor, while Chapter 8 focuses on a global branch outcome predictor. Chapter 9 concludes the thesis and discusses future work.

1.4 Contributions

The main contribution of this thesis is a set of experimental flows and microbenchmarks for determining organization of modern branch predictor units. The experimental flows and microbenchmarks are employed on Intel's Pentium M processor (Dothan core) giving the following insights into the branch predictor organization.

The Intel Pentium M branch predictor unit consists of five main parts as follows:

- Branch target buffer (BTB). The BTB caches target addresses for most the frequently used conditional and unconditional branches.
- Indirect Branch target buffer (iBTB). The iBTB caches the most frequently used target addresses of indirect branches.
- Bimodal predictor. The bimodal predictor is a simple outcome predictor that gives a first level outcome prediction for all conditional branches.
- Loop predictor. The loop predictor is a specialized predictor used to provide the second level outcome prediction for conditional branches with loop behavior.
- Global predictor. The global predictor is a specialized predictor used to provide the third level outcome prediction for conditional branches.

Branch Target Buffer

The BTB is a 4-way cache organized in 512 ways (the total size is 2048 entries). The BTB is accessed by a 16-byte instruction block address and it is indexed by the branch address bits IP[12:4]. Tag bits are branch address bits IP[21:13]. Branch addresses are determined as follows: it is the address of the first byte of the branch instruction in memory if that instruction belongs to a single 16-byte block; it is the address of the last byte in memory if the branch instruction spans multiple 16-byte blocks. A BTB set can store multiple entries with the same tag bits, thanks to an offset field in the BTB (Offset field are IP address bits [3:0]) and a so called offset mechanism. The offset mechanism selects a target address by selecting an entry with the lowest offset which is not smaller than the current instruction pointer.

In case of a BTB miss or incorrect target address on a BTB hit, the instruction decoder supplies a true target at the end of instruction decoding, assuming that outcome prediction is correct. The BTB may give a BTB hit for a non branch instruction due to partial tag fields. In this case, the instruction decoder evicts the selected entry in the BTB. The BTB employs a specific allocation/replacement policy. The replacement policy is "least recently used" (LRU) –based, but there is an indication that a branch needs to occur at least twice before this policy is reinforced.

Indirect Target Buffer

The indirect branch target buffer (iBTB) is a direct–mapped cache organized in 256 sets. Each iBTB entry keeps a target address of an indirect branch. The iBTB access is controlled by an index and a tag field. The index and tag are calculated as a hash between a path information register (PIR) and the indirect branch address.

The PIR is a 15-bit long shift register that keeps path history for the last 8 relevant branch instructions. Only conditional taken branches and indirect branches affect the PIR. The PIR register is updated as follows. When a new branch is encountered, the PIR is shifted to the left for two bits and 15 bits of the branch IP address or the branch target address are XOR-ed with the PIR. For a conditional taken branch, the branch address bits IP[18:4] are XOR-ed with the PIR. For an indirect branch, the branch target address bits TA[5:0] are XOR-ed with the PIR bits PIR[5:0], and the address bits IP[18:10] are XOR-ed with the PIR bits PIR[14:6].

The iBTB is indexed by the hash function represented by the XOR between the PIR and the indirect branch IP as follows. The PIR bits PIR[13:6] are XOR-ed with the indirect branch address bits IP[11:4] to provide the index for the iBTB. The PIR bits

[14,5:0] are XOR-ed with the indirect branch IP bits [18:12] to provide the tag for the iBTB.

The direct-mapped organization of the iBTB indicates that the iBTB lookup is serialized with the corresponding BTB lookup. The iBTB lookup is performed only if the BTB indicated an indirect branch. This serialization incurs additional delay, but it saves energy. The increased latency is reduced by using a direct–mapped cache for the iBTB. If the BTB gives a hit and the iBTB gives a miss, the BTB will provide the target address.

Loop Predictor

The loop predictor is a two-way cache structure with 64 sets (the total size is 128 entries). The loop predictor is indexed by the branch address bits IP [9:4]. The tag bits are the IP[15:10]. Each entry in the loop predictor has two 6-bit counters. The first counter keeps the current iteration number for the allocated branch with loop behavior. The second counter keeps the maximum counter value for the allocated branch. A loop predictor hit is conditional upon a BTB hit. The regular BTB has a longer tag field providing more accurate branch identification.

Global Predictor

The global predictor is a 4-way cache structure with 512 sets (total of 2048 entries). Entries in the global predictor are two bit-saturating counters. The global predictor is indexed by a hash function — an XOR between the path information register (PIR) and the conditional branch address. The PIR bits PIR[14:6] are XOR-ed with the branch address bits IP[12:4] to provide the index for the global predictor cache. The PIR bits PIR[5:0] are XOR-ed with the conditional branch address bit IP[18:13] to provide the

tag for the global predictor cache. The global predictor overrides the prediction from the loop predictor.

Bimodal Predictor

The bimodal predictor is a flat structure with 4096 entries. An entry in the bimodal predictor is a two-bit saturating counter. The bimodal predictor is indexed by branch address bits IP[11:0]. The bimodal predictor always gives the prediction; hence, no static prediction mechanism is used.

Putting It All Together

The branch predictor unit in the Pentium M is very similar to McFarling's patent [7] and combines the best efforts in achieving the best possible prediction rate at minimal cost. Based on our reverse engineering effort, we calculate that the Intel's Pentium M (Dothan Core) predictor size is approximately 135 Kbits. Figure 1.1 shows a complete schematic of the Pentium M branch predictor unit.

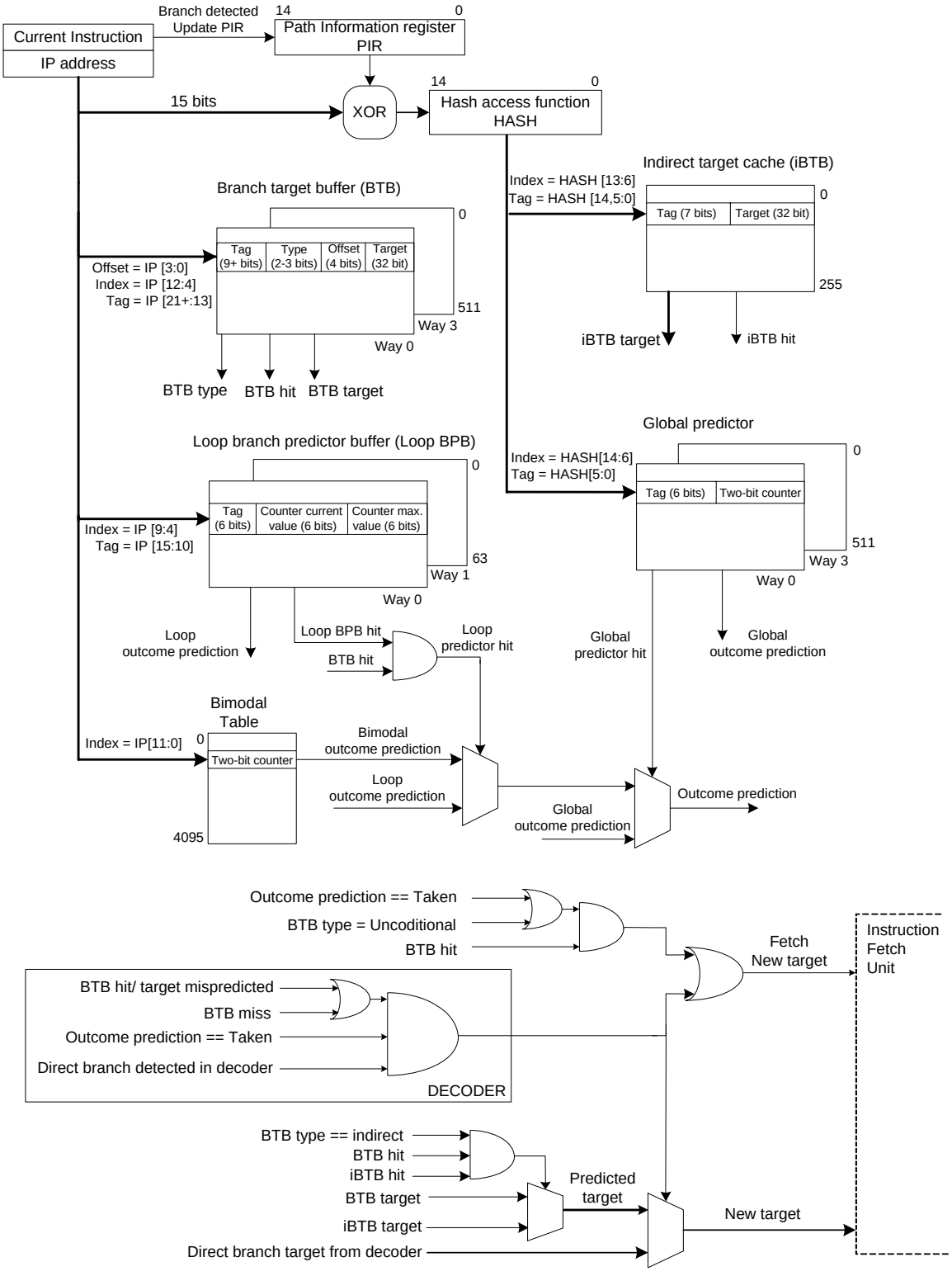

Figure 1.1 Pentium M branch predictor

CHAPTER 2

BRANCH PREDICTION TECHNIQUES

Modern branch predictor designs converge to a general organization consisting of a branch target buffer (BTB) and a branch outcome predictor. The branch outcome predictor can be coupled with or decoupled from the BTB. When the outcome predictor and the BTB are coupled, only branches that hit in the BTB are predicted, while a static prediction algorithm is used on a BTB miss. When the outcome predictor and the BTB are decoupled, all branch outcomes are predicted using the outcome predictor.

This chapter discusses pipelining and speculative execution in modern processors and surveys the historical development of branch prediction techniques, from basic concepts to the most advanced concepts. We also discuss two important issues in achieving higher accuracy of branch predictors at minimal cost, branch de-interference and branch classification or filtering.

2.1 Pipelining and the Speculative Execution

Pipelining is an execution technique where multiple instructions are overlapped in execution, taking advantage of parallelism that exists among the actions needed to

execute an instruction. Today, pipelining is a key technique used to make fast processors. Modern processors mainly operate as reduced instruction set computer (RISC) machines. A classical RISC implementation of an instruction execution encompasses five pipeline stages.

IF – Instruction Fetch. A new instruction is fetched from instruction memory and the program counter is updated.

ID – Instruction Decoding. The instruction is decoded and operands are read from the specified general purpose registers.

EX – Execution. A functional unit performs the specified operation on the operands prepared in the ID stage (arithmetic logic operation, or effective address calculation, or a branch target calculation).

MEM – Memory Access. If the instruction is a load or a store, a memory read or write is performed.

WB – Write Back. The result is written back to the register file.

Hazards are events that prevent execution of the next instruction in the pipeline stream in its designated clock cycle. There are 3 types of hazards: structural, data and control. Structural hazards arise from resource conflicts when 2 or more instructions compete for a single resource. Data hazards arise when an instruction depends on the results of a preceding instruction, and control hazards arise from instructions that change the program flow.

Control or branch hazards can cause a significant performance loss. When a branch is executed, it may change the Program Counter (PC) to its target address; it is a taken branch. When a branch falls through, it is not taken. The outcome and the target

address of a branch are typically known in execution stages of the pipeline. By the time they are known, the pipeline already fetched a certain number of following instructions. If the branch is taken, these instructions need to be flushed, and the pipeline starts fetching instruction from the branch target address. In this case, a number of clock cycles is wasted.

To handle control hazards, we can freeze the pipeline when a branch instruction is detected or we can assume predict-not-taken (treat each branch as not taken initially and correct it otherwise) or predict-taken (treat each branch as taken and correct it otherwise) approaches. An alternative approach is to rely on compiler (static) techniques – a very effective approach when branches have behavior predictable at compile time [1]. With dynamic branch prediction branches are predicted dynamically by the hardware at execution time. Almost all modern processors have a hardware resource – a branch predictor unit, responsible for handling branch prediction. The branch predictor unit is placed in the instruction fetch stages and needs to recognize an incoming branch instruction and give a correct prediction about the branch outcome (taken or not taken) and the branch target if the predicted outcome is taken. If the branch is taken and the predictor gives correct prediction, the processor front-end will start fetching instructions from the branch target address without stalling the pipeline.

Modern processors require highly accurate branch predictors, yet their complexity should be relatively small to ensure low latency and ease of verification during the design phase. The task of an architect is to carefully examine the design trade-offs and to achieve the best possible predictor accuracy with minimal cost, latency, and power consumption.

2.2 Branch Types

Branch instructions can be classified based on branch outcome to conditional (branch outcome can be taken or not taken) and unconditional (the branch is always taken). Branch instructions can be further classified based on the branch target into direct (the branch target is known in compile time) and indirect (the target is not known in compile time). Bellow we analyze various branch types and corresponding branch penalties using an example pipeline illustrated in Figure 2.1. The pipeline has 10 stages and the branch target is calculated in stage D2 for direct branches and in stage E3 for indirect branches. The branch outcome is resolved in stage E3. We assume that branch prediction is performed in the first pipeline stage.

Branch outcome penalties:

1. Conditional – Direction of the branch has to be determined during execution stages. Prediction is needed for the branch direction and branch target. Seven instructions that follow the branch are flushed from the pipeline in case of taken branch outcome misprediction (Figure 2.1).
2. Unconditional – Direction of the branch is always taken. Prediction is needed for the branch target. Three instructions that follow the branch are flushed from the pipeline in case of a branch outcome misprediction (Figure 2.1).

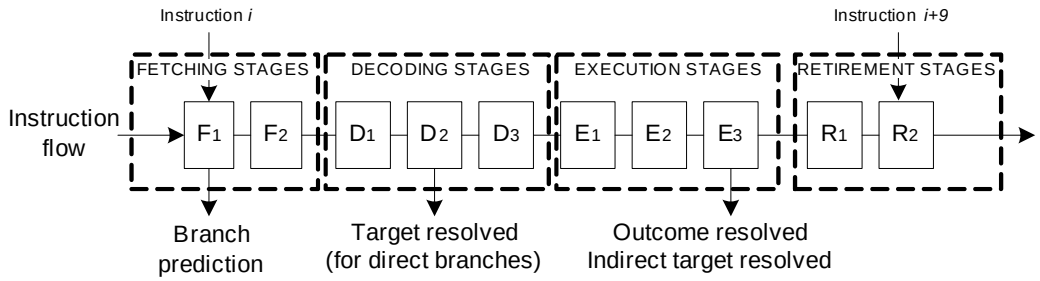

Figure 2.1 Pipeline example

Branch target address penalties:

1. Direct – Branch target address is specified within the branch instruction and it is known at compile time. Three instructions that follow the branch are flushed from the pipeline in case of a branch target misprediction (Figure 2.1).
2. Indirect – Branch target address is not known at compile time, rather it is determined during instruction execution. Seven instructions that follow the branch are flushed from the pipeline in case of a branch target misprediction (Figure 2.1).

Consequently, the prediction of the conditional branch outcome and indirect branch targets are more weighted because they incur higher penalties.

2.3 Branch Target Prediction

The target of a branch is usually predicted by a Branch Target Buffer (BTB). The BTB is a cache-like structure that keeps branch target addresses as its entries. The BTB is indexed by a portion of the branch instruction address. Each BTB entry typically includes the tag field, the valid bit, and the replacement bits for multi-way BTBs (see Figure 2.2).

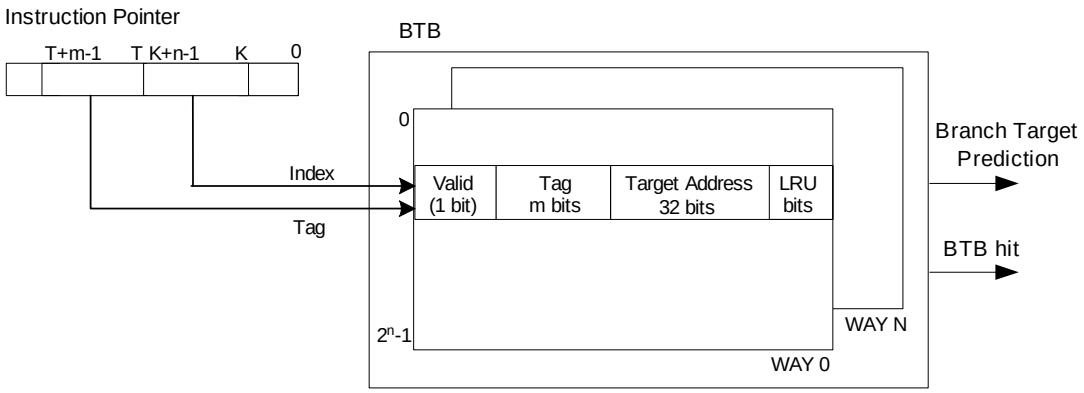

Figure 2.2 Branch-target buffer

The tag field includes another portion of the branch address or a compressed version of the remaining address bits. This reduces the size of the tag field (and consequently the number of transistors needed to implement the BTB) with minimal or no negative influence on prediction accuracy. If the branch target provided by the BTB turns out to be incorrect (mispredicted target address), the corresponding BTB entry is updated with a new branch target address after the branch is resolved.

An indirect branch can have multiple branch targets. The BTB is addressed by the branch IP address only, allowing only for one branch target address to be successfully predicted. Once, the target is changed, the BTB makes misprediction. A separate hardware structure named an indirect branch target buffer (iBTB) can be employed to handle multiple target address of indirect branches. The iBTB can override the target address coming from the BTB.

The target of an indirect branch correlates to a program path taken to reach the particular indirect branch target. In hardware, the path can be represented by a shift register containing different branch information (for example address bits, and branch

outcome). This information can be used to address the iBTB. The Pentium M processor includes an indirect branch predictor with an entry as shown in Figure 2.3. If a branch is marked as an indirect one, an iBTB lookup is performed in order to retrieve a correct target address.

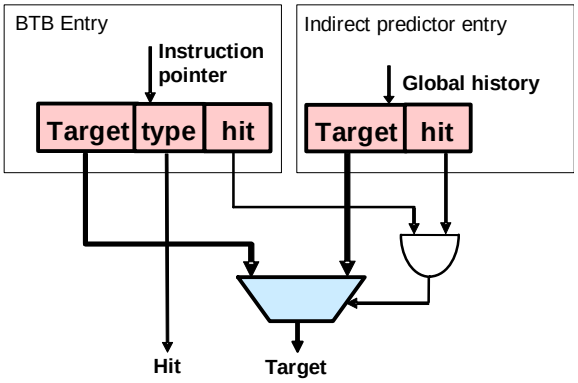

Figure 2.3 Indirect branch target address prediction in Pentium M (as presented in [2])

2.4 Static Branch Outcome Prediction

The static prediction mechanism is a simple decision on outcome based on the branch type and displacement. For example, backward branches can be statically predicted as taken, because they tend to be taken more often than not taken, for example in loops. Similarly, forward branches can be statically predicted as not taken. Behavior of program branches can be observed using program profiling, and special compiler hints or instructions can be used to enforce favorable prediction for a particular branch. However, to statically predict a branch, the prediction mechanism waits for the branch to

be decoded resulting in pipeline stalls. Static prediction techniques are sometimes used in modern processors when the dynamic prediction is not available for a given branch.

2.5 1–Bit Outcome Predictor

Branch outcomes can be dynamically predicted using a table where each table entry contains a bit that says whether the branch was recently taken or not. A table of 1-bit predictors is usually addressed by a part of the branch address. Consequently, the branch outcome prediction is the same as the outcome of the last branch that addressed the same entry.

2.6 2–Bit Outcome Predictor

The main disadvantage of the 1-bit predictor is its inability to accurately predict loop branches. Each loop branch has two mispredictions per loop: the loop exit and the first loop iteration. A 2-bit saturating counter allows the outcome to change its direction once before the prediction gets changed. The two-bit saturating counter (bimodal) is a four state finite state machine. The states are “Strongly Taken” (ST), “Weakly Taken” (WT), “Strongly Not Taken” (NT), “Weakly Not Taken” (WN). The transitions between states are controlled by the branch outcomes (T or NT). There have been many different implementations of the transitions between the states and the most frequently used one is shown in Figure 2.4. The bimodal counter’s MSB bit determines the outcome prediction. During execution of the program code, the counter is decremented or incremented according to the branch outcome.

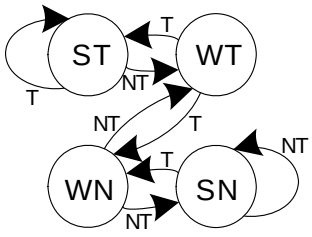

Figure 2.4 Bimodal saturating counter

2.7 Two-level Predictors

Two-level adaptive predictors are considered as a basic point for development of the modern branch prediction units. The first two-level adaptive predictor was introduced by Yeh and Patt [8]. The same authors present thorough analyses of different two-level predictor schemes and their accuracy in different applications [9] [10].

The two-level predictor has two major structures: the branch history register (BHR) and the branch history table (BHT). The branch history register is a shift register containing the outcomes of the recent program branches. A branch predictor may use an array of individual BHR registers, each tracking a local history of a branch; this array of BHRs is known as a history register table (HRT). The BHR is used as an index to the BHT to select a bimodal entry that will provide the prediction. Depending on the number of BHR registers, three main classes of 2-level predictors are developed as follows.

- Per-address schemes (PA) select an appropriate BHR in the HRT table by a part of the branch address. If *N* address bits are used to address the HRT, there are 2^N BHR registers.

- Per-set schemes (SA) select an appropriate BHR in the HRT table by a set address obtained from the branch IP address. Consequently, the number of BHRs is smaller than in the PA schemes.
- Global schemes (GA) use one BHR register for all program branches.

Each of the three classes of schemes can be further divided into sub-schemes according to the number of BHT tables. Figure 2.5 shows three sub-schemes of the PA scheme. Figure 2.5(A) shows the Per-Address Global scheme (PAg). This scheme uses one BHT table. Figure 2.5(B) shows the Per-Address address scheme (PAp). This scheme uses 2^k BHT tables, where k is the number of address bits used to access the BHT. Figure 2.5(C) shows the Per-Address per-set scheme (PAs). This scheme divides a number of address bits used to access the BHT tables into S sets to lower the number of BHT tables.

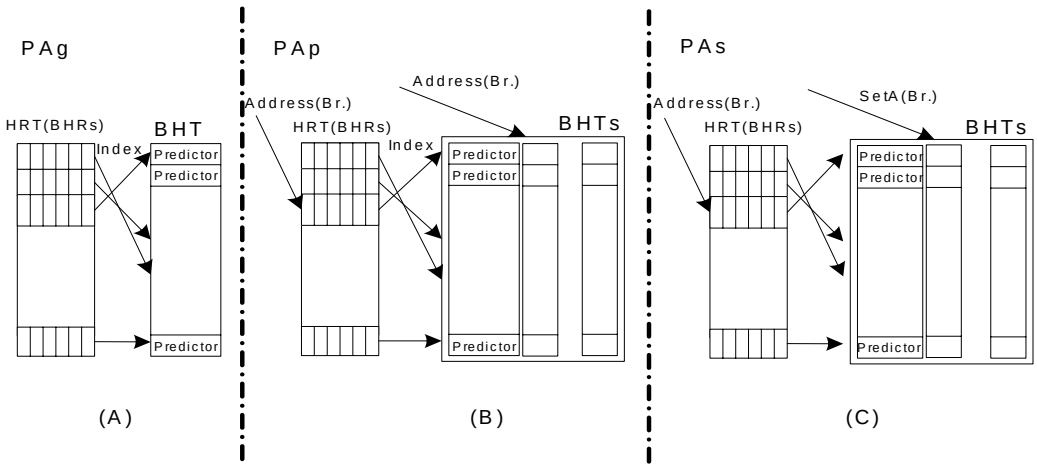

Figure 2.5 PA two-level schemes as presented in [8]

Figure 2.6 shows the Global-Address global scheme (GAg). The GAg scheme uses one BHR and one BHT table. The GAg predictor is a basis for the majority of modern branch prediction units. Predictors described further are all two-level predictors with one BHR and one BHT, although the BHR may be combined with other branch information to access the BHT and BHT and may be divided into several tables to look closely to GAs scheme.

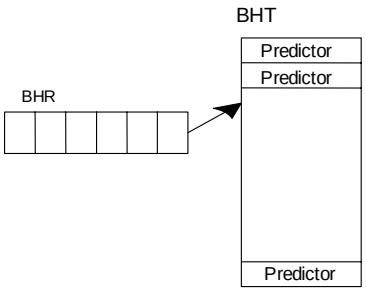

Figure 2.6 GAg two-level scheme

2.8 GShare Predictor

In the GAg scheme, indexing in the BHT is based solely on the BHR, which is affected by the last *N* outcomes of the program branches. McFarling proposed a so called *GShare* predictor [11]. The original GAg scheme is changed and the BHR is XOR-ed with a part of the branch address to create a hash function to access the BHT. This approach proved to be more accurate than the other existing schemes.

With the introduction of the *GShare*, two-level predictors become focused on the GAg scheme rather than on the PA schemes. *GShare* is still widely used in commercial processors and many processor simulators include the *GShare* as the outcome predictor.

2.9 Hybrid Predictors

McFarling [11] introduced a hybrid predictor. The bimodal predictor augments the *Gshare* predictor (see Figure 2.7). The bimodal predictor is used for highly biased branches where constant decision on NT or T outcome is enough for accurate prediction. *Chooser* is logic needed to make the final prediction from two outcome predictions coming from the bimodal and the *GShare* predictor. This logic can be as simple as a 2-bit saturating counter table (the same one used for bimodal table). The chooser counter's MSB bit selects the final prediction between the two provided.

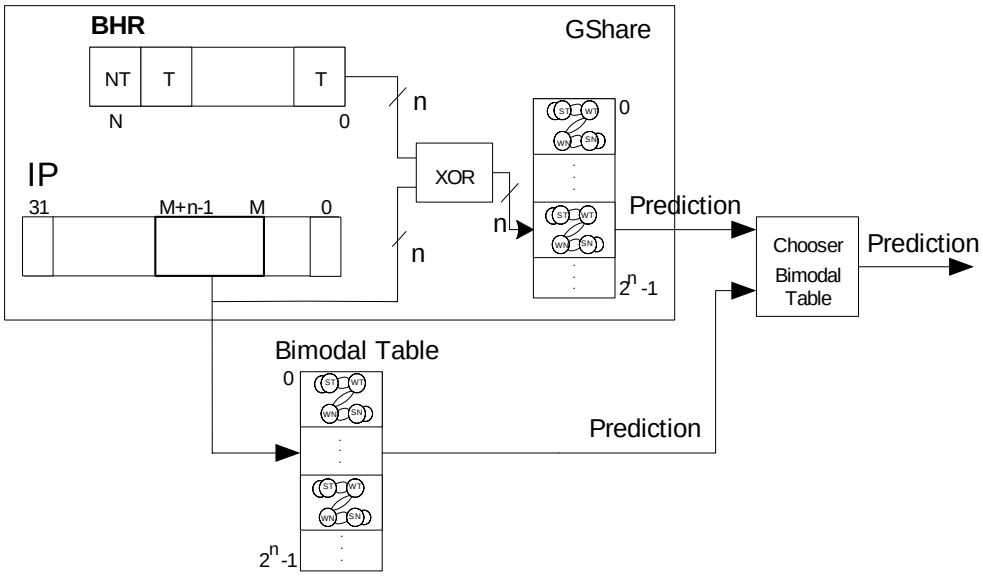

Figure 2.7 Hybrid predictor

The tournament predictor is a hybrid predictor, where *GShare* (the global predictor) is working in parallel with a local predictor instead of the bimodal predictor [1]. The tournament predictors are accurate since they cover two sides of branch correlations (local and global), but they are not cost-effective, because local and global correlations are just partially orthogonal. This leads to having redundant information in the local and global predictors.

2.10 De-interference Techniques

Negative interferences occur when two branches with opposite outcomes compete for the same predictor's entry. One or both outcomes are mispredicted. Interference misses are shown to be a more important limitation in achieving higher prediction rates than the predictor size. Here we give a brief description of 3 of most influential de-interference techniques: (a) Agree predictor, (b) Bi-mode predictor, and (c) Skewed predictor.

2.10.1 Agree Predictor

The Agree predictor [12] is one of the first predictors to try and cope with negative interferences. The Agree predictor introduces a bias bit dedicated for each branch. It assumes that two branches with opposite outcomes that compete for the same predictor's entry may have correct bias bit. The bias bit is usually allocated in the BTB and reflects the first occurrence of a branch that is usually a dominant one during the program execution.

Instead of providing prediction for a particular branch, the saturating counter provides information on whether the predictor agrees or disagrees with the bias bit. If we

assume that two branches with opposite outcomes have correct bias bits, the global predictor may provide the same information for both branches, e.g., “agree,” and both branches will be correctly predicted. The Agree predictor suffers from the bias bit implementation issues and moreover, in modern predictors, the bias bit is not considered accurate enough to have prediction relying on it.

2.10.2 Bi-mode Predictor

The Bi-mode predictor [13] introduces two BHT tables, a not taken table (NT.BHT) and a taken table (T.BHT). Each table stores only one type of outcome. A bimodal table is used to select between T.BHT and NT.BHT. An index hash function is used to access the selected table (see Figure 2.8).

The Bi-mode predictor translates a significant amount of negative interferences into neutral interferences. As long as the bimodal table selects the correct table, negative interferences are not possible.

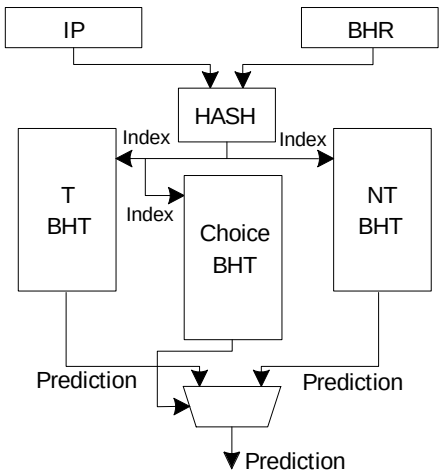

Figure 2.8 Bi-mode predictor

2.10.3 Skewed Predictor

The skewed predictor [14] resolves a source of collision by trying to increase associativity of the BHT. Since a tagged BHT is too expensive to implement, a special function named Skew function emulates a tag. The skewed branch predictor splits the PHT into three even banks and hashes each index to a 2-bit saturating counter in each bank using a unique hashing function per bank (f1, f2 and f3). The prediction is made according to a majority vote among the three banks. If the prediction is incorrect, all three banks are updated. If the prediction is correct, only the banks that made a correct prediction will be updated. The reasoning behind partial updating is that if a bank gives a misprediction while the other two give correct predictions, the bank with the misprediction probably holds information which belongs to a different branch. In order to maintain the accuracy of the other branch, this bank is not updated. The skewed predictor stores each branch outcome in two or three banks. This redundancy of 1/3 to 2/3 of the PHT size creates capacity aliasing but eliminates much more conflict aliasing, resulting in a lower misprediction rate.

2.11 Filtering and Branch Classification

De-interference methods use a global predictor as the base predictor and then augment it to improve its accuracy. Filtering methods try to cope with redundancy of allocated data in the hybrid predictor as well as with enabling smaller predictors. Filtering relies on classifying branches into different groups, where branches from each group are guided into a separate predictor structure tailored for that class of branches. The majority of the program branches are *always taken* or *always not taken* branches and

are easily predictable by the less costly bimodal predictor. A significant number of branches are highly locally correlated such as loop branches.

2.11.1 Counter and Bias-bit Based Filtering

Chang *et al.*[15] present a filtering technique that uses a counter in the BTB as well as a bias bit. The bias bit is set by the first branch outcome and is toggled every time the branch changes its outcome. The counter is used as a bias bit confidence value. The counter is incremented every time a branch outcome complies with the bias bit. If the counter is in saturation, a bias bit is used to predict the branch and the global predictor is not used nor updated. If the branch is mispredicted, the counter is reset. This way, the contention in the global predictor is reduced.

This filter technique involves tight relations with the BTB, making it difficult to implement. Moreover, by allowing the less accurate bias bit to predict the outcome until the counter is in saturation makes this approach less suitable for modern branch predictors.

2.11.2 YAGS Predictor

The YAGS predictor combines filtering and de-interference techniques [16]. It is based on the Bi-Mode predictor, but the NT.BHT and T.BHT tables are tagged in order to achieve a better filtering. Both tables can allocate only the branches with outcomes that do not comply with a bimodal predictor. The bimodal table is used as the selector between NT.BHT and T.BHT as well as the outcome predictor. If the bimodal predictor is able to predict the branch correctly, NT.BHT and T.BHT tables are not updated.

2.11.3 Serial-BLG Predictor

McFarling's Serial-BLG predictor [7] uses multiple predictor stages, each of which passes its prediction onto the next stage. Each stage overrides the prediction from the previous stage only if it can refine the current prediction. The serial predictor significantly reduces size of each stage; each stage handles only those branches that are not predicted accurately by the previous stages. All predictor stages allocate an entry for a particular branch as in the parallel hybrid predictor, but the stage does not update the replacement bits if it was unable to offer a better prediction than the previous stage. This way redundant entry in later stages is going to be replaced sooner.

The serial-BLG predictor implements three stages (see Figure 2.9) and handles four different classes of branches: (a) biased branches which are easily predictable with a bimodal predictor; (b) locally correlated branches which are predictable by a local predictor; (c) loop branches, predictable by a local loop predictor; and (d) all other branches that are handled by a global predictor.

The serial-BLG successfully copes with negative interferences and offers an excellent way for branch classification. Three predictors use relatively separated stages where each can be designed separately with well defined interfaces that connect them.

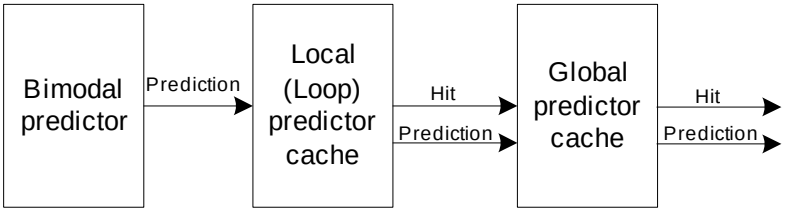

Figure 2.9 Serial-BLG predictor organization

2.11.4 Loop Predictor

The loop predictor is a cache structure used to recognize and predict loop branches. The loop predictor is trained to recognize the maximum loop count. It provides the opposite prediction when the current iteration counter reaches the maximum loop count. The Intel Pentium M's loop branch predictor unit uses a 2-counter scheme as shown in Figure 2.10. The first counter tracks current loop branch iteration and the second counter keeps the maximum loop value determined during previous loop runs. When counters match, a prediction provided by the loop predictor flips from the default prediction.

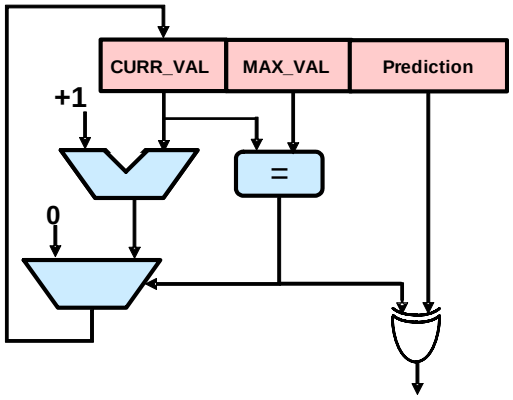

Figure 2.10 Loop predictor counters in Pentium M (as presented in [2])

2.12 Perceptron

Perceptrons are learning elements introduced in 60's in theory of neural networks. Neural branch prediction is first proposed by Vintan [17]. Vintan considers branch prediction as a particular problem in a broader class of pattern recognition problems that

can be solved by neural networks. The neural predictor has ability to exploit long histories at the cost of a linear resource growth. Classical predictors impose an exponential resource growth. Further development of the perceptron predictor was taken by D. Jimenez. Jimenez [18] presents a hybrid predictor similar to the *GShare* predictor [11], but with significant improvements in accuracy. The main disadvantage of the perceptron predictor is its high latency. Fast-path neural predictor presented by the same author [19] is a predictor with latency comparable to the state-of-the-art predictors used in industry. However, the perceptron prediction relies on arithmetic functions (add) that increase the predictor's latency.

Inputs to the perceptron are branch outcome histories just like in the two-level adaptive branch prediction. The output of the perceptron is non-negative (branch predicted taken) and negative (branch predicted not taken). One perceptron element is shown in Figure 2.11.

The inputs (x's) come from the branch history and they are equal to -1 or +1. The output (y) is a product of x's and w's. A training mechanism finds correlations between the history and the outcome (*w factors*). The bias weight, w_0, is proportional to the probability that the branch is taken.

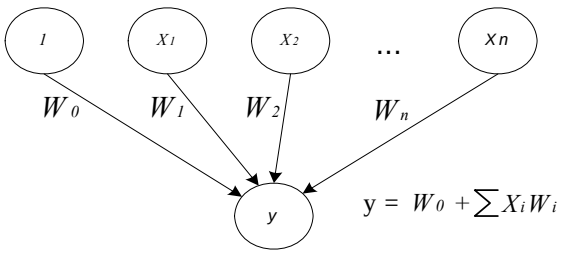

Figure 2.11 Perceptron basic element

2.13 Confidence Value

Confidence is the measure of accuracy of the branch prediction. A basic confidence value is for example Strongly-Taken and Weakly-Taken branch prediction states in the bimodal predictor. The hybrid predictor's chooser may use the confidence value in selecting the prediction from a predictor with the highest confidence value. It can also be used in performance-power trade-offs. For example, we may opt not to speculatively pursue a program path as indicated by the prediction if the confidence in that prediction is low. Even though this approach may result in performance degradation, it could be beneficial for power consumption.

Jacobsen *et al*. [20] propose a hardware mechanism that partitions conditional branch predictions into two sets: those that are predominantly accurate and those that are less accurate. The objective is to concentrate as many of the mispredictions as practical into a relatively small set of low confidence dynamic branches. Confidence value can be used to allow threads, predicted with a high confidence, to have priority over those with low confidence. This research indicates that a relatively simple confidence mechanism can isolate 89% of the mispredictions into a set containing 20% of the dynamic branches.

CHAPTER 3

INDUSTRIAL IMPEMENTATIONS OF THE BRANCH PREDICTORS

This section presents an overview of known branch predictors' implementations in commercial processors, with an emphasis on the branch predictor unit found in the Intel's P6 architecture.

3.1 Branch Prediction Unit in Intel's P6 Architecture

Several details about the branch predictor unit found in Intel's P6 architecture have emerged. Some of these details are implied in the Intel's software optimization manual for P6 architecture [21]; for example, the manual indicates the existence of a local predictor with a 4-bit long branch history register. More details have emerged from an early reverse engineering effort by Milenkovic *et al*. [3]. They found the following: The P6 architecture has a BTB organized in a cache structure with 128 sets and 4 ways (the total number of entries is 512). Address bits IP[10:4] are used as thr BTB index. The outcome predictor includes only a local predictor with 4-bit long BHR with no global

component. The findings from [3] are later confirmed by chief architects of P6 architecture in the book by Shen and Lipasti [22].

More details about the P6 branch predictor unit emerge in an Intel patent [23]. This patent presents a detailed BTB organization identical to the one found in [22]. The patent also describes the place of the branch predictor unit (BPU) in the pipeline and its interaction with other units in the processor's front-end (see Figure 3.1).

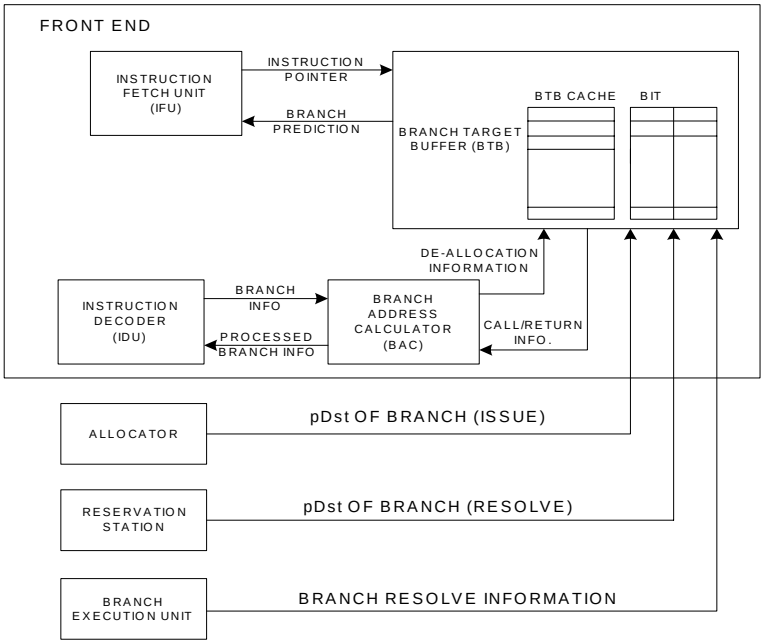

Figure 3.1 Pentium P6 Front-End and its branch predictor unit (as presented in [23])

The instruction fetch unit (IFU) sends the instruction pointer IP to the BTB. The BTB uses a portion of the instruction pointer to select a set in the BTB. The tag field of the incoming instruction pointer is compared to the tag fields of 4 entries in the selected set (see Figure 3.2). If a matching entry is found, the offset field is used to determine

whether hit will be reported or not. The offset field is used to select a right entry if multiple entries in the selected BTB set have matching tags. If there are multiple hits in the BTB, an entry with the lowest offset that is still larger than or equal to the instruction pointer is selected. The BTB tag is 9 bit long and 7 bits are obtained by compressing address bits IP[29:11]. Two most significant tag bits are address bits IP[31:30] to allow better detection of the jumps to the operating system service routines. If the incoming tag does not match, a new entry is allocated. The replacement policy bits (LRR bits) select an entry for allocation.

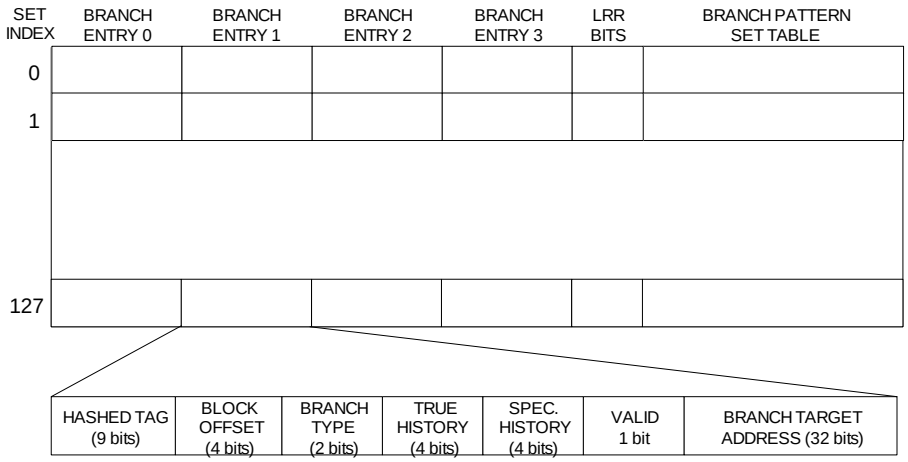

Figure 3.2 Organization of the BTB and layout of one BTB entry (as presented in [23])

The BAC unit provides the target address to be stored in the selected BTB entry in case of a BTB miss or in the case that the provided target address is incorrect (BTB hit, but the target address is mispredicted).

In addition, if we have a BTB hit that comes from a non-branch instruction (a bogus branch), the selected BTB entry is invalidated. This early BTB update is desirable due to superscalar execution. If the branch has a mispredicted outcome, the branch history table needs to be updated with the new outcome.

Multiple branches exist at any point in different pipeline stages. An incoming branch may not see its prehistory retired yet, and therefore the outcome history for that branch may have entries related to more distant branches. This is a reason why the BTB along with the outcome history employs a speculative outcome history. If it is accurate, a speculative bit will be set to inform internal logic to take a speculative history as a valid one. Otherwise a real branch history (outcome of retired instructions) is used.

Figure 3.3 shows the organization of instruction fetch lines in Intel's P6 architecture. If a branch instruction spans multiple 16-byte lines (as illustrated in Figure 3.3), the BTB will handle that branch (tag, index, and offset) based on the address of the last byte of that instruction. Consequently, the offset field will be 1h, and index field will be 2h.

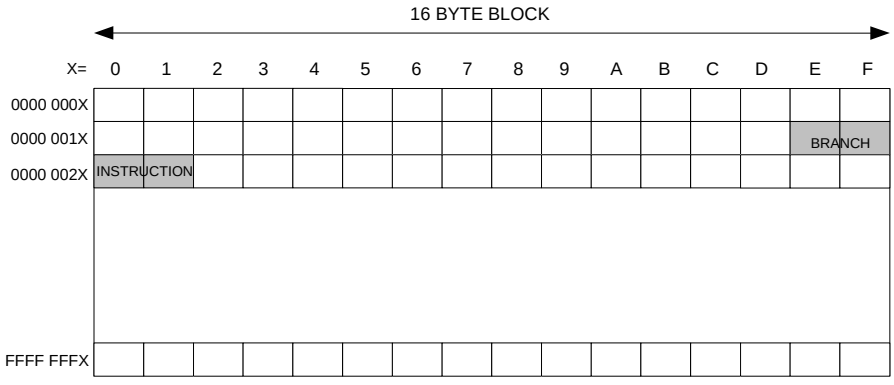

Figure 3.3 Fetch line in P6 architecture (as presented in [23])

3.2 Branch Prediction Unit in Intel's NetBurst Architecture

Starting with the Pentium 4, Intel claims significant improvements in the branch prediction accuracy [24]. Milenkovic *et al.* [3] showed that NetBurst architecture (Northwood core) uses a global predictor decoupled from the BTB. The BTB is a 4-way structure organized in 1024 sets (the total number of entries is 4096). Address bits IP[13:4] are used as the BTB index. The global predictor is addressed by a single BHR which is 16-bit long. The internal organization of the global predictor is unknown, but it is indicated that it has 4096 entries. Intel specifically refers to the global predictor as a proven to be better than the *GShare* predictor. NetBurst architecture uses static prediction mechanism if the branch is not found in the BTB.

3.3 Branch Prediction Unit in Intel's Pentium M

Intel technology journal [2] describes the Pentium M branch prediction as the same one used in the NetBurst architecture with addition of a loop and an indirect predictor. The loop predictor uses two counters as described in Section 2.11.4. The indirect target predictor is presented as a separate entry. The indirect branch prediction from the indirect predictor is conditional upon the BTB hit. If the indirect predictor misses, the BTB provides the target prediction (see Figure 2.3). The Intel technology journal [25] indicates that the outcome predictor is designed from scratch. The optimization manual indicates that all branches are predicted dynamically, which likely means that the outcome predictor includes a bimodal table, in addition to a global predictor. This is a starting assumption in our investigation.

3.4 AMD K6 and K7

The AMD K6 line of microprocessors features a highly accurate 8K-entry GAs branch predictor; however, the K7 AMD Athlon microprocessor has scaled-back to the 2K-entry predictor [26]. The predictor does include a few simple but proprietary enhancements to improve its behavior in important special cases. It seems that power consumption constraints and branch predictor latency drive designs toward smaller, but more sophisticated branch predictors with support for de-interference and classification. Athlon K7 predicts branch target addresses using a 2K-entry branch target address cache.

3.5 Alpha 21264

Alpha 21264 processor implements a tournament predictor consisted of one GAg predictor and one PAg predictor [27]. The GAg predictor has a 12-bit BHR and a 4K-entry BHT with 2 bit counters. The PAg has a 10 bit BHR and 1K-entry with 3-bit counters.

3.6 Sun UltraSPARC-IIIi

Sun UltraSPARC-IIIi processor uses 16K-entry GShare predictor with two bit counters [22]. Bimodal predictor is also implemented and it XOR address bits with global history register (except 3 lower order bits) to reduce aliasing.

CHAPTER 4

EXPERIMENTAL ENVIRONMENT

This section describes an approach used in reverse engineering of branch predictor units. We give a brief description of performance monitoring registers and a list of branch-related events that can be monitored in Pentium M processors. We also give a short description of Intel's VTune, a performance-tuning tool we used to run microbenchmarks and perform measurements of interest.

4.1 Reverse Engineering Flow

Figure 4.1 shows a generalized experiment flow of the reverse engineering process. In Step 1, a reverse engineer makes a hypothesis on a particular predictor structure or a mechanism and assesses a testing space that will cover all design structure details. In Step 2, a microbenchmark (or a series of microbenchmarks) is developed in C and/or assembly language. The microbenchmark has to (a) identify and isolate parameters/mechanisms for hypothesis testing (b) amplify the parameters/mechanisms of interests so they can be easily measured or observed, and (c) mask out effects of all other

microarchitectural parameters/mechanisms. The reverse engineer also needs to select a list of observable events on the given architecture of interest for the mechanisms/parameters tested in the hypothesis. This step ends with the reverse engineer making a simulation by hand and establishing expectations for the given hypothesis.

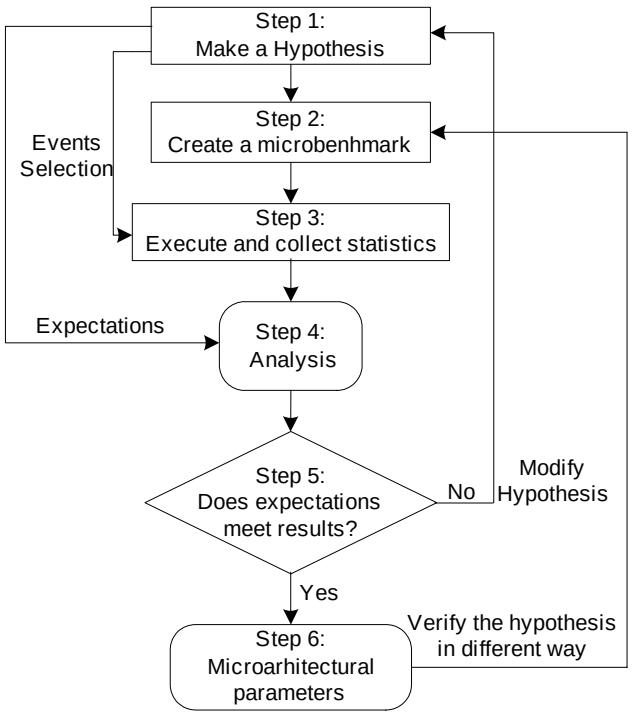

Figure 4.1 Reverse engineering flow

In Step 3, microbenchmarks are executed and microarchitectural events are collected using a performance monitoring tool (Intel's VTune in this thesis). In Step 4, collected microarchitectural events are analyzed and compared with the expectations made in Step 1. In Step 5, we decide whether the hypothesis is confirmed or not. If the

hypothesis fails, we modify the hypothesis and accordingly the microbenchmark. In Step 6, after the hypothesis is possibly confirmed, the reverse engineer can try to find a slightly different experiment to re-confirm the hypothesis (for example, a number of different experiments can be used to find the number of ways in a set-associative cache structure).

4.2 Performance Monitoring Registers

Modern processors include performance monitoring counters and corresponding logic that allows programmers to specify one or more microarchitectural events that can be monitored (measured) in real-time. For example, programmers can measure the number of clock cycles a program or a part of the program takes to execute, they can monitor the number of cache misses, the number of branch instructions, and the number of mispredicted branches, to name just a few hardware events. Insights gained in this process can be used for manual or guided program optimization. For example, a significant number of data misses can be reduced by changing the data layout in memory or by dividing program data sets into smaller pieces that better fit the memory hierarchy in a given processor. When a performance monitoring counter overflows, the corresponding logic triggers an exception, so the exception service routine can keep the track of the number of events.

4.3 Branch related microarchitectural events

Here we describe branch-related events specific for the Pentium M processors which are used in our reverse engineering effort. Table 4.1 shows the branch related events in Pentium M and the description provided from the Intel's VTune.

Table 4.1 Branch related microarchitectural events in Pentium M processor

Event name	Event description
Branch Instructions Decoded	Event indicates the number of branch instructions decoded.
Branch Instructions Executed	Event counts all executed branches (not necessarily retired). This includes only macro instructions and not micro branches.
Branch Instructions Retired	Event indicates all retired branches. This includes only macro instructions and not micro branches.
Branch Mispredictions Retired	Event measures all retired executed to completion branch mispredictions. This includes: Branches that were incorrectly predicted as "taken" and were discovered to be "not taken" only after retirement, Branches that were incorrectly predicted as not taken and were discovered to be taken after retirement.
BACLEARS Asserted	Event indicates branch mispredictions, where the branch decoder decides to make a branch prediction because the BTB did not, or, rarely, tries to override the BTB's prediction. Each branch misprediction of this type costs approximately 5 cycles of instruction fetches. The effect on total execution time depends on the surrounding code
Mispredicted Branch Instructions (Mispredicted at Decoding)	Event counts the number of branch instructions that were mispredicted at decoding.
Mispredicted Branch Instructions (Mispredicted at Execution)	Event counts the number of branch instruction mispredicted at execution.
Mispredicted Conditional Branch Instructions Executed	Event counts the number of mispredicted conditional branch instructions that were executed.
Mispredicted Indirect Branch Instructions Executed	Event counts the number of mispredicted indirect branch instructions that were executed.
Taken Branch Mispredictions	Event, the VTune Performance Analyzer counts those branches that were incorrectly predicted as taken and were discovered to be not taken only after retirement.
Taken Branch Retired	Event, the VTune Performance Analyzer counts the number of taken branches that are retired or completed.

It should be noted that the available literature often gives very vague description of certain events, so their exact nature is somewhat open for interpretation. For example, Mispredicted Branch Instructions (Mispredicted at Decoding) is described as “Event counts the number of branch instructions that were mispredicted at decoding.” This makes the event difficult to associate with the parameters of our interest – it could be a BTB miss (the tag does not match) or it could a BTB hit with mispredicted branch target.

The following branch related events are of great interest for the effort presented in this thesis: (a) Mispredicted Branch Instructions (Mispredicted at Decoding) MBI_DEC, (b) Mispredicted Branch Instructions (Mispredicted at Execution) MBI_EXEC, and (c) the Mispredicted Indirect Branch Instructions Executed (MBIE).

Our experiments indicate that the event MBI_DEC counts both the number of BTB misses and the number of BTB target mispredictions when the outcome predictor predicts branches as taken. If the MBI_DEC event count is high and the MBI_EXEC event is low, we consider this as an indication that after decoding, branch target was known and the misprediction does not propagate further in the pipeline to execution stages. We relate the MBI_EXEC event to the branch outcome misprediction. The MIBIE event in experiments related to the indirect branch predictor in the Pentium M and can also expedite the experimental flow used in determining organization of the global predictor.

4.4 VTune - Tool for Collection and Selecting Hardware Events

VTune is a performance tuning environment for Windows and Linux developers from Intel [28]. VTune provides more capabilities than just an observation of the hardware events. VTune monitors the performance of all active software and is able to

identify “HotSpots” or bottlenecks in a program and analyze program performance as it executes on an Intel microprocessor platform. VTune can examine each instruction and uncover problems at machine code level including optimization of the code using context-sensitive on-line tuning suggestions.

We use the VTune just as a tool to collect microarchitectural events related to branch instructions in order to determine the branch predictor unit hardware organization.

CHAPTER 5

MICROBENCHMARKS FOR THE REVERSE ENGINEERING OF THE BRANCH TARGET BUFFER

5.1 Objectives

The goal of this section is to develop an experimental flow and a set of microbenchmarks that will help us determine the size and organization of the branch target buffer. We expect the branch target buffer to have a cache-like structure and we want to determine the branch target buffer cache parameters (size, sets, ways, index, tag, replacement policy).

5.2 Contributions

We developed an experimental flow and a set of microbenchmarks for determining organization of the branch target buffer. The experimental flow and microbenchmarks applied on a Pentium M processor provide the following insights.

1) The BTB is decoupled from the outcome predictor and always not taken branches are not allocated in the BTB.
2) The BTB number of ways is a 4-way structure with 512 sets.
3) The BTB index bits are instructions pointer bits [12:4].
4) The BTB tag bits are instructions pointer bits [21:13].
5) The BTB replacement policy is based on the LRU policy but this policy is reinforced only on branches that at least once hit in the BTB.
6) BTB can allocate multiple branches with the identical tag in the same set. Each BTB entry includes offset field to achieve this functionality. The offset field is equal to the last four bits of the branch instruction pointer. Therefore, multiple hits are possible. Among all BTB hits, the offset mechanism selects the entry with the smallest offset yet not smaller that the instruction pointer.
   False hit in the BTB (a bogus branch) causes eviction of the whole BTB set.

5.3 Background

The BTB is typically implemented as a set-associative cache structure (Figure 5.1), with each entry storing critical information about the branch: the full branch target address, the branch type (for example, direct or indirect), the branch offset, and the tag. The BTB can be indexed by a portion of the branch address. An alternative implementation may combine a portion of the branch address with a path history register (e.g., the index is an exclusive-or function of a portion of the branch address and the path history register). The tag field can be another portion of the branch address or a compressed version of remaining 21 address bits, rather than a full address tag.

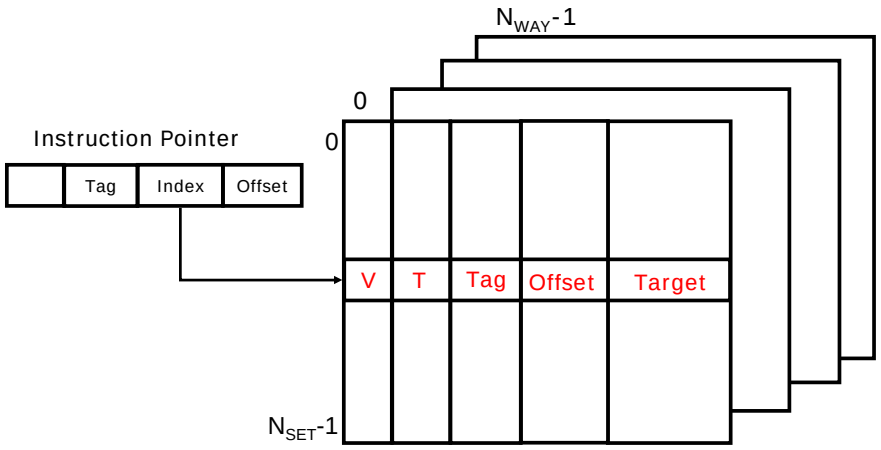

Figure 5.1 Branch Target Buffer

In reverse engineering of a BTB the following information is sought:

1) BTB organization;

BTB size (number of entries)

Number of ways

Number of sets

INDEX bits, TAG bits

2) Replacement policy (Way replacement)

3) Allocation/Eviction policy

Before going into details about branch target buffer experiments, we shall introduce several assumptions related to Intel architectures. The instruction fetch unit is fetching 16-byte instructions lines; branch instructions that are never taken do not allocate an entry in the BTB; and the branch address of an instruction that spans two 16-byte instruction lines is determined by the address of the last byte in that instruction and it belongs to the second 16-byte line. All these assumptions could be easily verified with a set of specific microbenchmarks that can be applied in a later stage once more details become available.

5.4 BTB Organization Tests

In determining BTB organization we start with an experimental flow first introduced by Milenkovic *et al.* [3]. A so called BTB-capacity test stresses the BTB structure trying to find the maximum number of branches that can fit in the BTB. A so called BTB-set test stresses the BTB structure trying to find the maximum number of branches that can fit in a single BTB set.

5.4.1 BTB-capacity Tests

A number of conditional taken branches (or unconditional direct or indirect taken branches), *B*, are placed at equidistant addresses in memory with distance *D* (see Figure 5.2). By varying the parameters B and D, we can, under certain conditions, determine the BTB size. For example, let us assume a BTB with 512 entries, organized in a 4-way cache structure, where branch address bits IP [10:4] are used as the BTB index. By varying B=128, 256, 512, 1024, 2048 and D=2, 4, 8, 16, 32, 64 and measuring the misprediction rate for the branches mispredicted at decoding, we expect the results as shown in Figure 5.3. When the number of branches is equal to the number of entries in the BTB, there exists 3 "fitting" distances *D* (*D=4, 8, 16*) that result in a very low misprediction rate. When the number of branches exceeds the number of entries in the BTB, the mispredictions rate is high (close to 100%). By lowering the number of branches, the number of fitting distances increases; for example, we have 4 fitting distances for *B=256* (*D=4, 8, 16, 32*) and 5 fitting distances (*D=4, 8, 16, 32, 64*) for *B=128*.

```
Address     Code
            void main(void) {
               int long unsigned i;
               int long unsigned liter = 1000000;
               for(i=0;i<liter;i++){
               _asm {
                        // dummy non-branch instructions
@A                      jle l0     // always taken
                        // dummy non-branch instructions
@A+D          l0:       jle l1     // always taken
                        // dummy non-branch instructions
@A+2D         l1:       jle l2
                        ...
@A+4094D      l4093:    jle l4094  // always taken
                        // dummy non-branch instructions
               }
@A+4095D      }                    // always taken
            }
```

Figure 5.2 BTB-capacity microbenchmark example for B=4096

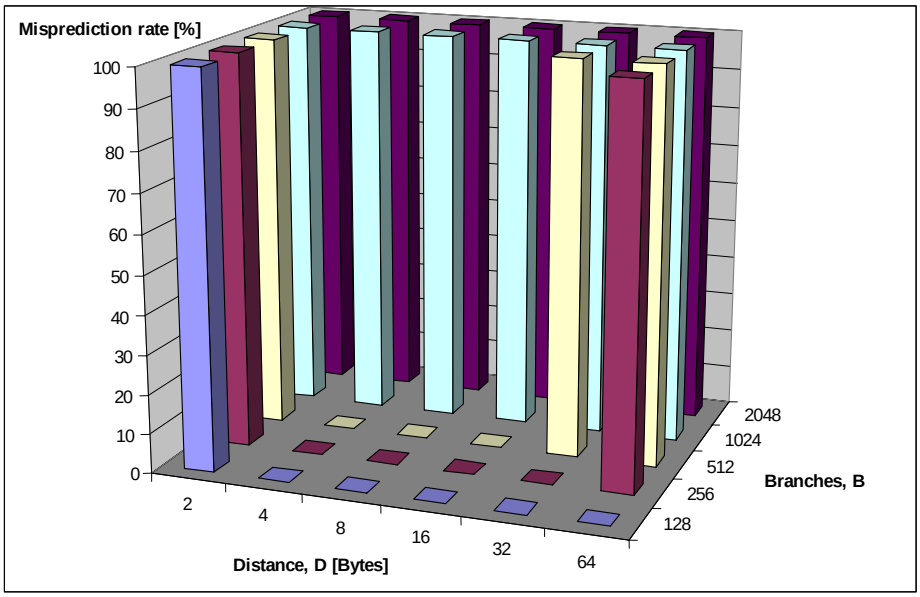

Figure 5.3 Expected misprediction rate as a function of the number of branches B, and the distance between branches, *D* for a BTB organized in 128x4 cache structure

In the general case, when the BTB-capacity test with $B = N_{BTB}$ spy branches gives *m* "fitting" distances, the number of ways in the BTB is $N_{BTBWAYS} = 2^{m-1}$ and the index bits used are IP[i+j-m:i], where the maximum fitting distance $D_{MAX} = 2^i$, and $j=\log_2 N_{BTB}$. A generalized equation used in the BTB-capacity test analysis is shown in Equation (5.1)

$$MB@dec \approx \left\{ \begin{matrix} 0,\ B \le N_{BTB}\ and\ i-m \le k \le i+j-l \\ 100\%,\ B > N_{BTB} \end{matrix} \right\}, B = 2^l, D = 2^k. \qquad (5.1)$$

It should be noted that this reasoning can be applied only if certain conditions are satisfied: (a) a portion of the branch address is used as the BTB index, (b) address bits above the index field are used as the tag in the BTB, (c) branch address offset bits are stored in the BTB, and (d) replacement policy is the round-robin, the least recently used or one of its derivatives.

We measure the number of branches mispredicted at decoding (MBI_DEC). Figure 5.4 shows the misprediction rate calculated as the MBI_DEC divided with the number of spy branches as a function of B (B=512–4096) and D (D =2–128).

When B=1024, there exists 3 values for D (D = 4, 8, 16) with no mispredictions. This result indicates a BTB organized as a 4-way cache structure with 1024 entries.

When B=2048, there exists only one value D (D=4) with no mispredictions. If we assume that the BTB size is 1024 entries, the expected misprediction rate should be high for all distances with B=2048. One possible explanation for this anomaly is that the branch predictor unit does not allocate an entry in the BTB for every branch instruction because three out of four branches have targets in the same 16-byte fetch line. This assumption does not hold because a similar reasoning should apply when B=4096 and

D=4 as well as for all experiments with D=2. However, the misprediction rate in those cases is relatively high, though not close to 100%. Another possible explanation is that the BTB is a 2-way structure with 1024 entries. However, this assumption does not hold either, because in that case we should see a high misprediction rate for B=1024 and D=4 (4 branches with the same tag would miss in a 2-way structure). Rather, we see a low misprediction rate. Yet another possible explanation is that the BTB is a 4-way structure with 2048 entries, but the BTB allocation policy and/or replacement policy are a source of "unexpected" behavior when B=2048 and D=8 and D=16 (we observe relatively high misprediction rate of 60% for D=8, though we expect no mispredictions under these conditions). Consequently, we conclude that the BTB-capacity tests alone is not enough to decisively determine the size and organization of the BTB and more experiments that will stress a BTB set and BTB allocation/replacement policy are required.

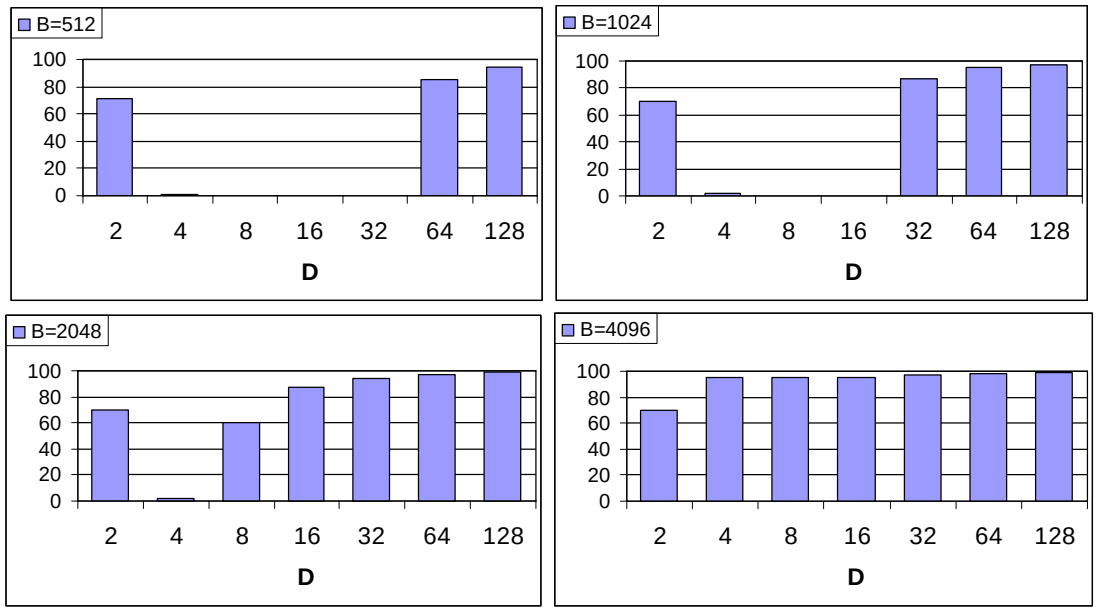

Figure 5.4 BTB-capacity test results for B=512–4096 and D=2–128

5.4.2 BTB-set Test

Instead of finding the number of branches B that fills the whole BTB, an alternative experiment finds the number of branches that fills a BTB set, and a distance D_S such that those branches map into the same set (see Figure 5.5). These types of experiments are called BTB-set tests. We analyze the misprediction rate as a function of the distance D_S and the number of branches B. When the distance between branches exceeds $2^{MSB(Index)}$ bytes ($D_S > 2^{MSB(Index)}$), the curve knee point for the misprediction rate is when B becomes equal to the (*number of ways* +1), and it does not depend on D_S. When $D_S \leq 2^{MSB(Index)}$, the curve knee point for the misprediction rate is a function of both B and D_S. When $D_S > 2^{MSB(Tag)}$, the curve knee point for the misprediction rate is when B=2. The corresponding microbenchmark is similar to the BTB-capacity test, but we deal with a smaller number of branches B placed at equidistant locations with larger distances D_S.

```
Address     Code
            int long unsigned i, a=1, liter = 1000000;
            do {
@A             if(a==0){ // dummy non-branch instructions (skipped)
               }
@A+x           // dummy non-branch instructions (executed)
@A+DS          if(a==0){ // dummy non-branch instructions (skipped)
               }
@A+DS+x        // dummy non-branch instructions (executed)
@A+2DS         if(a==0){ // dummy non-branch instructions (skipped)
               }
@A+2DS+x       // dummy non-branch instructions (executed)
@A+3DS         if(a==0){ // dummy non-branch instructions (skipped)
@A+3DS+x       // dummy non-branch instructions (executed)
               liter--;
            } while(liter>0);
```

Figure 5.5 BTB-set microbenchmark

The microbenchmark shown in Figure 5.5 includes 4 spy conditional branches placed at equidistant locations. It should be noted that a number of non-branch dummy instructions is executed between two spy branches; this way we ensure that occurrences of the spy branches are separated in time, giving each branch instruction enough time to update the BTB before the next one is executed. However, time separation is not possible in the experiments with rather small distances D_S.

In the BTB example described above (128 sets, 4 ways, with index bits IP [10:4]) B=4 spy branches placed at distance D=2 Kbytes map in the same BTB set. They all fit in the BTB producing no mispredictions if their tags are unique (for example, let us assume a 9-bit tag, Tag=IP [19:11]). If we try to map 5 branches with the same distance, we should see an increase in the misprediction rate. The actual misprediction rate depends on the replacement policy and should be close to 100% if the round-robin or the LRU replacement policy is implemented. A further increase in the number of branches mapped in the same BTB set will result in high misprediction rate.

A set of BTB-set microbenchmarks can be used not only to verify findings of the BTB-capacity test (e.g., the number of sets, the number of ways, index bits, and offset bits), but also in determining address bits used for the tag field in the BTB. A generalized experimental flow is shown in Figure 5.6. Appendix B shows the BTB-set algorithm flow applied to the 4-way BTB with 512 entries, where branch address bits IP[10:4] are used as the BTB index and branch address bits IP[10:4] are used as the BTB tag.

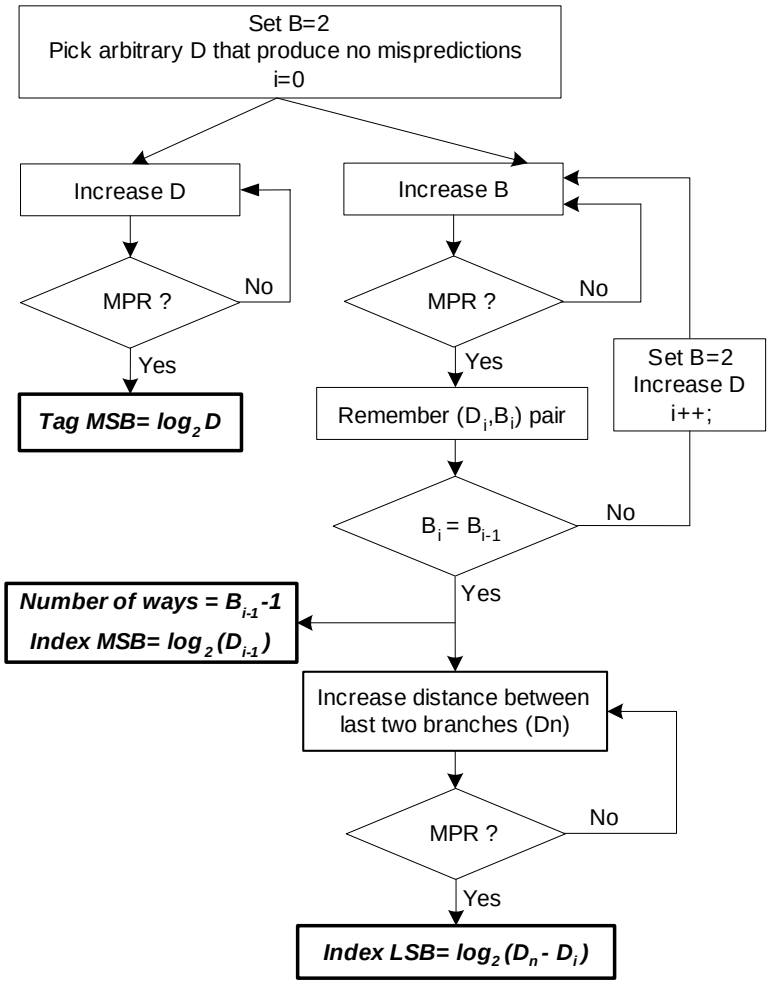

Figure 5.6 Searching for tag and index bits using the BTB-set test

We measure the number of branches mispredicted at decoding (MBI_DEC). Figure 5.7 shows the misprediction rate calculated as the MBI_DEC divided with the number of spy branches as a function of B (B = 2…9) and *D* (D=800h–4000h). These results are only a subset of the results collected for the generalized experimental flow shown in Figure 5.6, but they are found sufficient for further analysis.

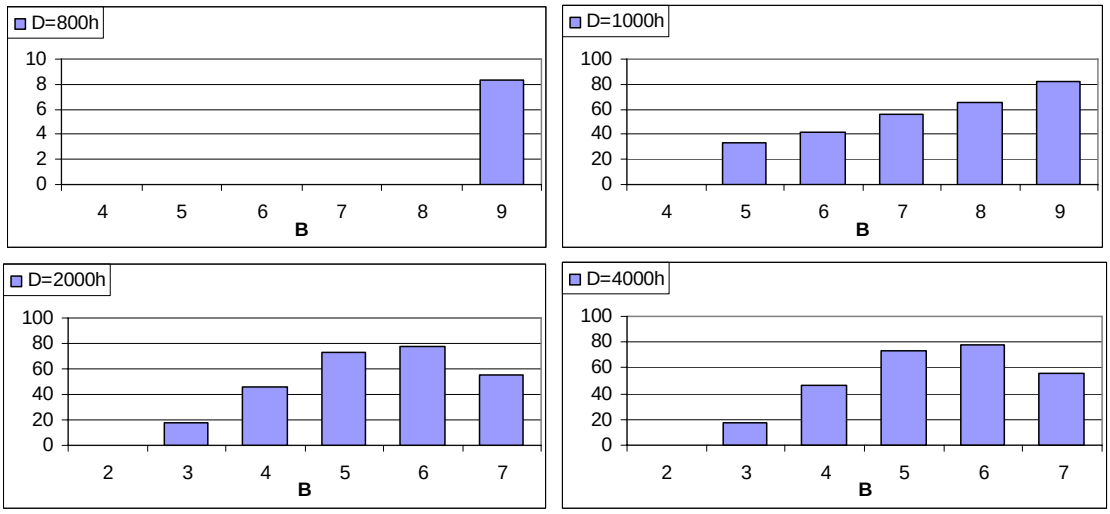

Figure 5.7 BTB-set test results

For D=2000h and up we observe no mispredictions when B = 2 and an increase in the misprediction rate when B=3. According to BTB-set algorithm, we choose pair (Bi, Di) = (2, 1000h). This indicates a 2-way BTB with address bit IP[12] being the most significant bit of the index field. The misprediction rate for B=3 is approximately 20%. An ideal replacement policy in a 2-way structure would be able to recognize that 3 branches compete for 2 slots in a set, so the third branch would never be allocated, leading to a misprediction rate of 33%. However, the misprediction rate in this experiment actually varies among different microbenchmark runs in the range between 10% and 80%. This is another indication that the BTB replacement/allocation policies are not conventional ones (LRU, allocate on first miss). Consequently, additional tests are required to determine the number of ways and the index bits.

Alternatively, if we assume that IP[12] is the index MSB bit (D=2000h) and a 4-way BTB, then the allocation and replacement policy are likely causes of

mispredictions even when B<5. The motivation for such unconventional replacement/allocation policy could come from BTB design trade-offs (runtime resizing/power savings/speed). This assumption does not conflict with results for D=1000h.

Determining Tag Bits

To determine tag bits, we use the BTB-set test with B=2 while varying the distance D_S. Figure 5.8 shows the misprediction rate, calculated as the MBI_DEC divided with the number of spy branches, as a function of D_S (D_S =2000h–800000h). The results indicate that the IP address bits [21:13] are used for the BTB tag match. According to analysis in Section 3.1, we may expect more than 9 bits to be used (IP address bits [31:30] possibly included), but the test cannot be performed for this IP address bits.

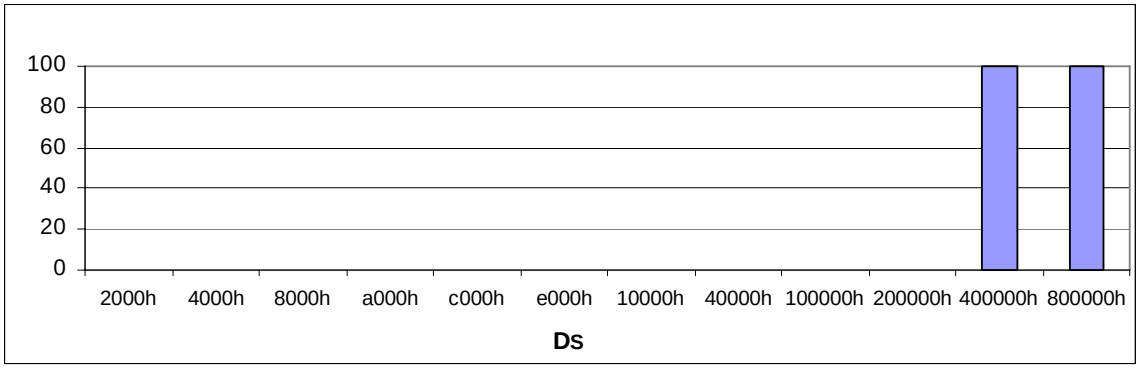

Figure 5.8 BTB-set tag testing results

Determining Index LSB Bit

The last portion of the BTB-Set test focuses on determining the least significant bit of the index field. We start from the test with D_S=2000h and B=3 because it incurs a certain number of mispredictions. The distance between the second and third branch D' is increased (D'–D_S = 1, 2, 4, 8, …) until mispredictions disappear. The value of D'–D_S that produces low misprediction rate is the index LSB distance; Index LSB=$\log_2$(D'–D_S).

Figure 5.9 shows the misprediction rate, calculated as the MBI_DEC divided with the number of spy branches, as a function of the difference between IP addresses of the 2nd and 3rd branch instruction. When the difference reaches 11 bytes, we observe no mispredictions. The difference of 11 bytes is equivalent to the difference of 16 bytes in the branch instruction addresses (the instruction length is 6 bytes). As noted before, the branch address of a branch instruction that spans two 16-byte blocks is determined as the IP address of the last byte of that instruction. The results indicate that the Index LSB bit is IP[4].

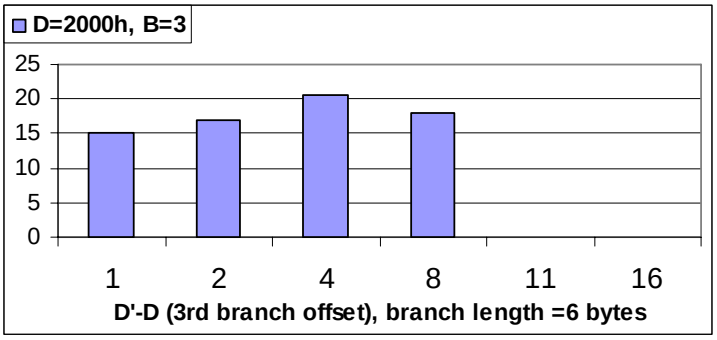

Figure 5.9 BTB-set Index LSB testing results

5.5 Modified BTB-capacity Test

The original BTB-capacity and BTB-set tests indicate that the BTB size is either 1024 or 2048 entries. Here, we want to further investigate the size of the BTB. Let us consider the following examples first.

Example #1. We assume the following BTB organization: 2048 entries organized in 512 sets and 4 ways, Index=IP [12:4], the LRU replacement policy, and allocate on first miss allocation policy. Let us consider an execution of the BTB-capacity test with B=1280 and D=8. Two consecutive branches reside in a single 16-byte instruction block. An execution of the BTB-capacity test will result in the following execution trace.

- *Branches 0-511 (512) map into the 1st and 2nd way of the BTB lower half.*
- *Branches 512-1023 (512) map into the 1st and 2nd way of the BTB higher half.*
- *Branches 1024-1280 (256) map into the 3rd and 4th way of the BTB lowest quarter.*

We see that four branches map into each set in the lowest BTB quarter and two branches map into each set in the other 3 quarters of the BTB. The expected number of mispredictions should be close to zero.

Example #2. Let us assume that the BTB size is 1024 entries, organized in 256 sets and 4 ways, Index=IP[11:4], the LRU replacement policy, and allocation is done on each BTB miss. Consider the same BTB-capacity test as the one described in Example #1. The number of branches in this test exceeds the number of entries in the BTB, so we should see an increased number of mispredictions.

Our next step is to design a new microbenchmark, here called *CapMod*. In creating this microbenchmark, we start from the BTB-capacity test with B=1280, D=8. The corresponding *CapMod* test is created by preserving the BTB-capacity benchmark layout,

but all branch instructions are removed except the first 256 branches (0-255) and the last 256 branches (1024-1279).

Example #3. If the BTB size is 2048 entries (512x4) and four branches that target the same set cause mispredictions, the number of mispredictions for the *CapMod* test will be the same as in the original BTB-capacity test with B=1280 and D=8 since we removed only branches that were two per set (do not add to the number of mispredictions).

Example #4. If the BTB size is 1024 entries, CapMod will not produce mispredictions because CapMod has only 512 branches.

According to these examples, we create two tests, *CapMod1* and *CapMod2*. *CapMod1* is based on the BTB-capacity test with B=1280 and D=8 -- it preserves first 256 branches and the last 256 branches (the total B is B=512). *CapMod2* is based on the BTB-capacity test with D=1152 and D=16 -- it preserves the first 128 and the last 128 branches and the branches 512-768 (B=378).

Figure 5.10 shows the number of branches mispredicted at decoding (MBI_DEC) per one program iteration. According to the assumption in *Example #3,* the results indicate that that the four branches that target the same set in the 4-way BTB with 2048 entries cause the mispredictions. We conclude that the BTB has unconventional allocation/replacement policies.

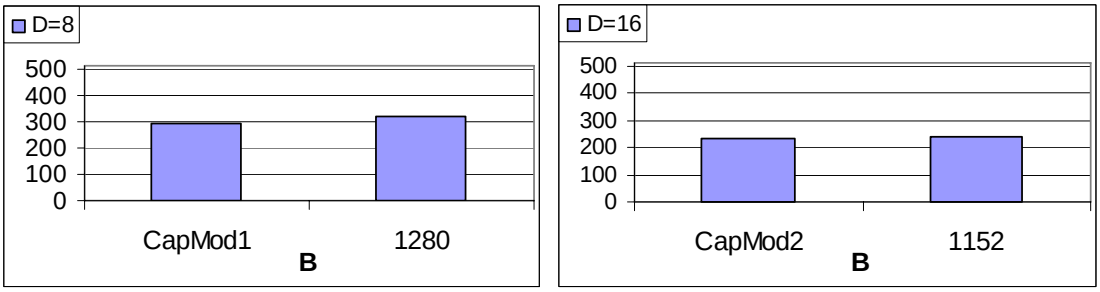

Figure 5.10 Modified BTB-capacity tests results

5.6 Cache-hit BTB-set Test

The results of the BTB-capacity and BTB-set tests were inconclusive and we were unable to determine the BTB size. The previous *CapMod* tests indicate a BTB of 2048 entries, but obviously the replacement policy and/or BTB allocation policy are not conventional ones, making our experimental flow insufficient in determining BTB organization. Consequently, our next step is to develop a microbenchmark that will stress the allocation policy.

In the BTB-set tests, the execution pattern for spy branches in one iteration of the benchmark main loop is as follows: $J_1, J_2, \ldots, J_B$. If allocation/replacement policy considers the number of times a BTB entry is accessed before eviction, this pattern will be insufficient in determining BTB organization. Consequently, the microbenchmark is modified to exhibit the following execution pattern: $J_1, J_1, J_2, J_2, J_3, J_3, \ldots, J_B, J_B$. Here, each branch is executed twice before going to the next branch in sequence.

The microbenchmark source code fragment is shown in Figure 5.11. We use an indirect branch as a setup branch to achieve execution of each spy branch twice consecutively. The rest of the source code can be found in Appendix B.

```
int long unsigned liter = 100000000;
for (i=0; i<liter; ++i){
   jmp  dword ptr [ebx] // setup indirect branch
   ...
   l0:   jmp edx // branch 0, "edx" stores _Exit IP
   // dummy code to set distance D
         jmp edx  // branch 1
   // dummy code to set distance D
   ...
   // dummy code to set distance D
         jmp edx  // branch B
 _Exit: clc
}
```

Figure 5.11 Cache-hit test source code fragment

We measure the number of branches mispredicted at decoding (MBI_DEC). Figure 5.12 shows the misprediction rate calculated as the MBI_DEC divided with the number of spy branches as a function of B (B=2…17) for D=800h–4000h.

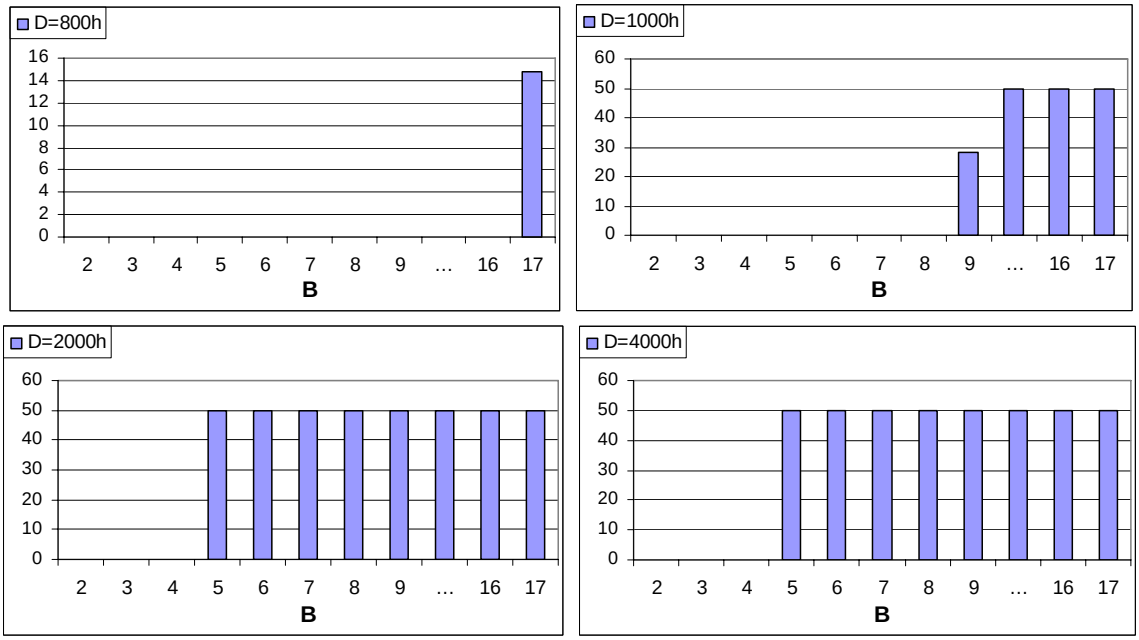

Figure 5.12 Cache-hit BTB-set test results

The results indicate that a 4-way structure is used with set size of 512 entries. When all branches target a single BTB set, the misprediction rate is 50%, indicating the LRU replacement policy; and allocate-on-first miss allocation policy. The branches that are just allocated can be evicted from the cache if they have not been touched again in the program execution. This explains why 3 branches in the BTB-Set test can cause some mispredictions.

5.7 Cache-hit BTB-capacity Test

The next step in the experimental flow is to confirm our findings about replacement policy. The Cache-hit BTB-set approach is applied to the BTB-capacity test, where we try to fit as many branches in the BTB as possible with no BTB mispredictions. The base implementation ensures that each spy branch is executed twice consecutively, before any other branch that maps in the same set. We call this microbenchmark Cache-hit BTB-capacity test.

We measure the number of branches mispredicted at decoding (MBI_DEC). Figure 5.13 shows the misprediction rate, calculated as the MBI_DEC divided with the number of spy branches, as a function of B (B=512–4096) and D (D=2–128). The results indicate that the BTB size is 2048 entries. The misprediction rate for non-fitting distance is 50% rather than 100%. This is can be explained as follows. Each branch misses in the BTB on its first occurrence, and its second occurrence results in a BTB hit. The BTB hit "verifies" the corresponding BTB entry and the LRU replacement policy is used in selecting a victim entry in the BTB set.

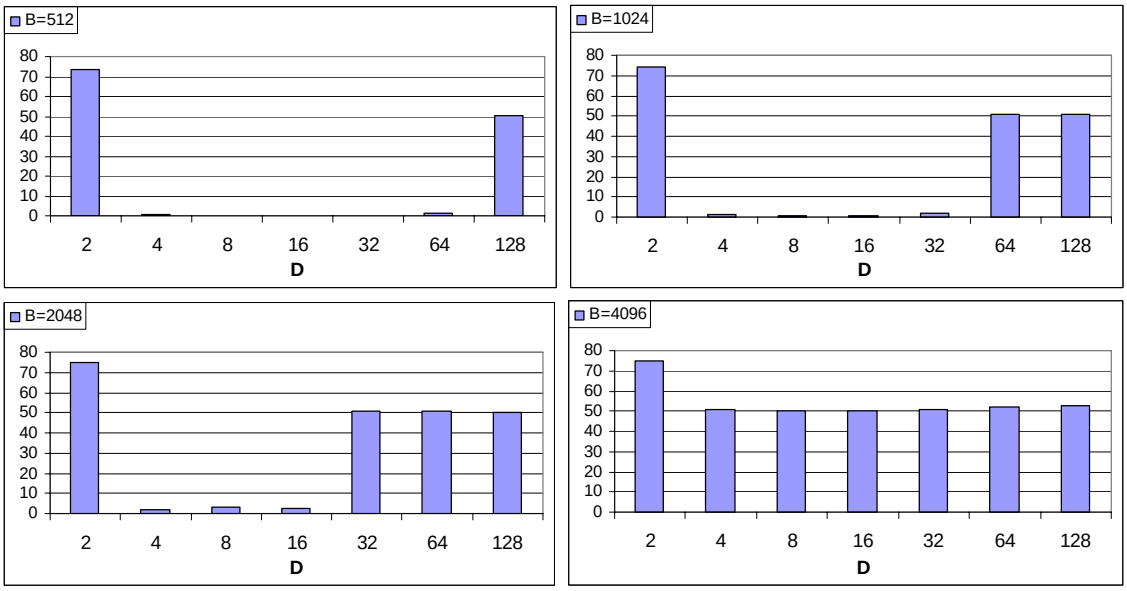

Figure 5.13 Cache-hit capacity test results

5.8 Other Issues

In this subsection we explore several issues related to the BTB operation. The first issue is related to bogus branches and their influence on the BTB. We have already concluded that the BTB uses specific allocation and replacement policies. These policies and their relations are partially revealed. This section presents some of the BTB evictions and allocation policies mainly related to the patent [23]. The intent is to show that the BTB is behaving close to the BTB design presented in [23]. We test following situations:

1) *BTB hit/ misprediction*
2) *BTB hit/ bogus branch detected*
3) *Offset algorithm*

5.8.1 BTB Hit/ misprediction

This situation refers to the occurrence of the branch that has the same index, offset and tag fields as a branch allocated in the BTB but has different target. This situation is observed during tag bits testing in Section 5.4.2 where test employed had a distance D=400000h. We observed that two branches evict each other. In this situation, the BTB updates the entry with the new branch target.

5.8.2 BTB Hit/ bogus Branch Detected

A bogus branch refers to a BTB hit event caused by a non-branch instruction that has the same tag and index fields as an allocated branch. When a bogus branch is executed, the original BTB entry may be left intact or it may be discarded in order to avoid pursuing false paths in the program execution. According to an Intel's patent [23], a bogus branch causes eviction of the corresponding branch from the BTB. Here, we design a microbenchmark that verifies the BTB's behavior related to bogus branches.

The microbenchmark is shown in Figure 5.14.

```
Address            Code
                   int long unsigned liter = 1000000;
                   int a=1;
                   do {
@A                    if(a==0){ // dummy non-branch instructions (skipped)
                      }
@A+x                  // a dummy non-branch instructions
@A+400000h            // a dummy non-branch instructions(a bogus branch)
                      liter--;
@A+400000h+y;y<x   } while(liter>0);
```

Figure 5.14 Bogus branch test source code

The program's main loop encompasses an always taken spy branch. The target address of the spy branch is tuned to move the program control flow so that *IP(Spy Target) - IP (Spy address)* $< 2^{MSB(TAG)}$. An instruction that follows the spy branch target instruction will have the same tag and index fields as the spy branch – i.e., the instruction acts as a bogus branch. If the bogus branch evicts the spy branch, we should see a high misprediction rate.

The measurements show that the spy branch is always mispredicted. Consequently, we confirm that a bogus branch evicts entries the BTB entry.

5.8.3 Offset Algorithm

The BTB may provide more than one hit for the fetched 16-byte block since the BTB design allows that multiple branches in a single BTB set have the same tag. The experiments in Section 5.4.1 for distances D=4 and D=8 confirm the previous statement. We could have four consecutive branches that map in a single BTB set, all having the same BTB tag field and all are predicted correctly.

Once a new 16-byte block is fetched from the memory, BTB provides multiple hits [23]. Then, an algorithm selects the correct target. The algorithm selects the target with the smallest offset value yet not smaller than the current instruction IP address.

In this test we want to observe a situation when two branches from the different 16-byte blocks have the same index and tag but different offset and therefore BTB provides two hits, but actually one branch will be detected.

Two situations are observed: (a) **Double hit –** two 16-byte blocks with each having one spy branch that match by both tag and index are entered from the lowest byte in the block and therefore, whenever BTB has two branches allocated, one will be a false

hit or a bogus branch. (b) **Single hit –** Second 16-bytes block is entered with the IP value that is in between offsets of the observed two branches that matched. In this case, upon second 16-byte block being entered, only one BTB hit is provided; upon first 16-byte block being entered, two BTB hits are provided if the both branches are allocated at the moment.

Double hit

The microbenchmark source code reuses the source code of the BTB-set test with D=400000h and B=2 with second, Spy2 branch moved slightly to have the different offset fields within the same set as shown in Figure 5.15.

The test is executed with changing of the Spy2 offset while keeping Spy1 and Spy2 to target the same set. The test reports the misprediction rate of 100% which is an indication that both branches are mispredicted. There was a possibility that *Spy1was* always predicted correctly because when the 16-byte block is entered, the BTB logic actually selects the BTB entry for the *Spy1* branch. Consequently we conclude that the false hit evicts all branches within the set.

```
Address                Code
                       int long unsigned liter = 1000000;
                       int a=1;
                       do {
@A                        if(a==0){    // Spy1
                            // dummy non-branch instructions (skipped)
                          }
@A+x                     // a dummy non-branch instructions
@A+400000h+z;z-x<16       if(a==0){    // Spy2
                            // dummy non-branch instructions (skipped)
                          }
                         liter--;
@A+400000h+y;y<x       } while(liter>0);
```

Figure 5.15 Double hit test source code

Single hit

The microbenchmark source code is shown in Figure 5.16. The 16-byte block that has *Spy2* within it is entered from a jump instruction and the entry point is in between *Spy1* and *Spy2* offsets.

The test performed did not produce any mispredictions. The correct prediction for the second 16-byte block with *Spy2* indicates that the offset algorithm did not consider the BTB hit from *Spy1*, as the *Spy1* offset was smaller from the instruction pointer. Correct prediction for the first 16-byte block indicates that the offset algorithm exists and the purpose is to select the entry with the smallest offset among multiple BTB hits.

```
Address                 Code
                        int a=1;
                        int long unsigned liter = 1000000;
                        do {
@A                         jmp l1        // Spy1
                           // dummy non-branch instructions (skipped)
l1                         // a dummy non-branch instructions
@l1+x                     jmp l2
                            // dummy non-branch instructions (skipped)
l1=@A+400000h+z;z-x<16     if(a==0) a=1  // Spy2
                          liter--;
@A+400000h+y;y<x        } while(liter>0);
```

Figure 5.16 Single hit test source code

CHAPTER 6

MICROBENCHMARKS FOR THE REVERSE ENGINEERING OF LOOP PREDICTORS

6.1 Objectives

The goal of this section is to develop an experimental flow and a set of microbenchmarks that will help us determine the structure of the loop predictor. We expect a loop predictor with a cache-like structure and we want to determine the loop predictor's cache parameters (size, sets, ways, index, tag and replacement policy), an algorithm used to recognize a loop branch type behavior, and the counter based algorithm used for the loop-branch outcome prediction. We also want to determine relationship between the loop predictor and the BTB and test for existence of other local predictors.

6.2 Contributions

We developed an experimental flow and a set of microbenchmarks for determining organization of the advanced loop predictor. The experimental flow and microbenchmarks applied on a Pentium M processor provide the following insights.

1) The loop predictor main part is a two-way cache structure named the loop branch prediction buffer, the loop BPB. The loop BPB is organized in 64 sets (total size is 128 entries). The loop BPB index field is IP [9:4]. The loop BPB tag field is IP [15:10]. The loop BPB employs the LRU replacement policy.
2) Each loop branch prediction buffer entry has two 6-bit counters. One counter counts the current loop iteration number and the other stores the maximum loop count. When the two values match, the predictor provides an opposite outcome prediction.
3) A branch is allocated in the loop branch prediction buffer when the opposite outcome of the branch is observed.
4) Once a branch is allocated in the loop BPB, the whole loop pattern is used to train the loop maximum counter value in the loop BPB.
5) A loop BPB hit is filtered with a regular BTB hit.
6) The branch prediction mechanism does not employ other local predictors.

6.3 Background

Loop predictor is a local predictor specialized for branch instructions that exhibit loop behavior. A branch exhibits loop behavior if it moves in one direction for a certain number of times interspersed with one outcome in the opposite direction. For example, a branch with a repeating outcome pattern, 10 times taken and 1 time not taken, can be marked as $\{\{T\}^{10}, NT\}^{k}$. A loop predictor should accurately predict branch outcomes when the outcome pattern length exceeds the size of the branch history buffer attached to a global branch predictor. The available information on the loop predictor indicates that once loop behavior of a branch is detected, a set of counters is allocated, without further

details about the loop predictor organization [2]. A number of questions arise from this scarce description. The first question is how to recognize a reoccurrence of the branch that exhibits loop behavior? We assume that a cache-like structure is used in the loop predictor, named a loop branch prediction buffer, the loop BPB. Similarly to the regular BTB, the loop BPB provides a hit if a particular IP address matches the tag in the loop BPB. In contrast to the regular BTB, the loop BPB keeps resources needed to carry out the correct outcome prediction. To detect loop behavior, two counters are assigned to each loop BPB entry [2] – the maximum counter and the iteration counter. The maximum counter gets trained by the pattern length to indicate the opposite branch outcome, and the iteration counter is incremented on each branch execution. When the iteration counter reaches the maximum counter, an opposite branch outcome is returned as a branch prediction, and the counter is cleared. This way, each branch outcome is predicted correctly. A loop BPB entry can also keep the target branch address, but reusing a BTB target addresses is a more likely option. Prior to allocation of an entry in the loop BPB, we would like to know if the particular branch has a loop-like behavior. This raises the question of the loop predictor training.

Consequently, in reverse engineering of the loop predictor, we would like to answer to the following questions:

1. *What is the maximum counter length?*
2. *What is the loop BTB organization?*
3. *How the loop predictor is trained?*
4. *What is loop predictor allocation policy?*
5. *What is relationship of the loop predictor with the regular branch target buffer?*

6.4 Maximum Counter Length

In determining counter length, we use a 'spy' branch that exhibits loop behavior. For example, the spy branch is executed many times, and it repeatedly goes through the following pattern: $\{\{T\}^{L-1}, nT\}$ – taken (L-1) times, and not taken once. The corresponding microbenchmark is shown in Figure 6.1.

```
#define L 65     /* pattern length */
void main(void){
   int long unsigned I;              /* loop index */
   int long unsigned I = 100000000;  /* number of iterations */
   for (i=0; i<I; ++i){
      if ((i%L) == 0) a=0;           /* spy branch */
   }
}
```

Figure 6.1 Microbenchmark for determining maximum counter length

We increase the parameter L, starting from *Lmin*, where *Lmin* is determined by the length of the global branch history register, i.e., $Lmin \geq length(BHR)$. The expected number of retired mispredictions in this microbenchmark is shown in Equation (6.1), where N is the length of loop counters. As long as the pattern can be caught by the loop counters, the number of mispredictions should be close to zero. Once the pattern length exceeds the size of loop counters, the number of mispredictions should be equal to I/L, where I is the number of the outer loop iterations.

$$MBI \approx \left\{ \begin{matrix} 0, L \leq 2^N \\ \frac{I}{L}, L > 2^N \end{matrix} \right\}. \tag{6.1}$$

We measure the number of branches mispredicted at execution (MBI_EXEC). Figure 6.2 shows the misprediction rate, calculated as the MBI_EXEC divided with the number of spy branches, as a function of L (L=8–128). The results indicate that pattern lengths of L≤64 can be successfully predicted, and when L=65, the number of mispredictions is equal to *I/65*. Consequently, the loop counters are 6 bits long.

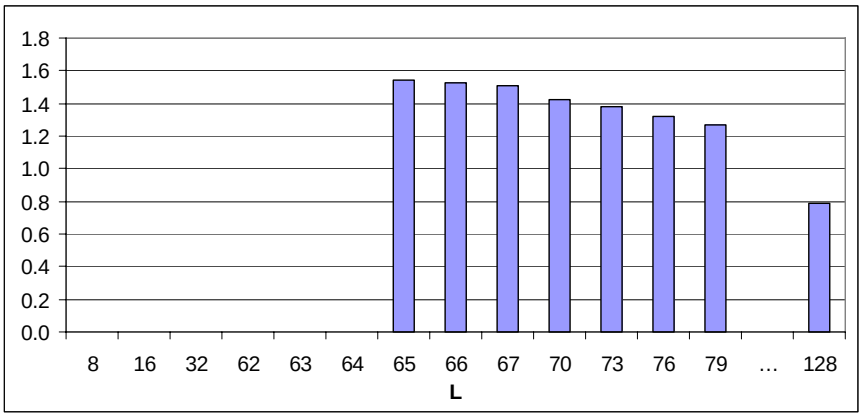

Figure 6.2 Maximum counter length test results

6.5 Loop BPB Organization

In determining loop BPB organization, we start from the BTB-capacity and BTB-set tests used in the experimental flow for determining regular BTB organization. We hypothesize that the loop BPB is organized as a cache-like structure that keeps relevant information about branches that exhibit loop-like behavior. Each entry is accessed using a portion of the branch address, has its own tag and two counters [2]. Each counter is 6 bits in length. We also assume that the target address is retrieved from the regular BTB.

Two microbenchmarks are used. A so called Loop-capacity test stresses the loop BPB structure trying to find the maximum number of branches that can fit in the loop BPB. A so called Loop-set test stresses the loop BPB structure trying to find the maximum number of branches that can fit in a single loop BPB set.

6.5.1 Loop-capacity Tests

A Loop-capacity microbenchmark reuses the algorithm used in the BTB-capacity tests. The Loop-capacity tests features B branches that exhibit loop behavior placed at equidistant memory locations with distance D. The microbenchmark layout is illustrated in Figure 6.3.

The microbenchmark source code is shown in Figure 6.4. A compiled code for a single loop from Figure 6.4 takes more space in memory than a simple branch used in the BTB-capacity test (3 instructions per loop). Consequently, the smallest distance between loops D_{MIN} will be larger than D_{MIN} in the BTB-capacity test (D_{MIN}=2). Here, D_{MIN}=8. To avoid a correct prediction coming from a global branch predictor, loop branches have pattern lengths of 64 (the branch is taken 63 times, not taken one time). The microbenchmark uses the highest possible count modulo because of the following reason; Due to speculative execution and pipelining, the global predictor's access shift register in the first level may not be updated by the loop branch outcome as fast as the loop code is fetched and decoded. It may happen that the one program loop is over before its outcomes are reflected in the global predictor. This may cause the global predictor to affect the prediction.

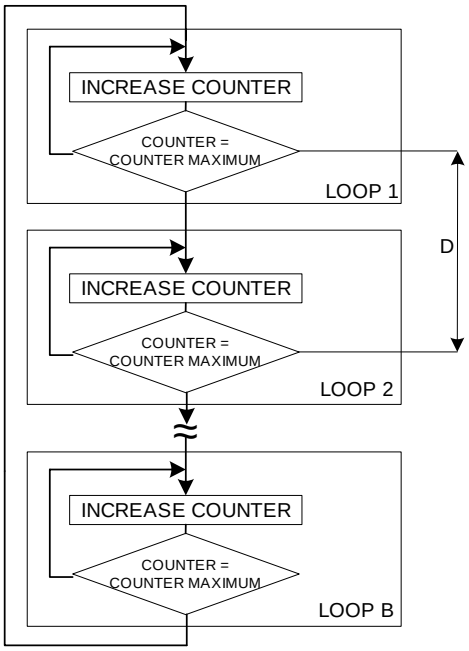

Figure 6.3 Layout of the Loop-capacity test

```
Address     Code
            int long unsigned liter = 1000000, modulo = 64;
            do {
               temp = modulo;
               _asm{
                    mov al, temp
                  l0: sub al,1
                    cmp al, 0
@A                  jne l0            // 1st spy loop branch
                    // dummy non-branch instructions
                    mov al, temp
                   l1: sub al,1
                    cmp al, 0
@A + Ds             jne l1          // 2nd spy loop branch
                    // dummy non-branch instructions
                    mov al, temp
                  lBm1: sub al,1
                    cmp al, 0
@A + B*Ds           jne lBm1        // (B-1)th spy loop branch
               } liter--;
            } while(liter>0);
```

Figure 6.4 Loop-capacity test source code

We measure the number of branches mispredicted at execution (MBI_EXEC). Figure 6.5 shows the loop misprediction rate, calculated as the MBI_EXEC normalized to the total number of program loops, as a function of the number of the executed loops B (B= 32–256) for D=8–64 respectively.

For tests with D=8 and D=16, mispredictions occur for B larger than 128 and for B=256, all loop are mispredicted. This indicates the loop BPB size of 128 entries. For tests with D=32, mispredictions occur for B=128. This is an indication that the distance becomes large enough and that every second set is jumped over. The tests for D=64 confirm this conclusion. Consequently, we conclude that the branch IP address bits [3:0] are not used to access the loop BPB.

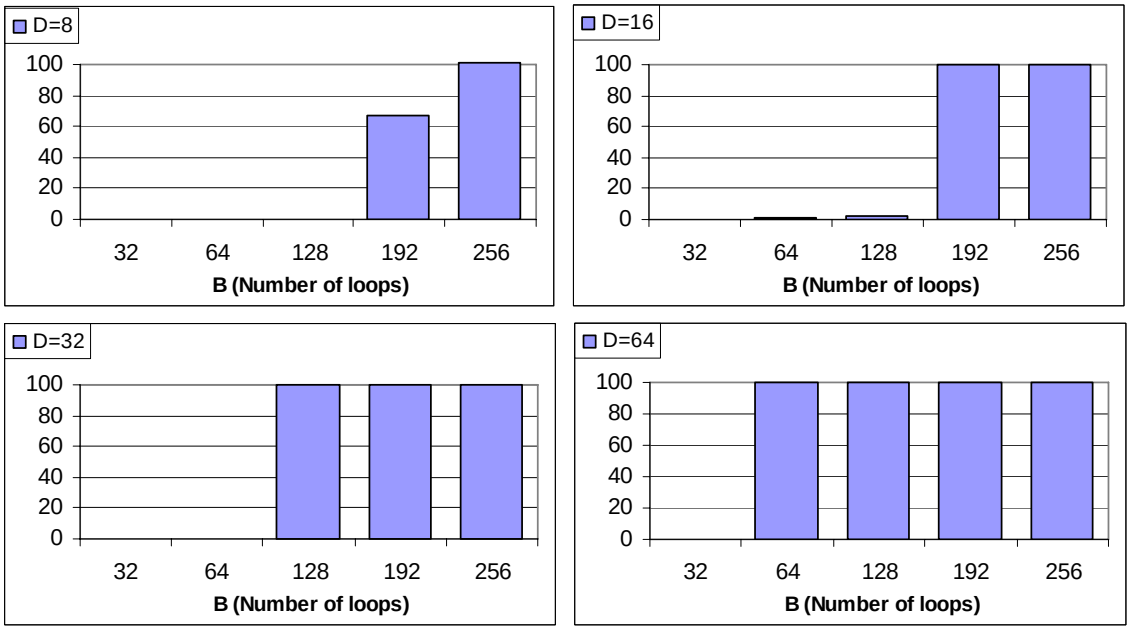

Figure 6.5 Loop-capacity test results

We cannot determine the number of ways in the loop BPB because of missing Loop-capacity tests for small distances (D=2 and D=4). The results for D=8 and D=16 report the same maximum number of branches B that cause no mispredictions, B=128. This is an indication that the minimum number of ways is two.

6.5.2 Loop-set Tests

In order to determine the loop BPB organization further, we develop a so-called Loop-set test. This microbenchmark is very similar to the BTB-set microbenchmark (see Figure 5.6). The microbenchmark includes loops instead of always taken branches and the loop branches are placed at larger distances D_S (see Figure 6.6).

```
Address     Code
            int long unsigned modulo = 64;
            do {
              temp1 = modulo-1;
              temp2 = modulo-2;
              ...
              tempB = modulo-B;
               _asm{
                    mov al, temp1
                  l0: sub al,1
                    cmp al, 0
@A                  jne l0           // 1st spy loop branch
                    // dummy non-branch instructions
                    mov al, temp2
                   l1: sub al,1
                    cmp al, 0
@A+ Ds              jne l1          // 2nd spy loop branch
                    // dummy non-branch instructions
                    mov al, tempB
                lBm1: sub al,1
                    cmp al, 0
@A+ B*Ds            jne lBm1        // (B-1)th spy loop branch
               } liter--;
            } while(liter>0);
```

Figure 6.6 Loop-BTB-Set test source code

For the part of the Loop-set algorithm used for the tag MSB testing, two branches that match both the index and the tag must have different loop modulus. Otherwise the loop predictor will predict them correctly as it sees them as the one loop branch. To avoid such a situation, the microbenchmark features loops with different modulus.

We measure the number of branches mispredicted at execution (MBI_EXEC).

Figure 6.7 shows the loop misprediction rate, calculated as the MBI_EXEC normalized to the total number of program loops, as a function of the distance between branches *D* (D=80h–10000h) for B=2. According to the BTB-set algorithm, the tag MSB bit is the IP address bit 15.

Figure 6.8 shows the loop misprediction rate, calculated as the MBI_EXEC normalized to the total number of program loops, as a function of the distance between branches *D* (D=80h–1000h) for B=3. The test indicates that the index MSB bit is the IP address bit 9 and the loop BPB is a two-way cache structure.

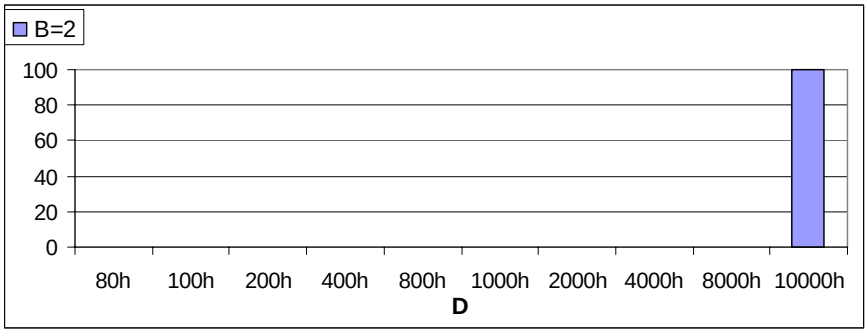

Figure 6.7 Loop-set test results for B=2

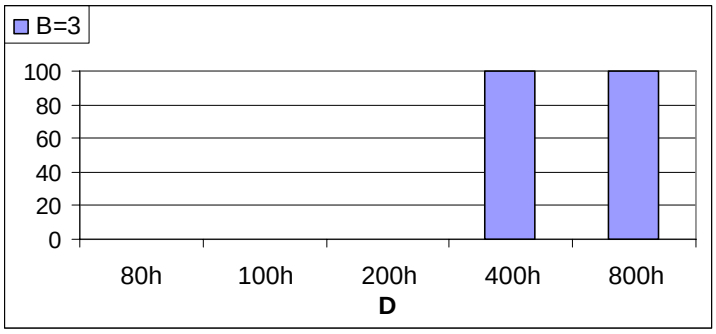

Figure 6.8 Loop-set Index MSB testing results

Figure 6.9 shows the loop misprediction rate, calculated as the MBI_EXEC normalized to the total number of program loops, as a function of the difference between memory addresses of the starting bytes of the 2nd and 3rd branch instruction. The test indicates that the Index LSB bit is the address bit IP[4].

According to the Loop-capacity and Loop-set tests, we conclude that the loop BPB is a 2-way cache structure with 128 entries, indexed by the IP[9:4] and tagged by the IP[15:10].

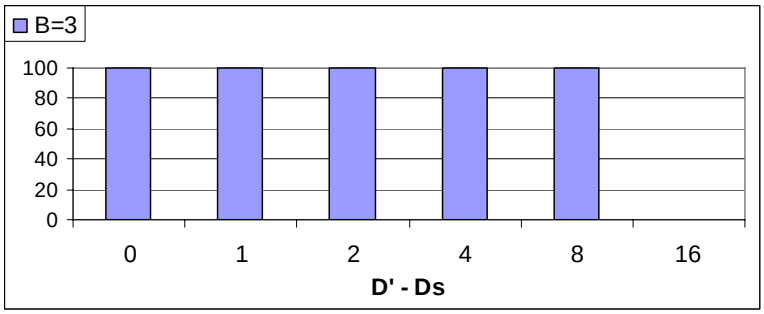

Figure 6.9 Loop-set Index LSB testing results

6.6 Loop Predictor Training Logic

The loop predictor must act upon the program branches and decide whether the branch is a potential loop branch. After the branch becomes the candidate for a loop branch, the loop predictor must set the loop counter maximum value before the loop reoccurrence may be monitored. We refer to this as the loop training process. After a branch becomes a candidate for a loop, the training process may take place in the loop BPB or in the separate training logic. A benefit from the separate training logic would be from saving the BTB entries from eviction in the case that a currently trained branch comes out not to be a loop. If a separate training logic is used, we expect that, upon training being done, new loop branch is transferred to the loop BPB. We expect that separate training logic is similar to the loop BPB but is a much smaller structure since we expect that only part of the program loop branches are in training at once.

A microbenchmark is developed to stress the training logic. It reuses BTB-capacity and BTB-set algorithms used for the regular BTB experiments (Equation .

(6.1) and Figure 5.6). The algorithm used here changes always taken branches from the original algorithm to the loop branches. In contrast to the loop BPB organization test (see Figure 6.3), the test loops are not executed consecutively, rather we execute each loop iteration consecutively. Therefore, all program loops are in the training phase at once. If the number of branches B is larger than the training logic size, mispredictions exist. To avoid correct predictions from the global predictor, the microbenchmark employs loop branches with different pattern lengths. Otherwise, all branches would change their outcome at the same time, at the counter maximum value,

making them predictable with the global predictor. The number of branches B is varied to determine the number of branches that can be trained consecutively.

Figure 6.10 shows a source code for one possible implementation of the microbenchmark for B=256 and 16 different pattern lengths assigned in round-robin fashion. Distance between branches is set to D=16 as we prove that D=16 allows for each branch to have its own loop BPB entry if the number of branches B is smaller than 128.

```
Address          Code
                 int long unsigned modulo = 4;
                 int long unsigned liter = 1000000;
                 do {
                   temp1 = liter%modulo;          // 1st branch modulo=4
                   temp2 = liter%(modulo+1);      // 2nd branch modulo=5
                   ...
                   temp16 = liter%(modulo+15);    // 16th  branch modulo=19
                    _asm{
                         mov eax, temp1
                      l0: sub eax,1
                         cmp eax, 0
@A                       jg l0            // 1st spy loop branch
                         // dummy non-branch instructions
                         mov eax, temp2
                       l1: sub eax,1
                         cmp eax, 0
@A + 10h                 jg l1          // 2nd spy loop branch
                         // dummy non-branch instructions
                         mov eax, temp16
                     l255: sub eax,1
                         cmp eax, 0
@A + 255*10h             jg l255        // Bth spy loop branch
                   } liter--;
                 } while(liter>0);
```

Figure 6.10 Loop training logic test source code example for D=16 and B=256

We measure the number of branches mispredicted at execution (MBI_EXEC). Test results are normalized to the total number of program loops MPR_{MAX}. Let us consider a microbenchmark with B=32 and MOD=8 (loops modulus used are 4…11). Equation . (6.2) shows the MPR_{MAX} value calculation for a given example.

$$\left(\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8}+\frac{1}{9}+\frac{1}{10}+\frac{1}{11}\right)*\frac{liter}{MOD}. \tag{6.2}$$

Figure 6.11 shows the loop misprediction rate, calculated as the MBI_EXEC normalized to the MPR_{MAX} value, as a function of the number of total branches B (B=64, 128, 256) for D=16 and MOD=16. Results indicate that the 128 branches can be trained at once. The results for B=192 and B=256 are similar to those shown in Figure 6.5 (upper left). We don't expect that the training logic for 128 entries is implemented separately from the loop BPB. Consequently, we conclude that the training of counters is carried out in the loop BPB after an entry in the loop BPB is allocated.

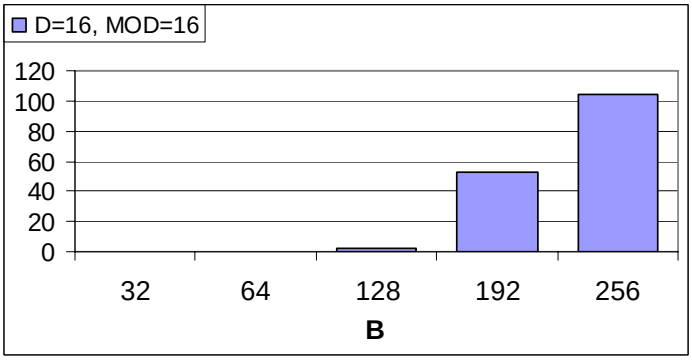

Figure 6.11 Loop training logic test results

6.7 Loop Predictor Allocation Policy

To allocate a loop BPB entry, we need to know if a branch exhibits loop behavior.

We could expect two possible allocation approaches depending on the moment when a branch is identified to have a loop behavior. With one approach, the allocation logic tags a branch as a loop branch as soon as the branch outcome moves in an opposite direction for the first time. For example, if a branch has a repeating outcome pattern *k*{T}.NT, the branch is marked as a loop branch after the $(k+1)^{th}$ occurrence of the branch. Once an entry in the loop-BPB is allocated, the maximum counter will get trained. We refer to this approach as *allocate on opposite outcome detection*. With the second approach, the allocation logic would wait for the branch to have confirmed loop behavior. For example, the earlier branch is tagged as a branch after the $(k+2)^{th}$ occurrence of the branch and only if the $(k+2)^{th}$ branch outcome is the taken one. We refer to this approach as *allocate on loop detection*.

The microbenchmark employs a branch that has a certain outcome pattern. The branch outcome pattern is set to make the branch allocated in the loop BPB if *allocate on opposite outcome detection* is implemented and not to be allocated in the loop BPB if the *allocate on loop detection* is implemented. We choose the branch with the outcome pattern {*3* T, 2* nT*} (3 times taken, two times not taken). The branch is named the *LoopLike*. The *LoopLike* branch is set to target the same loop BPB set as the two real loops (*Loop1* and *Loop2*).

If the *LoopLike* branch does not consume the loop BPB entry, mispredictions come only from the *LoopLike* branch and will be the same regardless of the *Loop1* and the *Loop2* count modulo. If the *LoopLike* branch does consume the loop BPB entry,

mispredictions come from the *LoopLike* branch, the *Loop1* branch, and the *Loop2* branch. The number of mispredicted branches will change with the change of the *Loop1* and the *Loop2* count modulo. The algorithm changes the *Loop1* and the *Loop2* branch modules. Change in the misprediction rate is an indication that the *LoopLike* branch consumes the loop BPB entry. The microbenchmark source code is shown in Figure 6.12. The *Loop1* count modulo is named *MOD1*. The *Loop2* count modulo is named *MOD2.*

```
Address        Code
               int long unsigned liter = 1000000;
               do {
                 temp1 = liter%MOD1;       // Loop1 modulo
                 temp2 = liter%MOD2;       // Loop2 modulo
                 temp3 = (liter%5)& 0xFE   // LoopLike outcome pattern
@A               8 x if(a==0) a=1;
                 // dummy instructions to allow outcomes update
@B               if(temp1==0) a=1;   // Loop1
                 // dummy code to control the distance
@A+ 400h         8 x if(a==0) a=1;
                 // dummy instructions to allow outcomes update
@B+ 400h         if(temp2==0) a=1;   // Loop2
                 // dummy code to control the distance
@A+ 800h         8 x if(a==0) a=1;
                 // dummy instructions to allow outcomes update
                 if(temp3==0) a=1;   // LoopLike
@B+ 800h         liter--;
               } while(liter>0);
```

Figure 6.12 Source code for the loop allocation policy test

Figure 6.13 shows the number of mispredicted branches (MBI_EXEC) per program iteration as a function of *Loop1* and *Loop2* count modulus (*MOD1* and *MOD2*). For *MOD1=MOD2=* 1, *Loop1* and *Loop2* branches are always taken and mispredictions come from the *LoopLike* branch only. The number of mispredictions per iteration is 0.6.

For *MOD1*=3 and *MOD2*=4, the maximum possible number of mispredictions per iteration is 0.6 + 1/4 + 1/3 = 1.83 mispredictions per iteration. Test results are approximately the same (1.08 mispredictions per iteration). For *MOD1*=15 and *MOD2*=16, the maximum possible number of mispredictions per iteration is 0.6 + 1/15 + 1/16 mispredictions per iteration. Test results are approximately the same (0.76 mispredictions per iteration).

Changes in *MOD1* and *MOD2* result in changing of the number of mispredictions. Therefore, the branch is allocated in the loop BPB immediately after detection of a branch opposite outcome.

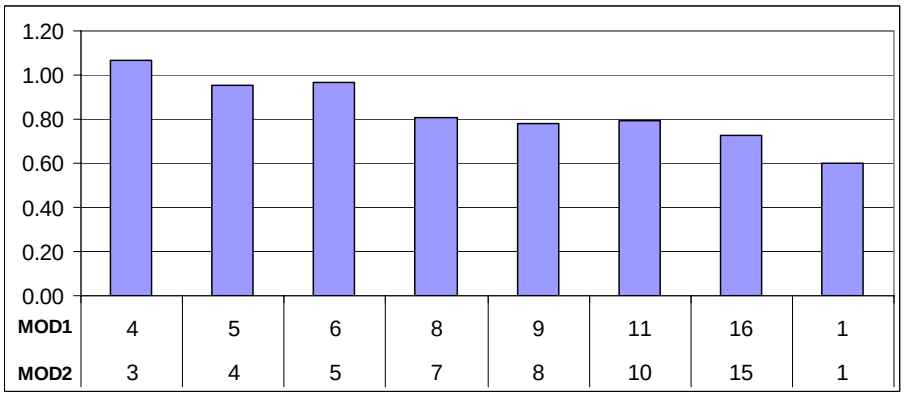

Figure 6.13 Loop predictor allocation policy differential test

6.8 Loop Predictor Relations with the BTB

Microbenchmark tries to find whether the loop BPB hit is conditional upon the BTB hit. The idea for such a test comes from the patent [23] related to the indirect predictor. The patent describes indirect predictor target address cache hit that is

conditional upon the BTB hit. The same logic can be applied here; if the loop BPB provides hit but BTB provides miss, the loop predictor prediction may be ignored. Conditioning with the BTB hit is useful because BTB uses more tag bits (at least 9) and is able to better identify the particular branch.

The microbenchmark puts many always taken branches in the same BTB set with the loop branch. Since always taken branches do not consume the loop BPB entries, the loop predictor will be able to provide the correct prediction for such a loop unless the loop predictor hit is conditional upon the BTB hit. Consequently, we expect mispredictions to exist if the BTB filtering is used. In this case, the expected misprediction rate is proportional to the number of program loops. The microbenchmark source code is shown in Figure 6.14. Four always taken branches are used, as it is enough to make the BTB miss for the spy loop branch.

```
Address          Code
                 int long unsigned liter = 1000000;
                 int a=1;
                 do {
                   temp = liter%MOD;
@A                  if(a==0) a=1;  // Always Taken
                    // dummy non-branch instructions
@A + 2000h          if(a==0) a=1;  // Always Taken
                    // dummy non-branch instructions
@A + 4000h          if(a==0) a=1;  // always Taken
                    // dummy non-branch instructions
@A + 6000h          if(a==0) a=1;  // always Taken
                    // dummy non-branch instructions
@A + 8000h          if(temp==0) a=1;  // Spy loop
                   liter--;
                 } while(liter>0);
```

Figure 6.14 BTB filtering test source code

We measure the number of branches mispredicted at execution (MBI_EXEC).

Figure 6.15 shows the misprediction rate calculated as the MBI_EXEC normalized to the total number of program loops (Loop misprediction rate).

The test is performed for three MOD values, MOD = 3, 5, 10. We observe that the loop misprediction rate is proportional to the number of program loops; thus, the final conclusion is that the BTB hit does filter the loop BPB hit

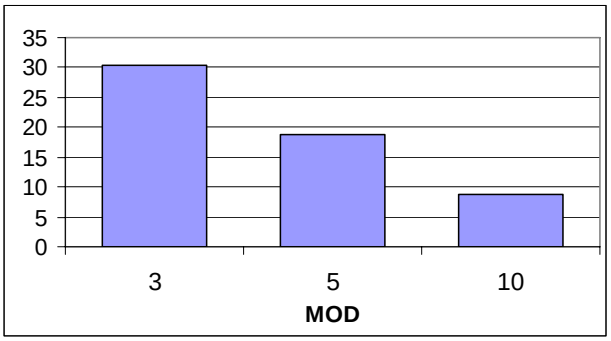

Figure 6.15 BTB filtering test results

6.9 Loop-BPB Replacement Policy

The loop-BPB is organized as a 2-way cache-like structure. This brings us to the question of the loop-BPB replacement policy. It could be random, First-In-First-Out (FIFO), or Least Recently Used (LRU). For both FIFO and LRU, one bit is needed per loop BPB set to point to an entry in the set that needs to be replaced next. This bit is updated on each miss in that BTB set for the FIFO policy, or on each miss and hit for the LRU replacement policy.

A microbenchmark is used to stress the loop BPB replacement policy. The microbenchmark encompasses three branches, here referred to as A, B, C. These branches are laid out in memory in such a way as to compete for one set in the loop BPB (similar to the Loop-set microbenchmark shown in Figure 6.6).

Three branches A, B, C have occurrence pattern {A, B, A, C} (see Figure 6.16).

If a hit affects the replacement bit, the branch A should always hit and branches B and C will compete for the remaining entry in the selected loop BPB set, and the expected misprediction rate is around 50%. If a hit does not affect the replacement bit, we should observe a misprediction rate close to 100%.

The code from Figure 6.16 produces a misprediction rate of 50%. Consequently, we conclude that the loop BPB employs the LRU replacement policy.

```
Address        Code
               int long unsigned liter = k, 1000000;
               do {
                 k = liter%4;
                 temp1 = liter%MOD1;       // A modulo
                 temp2 = liter%MOD2;       // B modulo
                 temp3 = liter%MOD3;       // C modulo
@A               if((k==0)|| (k==1))
@B                   if(temp1==0) a=1;   // A
                 // dummy code to control the distance
@A – 10h         if(k==2)
@B+ 400h             if(temp2==0) a=1;   // B
                 // dummy code to control the distance
@A – 20h         if(k==3)
@B+ 800h             if(temp3 == 0) a=1; // C
                 liter--;
               } while(liter>0);
```

Figure 6.16 Loop BPB replacement policy source code

6.10 Local Predictor

This section verifies the non-existence of any other type of local predictor. We assume that the local predictor would be a two-level predictor where the first stage, a shift history register, is accessed by the branch IP address. The detailed algorithm is not developed here due to certain observations; the smallest possible local predictor should have the first stage of length 4, consequently making the branch with the outcome pattern {T, T, nT, nT} predictable. Any smaller pattern (3 bits) is already predictable by the loop predictor.

The microbenchmark uses a branch with the outcome pattern {T, T, nT, nT} preceded by the number of conditional taken branches. Consequently, not all branch outcomes are predictable by the global predictor. Microbenchmark source code is shown in Figure 6.17.

The microbenchmark produces a certain misprediction rate. The existence of the misprediction rate indicates that the branch predictor does not employ any other local predictor than the loop predictor.

```
int L,a=1;
int long unsigned liter = 1000000;
 do{
   L = (liter%4) >>1;     // pattern {T,T,nT,nT}
   16 x if(a==0) a=1;     // repeat 16 times
   // dummy code to allow branches to retire
   if (L==0) a=1;a=1;
   liter--;
 } while (liter >0);
```

Figure 6.17 Source code for detection of the local predictor

CHAPTER 7

MICROBENCHMARKS FOR THE REVERSE ENGINEERING OF THE INDIRECT PREDICTOR

7.1 Objectives

The goal of this section is to develop an experimental flow and a set of microbenchmarks that will help us determine the organization of the indirect predictor. We expect the indirect predictor organized in a cache-like structure, with each entry keeping an indirect branch target address (we call such a structure Indirect Branch Target Buffer or iBTB). We want to determine iBTB parameters (size, sets, ways, index, tag) and we would like to determine relationship between the indirect BTB and the regular BTB.

7.2 Contributions

We developed an experimental flow and a set of microbenchmarks for determining indirect predictor organization and its associated logic. The experimental

flow and microbenchmarks applied on a Pentium M processor provide the following insights.

1. The indirect branch target buffer (iBTB) is organized as a direct-mapped cache structure with 256 entries. Each entry has a tag field and the target address.
2. The iBTB does not allocate entry for the branches predictable by the regular BTB.
3. The iBTB entry is updated on iBTB hit if the target address was mispredicted.
4. The iBTB entry is allocated on iBTB miss if the target address was mispredicted.
5. The replacement policy works together with the replacement policy in the BTB and interdependencies are not revealed in this thesis.
6. The index and tag fields for accessing the indirect predictor are taken from a path information register (PIR). We determined the size, update policy, type of branches that affect the PIR, and the branch address bits that affect the PIR.
   - The PIR width is 15 bits.
   - The PIR is updated as follows.
     a. Conditional taken branch address bits IP[18:4] are XOR-ed with the original PIR shifted by 2 bit positions to the left.
     b. Indirect branch address bits IP[18:10] and target address bits TA[5:0] are concatenated and XOR-ed with the original PIR shifted by 2 bit positions to the left.
     c. Conditional not taken branches, unconditional branches, branch outcomes, calls and returns do not affect the PIR.
7. The indirect predictor hash function is an XOR between the PIR and the indirect branch IP. Indirect branch address bits IP[18:4] are XOR-ed with the PIR bits

PIR[14:0]. The lower 6 bits of the hash function [5:0] and the highest bit [14] make a tag used in the iBTB lookup. The hash function result bits [13:6] are used as the index in the iBTB.

7.3 Background

An indirect predictor is a hardware structure in branch predictor units dedicated to handling indirect branches. Several academic proposals and patents from industry share a common approach in implementing the indirect predictor [29], [30], [31], [2] . The indirect predictor is a cache structure separated from the regular BTB, called iBTB (Indirect Branch Target Buffer). The iBTB stores target addresses of indirect branch instructions. The iBTB can be indexed either by a portion of a *path information register* (PIR) or by a hash of a portion of the PIR and the indirect branch instruction pointer, as illustrated in Figure 7.1.

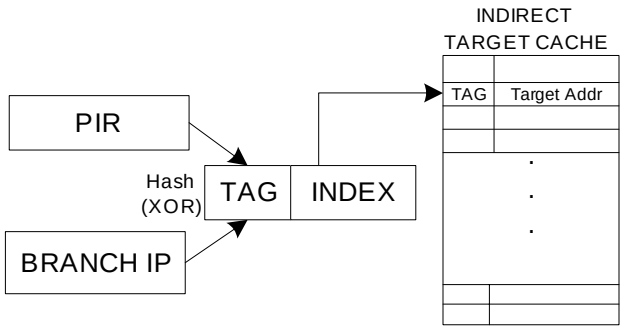

Figure 7.1 Indirect branch target buffer organization

The iBTB can be a direct-mapped or a set-associative cache structure. On an iBTB lookup, the index field is used to select an iBTB entry and the tag field stored in the selected entry is compared with the tag field calculated by a hash function. If we have an iBTB hit, the target address is read from the selected iBTB entry.

The PIR register keeps information about the program path. The PIR can be affected by all branch instructions or by branch instruction of certain types (e.g., conditional taken and indirect branches) and can combine a portion of branch addresses, branch targets, and branch outcomes, or some combination of these elements. Chang [30] and Driesen [29] introduce a PIR that combines branch address or targets bits, but do not include branch outcomes. The PIR is updated as follows: first, the current PIR is shifted for N bits to the left and an N-bit field from the branch address is shifted in. An alternative approach with interleaved shifting and insertion of new bits is used (see Figure 7.3) to achieve deeper history (more branches will influence the index). An Intel patent [31] uses an XOR function of the current PIR and the branch address and/or the branch target address of the current branch. This allows for more branch bits to be used. Three possible PIR update policies have been proposed so far: *Shift and add* [30], *Shift and add with interleave* [29] and *Shift and xor* [31].

PIR update policy: Shift and add. With this update policy, a portion (*N-bits*) of the branch IP or the branch target address is inserted into the PIR. The PIR keeps information about *M* most recently executed branch instructions that affect the PIR, so the total PIR width is N*M bits (see Figure 7.2). A portion of the PIR can be used as the tag and another portion can be used as the index field for the iBTB. Sometimes the PIR can be compressed (the number of bits is reduced) before using the tag and index fields.

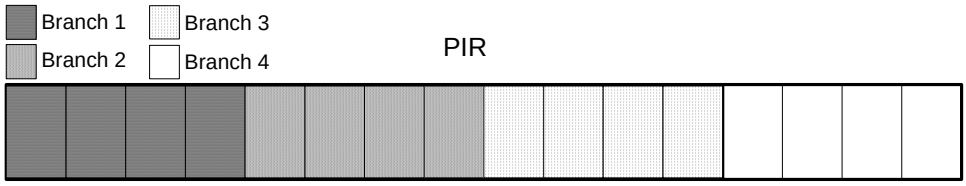

Figure 7.2 Shift and add PIR layout with M=4 and N=4

PIR update policy: Shift and add with interleaving. The *Shift and add* policy suffers from a relatively short history in the index and tag fields. For example, if the PIR from Figure 7.2 is divided into two halves with 8 bits each, and the upper one is used for the index to the iBTB, only two branches, branch 3 and branch 4, will have an effect on the index field. This can be insufficient for a good indexing function. To cope with this problem, an interleaved structure is used (see Figure 7.3). We can see that the previous *M (M=4* in our example*)* branches have an effect on the index to the iBTB.

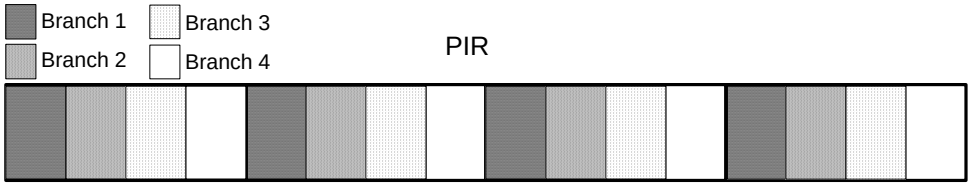

Figure 7.3 Shift and add with interleaving PIR layout with M=4 and N=4

PIR update policy: Shift and xor. With this policy, an incoming branch IP or its target address is XOR-ed with the current PIR. In this way, more bits from the branch address are affecting the PIR. Prior to XORing, the PIR is shifted left/right for a certain

number of bits (shift count). The PIR may be of the same width as the number of branch bits used for the XOR operation or it may be larger. Figure 7.4 shows the PIR of the same width as the number of branch bits used for the PIR.

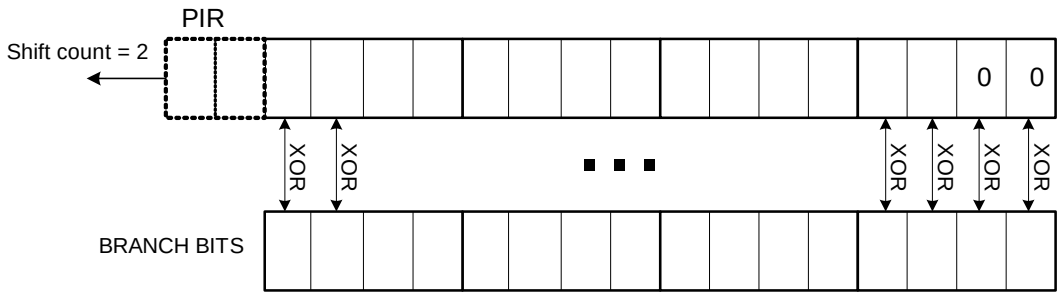

Figure 7.4 Shift and xor PIR layout

One important question related to the iBTB is its relationship with the regular BTB. A 16-byte instruction block fetched from memory triggers a lookup in the regular BTB and concurrently a lookup in the iBTB. If we have a BTB hit (selected entry is tagged as an "indirect" branch and tag field matches) and an iBTB hit, the branch target address can be provided by either the BTB or the iBTB. Gochman *et al.* [2] and an Intel patent [31] describe a BTB filtering methodology as a situation where the iBTB hit is conditional upon a BTB hit. We can have a hit or a miss in the BTB and iBTB, and in the case of hits, the BTBs can give a correct or an incorrect branch target. For each combination, the predictor may perform different update policies such as

- *BTB hit/iBTB hit – The iBTB gives prediction; if the target is mispredicted, update the selected entry in the iBTB; the selected entry in the BTB can be updated or left unchanged.*

- *BTB hit/iBTB miss – The BTB gives prediction; if the target is mispredicted, allocate a new entry in the iBTB; the BTB can be updated or left unchanged. If the BTB gives correct prediction, there will be no allocation in the iBTB.*
- *BTB miss/iBTB miss or hit – there is no prediction; allocate a new entry in the BTB; a new entry in the iBTB may or may not be allocated.*

Based on this preliminary discussion, our goal is to develop an experimental flow and microbenchmarks that uncover details about the indirect branch predictor. We strive to answer the following questions related to indirect branch predictor organization and functioning.

1. *iBTB organization*
    - *Number of entries*
    - *Number of ways*
    - *Number of TAG bits*
2. *PIR organization*
    - *Width (number of bits)*
    - *PIR update policy*
        - *Type of branches affecting PIR*
        - *Type of information used for PIR: Branch target and/or branch IP*
        - *Bits of used branch information affecting PIR*
        - *Bits insertion into PIR (adding, interleaving, xoring)*
        - *Shifting (direction and size)*
        - *Prehistory length (how many branches are seen in prehistory of ind. br)*
3. *Hash function to access iBTB*
    - *How is PIR related to the indirect branch to make an Index and TAG*
    - *What bits of indirect branches are used for hashing function?*
4. *Allocation policy*
    - *Relationships with the BTB*
    - *iBTB hit/miss/mispredicted, BTB hit/miss/mispredicted*

7.4 PIR Organization – Pattern/path Based PIR

The PIR may be affected by either a branch address, referred to as a pattern-based PIR, or by a branch instruction target, referred to as a path-based PIR.

A microbenchmark template used in determining whether the PIR is path- or pattern-based is illustrated in Figure 7.5. The microbenchmark has a spy indirect branch that alternates between two targets, *Target1* and *Target2*. Each target is reached through a unique path. When the program traverses *Path1,* the spy indirect branch target is *Target1*. When the program traverses *Path2,* the spy indirect branch target is *Target2*. Each path consists of N conditional taken branches.

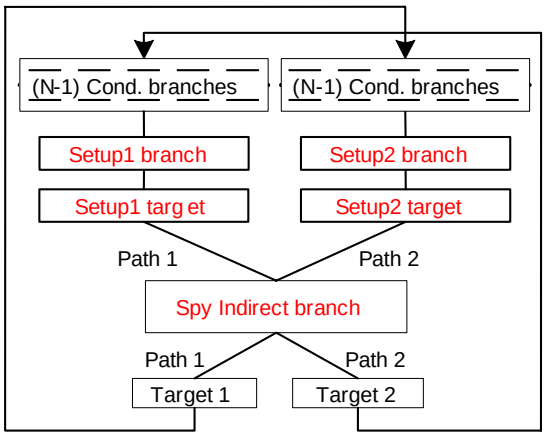

Figure 7.5 Microbenchmark for determining whether the PIR is path-based or pattern-based

The branch instructions are laid in the microbenchmark in such a way that the path-based PIR for *Path1* is equal to the path-based PIR for *Path2*. This is done by

setting branch addresses as follows: *IP(Path1.Bi)=IP(Path2.Bi) + Offset*, where *Offset*$\geq 2^k$ and *k* is the number of bits of the branch address used in calculating the PIR. For example, if we assume that the address bits IP[31:20] are not used in calculating the PIR, the minimum *Offset*=100000h. The number of branches in the paths, *N*, must be determined in such a way that N should be greater than or equal to the number of branches affecting PIR. In this way, the indirect branch targets will collide in the same entry in the iBTB. Consequently, this microbenchmark is forcing iBTB mispredictions.

The next step is to slightly modify the initial microbenchmark so that the path-based PIRs for *Path1* and *Path2* differ. This is accomplished by setting *Setup2* conditional branch in *Path2* to differ from *Setup1* conditional branch in *Path1*. The difference is achieved by changing either *Setup2* lower target bits or *Setup2* lower IP bits. As we do not know exactly which lower address bits are used for the PIR, we change for example lower 15 bits of the *Setup2* IP address or the target address. If *Setup2* differs from *Setup1* in IP address bits and the test produces no misprediction, then the PIR is pattern-based. If *Setup2* differs from *Setup1* in target address bits and the test produces no mispredictions, the PIR is path-based.

The microbenchmark source code is shown in Figure 7.6. It should be noted that the microbenchmark implementation requires one unconditional branch in *Path1* to jump over the *else* portion of the code. Consequently, this microbenchmark will work only if the assumption that unconditional branches do not affect the PIR is correct.

The path-based history test is performed first. The test produces one misprediction per iteration. This is an indication that the branch target bits do not affect the PIR.

Address	Code
	`int long unsigned L, liter = 1000000;`
	`int a=1;`
	`do{`
	`  L = liter%2;`
	`  if (L==0){  // execute one path per iteration`
`@A`	`  8x if(a==0)a=1;  // Repeat the statement 8 times`
`@B,B[14:0]="0"`	`  **if(a==0) a=1;**  // Setup1`
	`}  // Unconditional jump`
	`  else{ // dummy non-branch instructions`
`@A + Offset`	`  8x if(a==0)a=1;`
`@B,B[14:0]="1"`	`  **if(a==0) a=1;**  // Setup2`
	`  }`
	`  jmp dword ptr [ebx]  // Spy branch`
	`  Target1: clc  // ind. target for L==0`
	`  Target2: clc  // ind. target for L==1`
	`  liter --;`
	`} while (liter >0);`

Figure 7.6 Source code of the microbenchmark for determining whether the PIR is path- or pattern-based

The pattern-based test is performed by setting *Setup2*'s 15 lower address bits (IP[14:0]) to differ from the 15 lower address bits (IP[14:0]) of *Setup1*. We observe no mispredictions; hence, we conclude that the pattern-based PIR is used.

Note: Several assumptions are made in this test. If any of them was not correct, we would expect to see a low number of mispredictions in the pattern-based history test because *Path2* would be different from *Path1* even if *Setup1* and *Setup2* target addresses are the same. Therefore, assumptions are hence validated.

7.5 PIR Organization – Conditional Branch IP Address Effect on PIR

This test tries to find (a) branch address bits used for the PIR, (b) the PIR shift count, (c) the PIR history length (PIR.HL) -- the maximum number of branches, prior to

an indirect branch, affecting the PIR, and (d) the PIR width (the number of bits in the path information register).

The branch address bits used for the PIR are found using a test similar to the one used in the pattern-based history test (see Figure 7.5). The algorithm used in Figure 7.5 indicates usage of an unknown number of lower IP bits by setting 15 lower bits of the *Setup2* branch address to differ from the 15 lower bits in the *Setup1* branch address. Here, the algorithm advances by setting just one bit in the *Setup2* branch address to differ from the corresponding bit in *Setup1* branch address (bit *k* in the *Setup1* branch address is set to 0). The particular bit is referred to as *k* (*k*=0, 1, … $\log_2$*Offset*-1), and displacement D is defined as $D=2^k$. Therefore, IP(*Setup2*) = IP(*Setup1*) + D + *Offset.* Consequently, if bit *k* does not affect PIR, this microbenchmark is causing mispredictions in the iBTB.

A similar approach is used for testing the PIR shift count. We assume the *shift and xor* update policy. The algorithm is modified by inserting H conditional branches between the *Spy* and the *Setup1* (or *Setup2*) branches (see Figure 7.7).

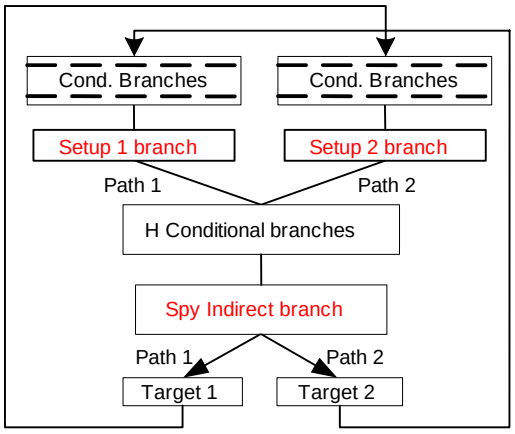

Figure 7.7 Layout of the microbenchmark for determining conditional branch address bits that affect PIR, PIR shifting policy, and PIR history length

These H branches are executed in both *Path1* and *Path2*; therefore, they influence the *Path1* PIR and the *Path2* PIR in the same way, that is, the *Path1* PIR and the *Path2* PIR will differ only if *Setup1* and *Setup2* differ. By increasing H, PIR bits shifted in by *Setup1* or *Setup2* branches are moved further down in the PIR history. The following examples explain the PIR shifting policy in terms of D and H.

Example 1. Let us make the following assumptions: the *shift and xor update policy* is used; the branch address bits IP [11:4] affect the PIR; the PIR width is 8 bits and the PIR shift count is 1.

If H = 0, Setup2's address bits IP[11:4] influence the Path2 PIR prior to the execution of the spy branch.
If H = 1, Setup2's address bits IP[10:4] influence the Path2 PIR prior to the execution of the spy branch.

Example 2. We use the same assumptions as in Example 1, except the PIR shift count which is 2.

If H = 0, Setup2's address bits IP[11:4] influence the Path2 PIR prior to the execution of the spy branch.
If H = 1, Setup2's address bits IP[9:4] influence the Path2 PI R prior to the execution of the spy branch.

In determining the shift count, we can observe from the examples above that it is equal to the difference between the number of address bits of the *Setup2* branch that do affect the PIR when H=0 and H=1. The PIR history length (PIR.HL) is equal to the minimum H for which misprediction is high regardless of parameter D. The following examples explain a process for determining the PIR width.

Example 3. We assume the following: branch address bits used for the PIR are IP[11:4]; the PIR width is 8 bits and the PIR shift count is 1.

- If D = 2^{11}, IP(*Setup2*) [11] (11^{th} bit of the *Setup2* branch address) is XOR-ed with the MSB bit of the PIR. If H = 1, this bit does not influence the *Path2* PIR because it is shifted out by the H_1 branch.
- If D = 2^4, IP(*Setup2*) f, this bit will still influence the *Path2* PIR. When H = 8, the considered bit will be shifted out of the *Path2* PIR.

Example 4. We use the same assumptions as the ones in Example 2, except that the shift count is equal to 2.

- If D = 2^{11} or D = 2^{10}, the relevant PIR bits will be shifted out after one branch in the H block (H=1).
- D = 2^4 or D = 2^5, the relevant PIR bits will be shifted out after 4 branches in the H block (H=4).

Based on the examples above, it is clear that the PIR width can be determined by Equation (7.1).

$$(\textit{shift count}) * \textit{PIR.HL} - (\textit{shift count}) < \textit{PIR Width} \leq (\textit{shift count}) * \textit{PIR.HL}. \quad (7.1)$$

All four questions require observation of the number of mispredictions as a function of parameters H and D. There is one exception where the described test will fail. Let us assume a direct-mapped iBTB and a displacement D affecting only a portion of the PIR used for the iBTB tag. In this case, both indirect branch targets will collide in the same iBTB entry making mispredictions, even though the bit *k* is used for the PIR. The number of mispredictions observed will depend on the iBTB update policy for iBTB miss/misprediction scenario. If bit *k* is not used for the PIR, the number of observed

mispredictions will depend on the iBTB update policy for the iBTB hit/misprediction scenario. These two update policies (iBTB hit/mispredictions and iBTB miss/mispredictions) may differ; in this case, it could be used to make a conclusion about the stated questions. The microbenchmark's source code is shown in Figure 7.8

```
Address              Code
                     int long unsigned L, liter = 1000000;
                     int a=1;
                     do{
                       L = liter%2;
                       if (L==0){           // execute one path per iteration
                          8x if(a==0)a=1;    // Repeat the statement 8 times
@A                        if(a==0) a=1;      // Setup1
                       }                    // Unconditional jump
                       else{ // dummy non-branch instructions
                           8x if(a==0)a=1;
@A + Offset + D            if(a==0) a=1;        // Setup2
                       }
                       H x if(a==0)a=1;       // The H block
                       jmp dword ptr [ebx]    // Spy branch
                       Target1: clc           // Spy target for L==0
                      Target2: clc            // Spy target for L==1
                       liter --;
                     } while (liter >0);
```

Figure 7.8 Source code of the microbenchmark for determining whether conditional branch address bits affect the PIR

We measure the number of mispredicted indirect branches (MIBIE).

Figure 7.9 shows the misprediction rate, calculated as the MIBIE divided with the number of indirect branches, as a function of the parameter D (D=1h–80000h) when H=0. For distances D=400h–20000h, the misprediction rate is zero, indicating that branch address bits IP [18:10] are used for the PIR. For distances 10h–200h and distance 40000h, the misprediction rate is approximately 40%. This is an indication that the

branch address bits IP [9:4] are used for the PIR and that iBTB is a direct-mapped structure; if they are used as an index, each target would have its own iBTB entry and there would be no mispredictions. Finally, if they are not used for the tag and index fields in the iBTB, we would have a misprediction rate of 100%. It is also expected to have tag bits sourced from the lower bits of the PIR as presented in patent [7]. There is not a good explanation for the misprediction rate of 40%. Most likely this number is a consequence of a complex interplay between the regular BTB and the iBTB; therefore, we can expect to have different allocation polices for *iBTB hit/misprediction* and *iBTB miss/misprediction* scenarios.

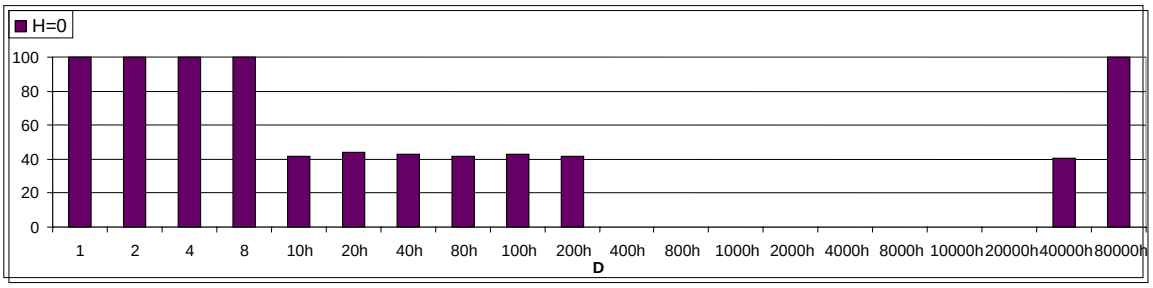

Figure 7.9 Results for detection of conditional branch IP bits effect on PIR test for H=0

Figure 7.10 shows the misprediction rate, calculated as the MIBIE divided with the number of indirect branches, as a function of the parameter D (D=1h–80000h), when H=1. For distances D=100h–8000h, the misprediction rate is zero. When compared to the results for H=0 (Figure 7.9), we see that displacements D producing no mispredictions are shifted to the left for the 2 bit positions. This indicates that *shift count* = 2.

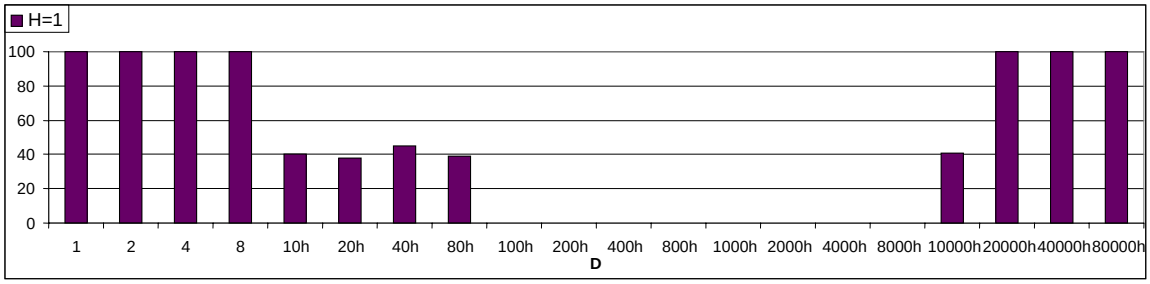

Figure 7.10 Results for detection of conditional branch IP bits effect on PIR test for H=1

Figure 7.11 shows the misprediction rate, calculated as the MIBIE divided with the number of indirect branches, as a function of the parameter D (D=1h–80000h), when H=2…8. The results confirm the previous observation regarding shift count and shift direction. The PIR history length is 8 (for H = 8 for all distances, the number of mispredictions is high).

PIR width. In determining the PIR width, we compare misprediction rates for two (H, D) pairs, when H=7, D=2^4 and when H=7, D=2^5. The latter results in a misprediction rate of 100%, indicating that the PIR is 15 bits long; a 16-bit long PIR would not result in a misprediction rate for D=2^5.

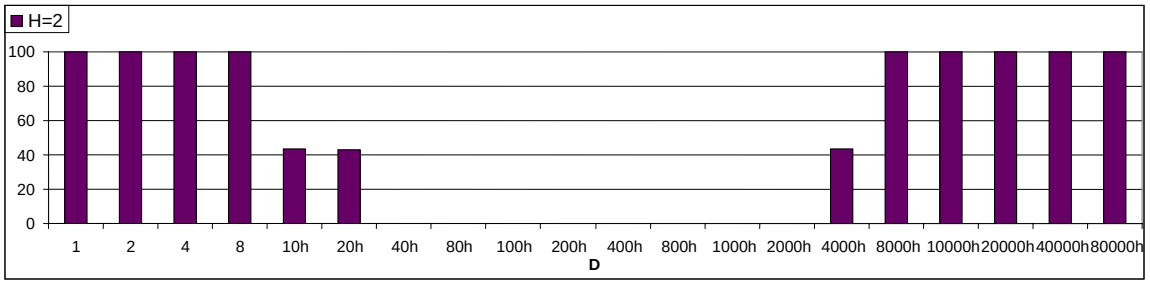

Figure 7.11 Results for detection of conditional branch IP bits effect on PIR test for H=2…8

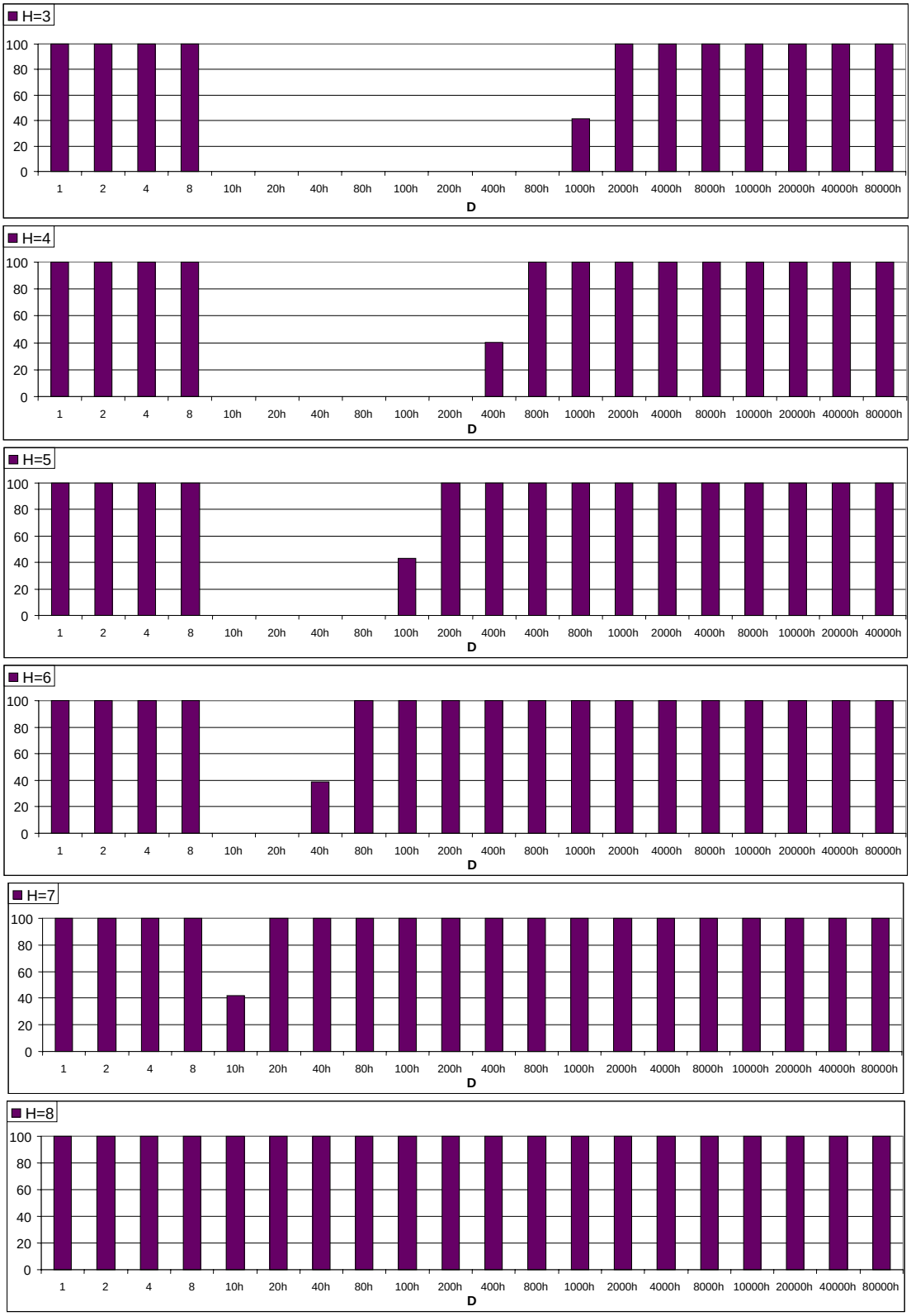

Figure 7.11 (Continued)

It should be noted that the findings about the branch address bits used in the PIR would hold even if our initial assumption about the *shift-and-xor* PIR is not correct. For example, with the *shift-and-add* policy, when H=0, we would observe results equivalent to those shown in Figure 7.9. There, 15 branch address bits IP [18:4] would be compressed to only 2 bits and then they would be shifted into the PIR. However, this policy does not appear to be practical -- there is a very little or no benefit in using 15 address bits that will be compressed to only 2 bits. Consequently, we stay with the original assumption about the *shift-and-xor* PIR.

Further experiments are carried out under the assumption that the PIR update policy is *shift and xor,* but we don't consider this assumption proven yet. Note: The performed test confirms that the number of branch address bits IP[18:4] and the PIR width are the same (15-bit long). Consequently, we can achieve full control over any bit of the PIR in our microbenchmark by controlling the difference between *Setup1* and *Setup2* branch addresses.

7.6 PIR Organization – Type of Branches Used

The goal of this test is to determine which types of branch instructions affect the PIR, apart from the conditional taken branches. We test the following branch types: (a) conditional not taken branches, (b) unconditional branches and (c) call and return jumps.

The test uses a slight modification of the microbenchmark shown in Figure 7.7. The H-block is modified to include a number of conditional taken branches (H_C, $H_C <$ PIR.HL), followed by a number of branches of other types. A number of branches of other types, H_O is set to H_O=PIR.HL –H_C (see Figure 7.12).

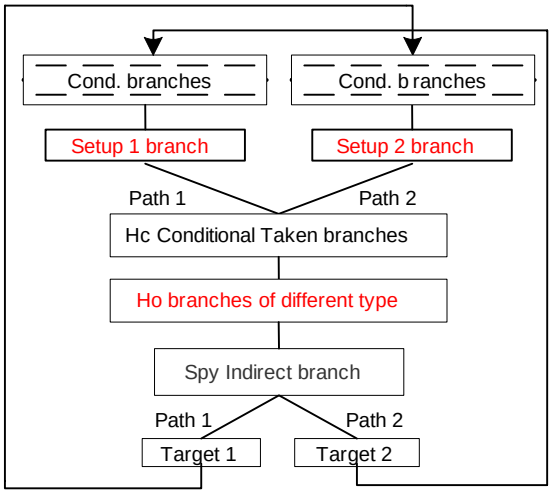

Figure 7.12 Algorithm for determining branch types affecting the PIR test

Path2 and *Path1* are set to be unique in the same way as in the benchmark shown in Figure 7.7, by placing *Setup2* at distance D from *Setup1*, where bit $k=log_2(D)$. If the H_O branches do affect the PIR, *Path2* will be the same as *Path1*, consequently producing indirect mispredictions. If the H_O branches do not affect the PIR, the misprediction rate will be zero.

Not taken direct conditional branches. Figure 7.13 shows a code snippet for the H_C and H_O branches for the microbenchmark described in Figure 7.7 when testing whether always not taken conditional branches influence the PIR.

Unconditional jumps. Figure 7.14 shows a code snippet for the H_C and H_O branches for the microbenchmark described in Figure 7.7 when testing whether unconditional taken branches influence the PIR.

Call/returns. Figure 7.15 shows a code snippet for the H_C and H_O branches for the microbenchmark described in Figure 7.7 when testing whether call/return branches influence the PIR.

```
 a=1;
 ...
 6 x if(a==0) a=1;a=1; // Hc = 6
 2 x if(a==1) a=1;a=1; // Not taken branches, Ho=2
 ...
 jmp   dword ptr [ebx] //spy indirect  branch
```

Figure 7.13 Source code fragment for testing of NT conditional branches effect on PIR

```
6 x if(a==0) a=1;a=1; //Hc = 6
_asm{jmp l1           //unconditional jumps
     l1: clc
     jmp l2
     l2: clc
 }
 jmp   dword ptr [ebx] //spy indirect branch
```

Figure 7.14 Source code fragment for testing of unconditional branches effect on PIR

```
_asm{
    jmp lcc
   _doit1:  mov ebx, 10
   _doit2:  mov ebx, 10
     ret
   lcc: clc
}
do{
   6 x if(a==0) a=1;a=1; //Hc=6
  _asm{                  //call/ret. Branches, Ho=2
   call    _doit1
   call    _doit2
  }
  jmp   dword ptr [ebx] //spy indirect branch
} while(liter>0);
```

Figure 7.15 Source code fragment for testing of call and return branches effect on PIR

All tests produce a misprediction rate of zero, indicating that always not taken conditional, unconditional, and call/return branches do not affect the PIR.

7.7 PIR Organization – Branch Outcome Effect on PIR

An important question regarding the PIR update policy is whether the branch outcomes affect the PIR. In answering this question, we develop a microbenchmark shown in Figure 7.16. The microbenchmark sets two paths to two spy indirect branch targets. *Path1* differs from *Path2* in the *Switch* branch behavior. The *Switch* branch outcome is "taken" for *Path1* and "not taken" for *Path2*. The *Switch* branch address bits do not affect the *Path2* PIR since it is a not taken branch. The *Switch* branch address bits do affect the *Path1* PIR since it is a "taken" branch. Consequently, *Path1* based PIR is affected by the following branches: <*Taken branch 8, Switch branch, Taken branch 7 – Taken branch 2*>. *Path2* based PIR is affected by the following branches: <*Taken branch 8 – Taken branch 1*>.

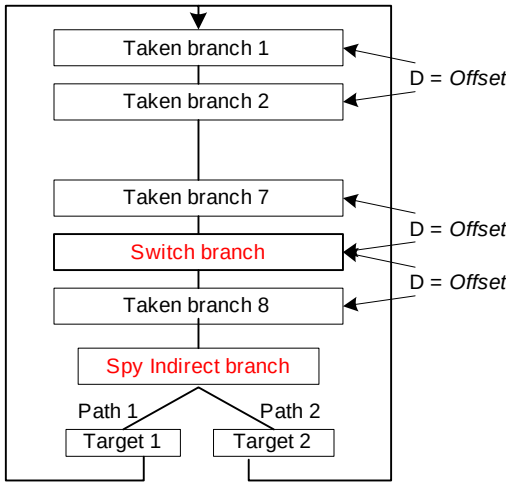

Figure 7.16 Layout of a microbenchmark for determining branch outcome effect on PIR

The microbenchmark sets the *Path2* PIR to be affected by the same branch address bits as the *Path1* PIR, regardless of the different branches' history. This is achieved by placing all branches at the distance *Offset*. Consequently, *Path2* based PIR will differ from the *Path1* based PIR only if the *Switch* branch outcome affects the PIR. We expect the number of mispredictions to be high if branch outcome does not affect the PIR. The microbenchmark's source code is shown in Figure 7.17. The *Offset* value is set to D=80000h.

The test produces a high misprediction rate, indicating that the branch outcome does not affect PIR.

```
Address            Code
                   int long unsigned L, liter = 1000000;
                   int a=1;
                   do{
@A                    if(a==0)a=1;    // Taken branch 1
                      // dummy non-branch instructions
                      ...
@A + 6*Offset         if(a==0)a=1;    // Taken branch 7
                      // dummy non-branch instructions
@A + 7*Offset         if(L==0)a=1;    // Switch branch
                      // dummy non-branch instructions
@A + 8*Offset         if(a==0)a=1;    // Taken branch 8
                      // dummy code to allow branches retirement
                      jmp dword ptr [ebx]    // Spy branch
                      Target1: clc           // Spy target for L==0
                      Target2: clc           // Spy target for L==1
                      liter --;
                   } while (liter >0);
```

Figure 7.17 Source code for determining branch outcome effect on PIR

7.8 PIR Organization – Indirect Branch Target Effect on PIR

We have developed a separate microbenchmark for determining whether the indirect branch target address affects the PIR. The microbenchmark reuses the *pattern-based PIR* test shown in Figure 7.5. The branches *Setup1* and *Setup2*, which were used to make the *Path1* PIR differ from the *Path2* PIR, are replaced with the indirect branches *Indirect1* and *Indirect2* (see Figure 7.18).

Indirect2 and *Indirect1* have target addresses that differ in only one address bit at position *k*. Consequently, if indirect branch target address bit *k* affects the PIR, the *Path2* PIR will differ from the *Path1* PIR and the benchmark will produce a low misprediction rate. To avoid possible indirect branch IP address affecting the PIR, the *Setup2* IP address is set to *Offset* distance from the *Setup1* IP address where *Offset=80000h.* The microbenchmark has the H block introduced in Figure 7.7 to test for target bits shifting through the PIR.

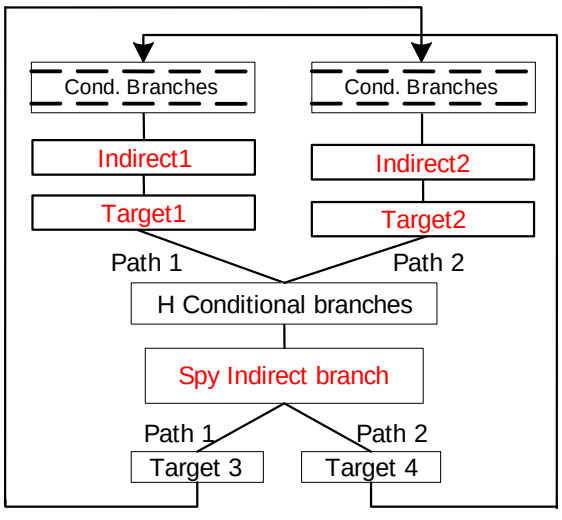

Figure 7.18 Indirect branch target bits effect on PIR test microbenchmark layout

The microbenchmark has two variables, D and H, where D is the difference between *Target1 and Target2* addresses and H is the number of branches in between the spy indirect branch and *Target1* or *Target2*. The information about target bits is determined from experiments varying D and H similarly to the experiments described in Section 7.5. *Indirect1* and *Indirect2* are made to be always mispredicted; therefore, the misprediction rate of 50% means no mispredictions for the spy indirect branch. The microbenchmark's source code is shown in Figure 7.19.

```
Address        Code
               int long unsigned L, liter = 1000000;
               int a=1;
               do{
                 L = liter%2;          // Setup for indirect br. pattern
                 8x if(a==0)a=1;       // Repeat the statement 8 times
                 jmp dword ptr [ebx]   // Setup branch
@A               Target1: clc          // Target1 (exec. when L=0)
@A + D           Target2: clc          // Target2 (exec. when L=1)
                 // dummy non-branch instructions
                 H x if(a==0)a=1;        // The H block
                 jmp dword ptr [ebx]     // Spy branch
                 Target3: clc            // Target3 (exec. when L=0)
                 Target4: clc            // Target4 (exec. when L=1)
                 liter --;
               } while (liter >0);
```

Figure 7.19 Indirect branch target bits effect on PIR test source code

We measure the number of mispredicted indirect branches (MIBIE).

Figure 7.20 shows the misprediction rat, calculated as the MIBIE divided with the number of indirect branches, as a function of the parameter D (D=1h–800h), when H=0. For distances D=1h–20h, the misprediction rate is ~65% indicating that the indirect

branch target bits [5:0] are used for the PIR. We assume that these bits are XOR-ed with the PIR bits [5:0] and are used for the iBTB tag

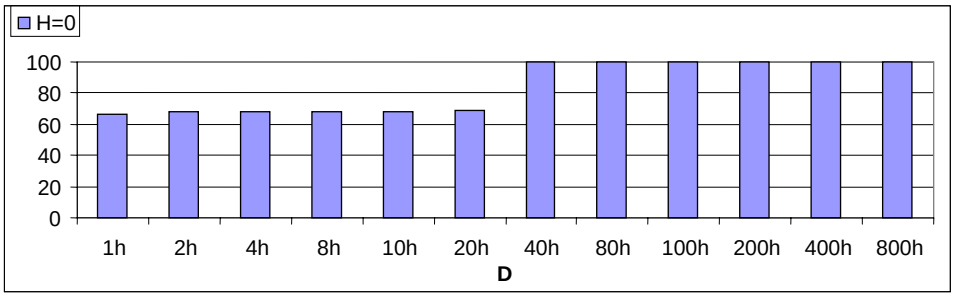

Figure 7.20 Indirect branch target bits effect on PIR test for H=0

Figure 7.21 shows the misprediction rate, calculated as the MIBIE divided by the number of indirect branches, as a function of the parameter D (D=1h–800h), when H=1. For distances D=1h–8h, the number of normalized mispredictions is ~0.65. For distances D=10h–20h, the spy branch produces no mispredictions. The results confirm the previous observation about XORing the indirect target address bits [5:0] with PIR bits [5:0]; target address bits [5:0] for H=1 are moved to the PIR positions [7:2]. PIR bits [7:6] produce no misprediction which is the situation we observed in Section 7.5. We prove the assumption about XORing indirect target address bits [5:0] with the PIR bits [5:0].

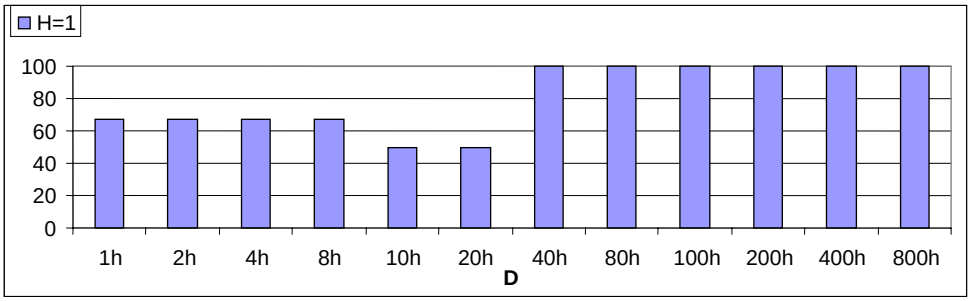

Figure 7.21 Indirect branch target bits effect on PIR test results for H=1

Figure 7.22 shows the misprediction rate, calculated as the MIBIE divided by the number of indirect branches, as a function of the parameter D (D=1h–800h), when H=2…8. The results confirm the previous observation about XORing the indirect target bits [5:0] with PIR bits [5:0]; for each increment of H, we see a shift of two bits through the PIR until for H=8, where all bits are shifted out of the PIR and all branches are mispredicted. As expected, we can see that for H=5, bit 4 reached the PIR highest bit which is assumed to be used for tag match and therefore mispredictions exist.

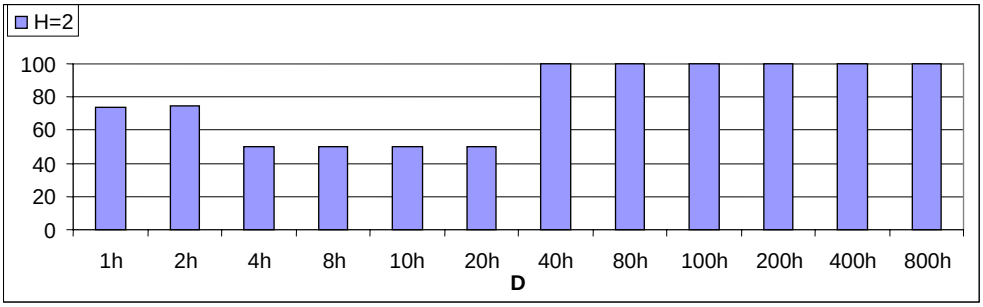

Figure 7.22 Indirect branch target bits effect on PIR test results for H=2…8

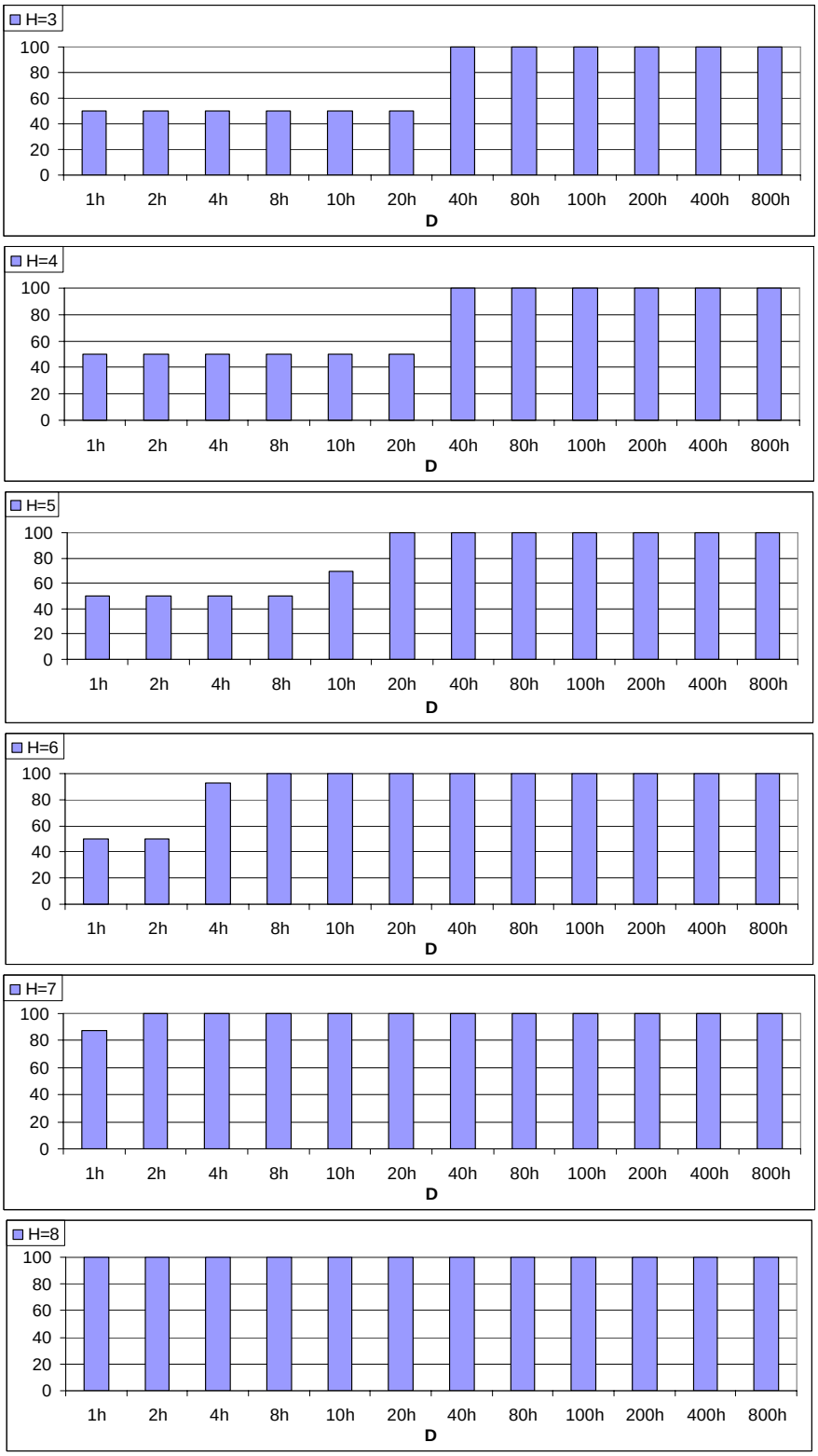

Figure 7.22 (Continued)

7.9 PIR organization – Indirect Branch IP Address Effect on PIR

Another question regarding the PIR update policy is whether addresses of indirect branches affect the PIR. In answering this question we will slightly modify a microbenchmark shown in Figure 7.18. Instead of having indirect branch target *Target1* and *Target2* differ at a particular address bit, *Indirect2* is set to have IP address bit *k* different from the address bit *k* of the *Indirect1* branch. Consequently, if address bits of indirect branches affect the PIR, the *Path2* PIR will differ from the *Path1* PIR and the test will have a low number of mispredictions. Bit *k* is set as follows: IP(*Indirect2*) = D + *Offset* + IP(*Indirect1*), where D =2^k and *Offset=80000h.* The target addresses *Indirect2* and *Indirect1* are set to have lower 6 bits equal to avoid indirect branch target addresses to influence the PIR. The block H is removed and the layout of the microbenchmark is shown in Figure 7.23. The microbenchmark's source code is shown in Figure 7.24.

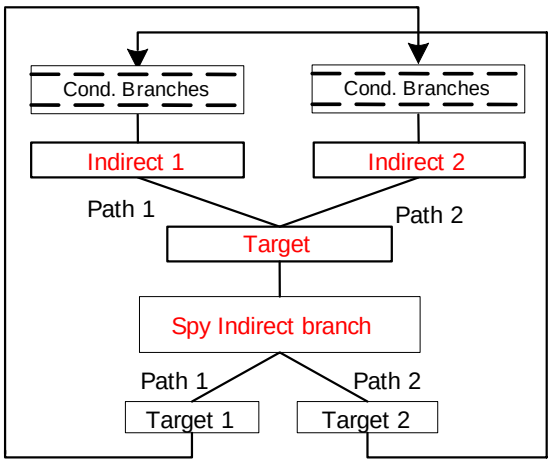

Figure 7.23 Layout of a microbenchmark for determining whether indirect branch address bits affect the PIR

Address	Code
	`int long unsigned L, liter = 1000000;`
	`int a=1;`
	`do{`
	`  L = liter%2;  // Setup for indirect br. pattern`
	`  if (L==0){`
	`    8x if(a==0)a=1;  // Repeat the statement 8 times`
@A	`    jmp dword ptr [ebx] // Indirect1`
@B	`    Target: clc  // Target`
	`  }`
	`  else{ // dummy non-branch instructions`
	`    8x if(a==0)a=1;  // Repeat the statement 8 times`
@A + Offset + D	`    jmp dword ptr [ebx] // Indirect2`
@B + Offset	`    Target: clc  // Target`
	`  }`
	`  jmp dword ptr [ebx]  // Spy`
	`  Target1: clc  // Target1 (exec. when L=0)`
	`  Target2: clc  // Target2 (exec. when L=1)`
	`  liter --;`
	`} while (liter >0};`

Figure 7.24 Source code of a microbenchmark for determining which indirect branch address bits affect the PIR

We measure the number of mispredicted indirect branches (MIBIE). Figure 7.25 shows the misprediction rate, calculated as the MIBIE divided by the number of indirect branches, as a function of the distance D (D=10h–80000h). The results indicate that the indirect branch IP address bits [18:10] are affecting the PIR. In Section 7.8 we observed that the indirect branch target bits [5:0] are affecting the PIR. The number of mispredictions in both cases allows us to implicitly conclude that the indirect branch IP address bits [18:10] are concatenated with the indirect branch target bits [5:0] and XOR-ed with the PIR due to the similarity to the results in Section 7.5. For the same reason, the block H and appropriate testing is considered redundant and is not included here.

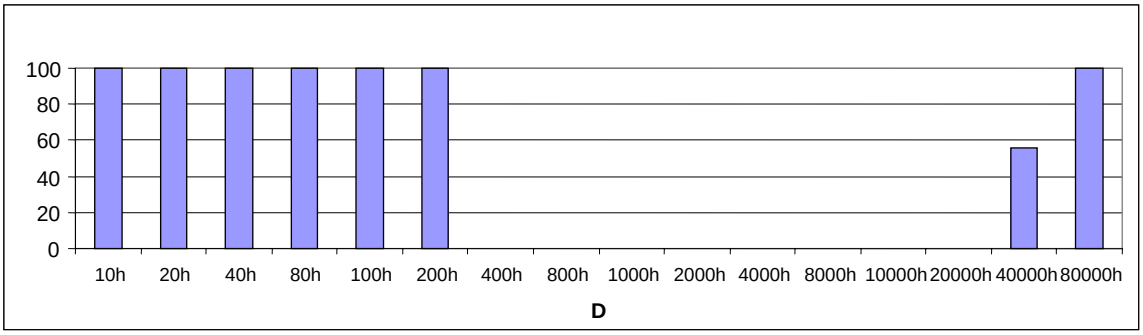

Figure 7.25 Results for detection of indirect branch IP bits effect on PIR test

7.10 PIR Organization – Update Policy

We have mentioned several PIR update policies, such as *shift and add, shift and add with interleaving* and *shift and xor*. Section 7.5 indicates that likely the *shift and xor* policy is used in Pentium M. Here we verify this assumption in a different way, using an alternative approach and developing a new microbenchmark. Before describing the experimental approach, let us walk through the following example.

Example. Assume an 8-bit PIR that uses branch address bits IP[11:4], and the shift count is 1.

- After *Branch1* is executed: *PIR = IP(Branch1*) *[11:4].*
- Prior to *Branch2* PIR is shifted first: *PIR = IP([Branch1] [10:4, '0'].*
- After *Branch2* is executed: *PIR = [IP(Branch1) [10]:* ***IP(Branch1) [4],*** *'0'] xor [IP(Branch2) [11]:* ***IP(Branch2) [5]****, IP(Branch2) [5]]*

Effectively, we consider that IP(*Branch1*)[4] and IP(*Branch2*) [5] are XOR-ed.

The microbenchmark shown in Figure 7.26 reuses the pattern-based microbenchmark from Figure 7.5. It includes two new branches, *Setup3* and *Setup4* that precede *Setup1* and *Setup2* from the pattern-based algorithm.

Following Example 1, we set the conditions for the test:

- *If Setup1[4] = Setup3[5] = '1', after both branches are executed, PIR[1] = '0'.*
- *If Setup2[4] = Setup3[4] = '1', after both branches are executed, PIR[1] = '0'.*

The test results in the *Path2's* PIR are the same as the *Path1's* PIR, and consequently produces mispredictions, but the conditional branches affecting the PIR are different.

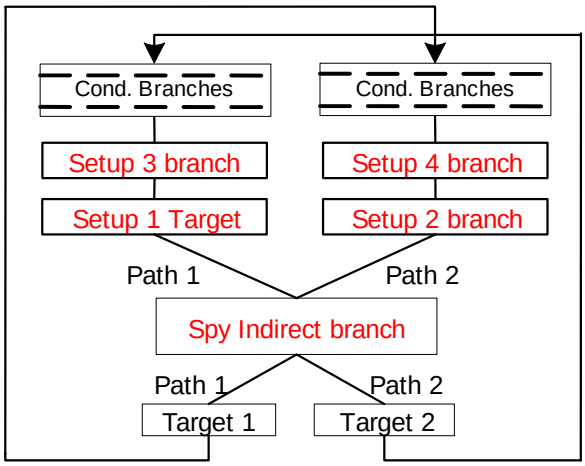

Figure 7.26 Layout of a microbenchmark for determining PIR update policy

The microbenchmark is designed to work with any shift count even though we have found that *shift count* = 2. Two variables are introduced as follows:

N1; $N1 = Setup3 - Setup1$

N2; $N2 = Setup4 - Setup2$

By changing *N2* and *N1,* we match the *Path2's* PIR to be equal to the *Path1's* PIR.

The microbenchmark source code is shown in Figure 7.27.

```
Address              Code
                     int long unsigned L, liter = 1000000;
                     int a=1;
                     for(i=0;i<liter;i++){
                       L=i%2;
                       if(L==0){
@A                        6x if(a==0)a=1;     // Repeat the statement 6 times
@B                        if(a==0)a=1;        // Setup3
@C                         if(a==0)a=1;        // Setup1
                       } else { // dummy non-branch instructions
@A + Offset               6x if(a==0)a=1;     // Repeat the statement 6 times
@B + Offset               if(a==0)a=1;        // Setup4
@C + Offset                if(a==0)a=1;        // Setup2
                       }
                       // dummy non-branch instructions
                       jmp dword ptr [ebx]  // Spy
                       _0: clc         // Target1 (exec. when L=0)
                       _1: clc         // Target2 (exec. when L=1)
                     }
```

Figure 7.27 Source code of the microbenchmark for determining PIR update policy

We measure the number of mispredicted indirect branches (MIBIE). Figure 7.28 shows the misprediction rate, calculated as the MIBIE divided by the number of indirect branches, as a function of the parameters *N1* and *N2*. Both tests verify our assumptions: the PIR update policy is *shift and xor* and the *shift count* = 2.

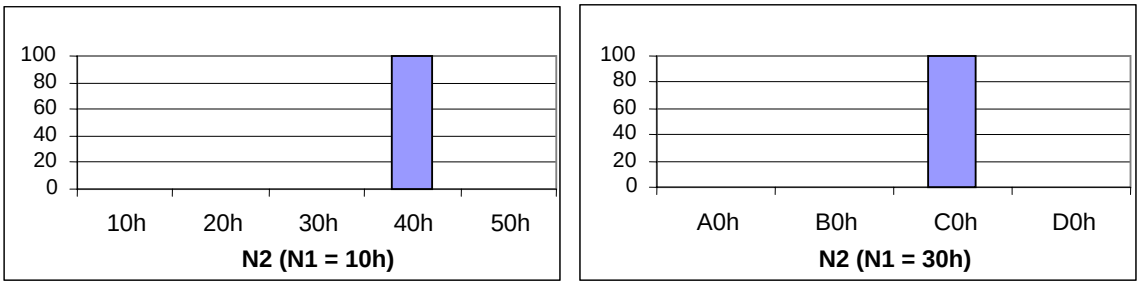

Figure 7.28 PIR shift and XOR update logic test results for N1=10h, 30h

7.11 Indirect Branch IP Effect on iBTB Access Hash Function

The index and tag fields for an iBTB lookup are a hash function of the indirect branch address bits and the current PIR register. Here we want to verify a hypothesis that the index and tag fields are determined by XORing the branch address and the PIR. First, we use a microbenchmark that tries to determine what address bits of an indirect branch are used for the iBTB access hash function. The microbenchmark includes two indirect spy branches, called *Spy1* and *Spy2*. We ensure that the PIR seen by *Spy1* is equal to the PIR seen by *Spy2*. Consequently, the hash functions for *Spy1* and *Spy2* will depend on the branch address of these two spy branches only (see Figure 7.29).

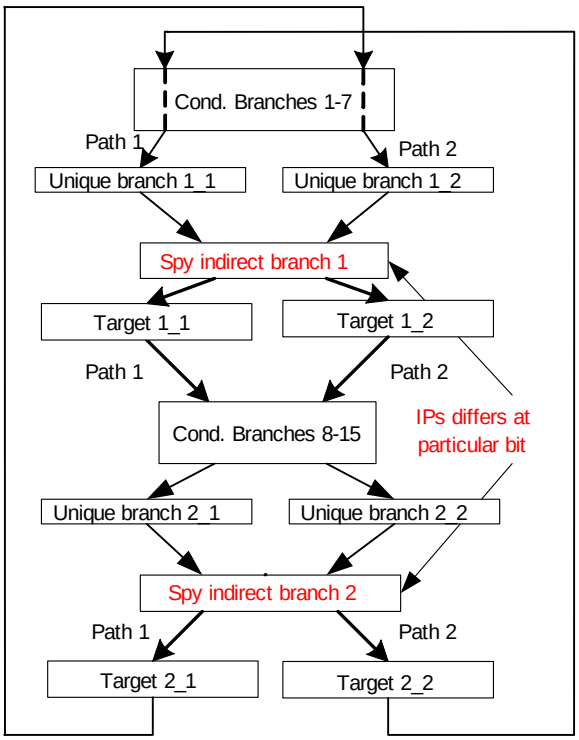

Figure 7.29 Layout of microbenchmark for determining Indirect branch IP address effects on hash function

It should be noted that *Spy1* and *Spy2* must have two targets if we want to cause collisions in the iBTB. Otherwise, the regular BTB will correctly predict both *Spy1* and *Spy2* target addresses. *Spy1* target addresses are *Target1_1* and *Target1_2*. *Spy2* target addresses are *Target2_1* and *Target2_2.*

The microbenchmark must ensure that *Spy1* sees two unique values in the PIR, depending on the execution path, one for *Path1* (the target address is *Target1_1*) and one for *Path2* (the target address is *Target1_2*). The same requirement must be satisfied for *Spy2*. The microbenchmark relies on placing unique branches *Unique1_1, Unique1_2, Unique2_1* and *Unique2_2* at such addresses to ensure unique PIRs for each path. The placement of unique branches must be such to satisfy the following requirements: the PIR seen by *Spy1* in *Path1* must be the same as the PIR seen by *Spy2* in *Path1* and the PIR seen by *Spy1* in *Path2* must be the same as the PIR seen by *Spy2* in *Path2*. This can be achieved with the following branch placement.

IP(Unique2_1) = IP(Unique1_1) + Offset; Offset = 800000h;
IP(Unique2_2) = IP(Unique1_2) + Offset; Offset = 800000h.

The next step in the setup is to place the spy indirect branches at a controlled distance: IP(*Spy2*) = IP(*Spy1*) + D, where $D=2^k$, and bit *k* is cleared in *Spy1* address. If the bit *k* of the *Spy2* branch is used for the hashing function, we should observe no mispredictions, and for the opposite, if bit k is not used for the hashing function, we should observe a high number of mispredictions. The microbenchmark source code is shown in Figure 7.30.

Address	Code
	`int long unsigned L,liter = 1000000;`
	`int a=1;`
	`do{`
	`  L = liter%2;`
	`  7x if(a==0)a=1;`
	`  if(L==0)a=1;  // make Unique1_1 & Unique1_2`
@A	**`  if(a==0)a=1;`** `// Unique1_1 & Unique1_2`
@B	`  jmp dword ptr [ebx]  // Spy1`
	`  _111: clc  // Target1_1`
	`  _112: clc  // Target1_2`
	`  // dummy non-branch instructions`
	`  7x if(a==0)a=1;`
	`  if(L==0)a=1;  // make Unique2_1 & Unique2_2`
@A + Offset + D	**`  if(a==0)a=1;`** `// Unique2_1 & Unique2_2`
@B + Offset	`  jmp dword ptr [ebx]  // Spy2`
	`  _121: clc  // Target2_1`
	`  _122: clc  // Target2_2`
	`  liter --;`
	`} while (liter >0);`

Figure 7.30 Indirect branch IP effect on hash function test source code

We measure the number of mispredicted indirect branches (MIBIE). Figure 7.31 shows the misprediction rate, calculated as the MIBIE divided by the number of indirect branches, as a function of the parameter D (D=1h-80000h). The results indicate that 14 indirect branch IP address bits are affecting the hash function.

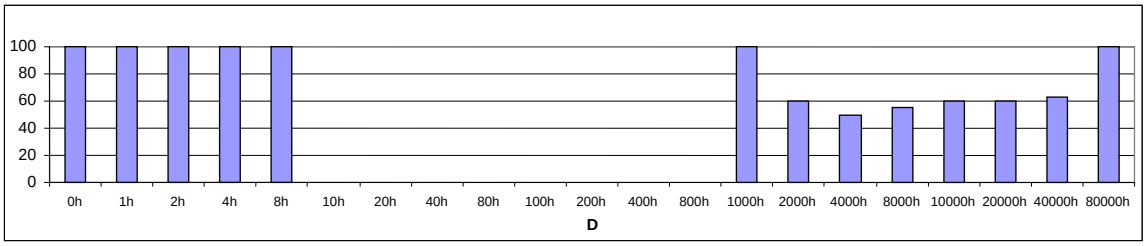

Figure 7.31 Indirect branch IP effect on hash function test results

For D = 10h–800h the test produces no mispredictions. This is an indication that particular bits are used as the index bits to the iBTB according to the observation made for the PIR bits in Section 7.5. For D = 2000h–40000h, the misprediction rate is ~60%. This is an indication that particular bits are used as the tag bits to the iBTB according to the observation made for the PIR bits in Section 7.5. For D = 1000h, the misprediction rate is 100%. Due to similarity to the PIR bits, we expect this bit to be only a part of the tag field or unused at all.

7.12 iBTB Access Hash Function

The iBTB access function is usually a XOR between the PIR and the part of the indirect branch IP address. We have determined the size and update policy for the PIR and indirect branch address bits used in the hash function. The goal here is to develop a microbenchmark that will help determine the hash function. The PIR and address bits can be XOR-ed without folding (e.g., PIR[14:0] is combined with IP[18:4]) or with folding (PIR[14] is combined with IP[4], etc.). The challenge is to find the hash function without a direct control over the PIR. Rather, the PIR is controlled through employment of a number of branches as explained earlier.

The microbenchmark is based on the microbenchmark shown in Figure 7.29 and is shown in Figure 7.32. A parameter in this benchmark is the distance D_{IP} between *Spy1* and *Spy2* indirect branches, D_{IP} = IP(*Spy2*) – IP(*Spy1*). Another parameter is the distance between the branches *Unique1_2* and *Unique2_2* or *Unique1_1* and *Unique2_1* D_{PIR} = IP(*Unique2_X*) – IP(*Unique1_X*). $k_{IP} = \log2(D_{IP})$ *and* $k_{PIR} = \log2(D_{PIR})$.

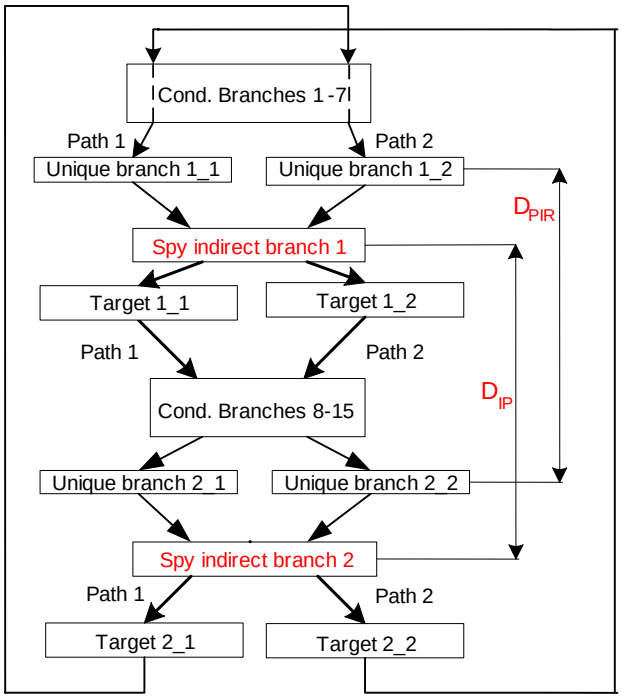

Figure 7.32 Layout of microbenchmark for determining iBTB hash access function

The following example illustrates the intricacies of the microbenchmark.

Example. Let us assume that PIR[1] is XOR-ed with the indirect branch address IP [4] to make a portion of either the tag or index field for the iBTB. IP(*UniqueX_X*)[4] is XOR-ed with the PIR[1]. The PIR values are the same in *Path2* and *Path1* prior to execution of *Unique1_2* and *Unique2_2*:

IP(Unique1_2) [4] ='0';

IP(Spy1) [4] ='0'

Produced hash function bit 1 for *Path2* is '0'.

IP(Unique2_2)[4] ='1';

IP(Spy2)[4] ='1'

Produced hash function bit 4 for Path2 is '0'.

Although the Path2's PIR differs from the Path1's PIR and the indirect branch IP address for Path1 differs from the indirect branch IP address for Path2, the hash function for both Path1 and Path2 will target the same iBTB entry resulting in mispredictions.

The microbenchmark finds the misprediction rate as a function of D_{IP} and D_{PIR}, where D_{IP} and D_{PIR} are distances of the changed bits k_{IP} and k_{PIR} :

IP(Unique2_X) = IP(Unique1_X) + D_{PIR}+ Offset; Offset = 80000h, $k_{PIR=}$ log_2 (D_{PIR})
IP(Spy2) = IP(Spy1)+ D_{IP} + Offset; Offset = 80000h, $k_{IP=}$ log_2 (D_{IP}).

If the misprediction rate is high, the PIR bit k_{PIR} is XOR-ed with the indirect branch address bit k_{IP}.

The microbenchmark's source code is shown in Figure 7.33.

```
Address          Code
                 int long unsigned i,k,l, liter = 1000000;
                 int a=1;
                 for(i=0;i<liter;i++){
                    L = i%2;
                    k = (i%(2*N))>>1; // execute 2x one spy branch consecutively
                    if (k==0){  // execute one target per iteration
@A                      7x if(a==0)a=1;
                        if(L==0)a=1;          // make Unique1_1 & Unique1_2
@B                      if(a==0)a=1;             // Unique1_1 & Unique1_2
@C                      jmp dword ptr [ebx]   // Spy1
                        _111: clc             // Target1_1
                        _112: clc             // Target1_2
                    }
                    else if (k==N){
@A + Offset             7x if(a==0)a=1;
                        if(l==0)a=1;          // make Unique2_1 & Unique2_2
@B + DPIR               if(a==0)a=1;             // Unique2_1 & Unique2_2
@C + DIP                jmp dword ptr [ebx]   // Spy2
                        _l21: clc             // Target2_1
                        _l22: clc             // Target2_2
                    }
                 }
```

Figure 7.33 Source code of the microbenchmark for determining iBTB hash function

Table 7.1 shows values of D_{PIR} and D_{IP} that matched, consequently producing a misprediction rate of 100%. The iBTB access function is illustrated in Figure 7.34

Table 7.1 PIR bits and indirect branch IP bits that XOR in iBTB hash access function

D_{PIR}	D_{IP}	Misprediction Rate
10h	2000h	100%
20h	4000h	100%
40h	8000h	100%
80h	10000h	100%
100h	20000h	100%
200h	40000h	100%
400h	10h	100%
800h	20h	100%
1000h	40h	100%
2000h	80h	100%
4000h	100h	100%
8000h	200h	100%
10000h	400h	100%
20000h	800h	100%
40000h	1000h	100%

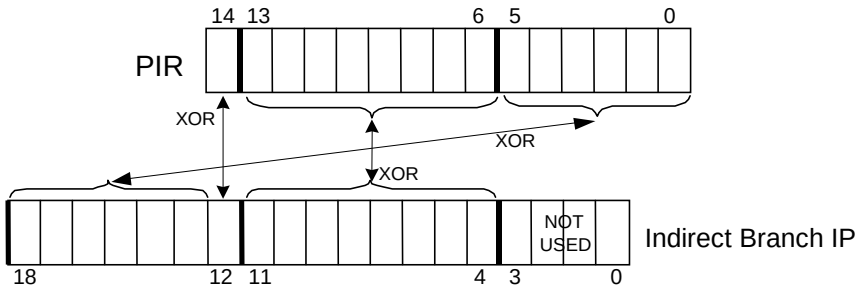

Figure 7.34 PIR bits and indirect branch IP bits that XOR in iBTB hash access function

7.13 iBTB Organization

This test tries to find which bits of the hash function are used for the tag and which for the index to the iBTB and provides an indication about the number of iBTB ways and sets. The approach is similar as in the pattern-based test (see Figure 7.5). There, the microbenchmark employs two indirect branch targets. By setting the spy indirect branch to have two PIR values prior to execution of each target, targets will be allocated in different iBTB entries.

The spy indirect branch in this microbenchmark must include N targets (N>2). Each of the N targets will target a different iBTB entry. The total number of targets that should be used will be different in different phases of the algorithm and will go up to the total number of the iBTB entries. To target a different iBTB entry, a similar idea is used as in Figure 7.5; each of the N targets must have a different PIR value prior to execution of the spy indirect branch.

The algorithm source code implementation is problematic. The microbenchmark in Figure 7.5 is implemented by shifting *Path2* from *Path1* at distance *Offset*. Consequently, the microbenchmark here would have to use N-1 offsets, which are not feasible to implement. To remedy this problem, two indirect branches are introduced. Setup indirect branch, *Setup,* serves to make N different paths; *Setup* has N targets and each target has one conditional branch within its target. These conditional branches are named *Unique0 – UniqueN* (see Figure 7.35*)*. *Unique0-UniqueN* help to achieve different N paths in the same way *Unique0 –Unique2* did in Figure 7.5. Each of *Unique0 – UniqueN* sets the PIR value to be different prior to execution of the spy indirect branch by setting *Unique0-UniqueN* at distance D from each other; consequently, $D = 2^k$.

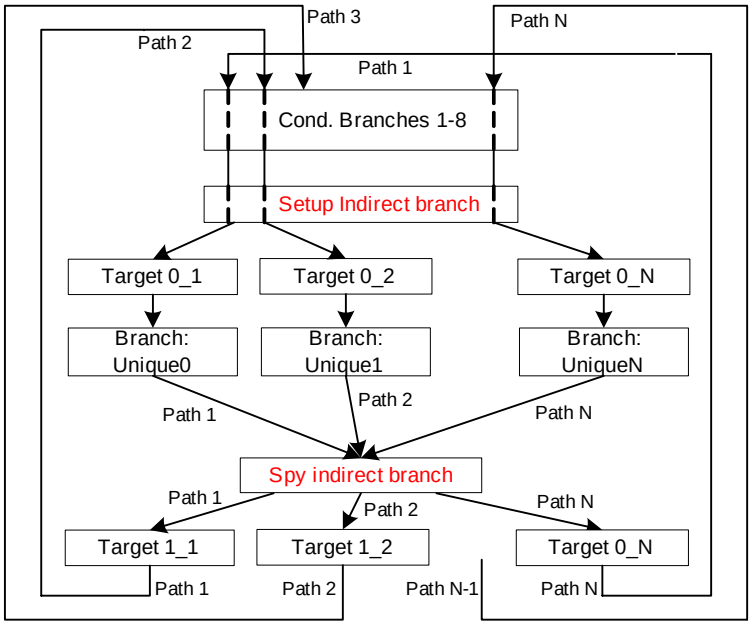

Figure 7.35 Layout of the microbenchmark for detection of iBTB organization

The setup indirect branch should consume only one iBTB entry. Therefore, *Setup* is preceded with the number of always taken branches. The iBTB entry reserved for the setup indirect branch may cause a collision with targets of the spy indirect branch, producing unwanted misprediction. During result analysis, we will identify and isolate these mispredictions.

We observe the misprediction rate as a function of D and N to make a conclusion on the iBTB number of sets and ways and the index bits and tag bits. The following example explains details:

- 4-way iBTB uses 7 lower PIR bits for the index, 6 consecutive for the tag:
  - *For D = 10h (lowest PIR bit used for the index), N up to $4*2^7$ will not produce mispredictions.*
  - *For D = 20h, N up to $4*2^6$ will not produce mispredictions.*

- 4-way iBTB uses 6 lower PIR bits for the tag, 7 consecutive for the index:
  - *For D= 10h (lowest PIR bit used for the index), N up to 4 will not produce mispredictions*
  - *For D= 2^{10}, N up to 2^{7} will not produce mispredictions.*
  - *For D = 2^{9}, N up to 2* 2^{7} will not produce mispredictions.*
  - *For D = 2^{8}, N up to 4* 2^{7} will not produce mispredictions.*

By observing the misprediction rate as a function of D and N, index and tag bits can be determined. Generalization of the example is not given and details will be explained during results analysis.

The microbenchmark source code is shown in Figure 7.36.

Address	**Code**
	`int long unsigned L,liter = 1000000;`
	`int a=1;`
	`do{`
	`L = liter%N;`
	`8x if(a==0)a=1;`
	`jmp dword ptr [ebx] // Setup branch`
`@A`	`_101: // Target0_1 (exec. when L=0)`
`@B`	**`jne l1;`** `// Unique0`
`@A+ 40h(80h,100h,…)`	`_102: clc // Target0_2 (exec. when L=1)`
`@B + D`	**`jne l1;`** `// Unique1`
	`...`
`@A+ 40h(80h,100h,…)`	`_10N: // Target0_N (exec. when L=N)`
`@B + (N-1)*D`	**`jne l1;`** `// UniqueN`
	`l1: clc`
	`jmp dword ptr [ebx] // Spy`
	`_111: clc // Target1_1 (exec. when L=0)`
	`_112: clc // Target1_2 (exec. when L=1)`
	`...`
	`_11N: clc // Target1_N (exec. when L=N)`
	`liter --;`
	`} while (liter >0)`

Figure 7.36 Source code of the microbenchmark for detection of iBTB organization

NOTE: The source code is unable to test for D = 40h and lower due to the fact that the target bits of the setup branch must be at D=40h as the minimum distance. Fortunately, for a small number of branches such as B=2 and B=3, code can be adjusted to achieve small D values, and, moreover, B=2, 3 will be enough to make conclusions.

We measure the number of mispredicted indirect branches (MIBIE).

Figure 7.37 shows the misprediction rate, calculated as the MIBIE divided by the number of indirect branches, as a function of the parameter D (D=10h–80000h) for B=2. Since B=2, these results are the same as results shown in Figure 7.9 and are discussed again as assumptions are made for further analysis. For D=400h–20000h, the spy branch produces no misprediction; this is an indication that the PIR bits [13:6] are the index bits in the iBTB. For D=10h–200h and D=40000h, the spy branch produces mispredictions, this is an indication that the PIR bits [5:0] are the tag bits in the iBTB.

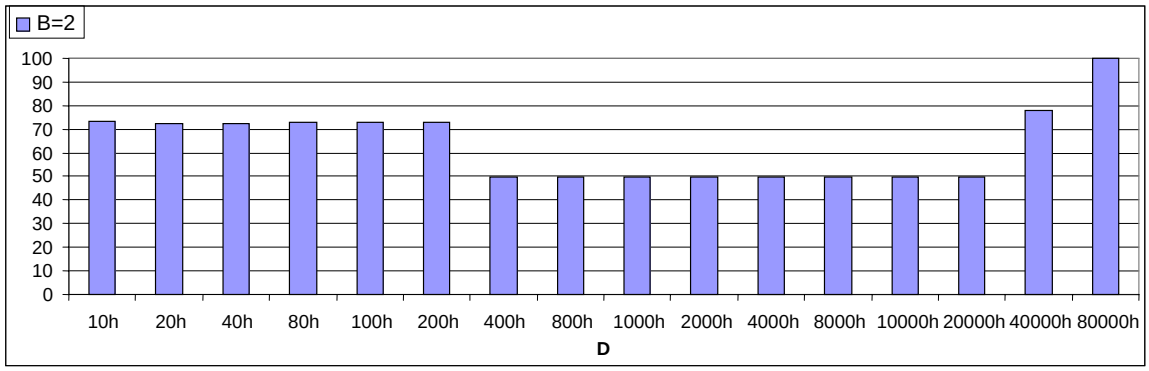

Figure 7.37 Results for detection of iBTB organization test for B=2

Figure 7.38 shows the misprediction rate, calculated as the MIBIE divided by the number of indirect branches, as a function of the parameter D (D=10h–80000h) for B=3.

For assumed PIR index bits PIR[13:6], the spy branch produces no mispredictions (D=400h–10000h) as expected. Note: For D=20000h and D=40000h, the spy branch produces the mispredictions because the 3rd spy target is effectively on distances D=40000h (assumed tag) and D=80000h (not used) respectively.

For assumed PIR tag bits PIR[5:0] (D=10h–200h), the spy branch is always mispredicted. This is an indication that the iBTB or/and the BTB update policy is based on *Allocate on 2nd misprediction* policy. *Allocate on 2nd misprediction* policy allows for two misprediction before the mispredicted entry updated and in the case of three mispredictions in a round robin manner, each target is mispredicted.

Results for tests where B>3 are not necessary for D<400h since Figure 7.38 shows that the misprediction rate is already 100% for B=3 and D<400h.

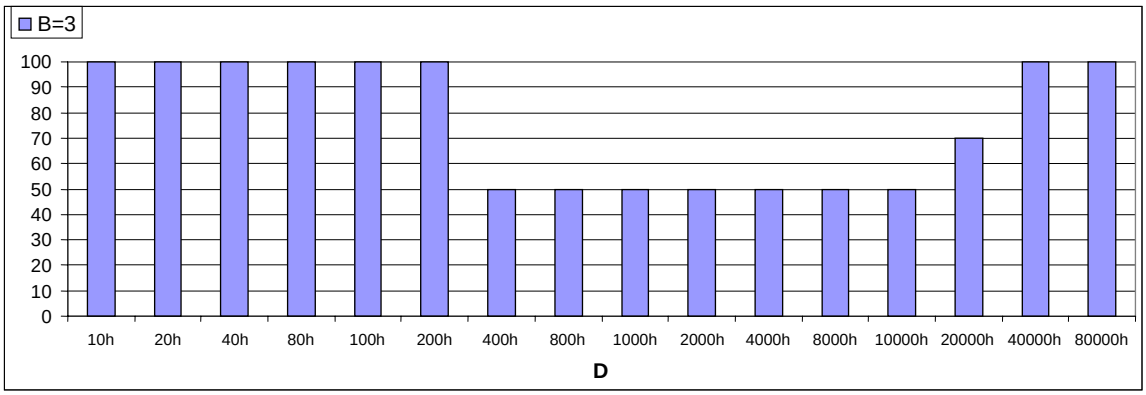

Figure 7.38 Results for detection of iBTB organization test for B=3

Figure 7.39 shows the misprediction rate calculated as the MIBIE divided by the number of spy branches as a function of the parameter B (B= 2–260) for D=400h. Results indicate that the iBTB can fit 256 branches per set. Since a direct-mapped structure is likely to be used, final indication is that the iBTB size is 256 entries.

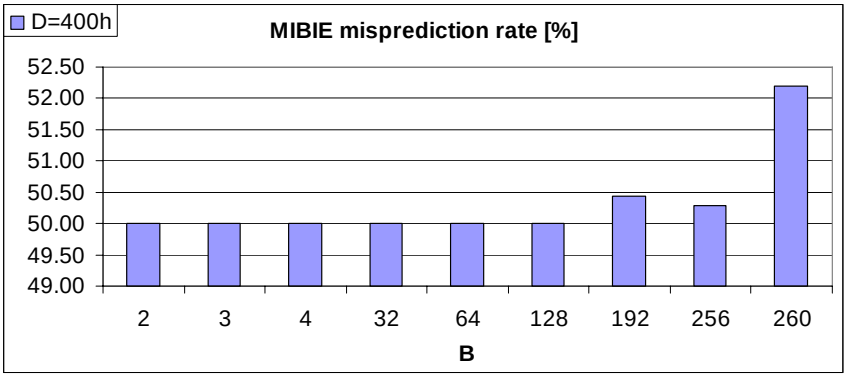

Figure 7.39 Results for detection of iBTB organization test for D=400h

NOTE: For B=192, the number of mispredictions is higher than expected. This is caused by the contention of one of the spy branch targets with the setup indirect branch entry since detailed calculation shows that mispredictions overhead are caused by two branches. Also, for B=256, this overhead is still proportional to the value of 2 branches.

7.14 iBTB Relations with the BTB

Indirect branch prediction from the BTB. First, we verify the assumption taken at the very beginning that the BTB provides the prediction for the monotonic indirect branch. The microbenchmark in Figure 7.31 is reused. Two indirect branches are set to have two different targets. A test is set to have *Path1* and *Path2* the same,

consequently producing mispredictions. A setup for each spy branch is changed to execute one target only. New microbenchmarks still produce contentions in the iBTB. The microbenchmark did not produce mispredictions, indicating that the BTB is used for the prediction of the monotonic indirect branches

iBTB miss/ misprediction. In Section 7.5 we concluded that the particular bits of the PIR are used as the tag in the iBTB. We made the assumption that the situation *iBTB miss/ misprediction* has a specific allocate policy based on an *allocate on iBTB on 2nd misprediction policy* basis. We further investigate this case. A new test reuses the algorithm from Section 7.5 with H=0 and D=2 with the test changed to execute the first target multiple times and the second target once.

We set pattern to execute the first target three times and the second target once. We observe a misprediction rate of 50%. It is obvious that the both second target and the very next first target occurrence is mispredicted. We conclude that the iBTB *allocate on 2nd misprediction policy* is not used. Consequently, we conclude that the both iBTB miss and the BTB have the *allocate on first misprediction allocation policy.*

The reason for the normalized number of mispredictions of 0.4 in Section 7.5 is seen in the specific iBTB dependencies to BTB. For example, if the allocation/replacement polices for the BTB and the iBTB direct each structure to update their entries, one of them may skip its update since the other structure will possibly predict correctly after an update. Consequently, there is a reduction in the number of mispredictions. We don't investigate this case further.

CHAPTER 8

MICROBENCHMARKS FOR THE REVERSE ENGINEERING OF THE GLOBAL PREDICTOR

8.1 Objectives

The goal of this section is to develop an experimental flow and a set of microbenchmarks that will help us determine the organization of the global and bimodal outcome predictors.

8.2 Contributions and Findings

We have developed an experimental flow and a set of microbenchmarks for determining the organization of the global and bimodal predictors. The experimental flow and microbenchmarks applied on a Pentium M processor provide the following insights. The outcome predictor is a multi-layer structure encompassing a global predictor and a bimodal predictor. The global predictor is organized as a cache-like structure, indexed by a portion of the path information register. More specifically, we have made the following findings.

1. The index and tag fields used for access to the global outcome predictor are based on the path information register (PIR), described in Section 7.2. The size and the update policy of the PIR are verified using a new set of microbenchmarks that feature only conditional branches. These experiments confirm the previous findings.
    - The PIR width is 15 bits.
    - The PIR is updated as follows.
        - Conditional taken branch address bits IP [18:4] are XOR-ed with the original PIR shifted by 2 bit positions to the left.
        - Indirect branch address bits IP[18:10] and target address bits TA[5:0] (total of 15 bits) are XOR-ed with the original PIR shifted by 2 bit positions to the left.
2. The global outcome predictor hash function is an XOR between the PIR and the conditional branch IP. Conditional branch IP bits [18:4] are XOR-ed with the PIR bits [14:0]. The lower 6 bits of the result are used as the tag. The higher 9 bits of the result are the index for the global predictor.
3. The global predictor is a 4-way structure organized into 512 sets.
4. Global predictor uses an LRU-like replacement policy with an unknown allocation policy. It is likely that a branch allocated to the predictor needs “verification” (that is, the branch is encountered at least twice). In the meanwhile, the branch may still be evicted by the other branches that target the same set, even if other entries in the particular set may be free.
5. The Bimodal outcome predictor is a flat structure of two-bit counters indexed by the instruction IP bits [11:0] with size of 4096 entries.

8.3 Background

Global outcome predictors have long been a focus of many research efforts in academia and industry striving to come up with a very accurate outcome prediction with small hardware complexity and small latency. Improving the predictor accuracy by increasing the predictor's size has become a non adequate solution. It has been shown that the negative interference in the predictors is a larger source of mispredictions than the capacity misses caused by a smaller outcome predictor.

An increase in the branch predictor size often leads to unacceptable predictor latency. The instruction fetch unit and consequently the branch predictor unit is on the critical path to sustain high issue rates. To reduce the predictor delay, the predictor size must be reduced. Jimenez [32] shows a benefit from a cascading predictor where a slower more accurate predictor may overrule the prediction coming from a faster and less accurate predictor.

To remedy the problem of predictor size and interference, researchers have offered many sophisticated solutions, and over time, outcome predictors have become multi-layer complex structures, mainly consisting of many best up-to-date outcome predictors that are closely coupled to each other. However, complex structures often impose a significant verification effort during predictor design, and this in turn affects ever tightening time-to-market.

We expect that the modern branch outcome predictor implementations are based on the *GShare* with additional structures to address issues such as filtering (branch classification) and negative interference.

8.3.1 Negative Interference

Negative interference occurs when two branches with opposite outcomes compete for the same entry in the global predictor, consequently making one or both outcomes mispredicted.

A number of research efforts have addressed the issue of negative interference. Here we focus our attention on the Bi-mode predictor [13] to set the background for expectations on resolving the negative interference problem in modern predictors. With Bi-mode, the global predictor is divided into two parts, not taken and taken branch history tables (NT.PHT and T.PHT). A third structure, a bimodal table is used to select which table, NT.PHT or T.PHT should provide an outcome prediction. This way, the global predictor will predict incorrectly only when two branches with opposite outcomes, have the same global predictor access function and the bimodal table gives an incorrect prediction selecting the opposite history table. The main flaw in the Bi-mode predictor is the non-existence of filtering. Branches that are always taken or not taken are also predicted by the global predictor although the bimodal predictor may predict this branch with fewer resources.

8.3.2 Branch Filtering

Branch filtering or branch classification is a technique where a specialized branch predictor structure is used to predict certain types of branches.

Specialized predictors introduce tags to be able to distinguish the occurrence of the certain branch type. We have already seen that the loop predictor is a cache structure where tags are used to identify the loop branches. We expect a similar approach in the implementation of the global predictor. The tagged global predictor may handle only

those branches that are not predicted correctly by both the loop predictor and the bimodal predictor. Our focus is on two academically proposed predictors, McFarling's Serial-BLG [7] and YAGS predictor [16].

YAGS Predictor

The YAGS predictor is based on the Bi-mode predictor with two global predictor tables, T.PHT and NT.PHT, but these tables are tagged in order to ensure filtering. However, it should be noted that tagging reduces positive and neutral interference potential of the Bi-mode predictor. For example, if two not taken branches compete for the same entry in T.PHT, they will evict each other if they have different tags. In a Bi-mode predictor, neutral interference would occur.

On the other hand, selection of the T.PHT and NT.PHT based on the bimodal predictor may reduce the filtering capabilities. For example, an entry in the global predictor may have a lookup hit, but the entry will not be considered if the bimodal table did not select a particular table the hit belongs to. Also, one table of the Global predictor may be more overloaded than the other table, resulting in unnecessary wasting of resources

Finally, we consider the YAGS scheme as a cache-like global predictor that works in tandem with the bimodal predictor with questionable global predictor way-selection function. This observation leads us to the Serial-BLG predictor [7].

Serial-BLG Predictor

The serial-BLG predictor consists of three stages (predictors), Bimodal, Local and Global. The bimodal is the default predictor. If the branch is detected to have a local behavior and cannot be successfully predicted by the bimodal predictor, the bimodal

prediction may be overridden in the local predictor. If the branch is detected to have a global behavior and cannot be successfully predicted by either the bimodal or the loop predictor, a prediction given by the bimodal predictor or the local predictor may be overridden by the global predictor. This way, the local and the global predictor may have much smaller sizes since the bimodal predictor handles the majority of the program branches. Therefore, filtering exists as in the YAGS predictor except that the global predictor access function is not related to the bimodal predictor. This fact allows for easier branch predictor development and verification.

8.3.3 Expectations

We expect the global predictor to be a multi-way cache structure. By introducing the tags, the number of global predictor entries may be significantly reduced compared to the version without tags. The global predictor needs to predict only branches that are unpredictable by the bimodal and loop predictors. A much smaller number of entries successfully compensates for an additional hardware overhead due to tags. Additionally, the reduced predictor size results in lower predictor latency, an equally important parameter in the design of modern branch predictor units.

A tagged structure also helps with the negative interference. The tags cannot remove negative interface but can identify it. For example, two branches that hit in the same global predictor entry with different tags, will still evict each other but, due to different tags, global predictors will not provide a hit, avoiding possible misprediction. Final prediction will rely on the bimodal predictor that can still give a correct prediction.

We expect first level of the global predictor to be a shift register containing branch history register, BHR. The BHR may be affected by different types of branches and branch information such as address bits, target bits or the branch outcome.

8.4 BHR Organization – Conditional Branch IP Address Bits used for BHR

We assume that the BHR is affected by the IP address of the last byte of a conditional branch, rather than by the branch target address. The BHR serves the same purpose as the PIR for the indirect branch predictor. Consequently, we assume that the BHR organization is a similar one to the PIR organization.

We develop a microbenchmark that tries to find the following: (a) branch address bits used for the BHR, (b) the BHR shift count, (c) the BHR history length (BHR.HL) -- the maximum number of branches, prior to a conditional branch, affecting the BHR, and (d) the BHR width (the number of bits in the branch history register).

The branch address bits used for the BHR are found using an algorithm similar to the one used in Figure 7.7 to find conditional branch IP bits that affect the PIR. The only change is that we replace the indirect spy branch with a conditional spy branch (see Figure 8.1). The spy conditional branch has two outcomes; the nT outcome and the T outcome. *Path1* is the path taken to reach the spy branch T outcome and *Path2* is the path taken to reach the spy branch nT outcome. To achieve different paths to work in the way described, a setup branch is introduced. The setup branch has the same outcome pattern as the spy branch. The setup branch is made to always be mispredicted by introducing a number of conditional taken branches before it. Consequently, the total misprediction rate of 50% means that the spy branch misprediction rate is 0%.

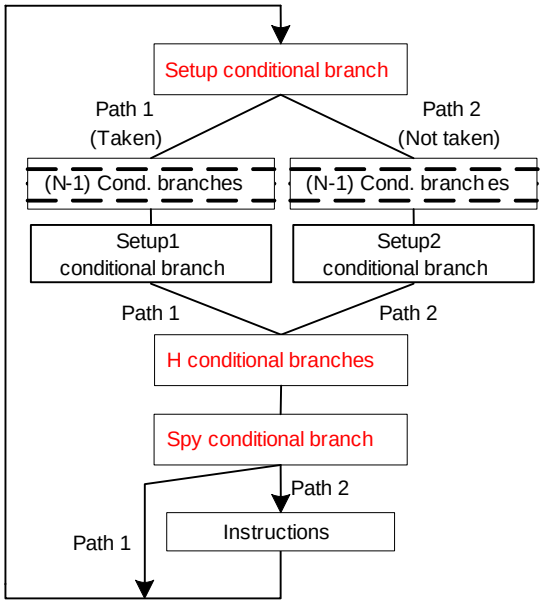

Figure 8.1 Conditional branch IP bits that affect BHR, BHR shifting policy and BHR history length microbenchmark layout

The spy branch must have the outcome pattern unrecognizable by the loop, the bimodal or any other local predictor. Since we know that the loop predictor has the maximum counter length of 64, a loop branch with the counter modulo larger than 64 can be used.

We choose a more generalized pattern: The spy branch will have pattern {N*Taken, N*Not Taken}, with N large enough to avoid any local predictor. This way we do not consider relations of the global predictor with any other predictor. A problem with this approach is the unknown expected number of mispredictions. However, when the spy branch starts to be mispredicted, we expect the total number of mispredictions to be twice what it is without the spy branch mispredictions, since the spy and the setup branches contribute equally to the total number of mispredictions.

The algorithm sets one bit in the *Setup2* branch address to differ from the *Setup1* branch address. The particular bit is referred to as k (k=0, 1, …$\log_2$Offset-1), and displacement D is defined as $D=2^k$. Therefore, IP(*Setup2*) = D + IP(*Setup1*) + *Offset*. Consequently, if bit k does not affect the BHR, the microbenchmark produces mispredictions in the global predictor. IP(*Setup1*) bit k must be set to zero for proper testing.

A similar approach is used for testing the BHR shift policy. The algorithm is modified by insertion of H conditional branches between the spy branch and *Setup1* (and *Setup2*) branches. These H branches are executed in both *Path1* and *Path2*; therefore, they influence the *Path1* based BHR in the same way as the *Path2* based BHR. Consequently, *Path1* based BHR and the *Path2* based BHR will differ only if *Setup1* and *Setup2* differ. By increasing H, BHR bits shifted in by *Setup1* or *Setup2* branches are moved further in the BHR history.

To find the BHR shift count, the same reasoning is used as in finding the PIR shift count (see Equation (7.1)) as shown in Equation (8.1).

$$(\text{shift count}) * BHR.HL - (\text{shift count}) < BHR\ Width \leq (\text{shift count}) * BHR.HL. \quad (8.1)$$

BHR history length is found in the same way as the PIR history length; BHR.HL is equal to the minimum H for which the number of mispredictions is high regardless of the parameter D. The microbenchmark source code is shown in Figure 8.2.

Address	Code
	`int long unsigned L, liter = 1000000;`
	`int a=1;`
	`do{`
	`  L = (liter%32)>>4;  // pattern 16*T, 16*nT`
	`  if (L==0){  // execute one path per iteration`
`@A`	`  8x if(a==0)a=1;  // Repeat the statement 8 times`
`@B,B[14:0]="0"`	`  if(a==0) a=1;  // Setup1`
	`  }  // Unconditional jump`
	`  else{ // dummy non-branch instructions`
`@A + Offset`	`  8x if(a==0)a=1;`
`@B,B[14:0]= "1"`	`  if(a==0) a=1;  // Setup2`
	`  }`
	`  if(L==0) a=1;  // Spy branch`
	`  liter --;`
	`} while (liter >0);`

Figure 8.2 Conditional branch IP bits effect on BHR test source code

We measure the number of branches mispredicted at execution (MBI_EXEC). Misprediction rate is calculated as the MBI_EXEC divided by the number of spy branches.

Figure 8.3 shows the misprediction rate as a function of the parameter D (D=1h–80000h), when H=0. For distances D=10h–40000h, the spy branch does not produce mispredictions, indicating that branch address bits IP [18:4] are used for the BHR.

Figure 8.4 shows the misprediction rate as a function of the parameter D (D=1h–80000h), when H=1. For distances D=10h–10000h, the spy branch does not produce mispredictions. According to analysis in Section 7.5, we conclude that the BHR shift count is two.

Figure 8.5 shows the misprediction rate as a function of the parameter D (D=1h–80000h), when H=7, 8. Not all H values are tested since the similarity with the PIR is obvious enough to consider results sufficient.

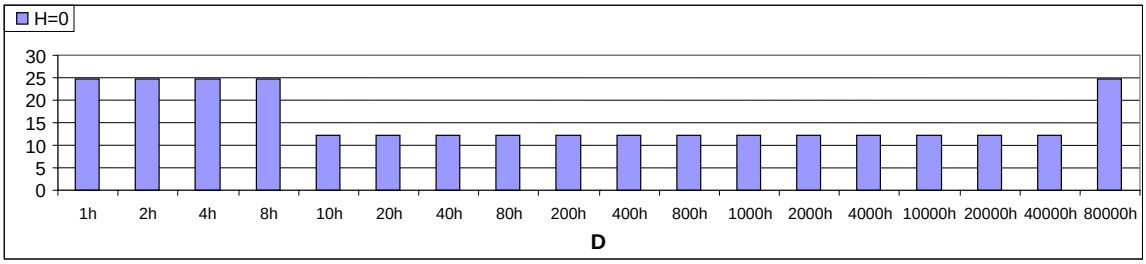

Figure 8.3 Conditional branch IP bits effect on BHR test results for H=0

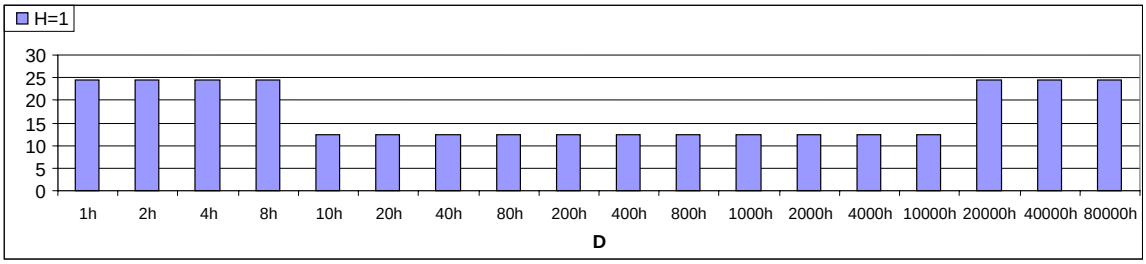

Figure 8.4 Conditional branch IP bits effect on BHR test results for H=1.

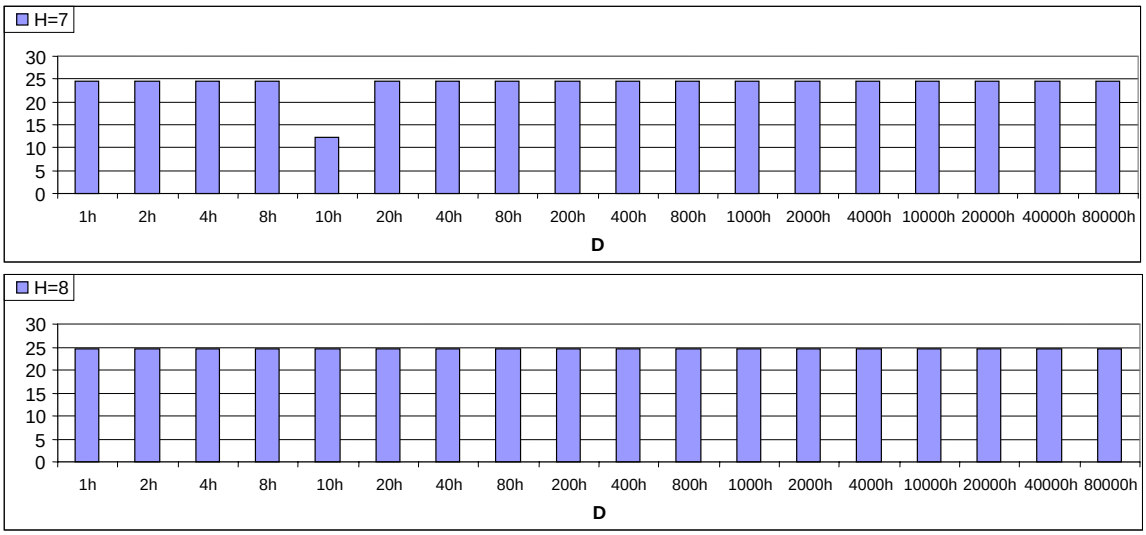

Figure 8.5 Conditional branch IP bits effect on BHR test results for H=7,8

8.5 BHR Organization – Type of Branches Used

The microbenchmark has been developed which determines which types of branch instructions affect the BHR, apart from conditional taken branches. We test the following branch types: (a) conditional not taken branches, (b) unconditional branches, (c) call and return jumps.

The microbenchmark uses a slightly modified approach presented in Figure 8.1. The H-block is modified to include a number of conditional taken branches (H_C, $H_C <$ BHR.HL), followed by a number of branches of other types H_O (see Figure 8.6).

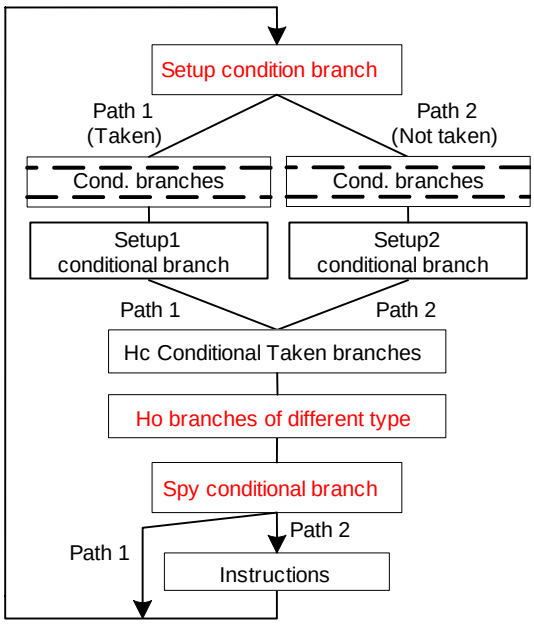

Figure 8.6 Branch types affecting the BHR microbenchmark layout

A number of branches of other types, H_O is set to the value H_O = BHR.HL–H_C. *Path2* and *Path1* are set to be unique in the same way as in the algorithm in Figure 8.6,

by setting the branch IP address of *Setup2* branch to be different from the branch IP address of *Setup1*. If the branches of other types do affect the BHR, *Path2* will be the same as *Path1*, consequently producing mispredictions. If the branches of other types do not affect the BHR, the total number of mispredictions should be zero.

Not taken direct conditional branches. Figure 8.7 shows the code snippet for Hc and Ho blocks for the microbenchmark described in Figure 8.6. The Ho block features direct conditional not taken branches.

Unconditional jumps. Figure 8.8 shows the code snippet for Hc and Ho blocks for the microbenchmark described in Figure 8.6. The Ho block features unconditionally taken branches.

```
a=1;
...
6 x if(a==0) a=1;a=1;    // Hc = 6
2 x if(a==1) a=1;a=1;    // Not taken branches, Ho=2
...
if(L==0)a=1;a=1;      //spy conditional branch
```

Figure 8.7 Source code fragment for testing of NT conditional branches effect on BHR

```
6 x if(a==0) a=1;a=1;    //Hc = 6
_asm{jmp l1              //unconditional jumps, Ho=2
    l1: clc
    jmp l2
    l2: clc
}
if(L==0)a=1;a=1;      //spy conditional branch
```

Figure 8.8 Source code fragment for testing of unconditional branches effect on BHR

Call/returns. Figure 8.9 shows the code snippet for Hc and Ho blocks for the microbenchmark described in Figure 8.6. The Ho block features Call/return branches.

All the tests produce no mispredictions, meaning that none of the tested branches, direct conditional not taken, unconditional, and call/returns are affecting the BHR.

```
_asm{
    jmp lcc
   _doit1:  mov ebx, 10
   _doit2:  mov ebx, 10
     ret
   lcc: clc
}
do{
   6 x if(a==0) a=1;a=1; // Hc=6
  _asm{                  // call & returns, Ho=2
     call    _doit1
     call    _doit2
  }
  if(L==0)a=1;a=1;      //Spy conditional branch
}
while(liter>0);
```

Figure 8.9 Source code fragment for testing of call and return branches effect on BHR

8.6 BHR Organization – Branch Outcome Effect on BHR

We develop a microbenchmark that reuses the algorithm shown in Figure 7.16. The switch branch and the spy indirect branch are replaced with two conditional branches with the same outcome pattern as shown in Figure 8.10. Consequently, we set different outcome histories values for two outcomes of the spy branches:

Spy taken outcome history: *<Taken branch 8, Switch, Taken branches 7–2>.*

Spy not taken outcome history: *<Taken branches 8–1>.*

To make both path histories dependant only on the switch branch outcome, the taken branches 8–1 and the *Switch* branch have the same lower bits, so they influence the BHR in the same way. The microbenchmark source code is shown in Figure 8.10.

The test doubles the number of mispredictions compared to the test when the *Switch* branch is simply replaced with the always taken branch. The *Switch* branch is crafted to produce mispredictions due to the same history for both its outcomes. Consequently, we conclude that the spy branch produces mispredictions because the outcome of the *Switch* branch did not affect the BHR.

```
Address              Code
                     int long unsigned L, liter = 1000000;
                     int a=1;
                     do{
                       L = (liter%32) >> 4; // pattern 16*T, 16*nT
@A                     if(a==0)a=1;    // Taken branch 1
                       // dummy non-branch instructions
                       ...
@A + 6*Offset          if(a==0)a=1;    // Taken branch 7
                       // dummy non-branch instructions
@A + 7*Offset          if(L==0)a=1;    // Switch branch
                       // dummy non-branch instructions
@A + 8*Offset          if(a==0)a=1;    // Taken branch 8
                       // dummy code to allow branches retirement
                       if(L==0)a=1;    // Spy branch
                       liter--;
                     } while (liter >0);
```

Figure 8.10 Detection of branch types affecting the BHR source code

8.7 BHR Organization – Indirect Branch Effect on BHR

A new microbenchmark is designed to test whether both indirect branch address and indirect branch target affect the BHR. The algorithm reuses the approach illustrated

in Figure 8.1. The new algorithm (see Figure 8.11) replaces the conditional branches *Setup1* and *Setup2* with indirect branches *ISetup1* and *ISetup2*. The indirect branches have targets, *Target1* and *Target2.*

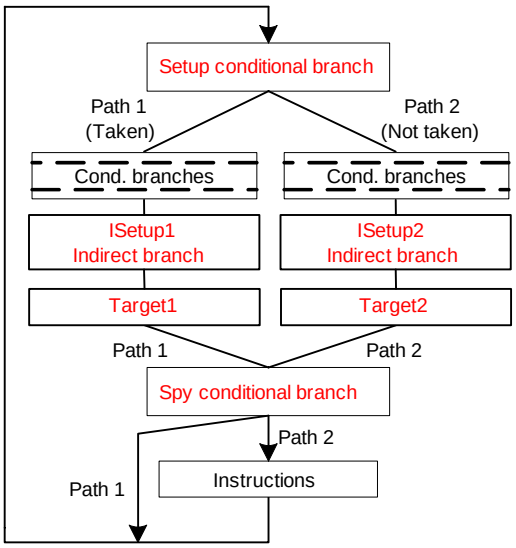

Figure 8.11 Detection of indirect branch effect on BHR microbenchmark layout

To test for the indirect branch target address impact on the BHR, *ISetup2* branch IP address is changed to have different target address bit *k* from the *ISetup1* branch IP address. Consequently, if indirect branch target address bit *k* affects the BHR, the misprediction rate should be close to zero. *ISetup1* and *ISetup2* are set at distance *Offset* not to affect the BHR. The difference at bit *k* is achieved by setting the *ISetup2* target address at distance D + *Offset* from the *Setup1* target address where $D = 2^k$ and *Offset* = 80000h. The microbenchmark source code is shown in Figure 8.12.

```
Address                Code
                       int long unsigned i,k,l, liter = 1000000;
                       int a=1;
                       do{
                          L = (liter%32)>>4;    // Pattern 16x Taken, 16x Not taken
                          8x if(a==0)a=1;       // make setup branch mispredicted
                          if (L==0){             // execute one target per iteration
@A                           7x if(a==0)a=1;    // Repeat the statement 7 times
@B                           jmp dword ptr [ebx]  // ISetup1
@C, C[14:0] = 0              _0: clc              // Target1
                          }
                           else{// dummy non-branch instructions
@A + Offset                  7x if(a==0)a=1;
@B + Offset                  jmp dword ptr [ebx]  // ISetup2
@C + Offset + D              _1: clc              // Target2
                           }
                          if(L==0) a=1;a=1;  // Spy branch
                          liter--;
                       } while (liter>0);
```

Figure 8.12 Indirect branch target address bits effect on BHR test source code

We measure the number of branches mispredicted at execution (MBI_EXEC).

Figure 8.13 shows the misprediction rate, calculated as the MBI_EXEC divided by the number of spy branches, as a function of the parameter D (D=1h–80h). The results are the same as the one for the PIR; BHR is affected by the indirect branch target address bits [5:0].

To test for the indirect branch IP address impact on the BHR, *ISetup2* is changed to have a different IP address bit *k* from the *ISetup1* branch IP address. Consequently, if the indirect branch IP address bit *k* affects the BHR, the induced number of mispredictions will be zero. *Target1* address and *Target2* are set at the distance D=40h not to affect the BHR. The difference at bit *k* is achieved by setting the *ISetup2* IP address at distance D + *Offset* from the *ISetup1* IP address, where $D=2^k$ and *Offset* = 80000h.

The microbenchmark source code is shown in Figure 8.14.

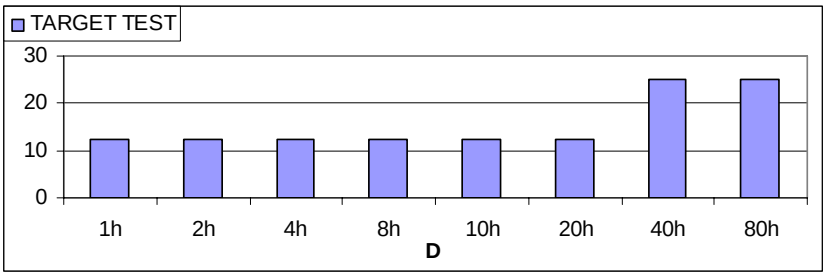

Figure 8.13 Indirect branch target address bits effect on BHR test results

Address	**Code**
	`int long unsigned i,k,l,liter = 1000000;`
	`int a=1;`
	`do{`
	`  L = (liter%32)>>4; // Pattern 16x Taken, 16x Not taken`
	`  8x if(a==0)a=1;  // make setup branch mispredicted`
	`  if (L==0){  // execute one target per iteration`
@A	`  7x if(a==0)a=1; // Repeat the statement 7 times`
@B, B[14:0] = 0	`  jmp dword ptr [ebx]  // Indirect1`
@C	`  _0: clc  // Target1`
	`  }`
	`  else{// dummy non-branch instructions`
@A + Offset	`  7x if(a==0)a=1;`
@B + Offset + D	`  jmp dword ptr [ebx]  // Indirect2`
@C + 40h	`  _1: clc  // Target2`
	`  }`
	`  if(L==0) a=1;a=1;  // Spy branch`
	`  liter--;`
	`} while (liter>0);`

Figure 8.14 Indirect branch IP bits effect on BHR source code

Figure 8.15 shows the misprediction rate, calculated as the MBI_EXEC divided by the number of spy branches, as a function of parameter D (D=10h–80000h). The results are the same as the one for the PIR; BHR is affected by the indirect branch IP bits [18:10].

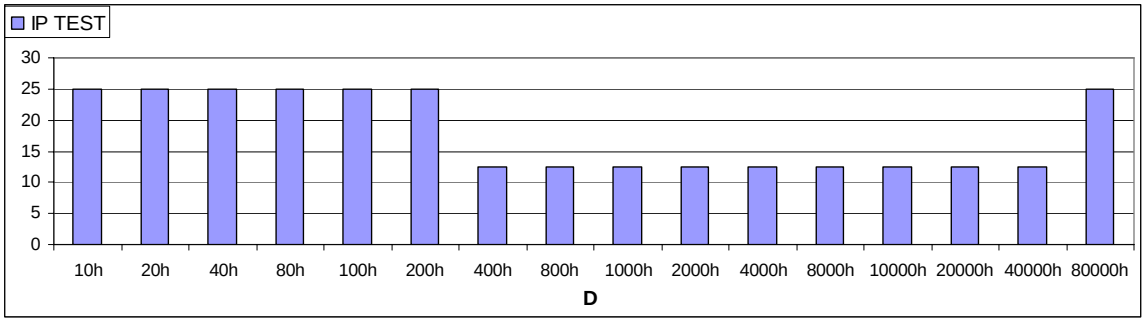

Figure 8.15 Indirect branch IP bits effect on BHR test results

We do not examine the effect of the indirect branch IP address or indirect branch target address with the “H block” as in Section 8.1 as we assume that the similarity with the PIR is sufficient to make conclusions even without additional testing.

NOTE: We are able to examine the BHR and conclude that the BHR has the same organization and behavior as the PIR. We conclude that the BHR is in fact the PIR; the branch predictor employs one shift register, the PIR, used to access both the global predictor and the indirect predictor.

8.8 Global Predictor Access Function

We assume that part of the conditional branch IP address is XOR-ed with the BHR to access the global predictor. A part of the XOR result is used as the index and another part as the tag in the global predictor.

Here we wanted to find conditional branch bits used for the hash function in the same way we did for the iBTB hash function in Section 7.5 and we wanted to find the hash access function in the same way as in Section 7.12. However, we cannot use an identical approach because of the bimodal predictor influence. A similar algorithm

would use two conditional branches, *Spy1* and *Spy2*, with the same histories in the BHR and change a particular *Spy2* IP address bit. This change involves changing of the bimodal entry for *Spy2,* making the misprediction rate impossible to analyze.

Moreover, we are assuming the possibility of the existence of two global predictor tables selected by the bimodal predictor as in the Bi-mode predictor. This requires us to exactly know the current state of the bimodal predictor and to keep it in one direction during the global predictor testing. Otherwise, two global predictor tables would make results analysis a challenging if not impossible task.

We develop an algorithm that relies on one of the basic properties of the two-bit counters. For the branch with outcome pattern {*3*T, 2*nT*}, a two-bit counter will incorrectly predict both not taken outcomes and one taken outcome that come just after the *nT* outcomes. If such branch outcomes all target the same bimodal predictor entry and the same global predictor entry, we analyze two cases. First, if the bimodal predictor does not choose between two tables then the misprediction rate is 60% as the two-bit counters were behaving the same as described. Second, if the bimodal predictor chooses between the two tables then the misprediction rate is smaller than 60%, because taken and not taken outcomes will be allocated in two different entries. Figure 6.13 already showed that the branch with the described pattern and targeting the same bimodal and global entry has a misprediction rate of 60%. Therefore, we conclude that there are not two global predictor tables. The algorithm used here (see Figure 8.16) uses two loops *Spy1* and *Spy2* with the same modulo *MOD* and both switch their outcomes at the same time.

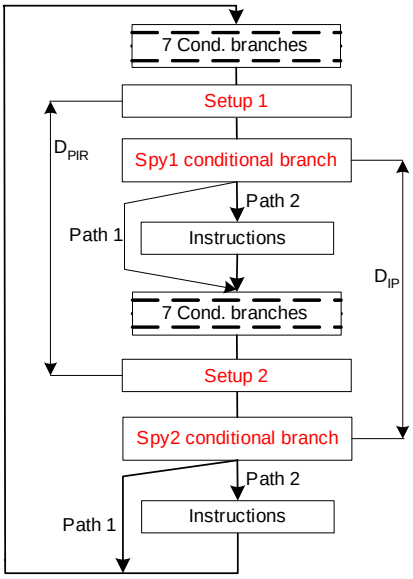

Figure 8.16 Global predictor access function microbenchmark layout

The *Spy2* IP address is changed at particular bit k_{IP} and therefore we expect *Spy2* to target a different global predictor entry, consequently making the total misprediction rate of 2/MOD. The distance $D_{IP} = 2^{k_{IP}}$.

Now, *Setup2* that precedes the *Spy2* is changed to differ from the *Setup1* branch at a particular bit k_{PIR}. The idea is to perform the same algorithm as in Section 7.12. If bits k_{IP} and k_{PIR} are XOR -ed in the hashing function, both *Spy1* and *Spy2* target the same entry in the global predictor, making the effective outcome pattern {*N*T, 2*nT}* and the expected misprediction rate is 3/MOD regardless of the bimodal predictor.

The microbenchmark source code is shown in Figure 8.17.

Table 8.2 shows values of D_{PIR} and D_{IP} that matched and consequently produced a misprediction rate of 3/MOD with MOD=65 in the performed test. The global predictor access function is illustrated in Figure 8.18.

Address	Code
	`int long unsigned L, liter = 1000000;`
	`int a=1;`
	`do{`
	`  L = temp%65;`
	`  7x if(a==0)a=1;  // repeat the statement 7 times`
@A, A[k_{PIR}] = 0	`  if(a==0)a=1;  // Setup1`
	`  // dummy code to allow previous branches to retire`
@B, B[k_{IP}] = 0	`  if(L==0)a=1;  // Spy1`
	`  // dummy non-branch instructions`
	`  7x if(a==0)a=1;`
@A+ 80000h+ D_{PIR}	`  if(a==0)a=1;  // Setup2`
	`  // dummy code to allow previous branches to retire`
@B+ 80000h+ D_{IP}	`  if(L==0)a=1;  // Spy2`
	`  liter--;`
	`} while(liter>0);`

Figure 8.17 Global predictor access function source code

Table 8.1 BHR bits and conditional branch IP address bits that are XOR-ed to create the global predictor access function

D_{PIR}	D_{IP}	MBI_EXEC misprediction rate
10h	2000h	3/MOD
20h	4000h	3/MOD
40h	8000h	3/MOD
80h	10000h	3/MOD
100h	20000h	3/MOD
200h	40000h	3/MOD
400h	10h	3/MOD
800h	20h	3/MOD
1000h	40h	3/MOD
2000h	80h	3/MOD
4000h	100h	3/MOD
`8000h	200h	3/MOD
10000h	400h	3/MOD
20000h	800h	3/MOD
40000h	1000h	3/MOD

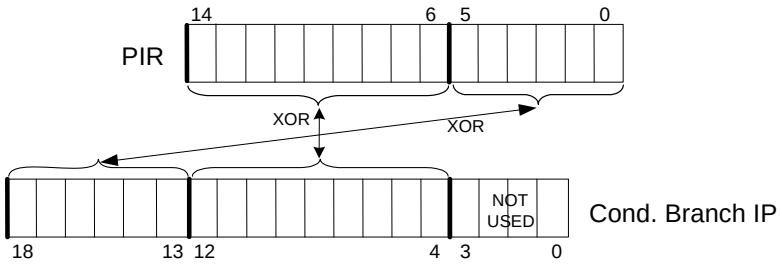

Figure 8.18 Global predictor access function

8.9 Global Predictor Organization

We expect the global predictor to be a multi-way cache. According to the results from the iBTB hash access function, we expect the lower part of hash function to be used as the index and higher bits as the tag in the global predictor.

In determining organization of the global predictor, we use a similar approach to the one used in determining organization of the iBTB (see Section 7.13). However, the problem here is somewhat more challenging. In crafting microbenchmarks, we need to neutralize the influence of the loop and the bimodal predictors. A new approach is needed that will ensure that spy branch behavior is tractable and dependable on the global predictor only. In the case of iBTB, the cached content was a unique target address. With the global predictor, an entry is likely a 2-bit saturating counter and interference is challenging.

We use an algorithm that considers N branches with unique tags competing for the same set in the global predictor with N-1 ways. Because the global predictor access causes misses, the branches will rely on the bimodal predictor that is set to give wrong predictions. This way we are able to test for the number of ways and the index and tag bits as will be explained below.

The microbenchmark has two spy branches -- an always taken *SpyT* and an always not taken *SpyN* (see Figure 8.19). The branches are placed at the distance *Offset*. *Offset* is large enough to ensure that the spy branches compete for a single entry in the global and the bimodal predictor. The *SpyN* branch is reached by N paths $PathN_1$ – $PathN_N$, while *SpyT* is reached by one path *PathT*. The program execution pattern is as follows: {$T*PathT, PathN_1, T*PathT\ PathN_2, \ldots, T*PathT, PathN_N, T*PathT$}.

Each *SpyN* occurrence must rely on the global predictor for the correct prediction as the bimodal predictor is always in the taken state set by the *SpyT.*

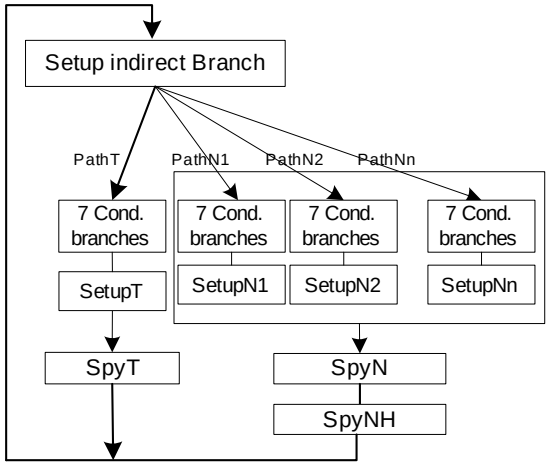

Figure 8.19 Global predictor organization microbenchmark layout

The parameter *T* should be large enough to avoid interference from the loop predictor. An alternative approach is to use a smaller value of *T* because larger *T* requires a high number of program iterations in order to observe results that are not statistical error (hardware performance counters count events imprecisely). We set T =4. To avoid

interference with the loop predictor, we insert another always not taken branch *SpyNH* that follows the occurrence of the *SpyN*. *SpyNH* is placed at a loop BPB tag distance from *SpyT*. The loop predictor sees *SpyN*, *SpyT* and the *SpyNH* as a one non-loop branch.

The microbenchmark sets $PathN_1 - PathN_N$ to differ in the same way as it is done in Section 8.1. $SetupN_i$ and $SetupN_{i-1}$ branches are at the distance D_G + *Offset* from each other where $D_G=2^k$. Therefore, the path based BHRs for each of the paths, $PathN_1 - PathN_N$, are set to be different from each other. The SpyN branch has N different values in the BHR, consequently occupying N different entries in the global predictor. If the distance D_G is set in the way that the BHR value for paths $PathN_1 - PathN_N$ differ only at the tag bits, subsequent *SpyN* occurrences will produce misses in the global predictor and mispredictions from the bimodal predictor.

The misprediction rate as a function of D_G and N gives us insight into the global predictor organization. The following example explains details about finding the global predictor organization:

- 4-way global predictor uses BHR[5:0] for the index, BHR[13:6] for the tag.
    - *For D_G = 10h (lowest BHR bit used for the index), N up to $4*2^6$ will not produce mispredictions.*
    - *For D_G = 20h, N up to $4*2^5$ will not produce mispredictions.*
- 4-way global predictor uses BHR bits [5:0] for the tag and BHR[12:6] for the index:
    - *For D_G = 10h, N up to 4 will not produce mispredictions.*
    - *For $D_G = 2^7$, N up to 2^7 will not produce mispredictions.*
    - *For $D_G = 2^6$, N up to $2* 2^7$ will not produce mispredictions.*
    - *For $D_G = 2^5$, N up to $4* 2^7$ will not produce mispredictions.*

The microbenchmark source code is shown in Figure 8.20. The program execution pattern is controlled by the Setup indirect branch. The indirect branch will

produce a known number of mispredictions due to a number of conditional taken branches that precede it. The Pentium M performance counters can count an event related to mispredictions on indirect branches exclusively. Furthermore, it is much easier to distinguish mispredictions that are from the spy branch only as a total number of mispredictions minus the number of indirect branch mispredictions.

```
Address                 Code
                        int long unsigned L, liter = 1000000;
                        int a=1;
                        do{
                           jmp dword ptr [ebx]
@PathN0                     7x if(a==0)a=1;      // repeat the statement 7 times
@A                          if(a==0)a=1;         // SetupN0
                            jmp _SpyN
                            // dummy non-branch instructions(skipped)
@PathN1                     7x if(a==0)a=1;
@A+ Offset+ D               if(a==0)a=1;         // SetupN1
                            jmp _SpyN
                            // dummy non-branch instructions(skipped)
@PathNN                     7x if(a==0)a=1;
@A+ N*Offset+ N*D           if(a==0)a=1;         // SetupN1
@_SpyN                      // dummy code to allow previous branches to retire
                            if(a==1)a=1;         // SpyN
@PathT                      7x if(a==0)a=1;
@A+ (N+1)*Offset+           if(a==0)a=1;         // SetupT
10101010101b
                            // dummy code to allow previous branches to retire
@_SpyN + Offset             if(a==0)a=1;         // SpyT
                            liter--;
                        } while(liter>0);
```

Figure 8.20 Global predictor organization test source code

During the test implementation, we find obstacles. The number of mispredictions (MBI_EXEC) is unexplainable in the same way it was for the BTB-capacity tests. For N=3, and D_G set to have all 3 branches changed only at the lower 6 BHR bits (assumed

tag bits), we observe that less than one branch is missed per iteration. For N=4 and D_G set to have all 3 branches changed only at the lower 6 BHR bits (assumed tag bits), we observe that approximately 3 branches are mispredicted per iteration. For D_G values large enough to make the *SetupN*$_i$ branches target the upper BHR bits to be different, we observe no mispredictions.

To cope with this issue, we use the same approach used in the BTB tests. Each of not taken paths *PathNi* is executed twice consecutively before starting with the new path. An indentical layout of the algorithm is used as in Figure 8.19, but the new program pattern is as follows:

*T*PathT, PathN*$_1$*, T*PathT, PathN*$_1$*, T*PathT, PathN*$_2$*, T*PathT, PathN*$_2$*, … , T*PathT, PathN*$_N$*, T*PathT, PathN*$_N$.

Setup indirect branch must use an appropriate pattern to achieve such a program pattern.

We measure the number of mispredicted indirect branches (MIBIE) and the number of branches mispredicted at execution (MBI_EXEC). The misprediction rate is calculated as the (MBI_EXEC– MIBIE) divided by the number of *SpyN* branches.

Figure 8.21 shows the misprediction rate as a function of D_G (D_G =10h–1000h) and N=5. For N=3, 4 results are not presented, as the test did not produce any mispredictions. For N=5 we observe mispredictions at the D_G < 80h. For D_G =80h, the 5th branch bit set is the IP bit 10 (Effective distance is 400h – BHR bit 7). We conclude that the index LSB bit is the hash function bit 7 and the global predictor is a 4-way structure.

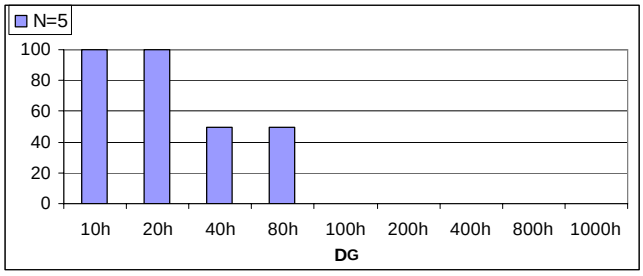

Figure 8.21 Global predictor organization tag test results

The logical way for the algorithm to advance the examination would be to set D_G at the lowest distance that produces no mispredictions for N=5 (D_G= 80h) and to increase N until mispredictions appear. This would mean that we are targeting each *SpyN* occurrence to occupy a different BTB entry. We are unable to create such a microbenchmark for N larger than ~10 and therefore another approach is used.

The microbenchmark reuses the source code shown in Figure 8.20 and sets N=5 and D=20h, so that mispredictions exist. Now, a particular bit of the $SetupN_0$ is changed by setting the $SetupN_0$ branch address at the distance D from its previous position. $SetupN_0$ IP address was a reference point for distance D_G and a new distance D is reflected as the distance that sets bits $k = \log_2(D)$. Distance D is increased to test for all higher BHR bits. The new distance is named D_I. Low misprediction on particular bit k_i set by the distance D_I, means that the bit k_i is the part of the index in the global predictor.

The microbenchmark's source code is similar to the one in Figure 8.20 except that the $SetupN_0$ IP address is changed from *"@A"* to "*@A*+ D_I*"*.

We measure the number of mispredicted indirect branches (MIBIE) and the number of branches mispredicted at execution (MBI_EXEC). The misprediction rate is calculated as the (MBI_EXEC– MIBIE) divided by the number of *SpyN* branches.

Figure 8.22 shows the misprediction rate as a function of D_I (D_I =100h–40000h) and N=5. We observe that the bits [18:10] are used for the index and therefore the global predictor set size is 512 entries. Consequently, the total size is 2048 entries.

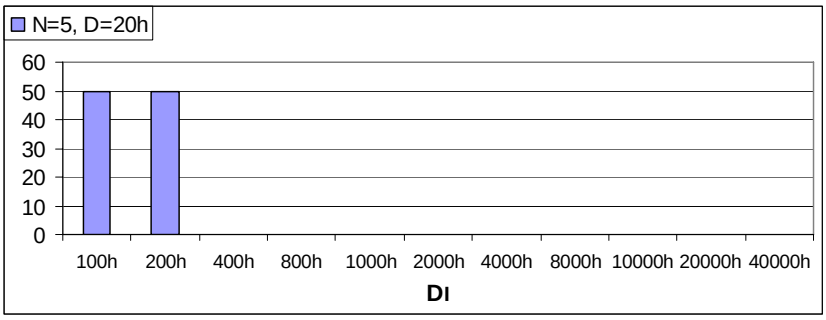

Figure 8.22 Global predictor organization index test results

8.10 Bimodal Predictor Organization

We expect the bimodal predictor to be a flat structure without tags and addressed by the branch IP address bits only. Until now, we have been able to see that the branch prediction mechanism relies on the bimodal predictor if the global predictor has a miss for a given branch.

In the previous experiment, we used one always taken branch to make one bimodal entry to be always in a taken state. The *SpyN* branch, which misses in the global predictor, is predicted by the same bimodal entry that was in the taken state, consequently causing mispredictions. Here we reuse the microbenchmark shown in Figure 8.19. A new version increases distances between *SpyN* and the *SpyT* branches: *SpyT* = *SpyN* + D_G + *Offset*; $D_G=2^k$. If the branch IP bit *k* is used to address the bimodal predictor, each

SpyN occurrence that misses in the global predictor will rely on a bimodal predictor entry that is in a not taken state. Consequently, no mispredictions exist.

The microbenchmark sets collisions in the global predictor by setting appropriate parameters D and N as shown in Figure 8.20. We set D=10h as we have proven that all *SpyN* occurrences target the same global predictor set and N=5 as we have proven that the global predictor is a 4-way structure. The microbenchmark finds the number of mispredictions as a function of the D_G. We expect that the low number of mispredictions is an indication that the branch IP address bit *k* is used for the index in the bimodal predictor. The microbenchmark is not shown here as it is almost identical to the one shown in Figure 8.20. *SpyT* branch has to be moved for offset D_G to test for the bit *k* effect on the bimodal predictor.

We measure the number of mispredicted indirect branches (MIBIE) and the number of branches mispredicted at execution (MBI_EXEC). The misprediction rate is calculated as the (MBI_EXEC– MIBIE) divided by the number of *SpyN* branches.

Figure 8.23 shows the misprediction rate. The results indicate that the bimodal predictor is addressed by the IP address bits [11:0]. Consequently, we conclude that the bimodal predictor size is 4K entries.

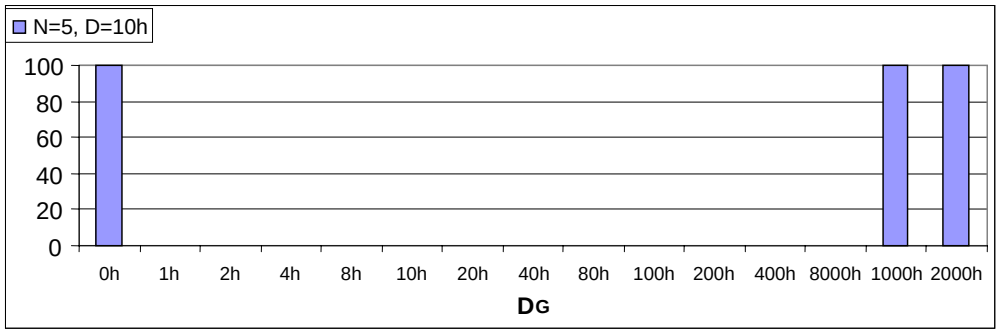

Figure 8.23 Bimodal predictor bits detection test results

We are able to see the bimodal predictor that works as a stage before the global predictor. Moreover, the bimodal predictor is not a cache structure. This leads us to the conclusion that the branch prediction unit does not use a static branch prediction mechanism (this is what we knew from documentation). The bimodal predictor always gives outcome prediction. On a BTB miss and with outcome prediction "taken," the decoder stages decide about the branch target address.

8.11 Global-Loop Predictors Relations

In this section we test whether a global predictor hit overrides a loop predictor hit. We develop a microbenchmark that uses a specific branch outcome pattern where one of the outcomes hits in both the loop BPB and the global predictor, but just one of the predictors predicts the outcome correctly. We set the loop predictor to predict the outcome incorrectly and the global predictor to predict it correctly. If a global predictor hit overrides a hit in the loop predictor, the branch outcome will not be mispredicted, otherwise the outcome is mispredicted. The microbenchmark uses the branch with the pattern shown in Table 8.2.

At iteration number 13, a branch has a “Not taken” outcome. Due to previous branch behavior, the branch is allocated in the loop BPB; therefore, the loop predictor provides a loop BPB hit and it is a misprediction. The same outcome is correctly predicted with the global predictor by reusing the microbenchmark source code from Figure 8.20 with N=1 and *SpyNH* removed. The indirect setup branch is set to produce the pattern in Table 8.2.

The test results in no mispredictions, indicating that indeed the global predictor sits on the top and overrides the prediction from the global predictor.

Table 8.2 Outcome pattern for the testing of Global hit priority over Loop hit

Iteration number	1	2	3	4	5	6	7	8	9	10	11	12	13
Branch outcome	**T**	**T**	**T**	**nT**	**T**	**T**	**T**	**nT**	**T**	**T**	**T**	**nT**	**nT**

CHAPTER 9

CONCLUSIONS AND FUTURE WORK

Branch predictor units are one of the crucial resources that ensure a full exploitation of potential performance benefits promised by deeply pipelined and wide-issue processors. They have been a focus of many research efforts in industry and academia. Unfortunately, the commercial implementations are rarely publicly disclosed. However, it has been demonstrated that the knowledge about exact branch predictor optimization can be used by optimizing compilers to improve overall performance.

In this thesis we present a systematic approach to reverse engineering of modern branch predictor units. We have developed a set of microbenchmarks and experimental flows that target various structures found in modern branch predictor units. The microbenchmark and experimental flows have been applied in reverse engineering of one of the most sophisticated commercial branch predictor units, the Pentium M branch predictor unit. We have found that the Pentium M's branch predictor unit encompasses the following resources. For speculative prediction of target addresses, we have a branch target buffer (BTB) and an indirect branch target buffer (iBTB). For speculative

prediction of branch outcome, a cascaded predictor is used that encompasses (a) a bimodal predictor, (b) a loop branch predictor buffer, and (c) a tagged global predictor. For each resource, we have found its size and organization, as well as update and allocation policies.

The presented framework can benefit not only future architecture-aware compilers, but also can be a valuable tool to branch predictor designers in their verification efforts. Last, but not least, unveiling a sophisticated commercial predictor unit can help future research efforts and contribute to a better understanding of modern branch predictors.

In the future the proposed framework can be tested and possibly extended to other modern processors. Eventually, software tools for architectural exploration can be implemented that will automatically generate microbenchmarks and process the results.

APPENDICES

Appendix A

BTB-set Flow Example

Assumed BTB architecture: 4-way, Index = IP[10:4], Tag = IP[16:11], Offset = IP[3:0].

Step 1: B=2:

D=10h: *different set => **No MPR** → Increase D:*

D=20h: *different set => **No MPR** → Increase D:*

D=40h: *different set => **No MPR** → Increase D:*

D=80h: *different set => **No MPR** → Increase D:*

D=100h: *different set => **No MPR** → Increase D:*

D=200h: *different set => **No MPR** → Increase D:*

D=400h: *different set => **No MPR** → Increase D:*

D=800h: *same set, diff. tag => **No MPR** → Increase D:*

D=1000h: *same set, diff. tag => **No MPR** → Increase D:*

D=2000h: *same set, diff. tag => **No MPR** → Increase D:*

D=4000h: *same set, diff. tag => No **MPR** → Increase D:*

D=8000h: *same set, diff. tag => **No MPR** → Increase D:*

D=10000h: *same set, diff. tag => **No MPR** → Increase D:*

D=20000h: *same set, same tag => **MPR high**! → **Stop***

Remember (D_i, B_i) pair (20000h, B=2)

Step 2: B=3:

D=10h: *all three diff. sets => **No MPR** → Increase D:*

D=20h: *all three diff. sets => **No MPR** → Increase D;*

D=40h: *all three diff. sets => **No MPR** → Increase D;*

D=80h: *all three diff. sets => **No MPR** → Increase D;*

D=100h: *all three diff. sets => **No MPR** → Increase D;*

D=200h: *all three diff. sets => **No MPR** → Increase D;*

D=400h: *1st & 3rd same set, 2nd other set => **No MPR** → Increase D;*

D=800h: *all three same sets, diff. tags => **No MPR** → Increase D;*

D=1000h: *all three same sets, diff. tags => **No MPR** → Increase D;*

D=2000h: *all three same sets, diff. tags => **No MPR** → Increase D;*

D=4000h: *all three same sets, diff. tags => **No MPR** → Increase D;*

D=8000h: *all three same sets, diff. tags => **No MPR** → Increase D;*

D=10000h: *two with same tags same set => **MPR high → Stop***

Remember (D_i, B_i) pair (10000h, B=3)

Step 3: B=4:

D=10h: *all four diff. sets => **No MPR** → Increase D;*

D=20h: *all four diff. sets => **No MPR** → Increase D;*

D=40h: *all four diff. sets => **No MPR** → Increase D;*

D=80h: *all four diff. sets => **No MPR** → Increase D;*

D=100h: *all four diff. sets => **No MPR** → Increase D;*

D=200h: *all four diff. sets => **No MPR** → Increase D;*

D=400h: *1st, 3rd same set, 2nd, 4th same set => **No MPR** → Increase D;*

D=800h: *all four same set, diff. tags => **No MPR** → Increase D;*

D=1000h: *all four same set, diff. tags => **No MPR** → Increase D;*

D=2000h: *all four same set, diff. tags => **No MPR** → Increase D;*

D=4000h: *all four same set, diff. tags => **No MPR** → Increase D;*

D=8000h: *two with same tags same set => **No MPR** → Increase D;*

Remember (D_i, B_i) pair = (8000h, B=4)

Step 4: B=5:

D=10h; *all five diff. sets => **No MPR** → Increase D;*

D=20h: *all five diff. sets => **No MPR** → Increase D;*

D=40h: *all five diff. sets => **No MPR** → Increase D;*

D=80h: *all five diff. sets => **No MPR** → Increase D;*

D=100h: *all five diff. sets => **No MPR** → Increase D;*

D=200h: *1st, 5th same set, diff. tags => **No MPR** → Increase D;*

D=400h: *1st, 3rd, 5th same set, 2nd, 4th same set => **No MPR** → Increase D;*

D=800h: *5x same set, diff. tag => **MPR high → Stop**.*

Remember (D_i, B_i) pair (800h, B=5)

For B_i=2, D_i =20000h, D_{i-1} =10000h **=> Tag MSB = IP [16]**

Smallest D is D_i =800h, D_{i-1} =400h **=> Index MSB = IP [10]**

For smallest D_i =3, Bi = 5, B_{i-1} = 4 **=> Number of ways is 4**

Step 5: Index LSB detection; pick (D_i, B_i) = (1000h, 5)

D (4th, 5th) = 1000h: all five same set => MPR high → Increase D (4th, 5th);

D (4th, 5th) = 1001h: all five same set=> MPR high → Increase D (4th, 5th);

D (4th, 5th) = 1002h: all five same set=> MPR high → Increase D (4th, 5th);

D (4th, 5th) = 1004h: all five same set=> MPR high → Increase D (4th, 5th);

D (4th, 5th) = 1008h: all five same set=> MPR high → Increase D (4th, 5th);

*D (4th, 5th) = 1010h: four to same set=> **No MPR → Stop.***

Index LSB bit = $\log_2$ [D (4^{th}, 5^{th}) – D] **= IP [4].**

Appendix B

Setup Code for Cache-hit BTB-set Test

```
void main(void) {
int long unsigned liter = 100000000;
int long unsigned offset1;     // setup indirect branch offset
int long unsigned i,a;
int long unsigned temp_i1,temp_i2;

int Dist = 128;                 // Dist = D
int Branches = 4096;            // Dist = B

_asm{sub   esp, 4014H}         // free stack space
for (i=0; i<Branches; ++i){    // allocate all ind. br targets in stack
  temp_i1 = 4*i;               // pointer to the current indirect target stack
                               // position, target is 4 bytes long
  temp_i2 = Dist*i;            // indirect target position in the code
  _asm {
    mov   eax,  l0
    add   eax,  temp_i2         //  prepare current target
    mov   ecx,  temp_i1
    add  ecx,  esp
    mov   dword ptr [ecx], eax  // allocate current target on the stack
  }
}

for (i=0; i<liter; ++i){
  offset1 = (i%(2*Branches))>>1;  // prepare offset, jump to each target twice
  offset1 =  offset1*4;           // target is 4 bytes long
  _asm {
    mov     edx, _Exit
    mov     ebx, offset1
    add     ebx, esp           // add offset to the value of the stack pointer
    jmp     dword ptr [ebx]    // jump to the target
   // final part of code already presented
```

Figure B.1 Indirect branch pattern for the Cache-hit BTB-capacity test in Section 5.7.

REFERENCES

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, IV ed, 2007.

[2] S. Gochman, et al., "The Intel Pentium M Processor: Microarchitecture and Performance," *Intel Technology Journal*, vol. 07, 2003, pp. 21-36.

[3] M. Milenkovic, et al., "Microbenchmarks for Determining Branch Predictor Organization," in *Software Practice and Experience*, vol. 34, April 2004, pp. 465-487.

[4] C. Coleman and J. Davidson, "Automatic Memory Hierarchy Characterization," in *IEEE International Symposium on Performance Analysis of Systems and Software*, Tucson, AZ, 2001, pp. 103-110.

[5] S. T. Gurumani and A. Milenkovic, "Execution Characteristics of Spec Cpu2000 Benchmarks: Intel C++ Vs. Microsoft Vc++," in *ACM Southeast Regional Conference*, 2004, pp. 261-266.

[6] D. Jimenez, "Code Placement for Improving Dynamic Branch Prediction Accuracy," in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, June 2005.

[7] S. McFarling, "Branch Predictor with Serially Connected Predictor Stages for Improving Branch Prediction Accuracy," U.S. Patent 6374349, 2002.

[8] T.-Y. Yeh and Y. N. Patt, "Two Level Adaptive Training Branch Prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991, pp. 51 - 61.

[9] T.-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Predictions," in *19th Annual International Symposium of Computer Architecture*, May 1992, pp. 124 - 134.

[10] T.-Y. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Prediction Techniques That Use Two Levels of Branch History," in *Proceedings of the 20th Annual International Symposium of Computer Architecture*, May 1993, pp. 257-266.

[11] S. McFarling, "Combining Branch Predictors," in *DEC WRL TN-36*, June 1993.

[12] E. Sprangle, et al., "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," in *24th Annual International Symposium on Microarchitecture*, May 1997, pp. 284-291.

[13] C.-C. Lee, et al., "The Bi-Mode Branch Predictor," in *30th Annual International Symposium on Microarchitecture*, December 1997, pp. 4-13.

[14] P. Michaud, et al., "Trading Conflict and Capacity Aliasing in Conditional Branch Predictors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997, pp. 292-303.

[15] P. Y. Chang, et al., "Improving Branch Prediction Accuracy by Reducing Pattern History Table Interference," in *Proceedings of the 1996 International Conference on Parallel Architectures and Compilation Techniques*, October 1996.

[16] A. Eden and T. Mudge, "The Yags Branch Prediction Scheme," in *31th International Symposium on Microarchitecture*, November 1998, pp. 69-77.

[17] L. N. Vintan and M. Iridon, "Towards a High Performance Neural Branch Predictor," in *Proceedings of the 9th International Joint Conference on Neural Networks*, 1999, pp. 868-873.

[18] D. Jimenez and C. Lin, "Dynamic Branch Prediction with Perceptrons," in *Proceedings of the 7th International Symposium on High Performance Computer Architecture*, 2001, pp. 197-206.

[19] D. Jimenez, "Fast Path-Based Neural Branch Prediction," in *Proceedings of the 36th Annual International Symposium on Microarchitecture*, December 2003.

[20] E. Jacobsen, et al., "Assigning Confidence to Conditional Branch Predictions," in *29th Annual International Symposium on Microarchitecture*, Paris, France, December 1996, pp. 142-152.

[21] www.intel.com, "Intel® Architecture Software Optimization Reference Manual."

[22] J. P. Shen and M. H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*, 2004.

[23] B. D. Hoyt, et al., "Method and Apparatus for Implementing a Set-Associative Branch Target Buffer," U.S. Patent 5574871, Intel Corporation, 1996.

[24] G. Hinton, et al., "The Microarchitecture of the Pentium 4 Processor," *Intel Technology Journal,* 2001.

[25] D. Genossar and N. Shamir, "Intel® Pentium® M Processor Power Estimation, Budgeting, Optimization, and Validation," *Intel Technology Journal*, vol. 07, 2003, pp. 44-49.

[26] K. Diefendorff, "K7 Challenges Intel " *Microprocessor Report*, vol. 12, October 1998.

[27] R. E. Kessler, et al., "The Alpha 21264 Microprocessor," *Micro, IEEE*, vol. 19, 1999, pp. 24-36.

[28] "Intel Vtune™ Performance Analyzer," www.intel.com/software/products/vtune/.

[29] K. Driesen and U. Hölze, "Accurate Indirect Branch Prediction " in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, 1998, pp. 167-168.

[30] P. Chang, et al., "Target Prediction for Indirect Jumps," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997, pp. 274-283.

[31] L. Rappoport, et al., "Method and System for Branch Target Prediction Using Path Information," U.S. Patent 6601161, Intel Corporation, 2003.

[32] D. Jimenez, et al., "The Impact of Delay on the Design of Branch Predictors," in *Proceedings of the 33th Annual International Symposium on Microarchitecture*, December 2000, pp. 67-76.