# ABSTRACT

The Micro-architectures of modern processors have become sophisticated and complex due to the advancement in VLSI technology. Modern computer architectures rely extensively on advanced branch prediction schemes to keep their ever lengthening pipelines filled with instructions. These mechanisms are so crucial to performance that hardware vendors publish very little about how they actually work, instead keeping the details as "secret sauce". The thesis consists of a set of micro benchmarks to automatically determine the underlying branch prediction architecture of Intel Nehalem and ARM processors.

A number of micro benchmarks have been developed and tested to reveal the organisation details of Branch Target Buffer Predictors and the Loop Predictors. The thesis targets ARM11 processors and Intel core i3 and core i5 processors. The branch misprediction results are collected for every micro benchmark designed and the analysis was performed to reveal the structure of the branch predictor.

An automated analyzer is developed to automatically analyse the results and reveal the branch prediction architecture without involving any manual analysis of the results. This helps greatly in designing compilers that are aware of their underlying architecture which can significantly enhance the performance of the application.

Based on the results of the developed micro benchmarks, the primary contributions of the experiments carried on the Nehalem architecture are:

- Provision to discover the organisation of underlying Branch Target Buffer predictor.
- Revealed the structure of the Loop Prediction Unit implemented.
- Developed an automated analyser that took the results obtained from the micro benchmarks developed, and generated the structure of the branch predictor.

# ACKNOWLEDGEMENTS

I would like to express my love and gratitude to my beloved family, for their understanding, support and endless love throughout the duration of my studies, especially my mother Shobha S Mukartihal, my father Shivaputra D Mukartihal and my brother Santosh.

The work carried out would not have been possible without the guidance and assistance of a number of people who certainly deserve special mention.

I would like to express my gratitude to my supervisor, Mr Simon McIntosh-Smith for his motivation, encouragement, constant guidance and support throughout my work.

My sincere thanks to Professor David May, whose valuable insights through the Advanced Computer Architecture course inspired me to choose a project on Branch Predictors.

I would like to mention my special thanks to Mr David Power, Viglen Ltd, for providing access to their machines upon request from my supervisor.

I would also like to thank, Mr Syed Rahman, IT Support Specialist, Zone E Team Support, for his support in setting up a core i5 system.

I am heartily thankful to my friend Pallavi Anantharamu, for her constant support and motivation to perform better each time.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1.

# INTRODUCTION

## 1.1 Aims and Objectives

The objectives of this project are as follows:

- Analyse and Study the branch prediction structure of various modern computer architectures.
- Decode the branch prediction structure of ARM processors.
- Develop algorithms to aid in determining the branch prediction structure of Intel Nehalem architecture and reveal its organisation details.
- Develop an automated analyser that takes the result data from the benchmark algorithms as input, generating the structure of the underlying branch predictor at its output.

## 1.2 Outline

The outline of the thesis is as follows:

**Chapter 1** provides an insight on the outline of the thesis and discusses the aims and objectives of the project and also talks about the motivation to carry out the work.

**Chapter 2** provides a brief introduction to branch prediction, the importance in modern architectures and different types of basic branch predictors currently used. It also includes discussion of the previous work done in this field.

**Chapter 3** talks about some of the industrial implemented branch predictors, specifically in ARM and Intel processors.

**Chapter 4** provides information about the tools and environment settings. It also briefly discusses the configuration and parameters of the tools used.

**Chapter 5** deals with the work done on decoding the branch prediction structure of ARM processors, specifically about the developed micro benchmarks and the analysis of the obtained results.

**Chapter 6** covers with the work done on determining the organisation of the Branch Target Buffer (BTB) Prediction unit of Intel's Nehalem architecture and talks about the algorithms developed to understand the underlying structure of the BTB.

**Chapter 7** describes the work done on decoding the structure of Loop Predictors and includes discussion on the developed set of micro benchmarks that reveal the structure of loop branch buffers.

**Chapter 8** explains the automatic analyser which is implemented to automatically provide the structure of the underlying branch predictor unit depending on the results fed to the analyser.

## 1.3 Motivation

Most medium to high end processor architectures rely heavily on their underlying branch prediction unit to utilize all its resources and achieve high performance. Most of the modern processor architectures implement deep pipeline levels with many parallel units in the architecture. The performance of these units greatly relies on efficiency of the branch predictor unit.(1)

Intel's Nehalem architecture includes branch prediction units that are amongst the most advanced branch prediction units implemented. In the thesis, we intend to develop a set of micro benchmarks to understand the underlying branch prediction architecture.

The developed algorithms and the automated analyser have great potential to benefit the industry and academicia in several ways:

### 1.3.1 Architecture aware compilers

Knowledge of the underlying branch prediction structure greatly enhances the performance of an architecture aware compiler. Study by (2) reveals that the Intel C++ compiler outperforms the Microsoft VC++ compiler for SPEC CPU200 benchmarks. This suggests that understanding underlying micro architecture features such as branch prediction structures greatly helps in improving the performance of software applications.

### 1.3.2 Branch predictor design verification

Ever increasing processor complexity and strict time to market constraints make the verification of the design in the first place very critical. Very often, architects tend to make changes in the implemented design to satisfy the power, performance and size constraints. Hence it is necessary that the changes are thoroughly tested before releasing. With tight constraints on time, often small micro benchmarks targeting a particular structure of the branch predictor structure become indispensible. Thus the automatic analyser and the micro benchmarks developed can be useful tool in a verification environment.

### 1.3.3 Bridge gap between Academia and Industry

Researchers often tend to concentrate on the accuracy of the branch predictor and rarely consider the area, power and timing constraints that are imposed by the chip industry. On the contrary designers in the industry strive hard to get efficient branch prediction implementations out of the available resources to meet the design constraints.

Although we developed a set of micro benchmarks based on Intel's Nehalem architecture, a slight modification to these benchmarks using a systematic approach can be used to determine the branch prediction structures of other architectures.

# Chapter 2.

# BACKGROUND RESEARCH

## 2.1 Introduction

The section provides a brief introduction to branch prediction, the importance of it in modern computer architectures and the types of branch predictors commonly used. Later, we discuss previous research carried out in this field, specifically a paper on reverse engineering of branch prediction units used in Intel Pentium processors.(3)

## 2.2 Branch Prediction

In modern computer architectures the concept of pipelining is implemented which includes many pipeline stages, such as instruction fetch, decode, register allocation, µop reordering, multiple execution phases, load/store, and memory write back (1). Pipeline implementations are used as these enable the performance of different operations at same time and also help in efficient resource utilization, as all units are operated at any given time. The next instruction is fetched when the previous instruction is in the decode phase and so on. But the greatest hazard for the smooth operation of pipelines is the branch instruction. Once a branch instruction is encountered in the fetch phase of the pipeline, the address of the following instruction is not known until this branch instruction completes the execute phase of pipeline. This results in pipeline stalls and hence causes significant problems.

To overcome pipeline stall, branch predictors are used in modern processors. Branch predictors guess whether a branch is likely to be taken or not and also where the target is likely to be if the branch is taken. Branch prediction deals with predicting the direction and target address of the branch before it reaches the execute phase of pipeline, thus helping next instructions to be fetched from correct path. Thus branches certainly have a great impact on performance in processors which implement more number of pipeline stages like Intel Nehalem.

## 2.3 Importance

Branch Predictors play a vital role in modern superscalar and instruction-level parallelism (ILP) processors. These processors have deeper pipelines and higher instruction issue rates, thus branch predictors with a high precision are mandatory. "*It has been predicted that within a few years branch prediction becomes the most limiting factor in the performance of a processor, more so than the memory system*". (4) Branch mis-prediction causes heavy penalty resulting in complete pipeline flushing and re-fetching of instructions from the correct location.

Sophisticated branch predictors are developed by designers to greatly reduce the frequency of mis-predictions in complex computer architectures with deep pipelines. Most of the processors use both *static* and *dynamic* branch predictors in their architecture to predict the outcome of a branch. The most commonly used static prediction is that all backward branches are predicted as ‘taken’ while forward branches are predicted as ‘not taken’. The most commonly used dynamic branch predictor is predicting the outcome of current branch based on its previous outcomes (its *history*).

*“Even with generous resource allocation and no limit on the amount of state reached in one cycle, branch prediction is expected to create the most limiting bottleneck in future processors”* (4).

### 2.3.1 Performance Impact

Table 2-1 shows the average branch distance and average branch ratio of some of the SPEC benchmark programs. The table is grouped into two types depending upon the type of operation carried out. The top half of the table gives details of the general purpose (integer) benchmark programs while the bottom half gives the details of the scientific (floating point) benchmark programs. It can be observed that the average branch distance in general purpose programs and scientific programs varies greatly. It is also clear that there are more branches in general purpose programs compared to scientific programs.

The average distance between branches in general programs is 4.6 instructions, while in scientific programs it is 9.2 instructions. This suggests that in general purpose programs every 4th - 6th instruction can be a conditional branch instruction. In contrast, every 9th – 15th instruction is a conditional branch instruction in scientific programs.

| Program | Type | Average branch distance | Average branch ratio |
|---|---|---|---|
| Eqntott | **General Purpose** | 2.86 | 35.0 |
| Espresso | | 4.81 | 20.8 |
| Li | | 6.05 | 16.5 |
| **Average** | | **4.57** | **21.9** |
| Doduc | **Scientific** | 10.59 | 9.4 |
| Matrix 3000 | | 5.05 | 19.8 |
| Nasa 7 | | 10.72 | 9.3 |
| Spice 2g6 | | 3.27 | 30.6 |
| Tomcat | | 16.28 | 6 |
| **Average** | | **9.18** | **10.9** |

**Table 2-1: average branch distance and average branch ratio for different SPEC standards (5)**

Superscalar and ILP processors boost performance by executing more and more instructions in parallel. But as the number of instructions executed in each cycle increases the probability of encountering a branch instruction also increases and thus limits the performance of the superscalar or ILP. For example consider a general purpose code in which on an average every sixth instruction is a branch instruction as shown in Figure 2.1. Observe that if the issue rate is only two instructions per cycle then on average every third issue contains a conditional branch. Now consider if the issue rate is increased to three, then every second issue contains a conditional branch. The situation gets worse as the issue rate is increased. For instance if the issue rate is increased to six then every issue will contain a branch instruction and this greatly degrades the performance of the processor and even suppresses the advantage of having many execution units and executing more instructions in parallel.

Thus from the above example it is clear that predicting the outcome of unresolved conditional branches becomes an important task to achieve high performance. Significant importance is given to the branch prediction unit in the architecture of a processor as it plays a vital role to determine the performance of the processor. The next section gives insight to some of the simple and most commonly used branch prediction techniques.

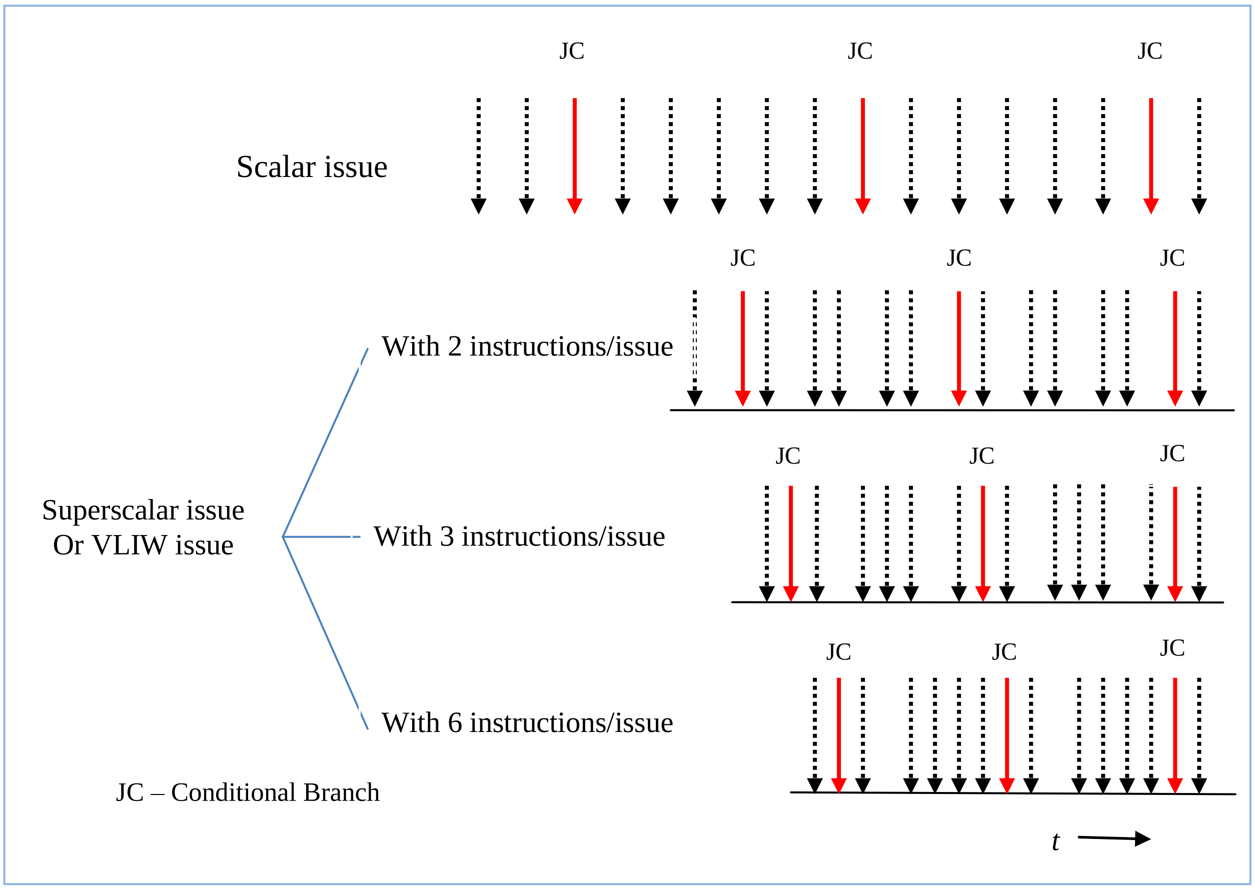


**Figure 2.1: Impact of increasing the instruction issue rate on the frequency of conditional branches per issue (5).**

## 2.4 Branch Prediction Techniques

On encountering the conditional jump, the branch prediction unit predicts the following:

- The target branch address.
- Whether the conditional jump will be taken or not.

The basic kinds of branch predictions used are:

- *Fixed prediction*: here the prediction is always the same, either always 'taken' or always 'not taken'. Also called 'one outcome guess'.

- *True Prediction*: also called ‘two outcome guess’, the prediction can be either ‘taken’ or ‘not taken’ and is divided into two types, namely:
  - Static Prediction: here predictions are made based on the object code.
  - Dynamic Prediction: here the prediction is made based on the execution history.

### 2.4.1 Static Branch Prediction

Here the prediction is made on particular attributes of the object code. The following are the most commonly used static branch prediction methods:

- *Opcode-based prediction*: The prediction of a branch depends on the opcode of the instruction. The branch can be ‘taken’ for certain opcodes and ‘not taken’ for others.
- *Displacement-based prediction*: The branch is predicted based on the displacement of the branch target address (BTA). If the BTA is less than zero (i.e. a backwards branch) then it is assumed to be ‘taken’, else it is a forwards branch and assumed to be ‘not taken’. This method is very popular among static branch prediction.
- *Compiler-directed prediction:* As the name indicates, the prediction is made by the compiler itself based on the control construct compiled and is indicated by either setting or clearing a ‘predict’ bit while encoding the branch instruction.

### 2.4.2 Dynamic Branch Prediction

In this scheme the branch history is used to predict the branch direction and target branch address. The following are the different types of dynamic branch prediction techniques.

#### 2.4.2.1 2-bit Dynamic Branch Prediction Technique

In a 2-bit dynamic branch prediction, two bits are used to maintain the last occurrence of the particular branch and as it makes use of 2-bits there are four possible states - strongly taken, weakly taken, weakly not taken and strongly not taken. The state diagram for a 2-bit dynamic branch prediction is as shown in Figure 2.2:

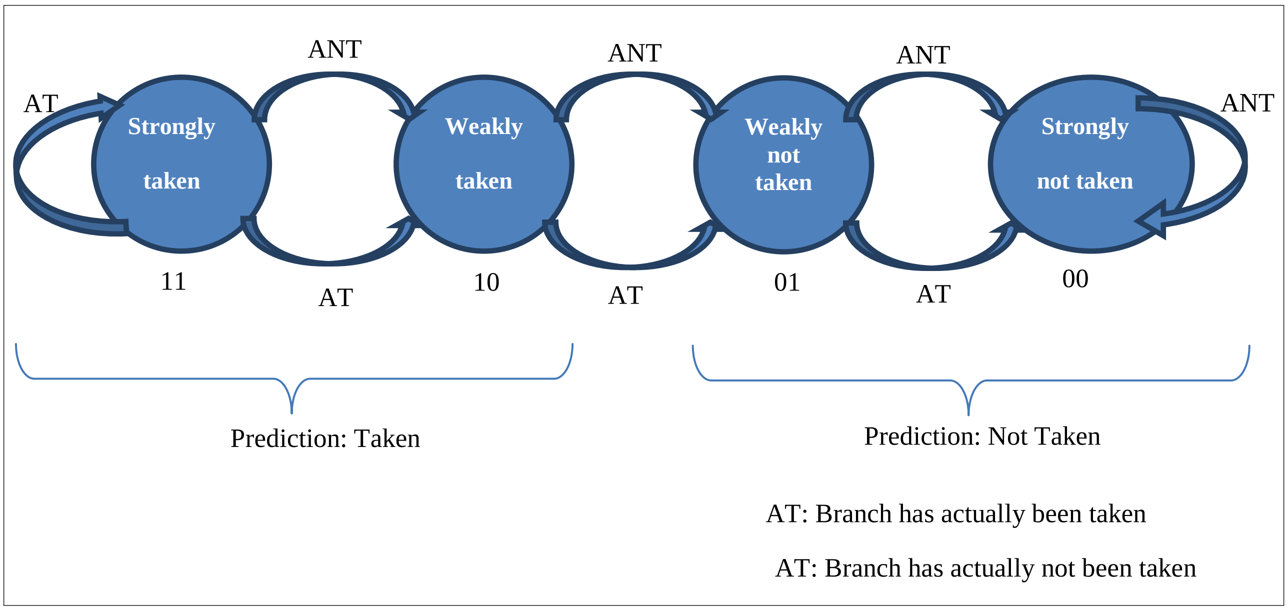


**Figure 2.2: 2-bit branch predictor**

The branch is predicted based on the status stored in the counter as follows. If the state is either ‘strongly taken’ or ‘weakly taken’, then the branch is predicted as ‘taken’ else if the state is either ‘weakly not taken’ or ‘strongly not taken’ then the branch is predicted as ‘not taken’. After the conditional branch is resolved, the status is updated as a two-bit saturating counter in the history table.

The advantage of this branch prediction is it performs well for a branch whose target address is the same most of the time, as is the case for many conditional branches. But its performance degrades if the branch target address changes too often (6).

### 2.4.2.2 Two Level Adaptive Predictor

A two level adaptive predictor maintains the history of the last *n* occurrences of the branch in a special cache called the pattern history table, as illustrated in Figure 2.3. Since there are *n* occurrences this table consists of $2^n$ history patterns and each entry internally is a saturating counter as was illustrated in section 2.4.2.1.

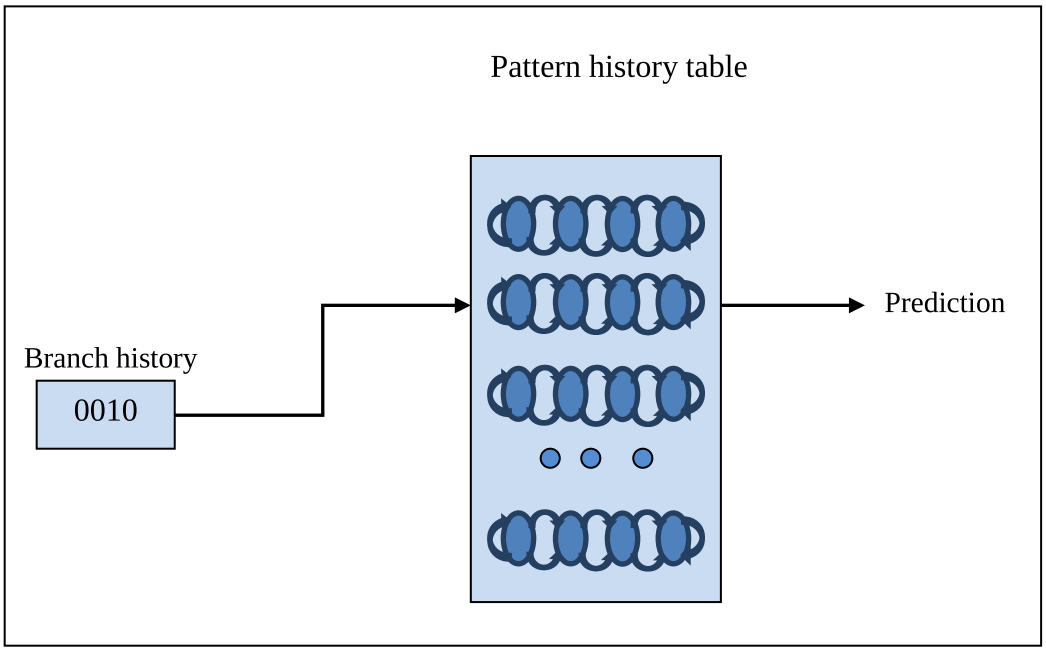


**Figure 2.3: Two level adaptive predictor**

Now, consider n=2. If n=2 then the branch history is a two bit entry and corresponding to $2^n$ = four different binary values (‘00’, ‘01’, ‘10’ and ‘11’, where ‘0’ → ‘not taken’ and ‘1’ → ‘taken’). Each entry in the pattern history table is a 2-bit saturating counter as discussed in Figure 2.2. The two bit value of each branch history register chooses which particular entry of the available four entries in the pattern history table is to be considered. If the branch history register is ‘00’ then the first saturating counter in the pattern history table is used and if the branch history register contains ’11’ then the last (fourth) entry from the pattern history table is used.

*The general rule for a two-level adaptive predictor with an n-bit branch history register is as follows:*
*Any repetitive pattern with a period of n+1 or less can be predicted perfectly after a warm-up time no longer than three periods. A repetitive pattern with a period p higher than n+1 and*

*less than or equal to 2n can be predicted perfectly if all the p n-bit sub-sequences are different* (7).

This method overcomes the cons of two-bit dynamic branch prediction techniques and quickly learns to predict the outcome of an arbitrary repetitive branch pattern.

### 2.4.2.3 Local Branch Prediction

A local branch predictor is the extension of the previously illustrated adaptive predictor. It makes use of a separate history buffer register and a pattern history table for each of the conditional jump instructions. Its implementation is illustrated in Figure 2.4. It predicts the outcome of a branch depending upon the previous ‘n’ outcomes of the branch instruction.

The Intel Pentium II and Pentium III processors use local branch predictors to predict the outcome of a conditional jump. They implement local branch predictors with local 4-bit history and a 16-entry local pattern history table for each conditional jump.

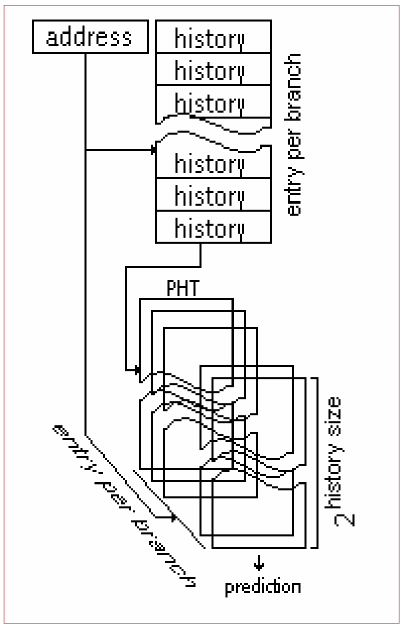


Figure 2.4: Local branch predictor (8)

### 2.4.2.4 Global Branch Prediction

In section 2.4.2.2, we discussed a two-level branch predictor. We note that it uses an ‘n’ bit history register and a ‘$2^n$’ pattern history table for each conditional branch. It should be noted that there is a practical limit as to how big the number of history bits ‘n’ can be, as the cache size requirement increases dramatically with ‘n’. One way to overcome this will be to use a shared branch history register and a shared pattern history table among all the branches. This arrangement leads to global branch prediction algorithm.

A global two-level branch predictor is shown in . This implementation maintains a single branch history register for all the conditional branches/jumps. It predicts the outcome of the branch based on the pattern of the outcome of the preceding ‘n’ branches. The advantage of this prediction is that it uses the correlation between branches to predict the outcome. But the performance of this predictor is poor if the different branches are not correlated.

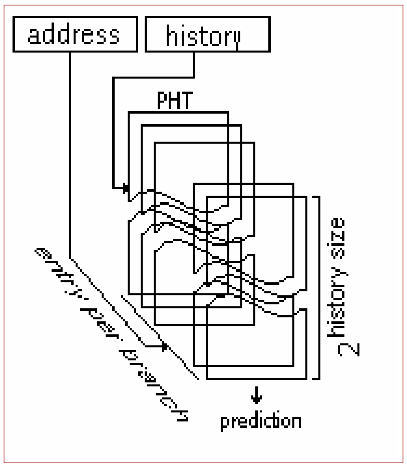


Figure 2.5: Global branch predictor (8)

This method has been implemented in processors such as some of AMD’s 64 bit processors (n=8) and P4E processor (n=16) (6).

### 2.4.2.5 The Agree Predictor

The disadvantage of using global branch predictors is that they make use of the correlation between branches and even the branches with no correlation share the same global pattern history table. This problem can be reduced by storing an extra bit called a ‘biasing bit’ in the BTB or instruction cache which can be used to indicate whether the branch is ‘mostly taken’ or ‘mostly not taken’. In an agree predictor, instead of indicating the direction of a branch, the pattern history table now indicates whether the branch will be ‘taken’ or ‘not taken’ in the direction of the ‘biasing bit’. This scheme effectively reduces the negative interference if a proper biasing bit is used.

An implementation of an agree predictor is as shown in Figure 2.6. Each branch implements a local predictor using a saturating counter and the indexing function is implemented by using an XOR function. The global branch history table is indexed by the global branch history and branch address, and indicates whether the branch is predicted to agree or disagree with the output of the local predictor (9).

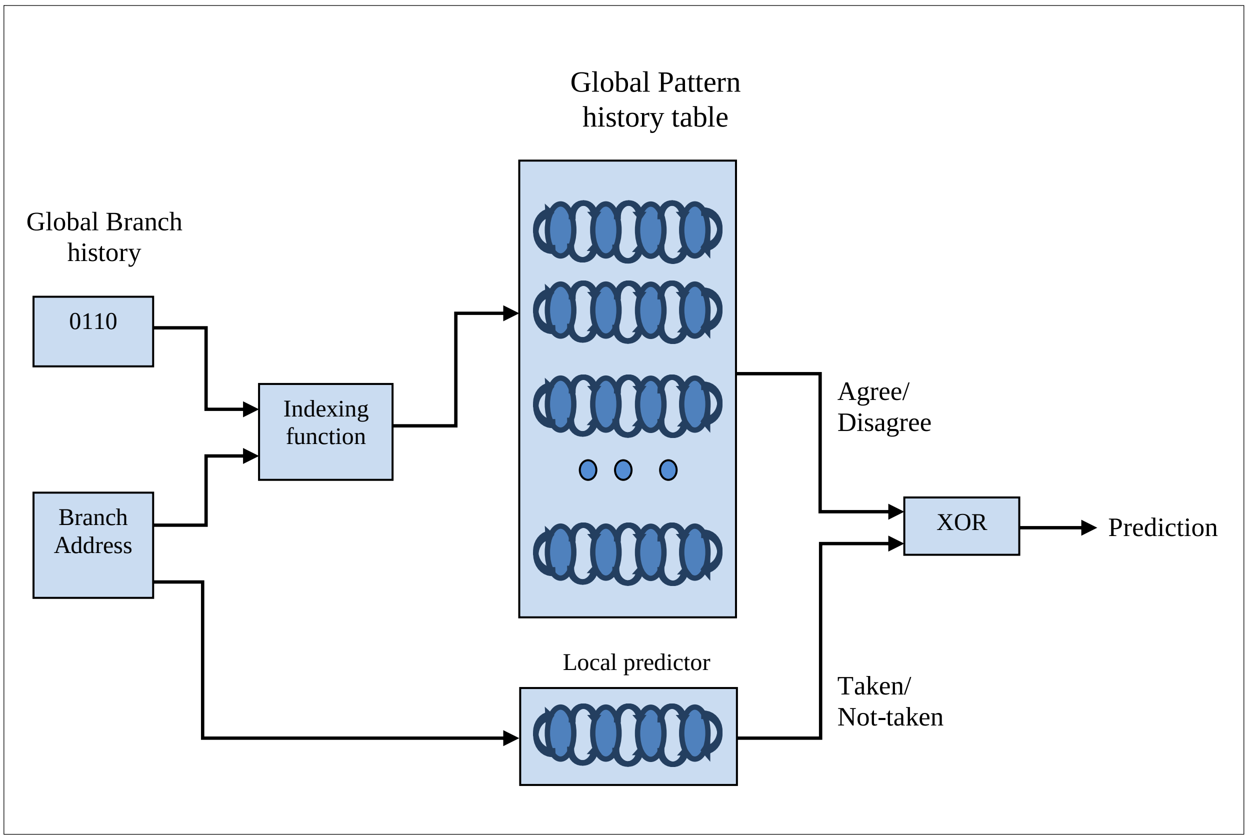


**Figure 2.6: Agree predictor**

### 2.4.2.6 Loop Predictor

Loop predictors are used to predict loop-delimiting branch instructions. For example consider a loop with ‘N’ iterations in it. A conditional branch inside a loop structure, when placed at the bottom of the loop will be ‘taken’ N-1 times and will be ‘not taken’ the last time. While the same conditional branch instruction when placed at the top of the loop will be ‘not taken’ N-1 times and will be ‘taken’ the last time. Such a conditional branch which goes in one direction for many times and then the other way is termed to show loop behaviour. Such a branch can be easily predicted by using a simple counter. A loop predictor is a type of hybrid predictor where a meta-predictor detects whether the particular branch exhibits loop behaviour.

Intel’s Pentium M and Core2 processors use a loop predictor with a loop counter of 6 bits. Any loops with a loop iteration count of less than 64 (=$2^6$) are predicted perfectly (6).

### 2.4.2.7 Indirect Branch Predictor

With more and more applications being developed using object oriented code, the correct prediction of indirect branches is becoming crucial. C++ code can generate indirect jumps by using function pointers, virtual function or by calls with switch statements. An indirect jump or call is a branch instruction that has more than two possible target addresses. Normally, the target branch address is part of the opcode, but in indirect branches it is specified by using a value in a register, a memory variable or an indexed pointer as the destination of the jump or call instruction.

Normally in many processors the BTB is implemented such that only one entry is made for every indirect branch or call. This results in a jump being predicted to be always to the same branch target address as last time, which is incorrect. Thus, an indirect branch prediction should be implemented by assigning a new BTB entry for every new branch target address that is encountered. The branch history register and the pattern history table must be suitably modified to hold more than one bit of information for every branch in order to distinguish between more than two possible targets (10).

*Working of Branch target buffer (BTB):*

A branch target buffer (BTB) is a special cache which is used to store the branch address and the branch target address of a branch instruction. When a conditional branch is encountered for the first time, the branch target address is stored in the BTB and during subsequent executions of the same branch, the branch target address stored in the cache is retrieved to fetch the next instruction into the pipeline, although the actual target address is not calculated until the branch instruction has been resolved in the execution stage of the pipeline.

### 2.4.2.8 Hybrid Predictor

*Researchers have shown that the most effective single scheme predictors use a large amount of branch history information and that two such predictors combined can outperform a single predictor at the same implementation cost* (11).

A hybrid predictor combines two or more different branch predictors, each selected for its ability to accurately predict a particular class of branch. Hybrid predictors make it advantageous to have more than one branch predictor in architecture. For example, most recent architectures have different branch mechanisms for indirect branches, loops, BTBs, return stacks etc.

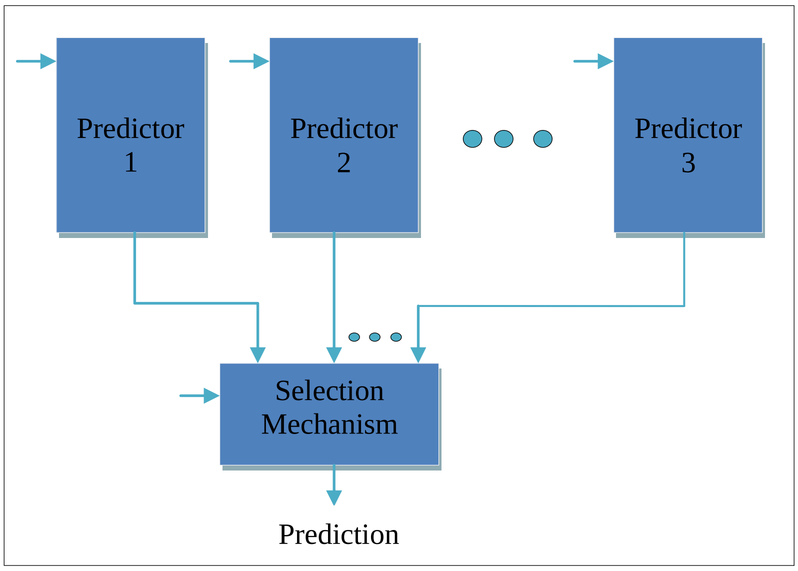


**Figure 2.7: hybrid prediction**

#### 2.4.2.9 Return stack buffer

The return stack is a buffer that is used to store the return addresses of call instructions. Each time a call instruction is encountered, the corresponding return address is stored in the return stack. When a return instruction is encountered, the corresponding return address is popped back from the stack. Since it is possible that the same procedure might be called from different locations it is necessary to pair up the return address with the corresponding subroutine calls in order to maintain correctness. Return instructions are a special case of indirect branches.

Unfortunately a return stack may be corrupted due to branch mis-prediction, an effect which is more likely to occur in deeper pipeline architectures. Inappropriate handling of this problem may seriously affect performance. Return stack corruption is overcome by recording the push and pop after any mis-predicted branch (12).

Although this section gave an insight on common branch predictors, many different kinds of branch predictors do exist (13),(14). It is likely that even more sophisticated branch prediction methods will be implemented in the future if pipelines are extended further.

## 2.5 Previous Work

Reverse engineering of micro architectural features has been a topic that has generated intense discussion over the years. Early work done by *Aleksandar Milenkovic and Milena Milenkovic*(15) on 'unveiling the structure and organisation of Intel branch predictors', gives significant insight on the way to perform analysis of complex branch predictors. The paper mainly concentrated on Intel NetBurst architecture and the Intel P6 architecture. Another paper by *Vladimir Uzelac and Aleksandar Milenkovic* (3) gives valuable insight on reverse engineering of the branch predictor unit of the Intel Pentium M. This paper mainly focuses on developing a set of microbenchmarks to decode the underlying structure of the branch prediction unit.

In the *Pentium Optimisation Manual* (6), *A. Fog* describes the research work carried on analysing the branch predictors of various Intel and AMD architectures. In this section we discuss the contributions made by these three papers and reveal the importance of these papers for analysing the branch prediction unit of Intel's Nehalem architecture.

The paper by *A.Milenkovic and M.Milenkovic*(15) mainly focuses on the BTB component and the Outcome predictor component used in Intel's Netburst and P6 architectures. The inferences from the paper are as follows:

- BTB organisation (P6 architecture):
  1. The total size of BTB is 1024 entries.
  2. The BTB is organised as a 4-way set. Each set has 128 entries.
  3. The BTB cache is addressed (indexed) by the branch address bits [10:4].
  4. The LSB bits [3:0] represent the offset.
- Global predictor: It confirms that the P6 architecture does not include any global prediction component.

- Front-end BTB organisation (NetBurst architecture) :
    1. The total size of BTB is 4096 entries.
    2. The BTB is organised as a 4-way set. Each set has 1024 entries.
    3. The BTB cache is addressed (indexed) by the branch address bits [13:4].
    4. The LSB bits [3:0] represent the offset.
- Global predictor: It confirms that the NetBurst architecture has a global prediction component with 16 global history bits.

The research work carried by A.Fog(6) reveals some important information about the organisation of the branch predictor structures used in Intel, AMD and VIA CPUs. Although it gives a good insight of most of the industrial branch predictors it does not show the exact details of how the experiments were carried out. It also mentions very little about the Intel Nehalem branch predictor structure. The research by A.Fog done on the Intel Nehalem and Intel Atom architectures is briefly discussed in sections 3.2.1 and 3.2.2. In this thesis we intend to verify some of the findings of A.Fog about the Nehalem architecture and also to extend these methods to reveal the complete structure of the branch predictor used.

*Uzelac and Milenkovic* (3) have published the complete branch predictor structure of the Pentium M processor architecture and have developed a number of micro-benchmarks to reveal the underlying branch prediction structure. The following are some of the important results of their experiments:

- The BTB is organised as a 4 way set associative structure with 512 entries in each set with a total number of 2048 entries. The BTB is indexed by the branch address bit [12:4] and the branch address bits [21:13] are used as a tag field.
- It implements an outcome predictor with a 4096 entry bimodal predictor.
- It also implements a loop predictor with total of 128 entries organised as a 2 way set associative cache indexed by branch address bits [9:4].
- A 4-way global predictor is incorporated with 2048 entries.

We intend to carry out our study of Intel Nehalem's branch prediction structure in a similar way as carried in (15) and (3), as the branch prediction unit of Nehalem is a derivative of that in the Intel Pentium M. We predict that the branch prediction structure of Nehalem is an extended version of its previous architecture. We intend to develop some of the algorithms that were used in the paper by *Uzelac and Milenkovic* (3) and build on these algorithms as required.

In this thesis we are mainly concentrating on two types of branch prediction structures:

1. Branch Target Buffer (BTB) Predictors.

2. Loop predictors.

# Chapter 3.

# INDUSTRIAL BRANCH PREDICTORS

This section introduces some of the branch predictors implemented in real processors. It describes the branch predictors that are published by the vendor or revealed by some research carried out previously. We cover the types of branch predictors that are implemented in ARM and Intel Processors (Intel Atom and Intel Nehalem architecture).

## 3.1 ARM PROCESSOR

ARM processors implement relatively simple branch predictor structures compared to Intel processors, for example. Following are the details of the branch predictors that are used in different families of ARM processors.

### 3.1.1 ARM 7

The ARM 7 series ARM7TDMI and ARM7EJ-S processors do not implement any branch prediction mechanism. These processors implement a simple three stage pipeline architecture with no pre-fetch unit, hence the target of the branch address is not known until execute stage of the pipeline, at which time it is known whether the branch is ‘taken’ or ‘not taken’.

In these processors all the branches are predicted as ‘not taken’ and the pipeline is filled with the instructions that follow the branch instruction.

### 3.1.2 ARM 8

The ARM 8 series ARM810 processor implements a four stage pipeline architecture with a pre-fetch unit. Due to the presence of pre-fetch unit the branch instructions are detected before they enter the core, hence it is possible to implement a branch prediction mechanism.

ARM 8 implements a simple static branch prediction mechanism. All the forward branching branches are predicted as ‘not taken’ and all the backward branching branches are predicted as ‘taken’. This results in the reduction of average branch Cycles Per Instruction (CPI) and thus improves the processor’s performance.

### 3.1.3 ARM 9

The ARM 9 series ARM9TDMI, ARM926EJ-S and ARM946EJ-S processors do not implement any branch prediction mechanism. These processors implement a five stage pipeline architecture with no pre-fetch unit, hence the target of the branch address is not known until execute stage of the pipeline, at which time it is known whether the branch is ‘taken’ or ‘not taken’.

In these processors all the branches are predicted as ‘not taken’ and the pipeline is filled with the instructions that follow the branch instruction.

### 3.1.4 ARM 11

The ARM 11 series ARM1136 and ARM11 MPcore processors have an 8-stage pipeline architecture and implement a branch prediction unit. The program flow prediction is carried out as follows:

- The Core implements Static Branch Prediction and a Return Stack.
- The Pre-fetch Unit implements Dynamic Branch Prediction.

**Dynamic branch prediction**

ARM1136JF-S makes use of a dynamic first level branch predictor. The dynamic branch predictor is implemented by using a single Branch Target Address Cache (BTAC) and includes a 128 entry directly mapped cache structure which is used to store the branch address, branch target address and prediction of whether the branch is ‘taken’ or ‘not taken’. Dynamic branch prediction is done by using the two bit saturating counter inside the BTAC. Once a branch is allocated in the BTAC, it is only evicted during a capacity clash or due to any address mapping conflicts.

A branch entry is allocated into the BTAC after the resolution of the branch in the execute stage of the pipeline. Whenever a branch is encountered, the BTAC is searched and if a BTAC hit occurs then the branch target address stored in the BTAC is used as a program counter to fetch the next instruction. If a BTAC hit occurs then the branch is predicted with a zero cycle delay.

**Static branch prediction**

ARM1136JF-S implements a second level of branch prediction called static branch prediction, where all unconditional branches are predicted as always ‘taken’. All conditional backward branches are predicted as always ‘taken’ and conditional forward branches are predicted as always ‘not taken’. All the branch instructions that are encountered for the very first time are predicted by using this static branch prediction scheme. The branches that are evicted from the BTB due to a capacity clash or address conflict are also predicted using the static branch predictor. Thus the first time any branch instruction is met it is predicted with the static branch predictor before subsequently being handled by the dynamic branch predictor. However the static branch prediction can be enabled or disabled by setting a bit in a configuration register.

**Return stack**

The return stack is implemented as a three-entry circular buffer, which is used to predict procedure calls and unconditional procedure returns.

“*Using a combination of dynamic and static branch prediction we can typically correctly predict whether the branch will be taken about 85% of the time. If the branch prediction fails, the core pipeline will need to be refilled and the current instruction pipeline flushed*”. (16)

It can be inferred that the ARM cores use fairly traditional branch predictors. Hence in our project we began by developing algorithms to analyse the branch predictors of ARM cores.

## 3.2 Branch Prediction Techniques - Intel

### 3.2.1 Branch Prediction in Intel Atom

The Intel Atom processor architecture makes use of a two level adaptive branch predictor with a global history table. The following details have been found on the Intel Atom architecture from the research done by Fog Agner(6). They have found that it uses a BTB with 128 entries, a branch history register with 12 bits and a pattern history table with 4096 entries. The pattern history table entries are shared between threads. And, conditional ‘taken’ and ‘never taken’ branches are entered into the global history table while unconditional branches are treated separately and do not have entries in the history table.

**Misprediction penalty**

The BTB stores the branch target address and in Intel Atom the BTB is much smaller than in other architectures. Thus it may happen that there might be a valid entry in the pattern history table for a branch but due to no entry in the BTB (a BTB miss), the branch is not predicted to completion. This results in a penalty of approximately 7 clock cycles. If a branch is completely mis-predicted then the penalty is 13 clock cycles. It is most likely that there is no separate branch predictor to predict the outcome of indirect branches, but instead indirect branches in Atom are predicted to jump to the same target as last time. It has a return stack buffer (RSB) with 8 entries (6).

### 3.2.2 Branch Prediction in Intel Nehalem Architecture

The Nehalem architecture is designed to be modular to target servers and high performance workstations. In this sub section we discuss the branch prediction architecture that is incorporated in this architecture.

#### 3.2.2.1 Branch Prediction Unit

Intel has not published much on the branch prediction mechanism used in the Nehalem architecture. There is very little literature that gives an insight on what might be the branch prediction mechanism used in this architecture. Following are the details of the possible branch prediction mechanisms used in the Nehalem micro architecture.

**Pattern recognition for conditional jumps**

As it is clear that Nehalem has a long mis-prediction delay, significant effort has been dedicated to develop an improved branch predictor. Below are the salient features of the branch prediction mechanism used:

- A hybrid branch predictor has been implemented consisting of a two-level branch predictor and a separate loop predictor.
- It has a mechanism to predict indirect jumps and indirect loops.

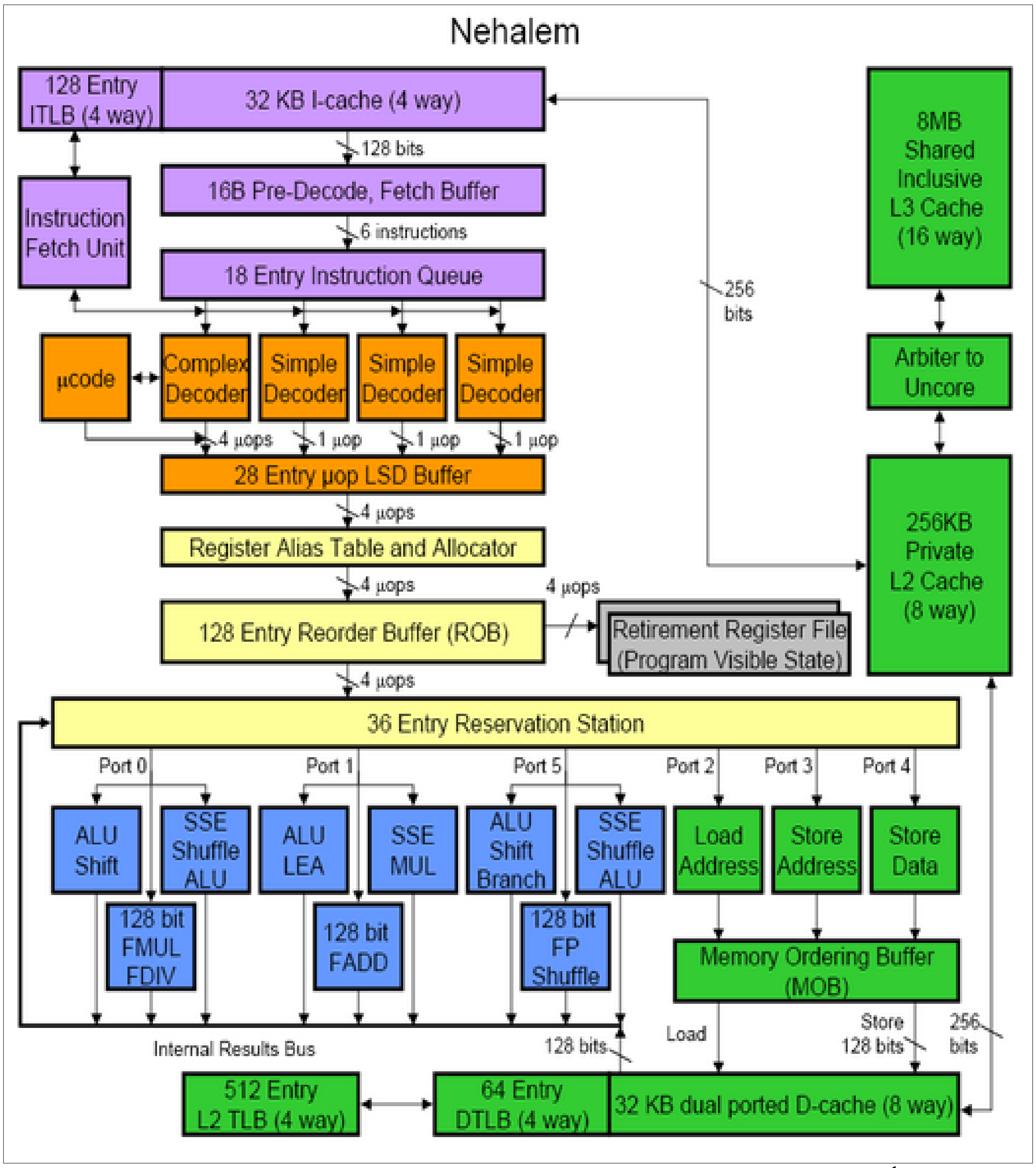


**Figure 3.1:Intel Nehalem Pipeline structure, Source: Real World Tech(17)**[1]

## Non loop behaviour branch prediction

A two level predictor is used to predict branches which do not show loop behaviour. It uses an 18-bit global history buffer with an associated pattern history table. The efficiency of the prediction of the branches with particular repetitive patterns depends on the number of jumps in the loop. It is revealed that Nehalem provides two 18-bit global history buffers: the first one includes all conditional and unconditional branches but excluding never taken branches. The second buffer includes only the important branches such as conditional branches.

[1] Figure is not of high quality as the source is a webpage.

**Return Stack Buffer**

The return stack buffer (RSB) is used to store the return address of procedure calls. However as seen in section 2.4.2.9, the return stack buffer may get corrupted when a branch predictor speculates a wrong path or due to stack overflow. Under these circumstances the Nehalem architecture renames the RSB which ensures that the RSB does not overflow while at the same time making sure that the mis-speculation does not corrupt the RSB. Also Nehalem implements a separate RSB for each of the threads ensuring no cross-contamination. (18)

**Misprediction penalty**

The mis-prediction penalty usually depends on the length of the pipeline used in the architecture. Intel Nehalem uses a longer pipeline than its predecessor the Core2. In fact the mis-prediction penalty in Nehalem is 17 clock cycles, which is substantial. The complex pipeline structure of Intel Nehalem is shown in Figure 3.1.

# Chapter 4.

# TOOLS AND ENVIRONMENT SETTINGS

This chapter focuses on various tools used to setup the environment for the experiments to be conducted. The performance monitor register subsection describes the event count registers that were used during the analysis. We also discuss the hardware events that were used to extract the branch related information such as branch misprediction percentages and the number of branches executed.

## 4.1 Tools

Following are the set of tools that are used in this project.

### 4.1.1 ARM Workbench IDE v4.0 and RVDS

- The ARM Workbench is an Integrated Development Environment (IDE) which can be used to create, debug, profile, flash and trace C/C++ projects for all ARM processor based targets.
- The RVDS 4.1 Professional IDE comes with an in-built ARM profiler which enables non-intrusive analysis of embedded software performance over long runs. It can gather profile information for code running over minutes, hours or even days. It can operate at a frequency of up to 400MHz.
- It has got a built-in fully featured assembler and C/C++ editor, built-in syntax highlighting and provides assembler files with customizable formatting of code so that it is readable.

The ARM Profiler provides instruction coverage, code coverage and also branch coverage. This information can be used to test the quality level and the effectiveness of the implementation of the embedded software. The Profiler also allows combining multiple analysis runs into a single report. The branch coverage information provides details of the total number of branch instructions in the code, the number of branch mis-predictions, etc. (19)

The RVDS tool was used to extract the branch target addresses and branch addresses. This information was used to analyse the behaviour of the branches for various different sets of micro-benchmarks.

RVDS was also used to monitor the ‘performance monitor control registers’ contents which included the events count register such as:

- Cycle Count Register (CCNT)
- Count Register 0 (PMN0)
- Count Register 1 (PMN1)

The Count Registers were configured to count the following events:

- Branch Instruction Executed.
- Branch Instruction Mis-predicted.

### 4.1.2 Intel Vtune

- Intel provides a tool called the Intel VTune Amplifier XE that can be used for performance analysis and tuning of the system and software applications. It has various different tools integrated into its environment and provides a user interface for collecting performance related data.
- The utility provides different analysis techniques allowing the user to add one or more tasks and also specify the type of performance data to be collected.
- Any analysis type provided by VTune is based on one of the following data collections types:
  - Hardware event-based sampling collection.
  - User-mode sampling and tracing collection.

"*The VTune Amplifier XE also enables you to create custom analysis types based on one of these collection types"*.(20)

- *Algorithm analysis* is the analysis type that can be used to tune software. It helps to improve the performance of an application by providing an environment to run various analyses and collect data to decide the best algorithm.
  - *Algorithm analysis* includes the following analysis types:
    - Lightweight Hotspots
    - Hotspots
    - Concurrency
    - Locks and Waits
- *Advanced Hardware Level Analysis*: This analysis is particularly targeted to Intel Core 2 processor family, Intel Nehalem micro-architecture and the Intel next generation Sandy Bridge micro-architecture. Data collection is by event-based sampling.
  - Following are the Intel micro-architecture code name Nehalem analysis types:
    - General Exploration
    - Cycles and uOps
    - Front End Investigation
    - Memory Access

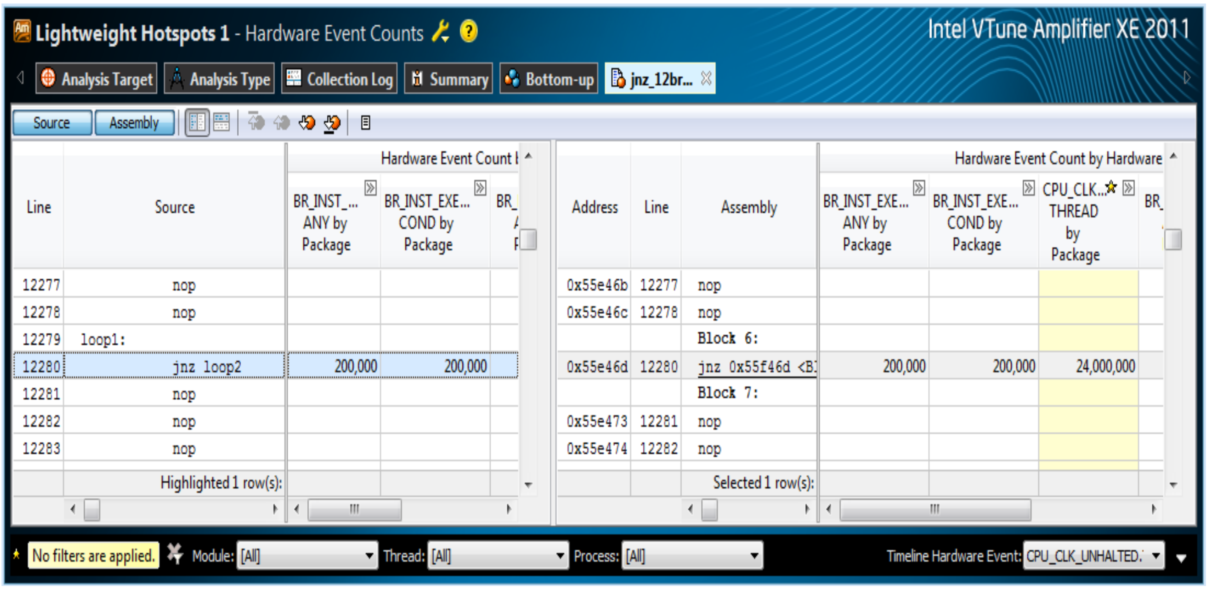
The Intel Vtune's 'Custom Analysis' type is used to get access to hardware event counters. The 'LightWeight Hotspot' Analysis is carried out under the 'Custom Analysis'. The following are the branch-related hardware events that are provided by the 'Intel Core Processor Family':

| Event Name | Event Description |
|---|---|
| BR_INST_DECODED | Branch Instructions Decoded |
| BR_INST_EXEC.ANY | Branch Instructions Executed |
| BR_INST_EXEC.COND | Conditional Branch Instructions Executed |
| BR_INST_EXEC.DIRECT | Unconditional Branches Executed |
| BR_INST_EXEC.DIRECT_NEAR_CALL | Unconditional Call Branches Executed |
| BR_INST_EXEC.INDIRECT_NEAR_CALL | Indirect Call Branches Executed |
| BR_INST_EXEC.INDIRECT_NON_CALL | Indirect Non Call Branches Executed |
| BR_INST_EXEC.NEAR_CALLS | Call Branches Executed |
| BR_INST_EXEC.NON_CALLS | All Non Call Branches Executed |
| BR_INST_EXEC.RETURN_NEAR | Indirect Return Branches Executed |
| BR_INST_EXEC.TAKEN | Taken Branches Executed |
| BR_INST_RETIRED.ALL_BRANCHES | Retired Branch Instructions |
| BR_INST_RETIRED.CONDITIONAL | Retired Conditional Branch Instructions |
| BR_INST_RETIRED.NEAR_CALL | Retired Near Call Instructions |
| BR_INST_RETIRED.NEAR_CALL_R3 | Retired Near Call Instructions Ring 3 only |
| BR_MISP_EXEC.ANY | Mispredicted Branches Executed |
| BR_MISP_EXEC.COND | Mispredicted Conditional Branches Executed |
| BR_MISP_EXEC.DIRECT | Mispredicted Unconditional Branches Executed |
| BR_MISP_EXEC.DIRECT_NEAR_CALL | Mispredicted Non Call Branches Executed |
| BR_MISP_EXEC.INDIRECT_NEAR_CALL | Mispredicted Indirect Call Branches Executed |
| BR_MISP_EXEC.INDIRECT_NON_CALL | Mispredicted Indirect Non Call Branches Executed |
| BR_MISP_EXEC.NEAR_CALLS | Mispredicted Call Branches Executed |
| BR_MISP_EXEC.NON_CALLS | Mispredicted Non Call Branches Executed |
| BR_MISP_EXEC.RETURN_NEAR | Mispredicted Return Branches Executed |

| | |
|---|---|
| BR_MISP_EXEC.TAKEN | Mispredicted Taken Branches Executed |
| BR_MISP_RETIRED.ALL_BRANCHES | Mispredicted Retired Branch Instructions |
| BR_MISP_RETIRED.CONDITIONAL | Mispredicted Conditional Retired Branches |
| BR_MISP_RETIRED.NEAR_CALL | Mispredicted Near Retired Calls |

Vtune enables assembly level code debugging which can be used to track the addresses of branch instructions. This helps in calculating the actual branch target address bits. This information is helpful in hand analysis of some critical cases.

The Figure 4.1 illustrates how the Vtune tool is used to view the assembly language and the branch address.



| | | Hardware Event Count b | | |
|---|---|---|---|---|
| Line | Source | BR_INST_... ANY by Package | BR_INST_EXE... COND by Package | BR_ A F |
| 12277 | nop | | | |
| 12278 | nop | | | |
| 12279 | loop1: | | | |
| 12280 | jnz loop2 | 200,000 | 200,000 | |
| 12281 | nop | | | |
| 12282 | nop | | | |
| 12283 | nop | | | |
| | Highlighted 1 row(s): | | | |

| | | | Hardware Event Count by Hardware | | | |
|---|---|---|---|---|---|---|
| Address | Line | Assembly | BR_INST_EXE... ANY by Package | BR_INST_EXE... COND by Package | CPU_CLK... THREAD by Package | BR_ |
| 0x55e46b | 12277 | nop | | | | |
| 0x55e46c | 12278 | nop | | | | |
| | | Block 6: | | | | |
| 0x55e46d | 12280 | jnz 0x55f46d <B | 200,000 | 200,000 | 24,000,000 | |
| | | Block 7: | | | | |
| 0x55e473 | 12281 | nop | | | | |
| 0x55e474 | 12282 | nop | | | | |
| | | Selected 1 row(s): | | | | |

**Figure 4.1: Example - Assembly Level Analysis**

## 4.2 Branch Prediction: Intel Nehalem Architecture

This section describes the implementation of different algorithms to unveil the underlying branch prediction structure of Intel Nehalem Micro-architecture. Although very little has been made public about the actual branch prediction structure used in the Nehalem architecture, it has been revealed from the background study that it implements a branch prediction unit with the following types of predictors in it:

- Branch Target Buffer (BTB) Predictor.
- Loop Branch Predictor.
- Indirect Branch Predictor.
- Hybrid Branch Predictor.

The white paper on Intel Microarchitecture (21) gives some insight on the possible types of branch predictors that are used in the Nehalem architecture. It publishes that a ‘New second-level BTB’ and a ‘New renamed return stack buffer’ has been implemented to reduce the number of branch mis-predictions in order to improve the performance of the processor and also the energy consumed.

In this thesis, algorithms have been developed and an automatic analyser developed to predict the structural organisation of two types of branch predictors that are used in the Intel Nehalem architecture:

- The Branch Target Buffer.
- The loop Predictor.

The following features about the above mentioned branch predictors are decoded:

- The Structure of the Predictor.
- The Access Mechanism.
- The Update Policy used.
- The Inter-dependencies.

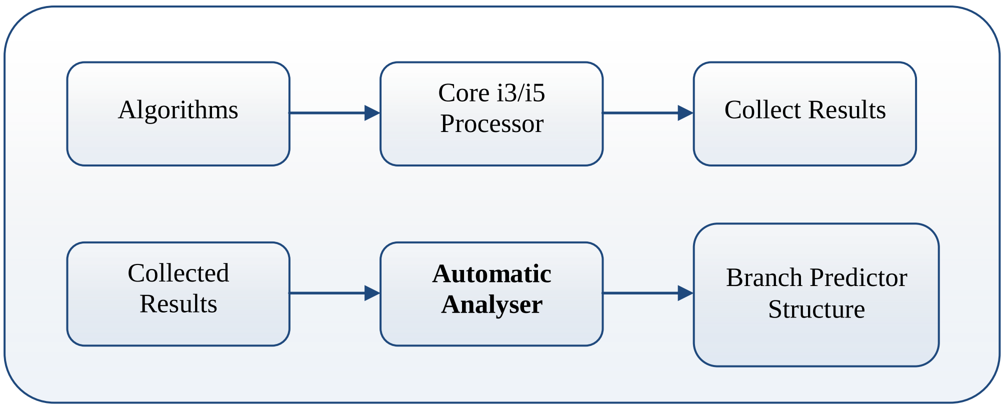


**Figure 4.2: Implementation Outline**

Figure 4.2 shows the abstract flow of the experiment undertaken. The Algorithm block indicates the algorithms (micro benchmarks) that are developed to analyse the structure of the branch predictor. The ‘Core i3’ block indicates that the algorithms were developed targeting the Intel Core i3 series of processors. The ‘Collect results’ block indicates the collection of branch related information such as the number of branches executed in the algorithm and the total number of branches mis-predicted during that run. This information is used to calculate the percentage mis-prediction of the branches.

The mis-prediction statistics from all the different algorithms are fed to the ‘automatic analyzer’. This block, developed as part of this project, carries out a series of actions to compute the structure of the branch predictor based on the received results as input. It should be noted that the automatic analyzer has been implemented targeting the core i3 processor. Thus it may fail to predict other types of branch structure when the algorithms are run on different processors but instead manual analysis of the results would need to be performed. Extension of the automatic analyser to a wider range of architectures could be performed in future work.

# Chapter 5.

# Branch Prediction structure of ARM

## 5.1 Introduction

This chapter describes the initial work done on developing the algorithms to analyze the branch prediction structure of ARM processors. To start with, the ARM architecture was chosen as it implements a relatively simple branch prediction structure and uses an easy to understand RISC instruction set. It also provides information about precise branch mis-predictions. It further provides a base to verify our algorithms and our analysis procedure. This section discusses the work done to decode the branch prediction structure of the ARM11 processor.

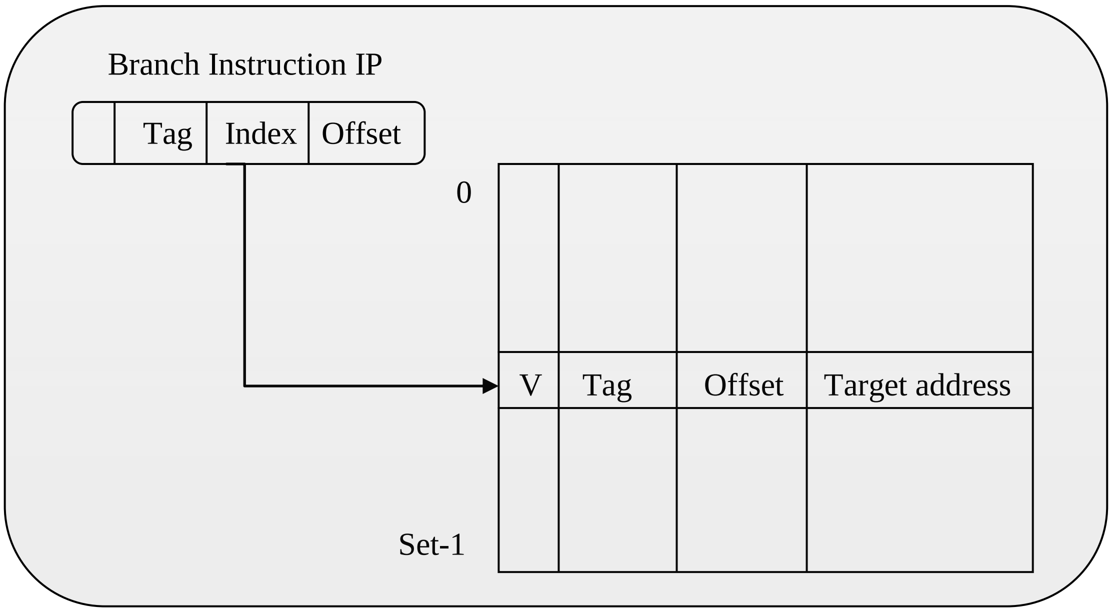
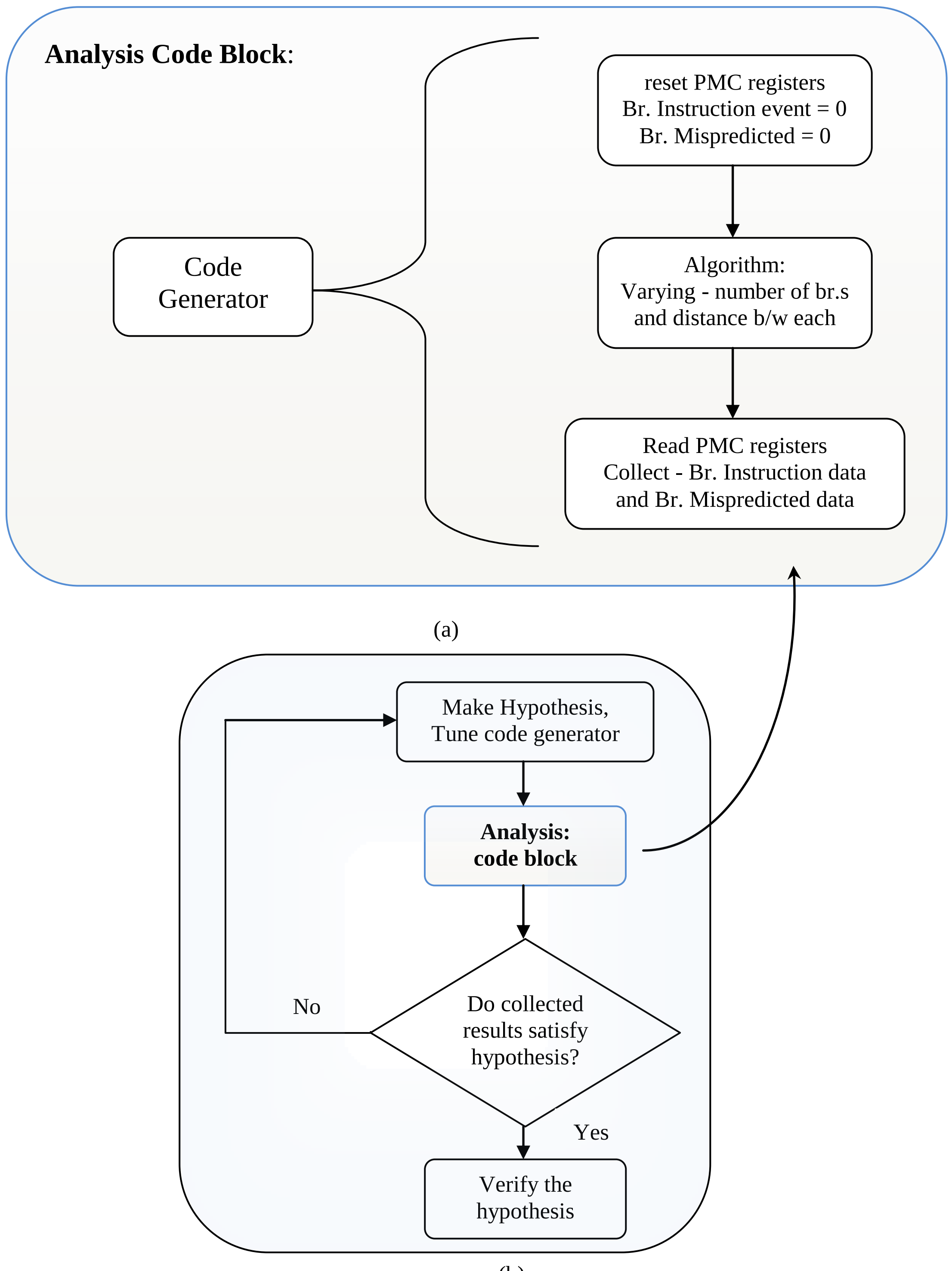


**Figure 5.1 BTB structure**

We assume the structure of the BTB as shown in Figure 5.1 BTB structure. The BTB cache consists of the tag field and the offset field. We aim to determine through a number of micro-benchmarks, the actual bits of the branch instruction address that are used to index this BTB. We also intend to develop algorithms to determine the MSB tag bit and the replacement policy used.

## 5.2 Algorithm flow developed

Figure 5.2 shows the general procedure/flow that is used to set up the environment to collect and analyse results.



(b)

**Figure 5.2 Flow Diagram: Branch Prediction Analysis - ARM**

Figure 5.2 (a) shows the code implementation details. The generator block is a piece of code that generates a code in assembly for the particular hypothesis. The code is written in assembly and it consists of broadly three parts. The first part sets the required bits in the configuration register to get access to the hardware events. The second block corresponds to the hypothesis code that usually consists of specific number of branches with a pre-determined distance between branches. The third block is a house-keeping block that collects the results, mostly hardware events such as branches executed, branches mispredicted etc, for further analysis.

Figure 5.2 (b) gives a broader view on how the analysis is performed on the ARM architecture. Initially a hypothesis is made so as to develop the algorithm to target a particular feature of the branch predictor. The code generator block generates the assembly level code targeting a particular hypothesis and the results are collected and analysed. It is checked whether the collected results match the pre-determined expectations. If the results are satisfactory then the verification of the hypothesis is carried out otherwise the hypothesis is suitably modified to reflect the changes in the results.

## 5.3 Branch Organisation Tests

A number of algorithms are developed to test branch prediction structure and behaviour. The branch mis-predicted percentage is determined for each test run and this information is used to determine the branch prediction unit structure. We specifically target the branch target buffer (BTB) organisation.

### 5.3.1 BTB capacity test

The capacity test is designed to determine the size of the BTB cache. The Figure 5.3 below shows the algorithm that is used to test the capacity of the BTB. A number of branches, varied in powers of 2, are placed at equidistant addresses in memory with distance D between them, which is also varied in powers of 2.

In this test, the distance D between branches and the number of branches are varied to trace the number of entries that the BTB can hold.

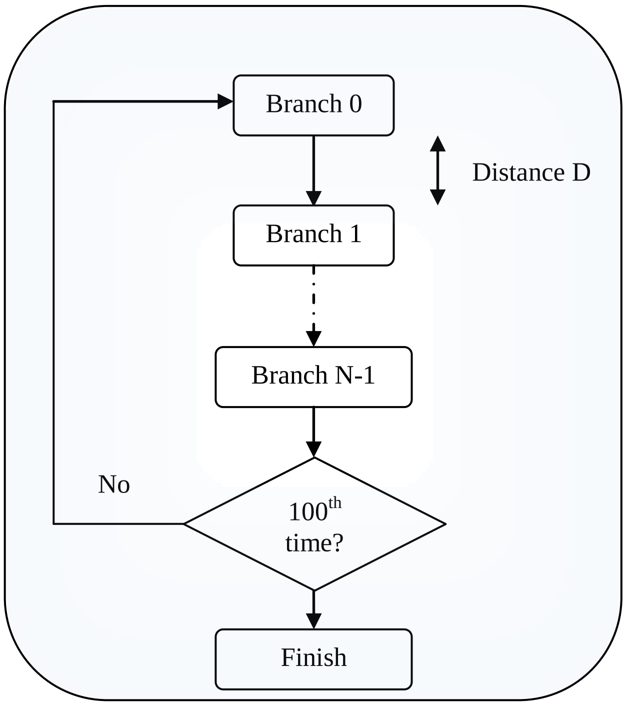


**Figure 5.3 BTB Capacity test**

Table 5-1 below shows the results taken by conducting different tests, by varying the number of branches and the distance between the branches.

| branches | distance | total_branches | total_mispredicted | mispredicted % |
|---|---|---|---|---|
| **64** | 4 | 6499 | 66 | 1 |
| **64** | 8 | 6499 | 262 | 1 |
| **64** | 16 | 6499 | 6499 | 100 |
| **64** | 32 | 6499 | 6499 | 100 |
| **64** | 64 | 6499 | 6499 | 100 |
| **64** | 128 | 6499 | 6499 | 100 |
| **64** | 256 | 6499 | 6499 | 100 |
| **64** | 512 | 6499 | 6499 | 100 |
| **64** | 1024 | 6499 | 6499 | 100 |

**Table 5-1 Results collected with Branches = 64**

It can be seen that the misprediction rate is close to 0% when the number of branches is 64 and the distance between the branches is 4 or 8. This indicates that the BTB can hold a minimum of 64 entries.

| Branches | Distance | Total branches | Total mispredicted | Mispredicted % |
|---|---|---|---|---|
| **128** | 4 | 12899 | 326 | 1 |
| **128** | 8 | 12899 | 12899 | 100 |
| **128** | 16 | 12899 | 12899 | 100 |
| **128** | 32 | 12899 | 12899 | 100 |
| **128** | 64 | 12899 | 12899 | 100 |
| **128** | 128 | 12899 | 12899 | 100 |
| **128** | 256 | 12899 | 12899 | 100 |
| **128** | 512 | 12899 | 12899 | 100 |
| **128** | 1024 | 12899 | 12899 | 100 |

**Table 5-2 Results collected with Branches = 128**

Table 5-2 shows the results with 128 branches and the distance between the branches is varied from 4 to 1024. We start from distance 4 as each instruction in ARM is stored as a 32 bit word. We observe from the above table that when the distance between branches is 4 the BTB can hold 128 branches and predict the outcome correctly. For all other distances it results in 100% misprediction.

Table 5-3 shows the same analysis but with 256 branches. Observe here that it results in 100% misprediction for all different distance cases. This suggests that the BTB can hold only 128 entries. Next we focus on decoding which particular branch address bits are used in indexing the BTB cache.

| Branches | Distance | Total branches | Total mispredicted | Mispredicted % |
|---|---|---|---|---|
| **256** | 4 | 25699 | 25699 | 100 |
| **256** | 8 | 25699 | 25699 | 100 |
| **256** | 16 | 25699 | 25699 | 100 |
| **256** | 32 | 25699 | 25699 | 100 |
| **256** | 64 | 25699 | 25699 | 100 |
| **256** | 128 | 25699 | 25699 | 100 |
| **256** | 256 | 25699 | 25699 | 100 |
| **256** | 512 | 25699 | 25699 | 100 |
| **256** | 1024 | 25699 | 25699 | 100 |

**Table 5-3 Results collected with Branches = 256**

**Conclusion**: we have revealed that the size of the BTB is 128.

### 5.3.2 BTB indexing tests

In this test we try to determine the access mechanism. From Figure 5.4 we observe that there are two fitting distances with no mis-prediction for branches = 64 and one fitting distance when branches = 128. As there is only one fitting distance when branches = 128, it gives a clue that the BTB is organised as a direct access cache. We predict that it is a 128 bit directly mapped structure.

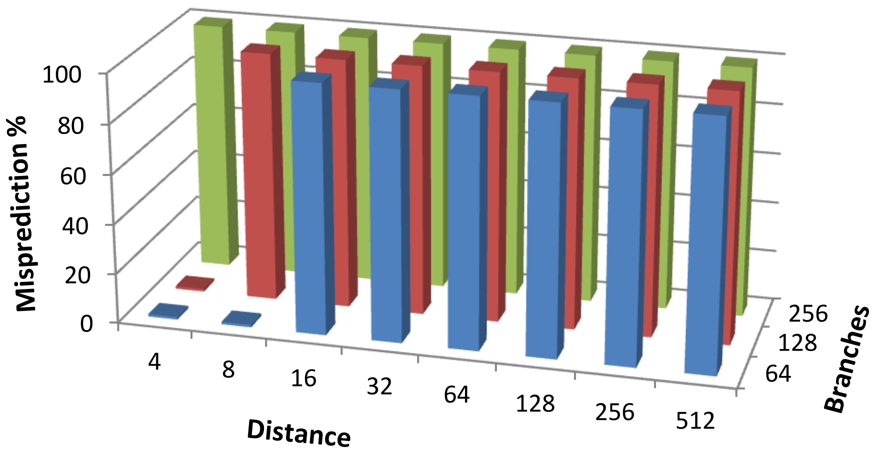


**Figure 5.4 BTB tests**

Instruction level analysis was performed to confirm these results. We found that the address bits [8:2] of the 129th branch were exactly the same as the corresponding bits of the first branch. After instruction level analysis we found that the 129th branch was evicting the first branch from the BTB. Similar behaviour was also found with the 130th and second branch pair and so on. Also note that the lower two bits [1:0] cannot be used to address as instructions are on 32 bit boundaries. Also, clearly from the graph we have only one fitting distance for 128 branches. Both these statements confirm that the BTB is a 128 entry directly mapped structure.

Test summary: 1. Concludes that address bits [8:2] are used to index the BTB.

2. Also concludes that the BTB is a 128 entry direct mapped structure.

### 5.3.3 BTB tag bits test

A tag bit test is used to reveal the number of branch instruction bits, after the MSB index bit that is stored in the BTB entry. The algorithm used to perform this test is shown in Figure 5.5. In this test, we use only two branch instructions but at a greater distance between them. The distance between them is increased, starting from $2^{(\text{msb index bit}+1)}$, until the misprediction is encountered.

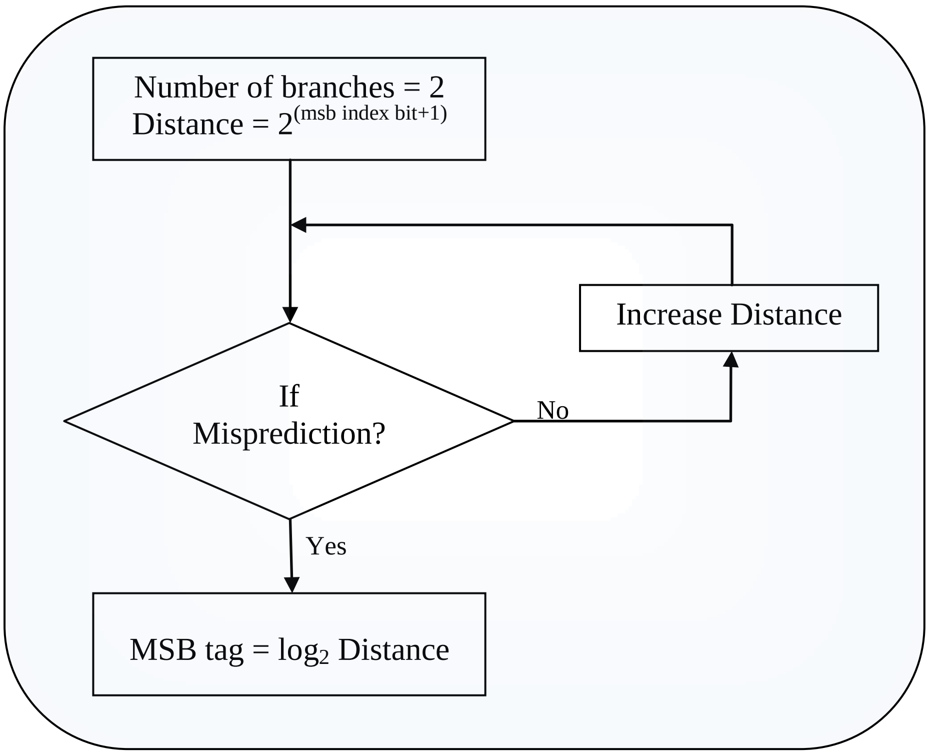

Number of branches = 2
Distance = $2^{(\text{msb index bit}+1)}$
Increase Distance
If Misprediction?
No
Yes
MSB tag = $\log_2$ Distance

**Figure 5.5 MSB tag analysis flow**

Table 5-4 shows the results of MSB tag analysis. It does not result in any mispredictions when the distance between the two branches is varied from 512 to $2^{22}$. This suggests that the address bits from 9 to 21 are stored in the BTB for each branch entry.

Analysis for distances greater than $2^{22}$ could not be performed due to restrictions in the ARM compiler and cycle accurate simulation software.

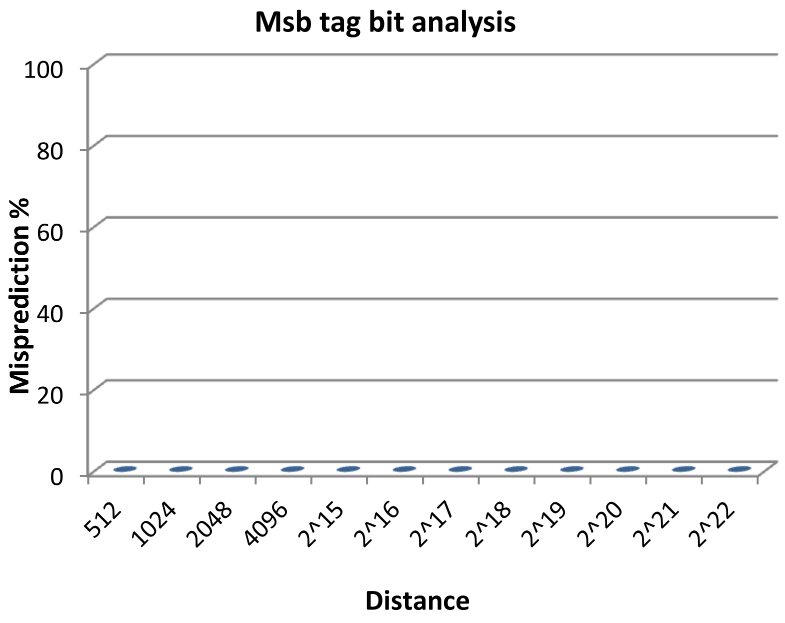


**Table 5-4 Tag MSB bit analysis**

**Test summary**: BTB stores branch address bits [22+:9] in the tag field.

### 5.3.4 2-bit saturation counters implementation

We assume that each entry in the BTB is organised as a 2-bit saturating counter. 2-bits are used to predict the direction of the branch. In order to study the behaviour we designed an algorithm targeting a single branch, by varying the direction (taken/not taken) of the branch during each iteration.

| Branch order | Mis-predict |
|---|---|
| T | 1 |
| T | 0 |
| T | 0 |
| NT | 1 |
| T | 0 |
| T | 0 |
| T | 0 |
| NT | 1 |
| T | 0 |
| T | 0 |
| T | 0 |
| NT | 1 |

(a)

| Branch order | Mis-predict |
|---|---|
| T | 1 |
| T | 0 |
| NT | 1 |
| NT | 1 |
| T | 0 |
| T | 0 |
| NT | 1 |
| NT | 1 |
| T | 0 |
| T | 0 |
| NT | 1 |
| NT | 1 |

(b)

| Branch order | Mis-predict |
|---|---|
| T | 1 |
| NT | 1 |
| NT | 0 |
| NT | 0 |
| T | 0 |
| NT | 0 |
| NT | 0 |
| NT | 0 |
| T | 0 |
| NT | 0 |
| NT | 0 |
| NT | 0 |

(c)

**Table 5-5 - 2-bit saturation counter tests**

‘1’ – indicates misprediction

‘0’ – indicates no misprediction

In Table 5-5 (a), we make a branch to be ‘taken’ three times and ‘not taken’ one time continuously. We observe that the branch predictor always predicts the branch to be ‘taken’ and hence it results in a mis-prediction when the branch is actually ‘not taken’. This behaviour is expected as the branch is ‘taken’ three times in succession and thus it predicts the branch to be ‘taken’ the next time.

Since the pattern is repetitive and it always results in mis-prediction during the ‘not taken’ branch, we can conclude that the ARM11 does not implement any loop predictor.

In Table 5-5 (b), we make the branch to be ‘taken’ two times and ‘not taken’ two times. We observe that the branch is predicted as ‘taken’ two times continuously when it is actually ‘not taken’ and the branch is not predicted when it is actually ‘taken’ two times in succession.

In Table 5-5 (c), we make the branch ‘taken’ one time and ‘not taken’ three times. It is observed that the branch is mis-predicted for the first time and since it involves three ‘not taken’ branches it is not predicted at all.  Also note that the BTB does not predict ‘not taken’ branches. Only taken branches are predicted.

| Branch order | Mis-predict |
|---|---|
| T | 1 |
| NT | 1 |
| T | 0 |
| NT | 0 |
| T | 0 |
| NT | 0 |
| T | 0 |
| NT | 0 |
| T | 0 |
| NT | 0 |
| T | 0 |
| NT | 0 |

(a)

| Branch order | Mis-predict |
|---|---|
| T | 1 |
| NT | 1 |
| NT | 0 |
| NT | 0 |
| T | 0 |
| T | 0 |
| NT | 1 |
| NT | 1 |
| T | 0 |
| T | 0 |
| T | 0 |
| NT | 1 |

(b)

**Table 5-6, 2-bit saturation counter tests**

‘1’ – indicates misprediction

‘0’ – indicates no misprediction

In Table 5-6 (a) we make a particular branch to be alternatively ‘taken’ and ‘not taken’. We observe that except for the first time the branch is not predicted at all. This is because the behaviour is not defined and is alternating between ‘taken’ and ‘not taken’.

In Table 5-6 (b), we have made the branch to be ‘taken’ and ‘not taken’ in a random pattern. We note that no matter how many times the branch is ‘not taken’ continuously, the moment it encounters two ‘taken’ in succession it predicts the next two instances of the branch as ‘taken’. And also the moment it encounters the branch to be ‘not taken’ two times in succession, it does not predict the outcome of the branch until it encounters two adjacent ‘taken’ branches.

We can summarise the following features about the 2-bit counter used:

- It allocates the branch into the BTB only if it is a taken branch. When it encounters a taken branch for the first time it makes an entry into the BTB and the branch is predicted as taken during the next execution of the branch.

- If it encounters a branch taken two times then it predicts the branch to be taken during the next two executions.
- If it observes the branch to be 'not taken' two times then it does not predict that branch until it encounters two 'taken' states continuously.
- If a branch shows alternate behaviour of 'taken' and 'not taken' branches then that branch is not predicted.
- All the above mentioned points suggest that in addition to a 2 bit saturating counter implementation, there should be an additional bit called 'valid bit' that is used to validate the entry.

## 5.4 Summary of BTB structure

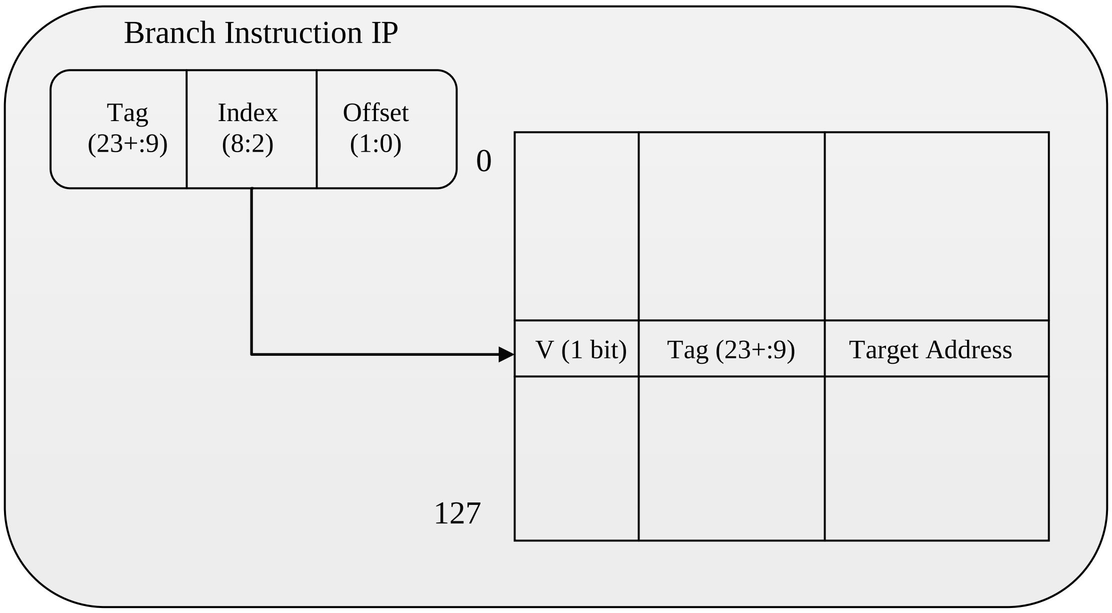


**Figure 5.6 Summary : BTB structure**

Figure 5.6 shows the summary of all the tests that were performed on ARM11. We observe the following features:

- The BTB is a 128 entry directly mapped structure.
- Each entry in the BTB implements a 2-bit saturating counter to store the state of the branch.
- The 'not taken' branches do not make an entry into the BTB.
- No replacement policy is implemented as the BTB cache is a direct mapped structure. A branch in the BTB is evicted during address conflicts with the new branch replacing the old branch entry in the BTB.

We tried to analyse the branch prediction mechanism that was used in ARM cortex processors, but unfortunately the debug tool (RVDS) is not a cycle level simulator and only models functional behaviour of the cortex family of processors. Thus we were unable to get hold of branch related event counts through the simulator.

An Amlogic board with a cortex A8 processor was ordered but the board was not available on time; as a result analysis of cortex processors was not performed.

# Chapter 6.

# Branch Target Buffer Predictor Analysis and Results

## 6.1 Introduction

This chapter describes the algorithms that were used to determine the branch target buffer structure of Intel's Nehalem architecture. The branch target buffer is similar to a cache structure that is used to hold branch related information such as the branch target address, some of the branch address bits, replacement bit etc.

Figure 6.1 shows the basic structure of the branch target buffer. In this section we focus on decoding the details of the BTB that is used in the Nehalem architecture. The following details about the BTB are can be determined from an analysis of the results obtained:

- The number of entries in the BTB.
- The tag bits used.
- The index bits used.
- The number of ways.

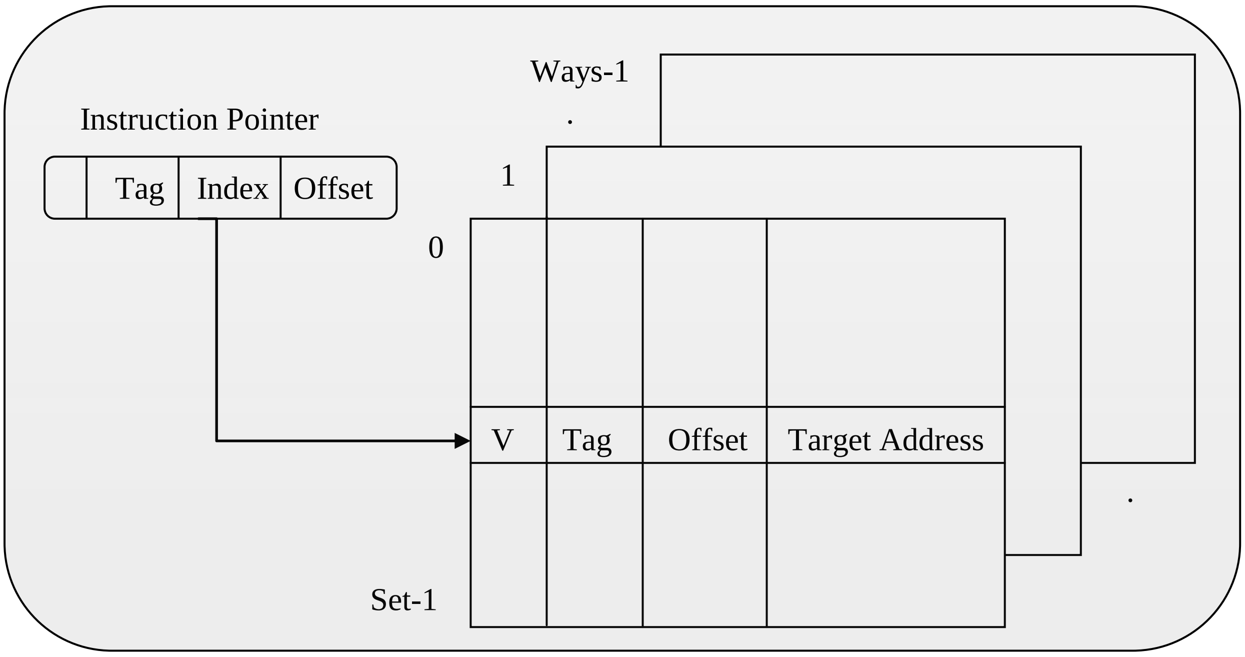


**Figure 6.1 General BTB Structure**

## 6.2 Contributions

The following results about the BTB structure are drawn as a result of various experiments carried out (explained in the next section).

- The BTB is organised as a cache structure capable of holding 2048 branch entries.
- 'Not Taken' branches do not make an entry into the BTB table.
- The BTB is organised as an 8-way structure with 256 entries in each set.
- Branch address bits [3:0] are not used in indexing the BTB. They are stored as an offset field in the BTB.
- Branch address bits [12:4] are used for indexing the BTB. Although these constitute to 9 bits they combine together (section 6.3.3.3, 8-bits) to address 256 entries in a set.
- Branch address bits [22+:13] are used as tag bits.

## 6.3 BTB Organisation tests

We intend to determine the size and organisation structure of the BTB. We extend the algorithm that was used by Uzelac and Milenkovic (3). A number of equidistant branches (in powers of 2) are placed in the test code. When the distance between the branches is varied in powers of two keeping the number of branches constant in the micro benchmark, we expect it to give a varying percentage of mis-predictions depending upon the structure of the BTB.

To illustrate the above statement let us consider an example where the BTB is organised as a 4-way set associative structure with 1024 entries. For this particular branch prediction architecture we perform the capacity test (refer back to section 6.3.1) by varying the number of branches from 128 to 4096 (128, 256, 512, 1024, 2048 and 4096). For a given test, the distance between the branches is varied from 2 to 1024 (2, 4, 8, 16, 32, 64, 128, 256, 512 and 1024).

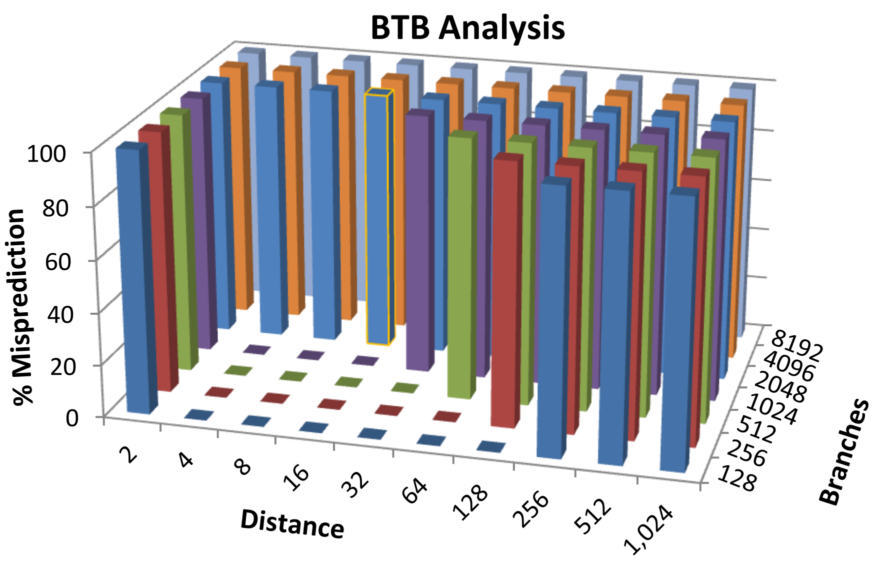


**Figure 6.2 Expected results for 4-Way BTB structure with 1024 entries**

Figure 6.2 shows the expected percentage of mis-predicted instructions for the various tests conducted. We observe that it results in almost 100% mis-prediction with 128 branches and when the distance between branch instructions is 256. This behaviour is due the organisation of the BTB. Observe that it results in no mis-prediction for three distances: D= 4, 8 and 16 when the number of branches = 1024. This suggests that the BTB is organised as a 4 way set associative structure with 256 entries in each set. Also note that it results in no misprediction for the B=1024 and D=16 pair. This indicates that the least significant four bits (since $2^4$=16) are not used for indexing the BTB. Hence we infer that the branch address bits [11:4] of the Instruction Pointer are used to index the BTB cache.

Having considered this hypothetical case, we can now perform similar tests on the branch prediction unit of the Nehalem architecture and understand its BTB organisation.

In our tests we make the following assumptions:

- The branches which are never taken do not make an entry in the BTB.
- Part of the branch instruction address bits are used as an index to the BTB cache.
- The BTB is organised as shown in Figure 6.1.

All the above mentioned assumptions are verified at the later stage when more details become available.

### 6.3.1 BTB capacity test and results

Figure 6.2 shows the algorithm that is used to determine the number of entries that the BTB can hold. We vary the number of branches and the distance between each branch and determine the misprediction rate for each test.

```
/*Address*/              /*Code*/
                         int main(){
                               _asm {
                            mov ebx, 1000000 // iterate for 10^6 times
                         loopup:
/* @x */                        jnz loop0      // always taken branch
                                     // dummy non branch instruction
                         loop0:
/* @x+D */                      jnz loop1      // always taken branch
                                     // dummy non branch instruction
                         loop1:
/* @x+2D */                     jnz loop2      // always taken branch
                                     // dummy non branch instruction
                         loop2:    …
                                   …
                         loop127:
                                sub ebx,0x1
                                cmp ebx,0x0
/* @x+127D */                   jnz loopup
                               }
                               int 0;
                         }
```

**Figure 6.3 Microbenchmark - BTB capacity test example with 128 branches**

Note that we run the branches $10^6$ times so that we can collect more accurate results. Each branch instruction is exactly placed at a distance 'D' (n = $\log_2$ D) from its previous branch so that the lower $2^n$ bits of all the branch instruction are the same for a particular test.

The following four hardware events are monitored using the Vtune performance analyser for every test conducted:

| Event Name | Event Description |
|---|---|
| BR_INST_EXEC.ANY | Branch Instructions Executed |
| BR_INST_EXEC.COND | Conditional Branch Instructions Executed |
| BR_MISP_EXEC.ANY | Mispredicted Branches Executed |
| BR_MISP_EXEC.COND | Mispredicted Conditional Branches Executed |

**Table 6-1 Branch events used in analysis**

The Table 6-2 shows the results collected for the BTB capacity test with branches = 128 and varying distance between the branches from 2 to 4096. It can be observed that it results in approximately 60% misprediction for distance = 2. This might be because some of the lower address bits of the branch instruction may not be used to index the BTB. This test also shows that mispredictions result for distances between branches greater than 512. This behaviour can be understood after carrying out some more tests.

| 128br | br_any | br_cond | mispred_any | mispred_cond | mispred % |
|---|---|---|---|---|---|
| **2** | 67,000,000 | 67,000,000 | 39,000,000 | 39,000,000 | 60 |
| **4** | 129,000,000 | 128,000,000 | 1,020,000 | 1,100,000 | 1 |
| **8** | 129,000,000 | 129,000,000 | 100,000 | 100,000 | 0 |
| **16** | 129,000,000 | 129,000,000 | 100,000 | 100,000 | 0 |
| **32** | 126,000,000 | 127,000,000 | 20,000 | 0 | 0 |
| **64** | 129,000,000 | 129,000,000 | 100,000 | 100,000 | 0 |
| **128** | 128,000,000 | 129,000,000 | 80,000 | 60,000 | 0 |
| **256** | 130,000,000 | 129,000,000 | 80,000 | 40,000 | 0 |
| **512** | 129,200,000 | 129,000,000 | 96,820,000 | 96,000,000 | 75 |
| **1024** | 129,000,000 | 129,000,000 | 121,000,000 | 121,000,000 | 94 |
| **2048** | 133,000,000 | 131,000,000 | 124,000,000 | 124,000,000 | 97 |
| **4096** | 133,000,000 | 131,800,000 | 125,360,000 | 125,280,000 | 97 |

**Table 6-2 BTB Capacity test with branches = 128**

We consider two branch events: 1. Total branches executed and 2. Total conditional branches executed. We also consider two mispredicted hardware events: 1. Total number of branch instructions mispredicted and 2. Total number of conditional branch instructions mispredicted.

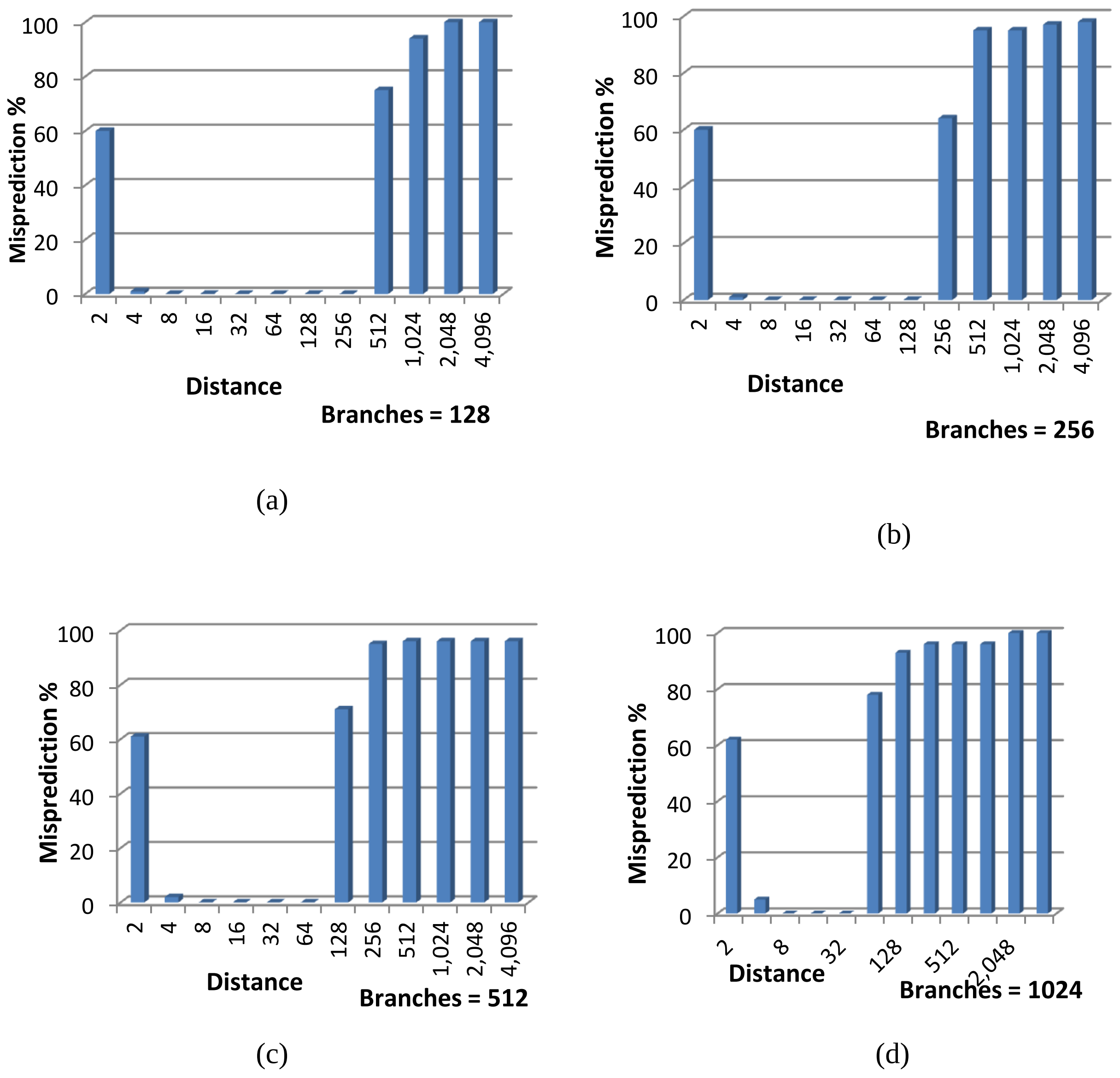
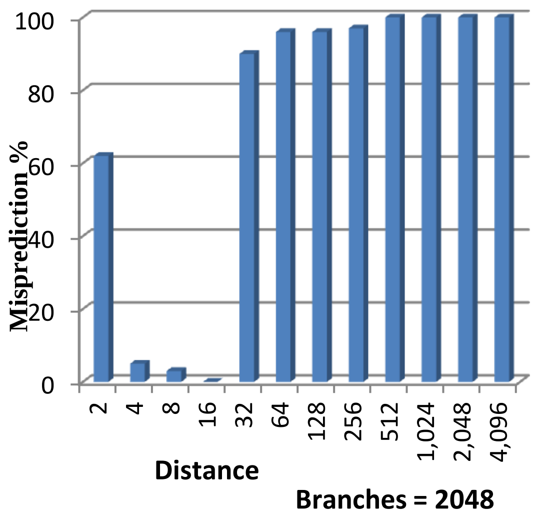


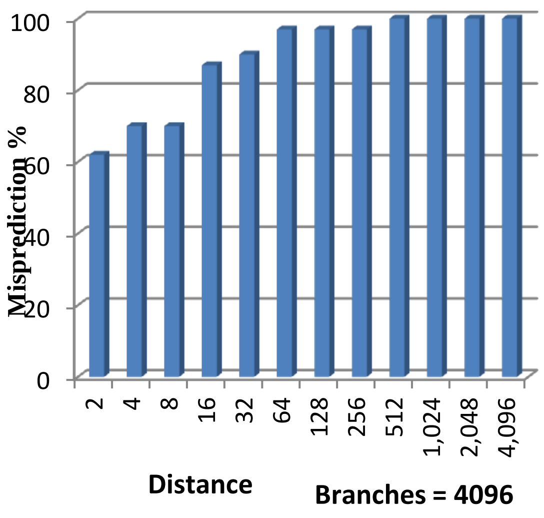
**Figure 6.4 BTB capacity test: branches 128 to 1024.**

Figure 6.4 (a) shows the misprediction rate over a range of distances with 128 branches. Figure 6.4 (b) shows the same but with 256 branches. We observe in analysis (b) the branch misprediction has resulted with lesser distance between branches compared to that with 128 branches. We also notice the same behaviour in Figure 6.4 (c), where the test is carried out with 512 branches. This gives a clue that the BTB is following a certain pattern. This is also verified from the Figure 6.4 (d) where analysis is performed with 1024 branches. In these four different types of analysis we observe that the misprediction rate is steadily increasing for lesser distance between branches as the number of branches in the test increases.

Figure 6.5 (a) shows the analysis with 2048 branches. Now there are only three distances for which the number of mispredictions is less than 5%. Observe from Figure 6.5 (b) that it results in misprediction for all distances when the branch count equals 4096.



(a)



(b)

**Figure 6.5 BTB capacity test: branches = 2048 and 4096**

These two analyses confirm that the BTB can hold 2048 entries in total. From (a), it is noted that there exists three fitting distances which result in no mispredictions when the branch count is 2048. Also at the same time there is no fitting distance which results in no mispredictions when the same analysis is performed on 4096 branches. This suggests that the BTB is organised as a 2-, 4- or 8-way set associative structure.

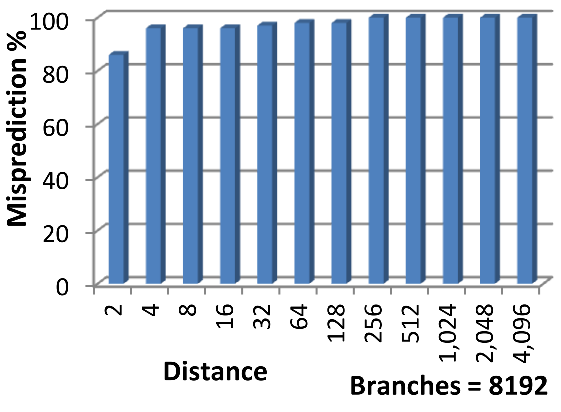


**Figure 6.6 BTB capacity test with 8192 branches**

Observe that, from Figure 6.5, the misprediction rate is not 100% for analysis with 4096 branches with 4 and 8 distances. This may be due to the replacement policy governing the BTB structure. But from Figure 6.6, where the same analysis is performed with 8192 branches, it is apparent that it results in 100% misprediction for all distances.

The Figure 6.7 shows the overall capacity test results with all the above discussed analysis put together.

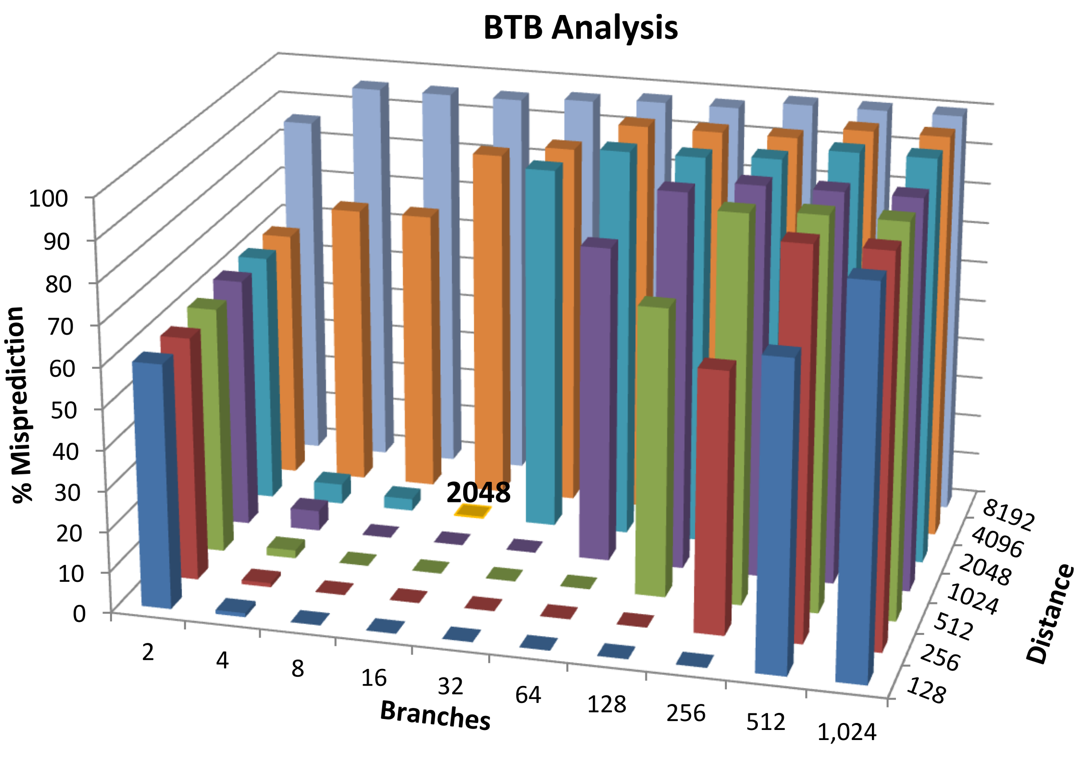


**Figure 6.7: Overall BTB Capacity Analysis**

From Figure 6.7 we can conclude that the BTB can hold 2048 entries. Next, we try to decode the set associativity of the BTB structure. It can be organised as a 2 way set associative structure with 1024 entries in each set or as a 4-way set associative structure with 512 entries in each set or perhaps even as an 8-way set associative structure with 256 entries in each set.

Consider the case where the BTB was organised as a 2-way set associative structure. If it was a 2-way structure with 1024 entries then it should have resulted in high rate of mis-predictions with 1024 branches and 4 distance (this would result in four branches having the same tag and this should result in a miss in 2-way structure). Thus it confirms that the BTB is not a 2-way structure with 1024 entries.

**'Not taken' branch analysis**

In the analysis described above, conditional 'taken' branches were used. In this analysis we intend to include branches which are not taken all time. Table 6-3 shows the results collected after performing the analysis with 2048 and 4096 'not taken' branches in the test (distance = 128).

| Branches | Misprediction |
|---|---|
| 2048 | 0 |
| 4096 | 0 |

**Table 6-3: Results: 'not taken' branch analysis**

The above results are a sub-set of the analysis that was performed. More low level analyses are performed to confirm the behaviour of the BTB.

**Test Summary:**

- The size of the BTB is 2048, meaning the BTB cache structure can hold 2048 branch entries in total.
- 'Not taken' branches do not make an entry into the BTB.

### 6.3.2 BTB set associative analysis and results

In this analysis we try to determine the set associative nature of the BTB. We try to decode whether the BTB is a 4-way or an 8-way structure. To perform this analysis we developed an algorithm that is as shown in Figure 6.8. We start with two branches in the test code and we choose the distance between branches sufficiently high (higher than the msb index bit [section 6.3.3.1]).

The micro benchmark shown in Figure 6.3 was modified to meet the requirements set in Figure 6.8. By this method, it is ensured in this analysis that all the branches have the same BTB index bits. Only the tag bits may vary. Thus by construction the number of branches that are correctly predicted in this analysis corresponds to the number of ways of the BTB structure.

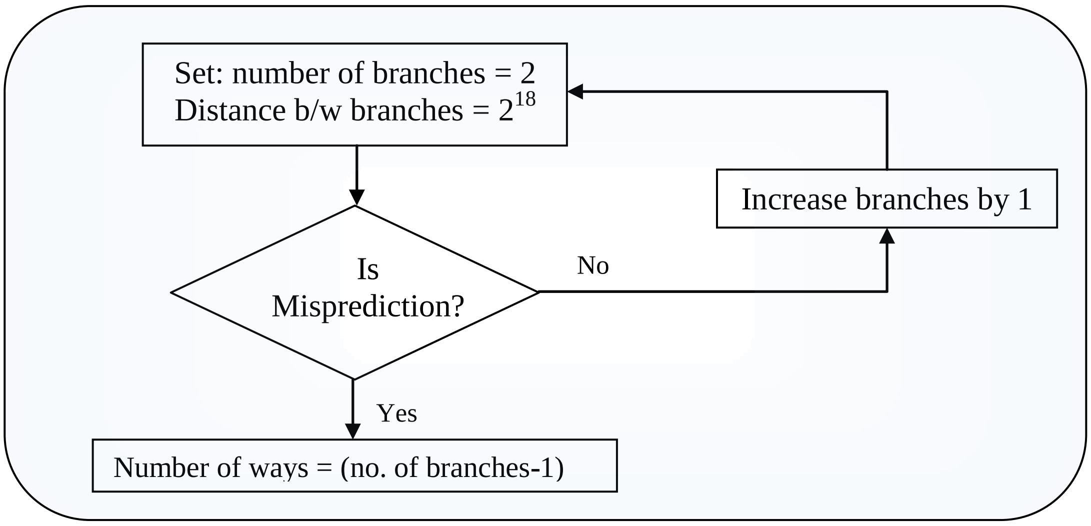

Set: number of branches = 2
Distance b/w branches = $2^{18}$
Increase branches by 1
Is Misprediction?
No
Yes
Number of ways = (no. of branches-1)

**Figure 6.8: set associative structure analysis**

The Figure 6.9 shows the results of the above analysis. It can be observed that for up to 8 branches no misprediction occurs. The misprediction starts from the 9th branch onwards. This concludes that the BTB structure is 8-way set associative with each set capable of holding 256 entries.

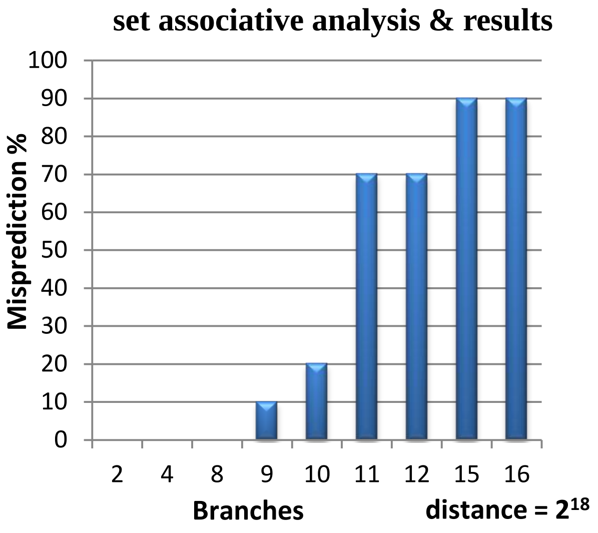


**Figure 6.9: Results: set associative analysis**

Test Summary: This analysis confirms that the BTB is 8-way set associative with 256 entries.

### 6.3.3 BTB indexing Analysis

From the previous section it is clear that the BTB is organised as an 8-way set associative structure. Now we intend to develop a set of tests which determine which bits of the branch address that are used to index the BTB.

#### 6.3.3.1 BTB MSB index bit analysis and results

This test is targeted to determine the msb that is used to access the Branch Target Buffer. The developed algorithm to determine the msb indexing BTB is as shown in Figure 6.10. In this analysis we keep the number of branches greater than the set associativity of the BTB (branches = 12). The distance between the branches is varied from 1024 until we get a misprediction. The micro benchmark shown in Figure 6.3 is modified to meet the requirements as described in this algorithm.

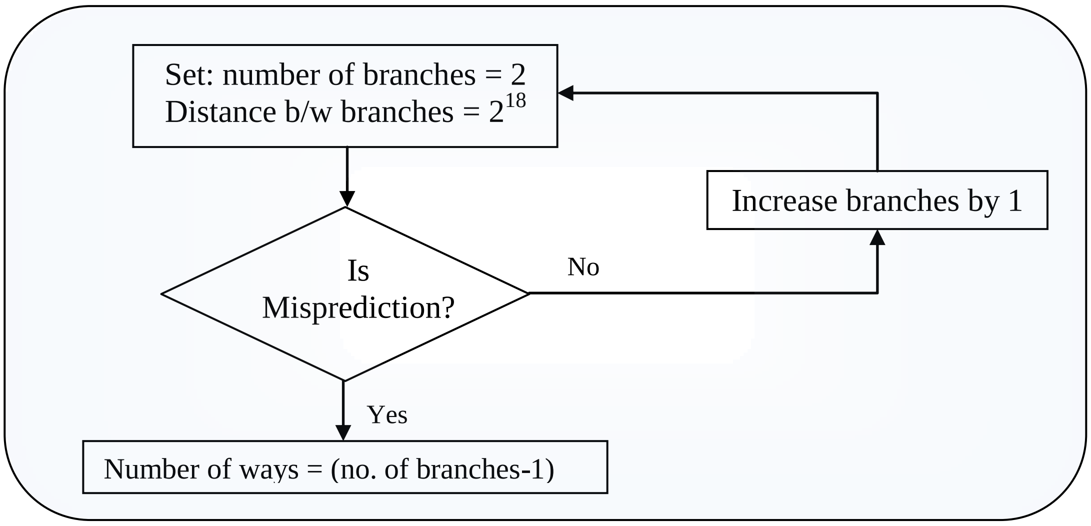

Set: number of branches = 2
Distance b/w branches = 2^18
Increase branches by 1
Is Misprediction?
No
Yes
Number of ways = (no. of branches-1)

**Figure 6.10: Algorithm to determine the msb index bit**

Figure 6.11 shows the results from the algorithm described above. Observe that it results in misprediction when the distance between branches is $2^{13}$. This suggests that the msb that is used to access the BTB is the 12th bit ($2^{12} = 4096$).

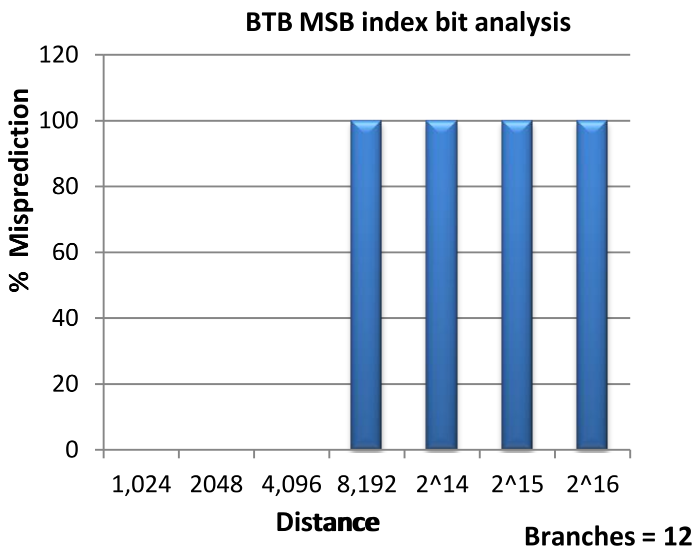


**Figure 6.11: Results: BTB msb index bit analysis**

Test Summary: This test confirms that addr[11] (12th bit) is used as an msb index bit to BTB.

### 6.3.3.2 BTB LSB index bit analysis and results

This test is targeted to determine the least significant bit (lsb) that is used to access the Branch Target Buffer. In this benchmark we include branches greater than the number of ways of the BTB structure (Branches = 9). Contrarily to the msb analysis, here we target the least significant bits. Thus our aim is to include branches with the distance of 1 to 32 between them. But we want to include all the nine branches such that they target a single set within the BTB.

This is achieved by placing the branches as explained in the micro benchmark, Figure 6.12. Note that the distance between the first eight branches is kept greater than 4096 (> msb index bit, $2^{12}$). Thus all the first eight branches target the same set in the BTB. The distance between the 8th and 9th branch is set to (8192+x). ‘x’ is varied from 1 to 32.

```
/*Address*/                 /*Code*/
                            int main()    {
                                   unsigned int long i;
                            for(i=0;i<1000000;i++) { //iterate 10^6 times
                                   _asm {
                            loopup:
/*@x*/                               jnz loop0  // always taken 1st branch
                                   //  dummy  non-branch  instruction
                    loop0:
/*@x+8192*/                          jnz loop1 // always taken 2nd  branch
                                   // dummy non-branch instruction
                            loop1:    …
                                      …

/*@x+(8192*7)*/                     jnz loop7 // always taken 8th branch
                                   // dummy non-branch instruction
                            loop7:
/*@x+((8192*7)+y)*/                jnz loopup  // always taken 9th branch
                                   }
                                   return 0;
                            }
* y is varied from 1 to 32
```

**Figure 6.12: micro benchmark to analyse lsb index bit**

Figure 6.13 shows the results obtained for the micro benchmark explained in Figure 6.12. Observe from the graph that no mispredictions occur for distance 16. This indicates that the 9th branch has moved to another set. This concludes that the lsb that is used to index the BTB is addr[4], or bit 5.

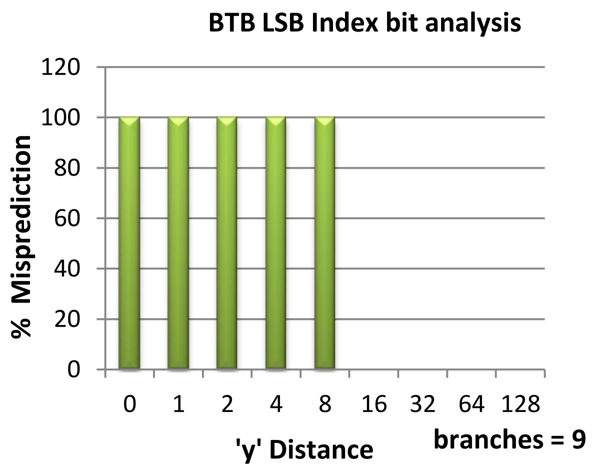


**Figure 6.13: Result: lsb index bit analysis**

In the BTB capacity test it was observed that for test (2048 branches, 16 distance) there were no mispredictions. This also suggested that the possible lsb index bit was addr[4]. Thus this test confirms that the lsb index bit used is addr[4].

Test Summary: addr[4] is the branch address lsb bit that is used to index the BTB.

#### 6.3.3.3 BTB index bit conclusion

We have obtained slightly conflicting results in section 6.3.3.1 and 6.3.3.2 for the msb and lsb analysis. The BTB is organised as an 8-way structure with 256 entries in each set. This indicates that 8 bits are used to index the BTB. But the lsb and msb analysis indicates that address bits [12:4] are the index bits, which actually indexes 512 entries.

From the above results we can infer that two address bits in [12:4] may be used to index a single bit of the BTB or a function of [12:4] bits are used to index effectively 256 ($2^8$) entries in the BTB.

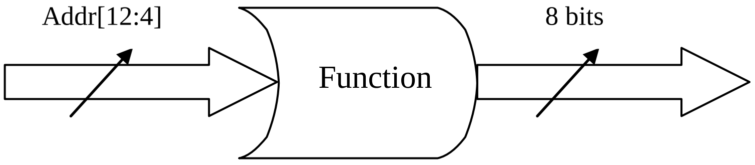


**Figure 6.14: indexing bits conversion function**

### 6.3.4 BTB MSB tag bit analysis and results

In this sub section, we try to find the number of tag bits that are stored in the BTB. In this test again we include two branches separated by a huge distance. We modify the BTB capacity micro benchmark to include two branches and the distance is varied from $2^{\text{msb index}}$ to until we get mispredictions.

Figure 6.15 shows the results collected after tag msb analysis. As we observe from the figure the experiment was carried out up to $2^{23}$ distance between the two branches. The test did not result in any misprediction. This suggests that the msb tag bit is higher than the 22nd branch address bit.

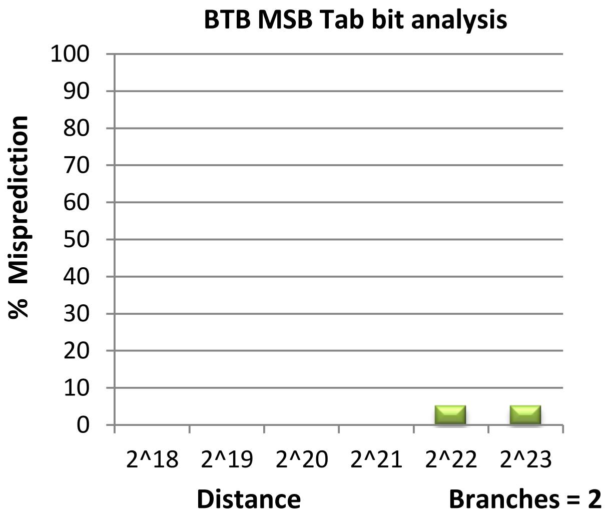


**Figure 6.15: Results: Tag MSB bit analysis**

Test Summary: address bits [22+:13] are stored in the tag field of BTB.

# Chapter 7.

# Loop Predictor Structure Analysis and Results

## 7.1 Introduction

In this chapter we intend to study the structure of the loop predictor that is used in the Nehalem architecture. We develop a set of micro benchmarks that are used to decode the structure, cache access mechanism and the update mechanism of the loop predictor.

Loop predictors use a cache structure called the loop prediction buffer. We intend to find the following about the loop predictor buffer:

- The size of the loop predictor buffer.
- The size of the counters.
- The number of ways in which the loop predictor buffer is organised.
- Which address bits of the branch instruction are used to index the loop predictor buffer.
- The tag bits that is stored in the loop predictor for a particular branch.
- The allocation policy used.
- The relationship between the loop predictor buffer and the BTB.

### 7.1.1 Background

Loop branches are characterised by branching in one direction for a certain number of times interspersed with one outcome in the other direction. For example, a branch is said to be a loop branch if it is 'taken' 'n-1' times and 'not taken' one time or if the branch is 'not taken' 'n-1' times and 'taken' one time.

The loop predictor's task is to predict the outcome of the loop branch when the outcome pattern length exceeds the size of the branch history register of the global branch predictor. The loop predictor implements a set of counters to detect the loop behaviour of the branch. We assume that the loop predictor is a cache structure similar to the BTB. But in addition to storing tag bits and the target address, it has to include a mechanism to detect the loop behaviour and predict the outcome of the branch correctly when it is taken in the other direction for one time.

The loop prediction buffer implements two counters to keep track of the loop behaviour of the branch. The 'maximum counter' field is trained during the first run of the loop branch to indicate the opposite outcome of the branch. In the subsequent encounter of the same branch, the 'iteration counter' is incremented until it reaches the 'maximum counter' value. Once it reaches this value the 'iteration counter' field is reset and the branch is predicted in the other direction. In this way, each loop branch outcome is predicted correctly.

Ideally a loop predictor buffer must also store the branch target address. But, the branch is also allocated in the BTB on its first encounter. Since the BTB stores the branch target address, it may happen that the loop predictor, in order to save space, retrieves the branch target address from the corresponding BTB entry of the loop branch.

The Figure 7.1 shows the general structure of the loop predictor. It consists of a loop predictor buffer that stores tag bits, the maximum value of the counter and the current counter value.

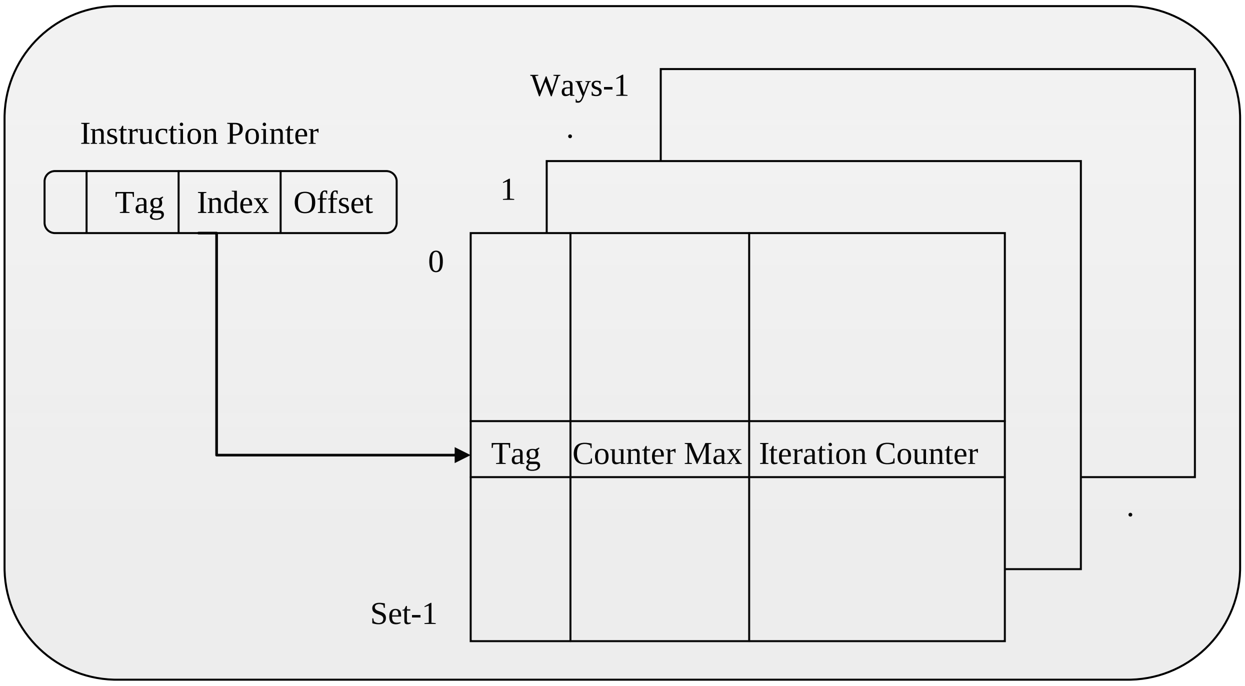


**Figure 7.1: Loop predictor general structure**

## 7.2 Contributions

We are able to reveal the following parameters about the loop predictor that is used in the Nehalem architecture based on the results collected by running the micro benchmarks developed (discussed in the next section).

- The loop predictor buffer can hold 32 entries.
- The loop predictor buffer is organised as a 2-way structure with 16 entries in each.
- The loop predictor buffer set is indexed by using the branch address bits [7:4].
- The branch address bits [12:8] are stored as the tag bits in the loop predictor buffer.
- A least recently used replacement policy is used to replace entries in the buffer.
- The loop branch prediction buffers implement two 6 bit counters. One counter is used to store the maximum loop count. The other is used to count the current loop iteration. When both counters match the predictor generates the opposite outcome.
- The loop predictor hit is conditionally dependent on the BTB hit.

## 7.3 Loop Predictor organisation analysis

### 7.3.1 Maximum Counter Length analysis and results

This micro-bench is designed to target the count value of the loop branch. In this micro benchmark we include one loop branch in the test and vary the iteration count from 16 to 256 (16, 32, 64, 128 and 256).

Figure 7.2 shows the general test procedure. We start the test with count = 16 and increase it until there is an increase in the percentage of the misprediction.

Figure 7.3 shows the micro-benchmark used to perform this test. The example code shown here uses a single loop branch with count = 32.

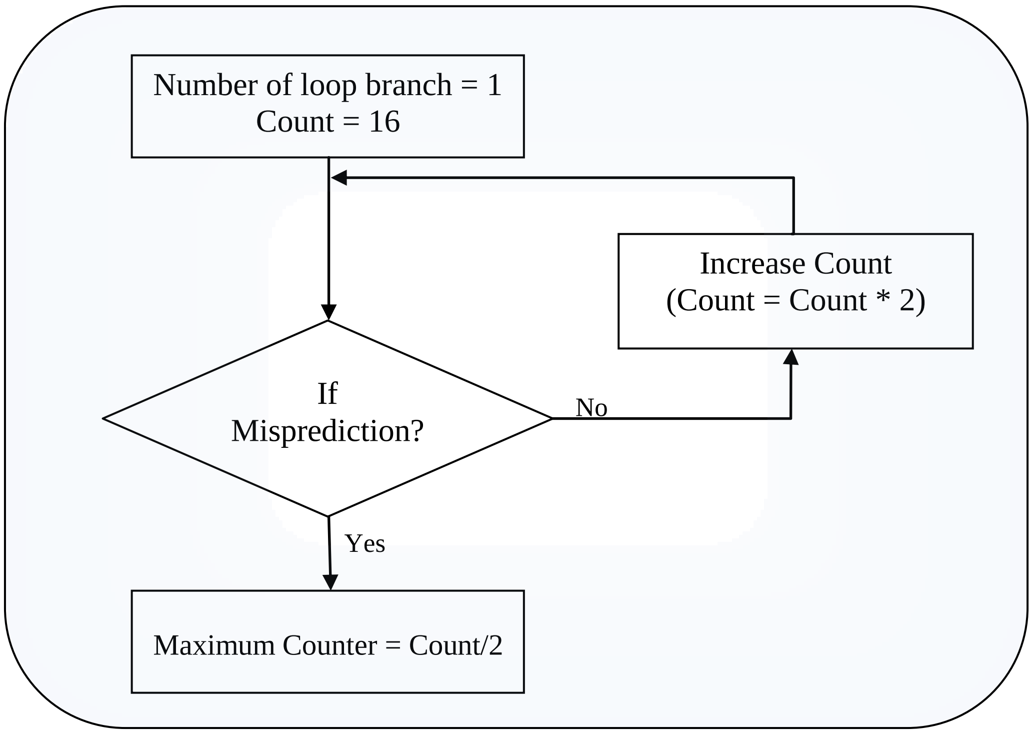


**Figure 7.2: Counter Length test flow**

```
int main(){
        long unsigned int i;
        for(i=0;i<32000000;i++){
                _asm   {
                        mov ecx, 32
loop_up:
                        sub ecx,1
                        cmp ecx,0
                        jnz loop_up
                }
        }
        return 0;
}
```

**Figure 7.3: micro benchmark - loop counter analysis.**

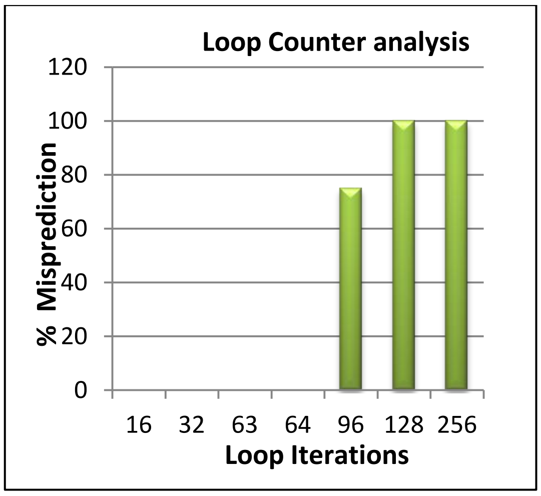


**Figure 7.4: Results: Loop counter analysis**

Figure 7.4 shows the results obtained for different iteration counts. The percentage misprediction is normalised by the loop count. It is clear from this analysis that the loop predictor is capable of predicting loops with maximum 64 counts correctly. It sets aside 6 bits to store the maximum counter value and 6 bits to store the iteration counter value.

**Test Summary:** Indicates loop counters are 6-bit long.

### 7.3.2 Loop capacity analysis and results

In this test we develop tests to determine the maximum number of entries that the loop predictor buffer can hold. We assume that the branch target address is retrieved from the BTB. We also assume that the loop prediction buffer is organised in a similar way as the BTB is organised. All these assumptions can be verified through a number of micro-benchmarks when more data is available. The loop capacity test is similar to the BTB capacity test as described in section 5.3.1.

Figure 7.5 shows the general structure of the loop capacity test. The loop capacity consists of a number of loop branches that are placed at equidistant locations in memory with distance D between them.

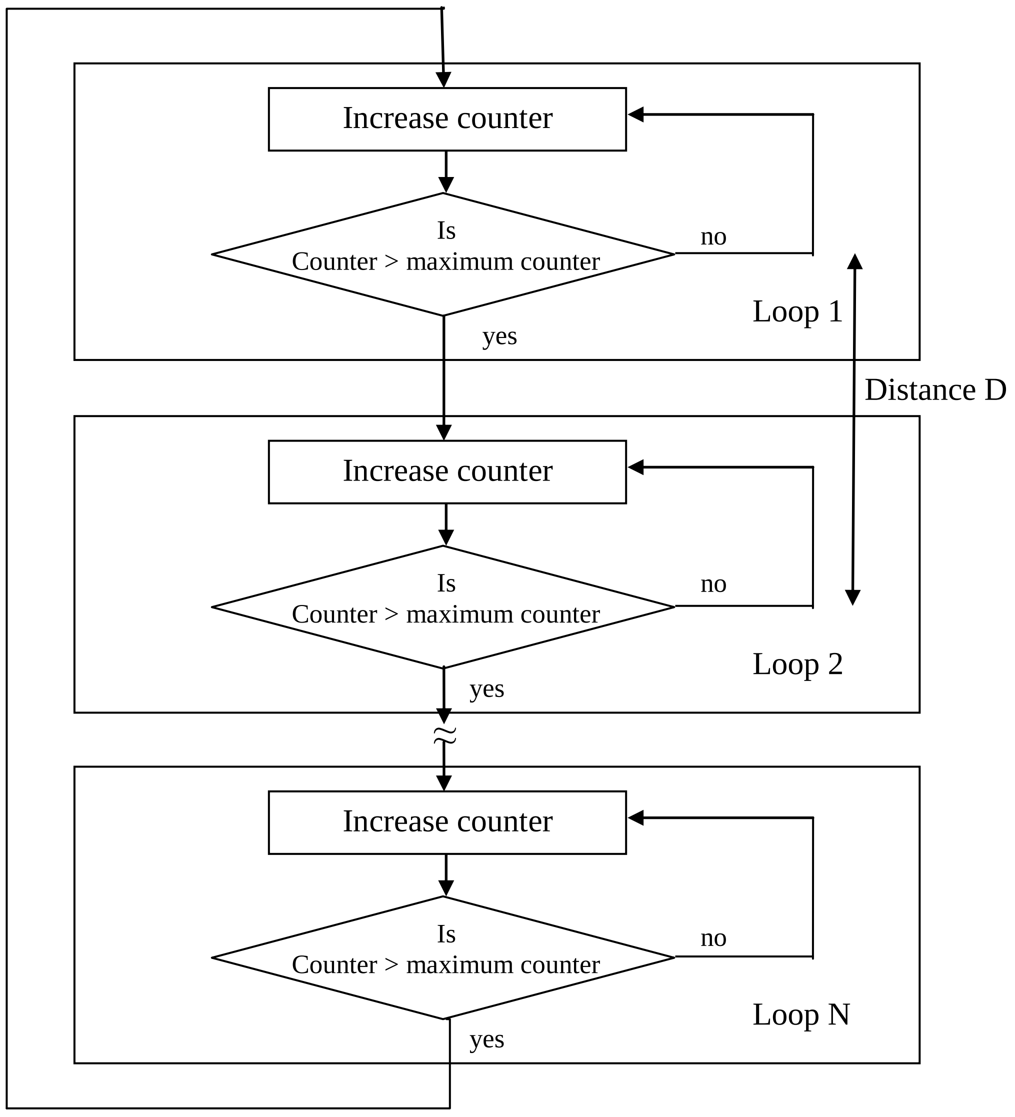


**Figure 7.5: structure of loop capacity test**

Figure 7.6 shows the micro benchmark that is used to perform the loop capacity analysis. Note that each branch is a loop branch that is ‘taken’ count number of times and is ‘not taken’ one time. The loop counter is set to 32 as it is sufficiently greater than the pattern history register used by the global branch predictor. This ensures that these branches are predicted by the loop predictor rather than the global predictor. The loop branches are again kept at an equidistant location in memory (in powers of 2) with distance ‘D’.

```
/*Address*/             /*code*/
                        int main(){
                        long unsigned int i;
                        for(i=0;i<1000000;i++){
                              _asm {
                                  mov ecx,32
                        loop0:
                                  sub ecx,1
                                  cmp ecx,0
/* @x */                          jnz loop0  // 1st loop
                        // dummy non branch instructions
                                  mov ecx,32
                        loop1:
                                  sub ecx,1
                                  cmp ecx,0
/* @x + D */                      jnz loop1  // 2nd loop
                        // dummy non branch instructions
                                  mov ecx,32
                        loop2:
                                    ...
                        loop15:
                                  sub ecx,1
                                  cmp ecx,0
/* @x + N*D */                    jnz loop15  // 16th loop
                                    }
                              }
                        return 0;
                        }
```

**Figure 7.6: Micro benchmark: loop capacity test**

The loop capacity test is carried out with branch iterations varying from 8 to 128 and distance between the branches is varied from 8 to 128. As described in section 6.3.1, in this test the four branch-related hardware events are monitored. Table 7-1 shows the results collected with 32 iterations and 16 distance between loops. The misprediction percentage is indicated by normalising the number of branch mis-predictions by the loop counter.

Figure 7.7 shows the results collected after performing the loop capacity analysis on all sets of tests with varying iteration counts and distances. It is observed that the distance between branches is started from a minimum of 8 in the analysis. This is because unlike the BTB capacity test, in the loop capacity test the branches cannot be placed with distance of 4 and 2 between them.

| Branches (dist = 16) | Branches any | Branches conditional | Mispredicted branches any | Mispredicted conditional | Misprediction % |
|---|---|---|---|---|---|
| **8** | 260,000,000 | 259,000,000 | 120,000 | 180,000 | 0 |
| **16** | 514,000,000 | 516,000,000 | 260,000 | 160,000 | 0 |
| **32** | 1,027,000,000 | 1,024,000,000 | 600,000 | 580,000 | 0 |
| **64** | 2,054,000,000 | 2,051,000,000 | 64,520,000 | 63,740,000 | 100 |
| **128** | 4,099,000,000 | 4,102,000,000 | 128,000,000 | 128,000,000 | 100 |
| **192** | 6,159,000,000 | 6,154,000,000 | 194,000,000 | 192,000,000 | 100 |
| **256** | 8,207,000,000 | 8,195,000,000 | 259,460,000 | 258,820,000 | 100 |

**Table 7-1: loop capacity results with distance = 16**

Observe from Figure 7.7 that the misprediction rate is the same for two sets of distances, 8 and 16. This gives a clue that the 3rd lsb may not used for indexing. From this test it is observed that up to 32 loops are predicted correctly (set: 32, 16).

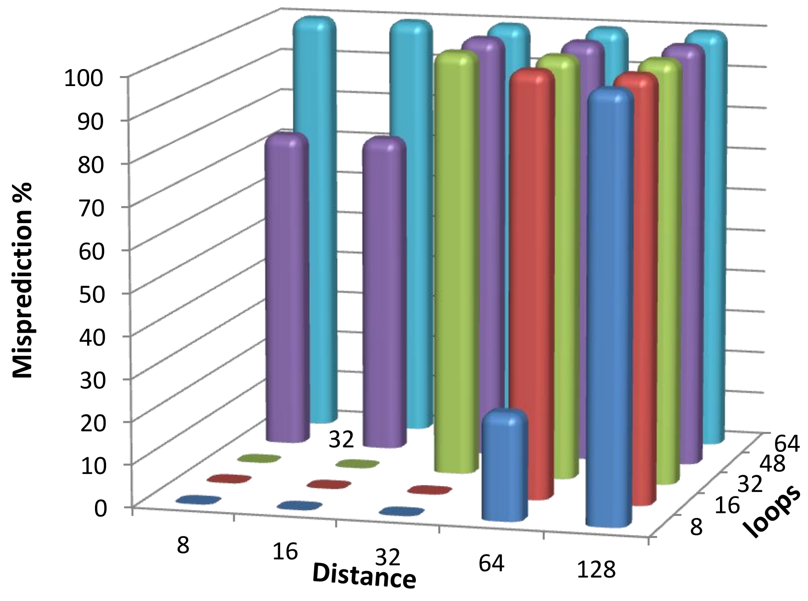


**Figure 7.7: loop capacity analysis – Results**

Test Summary: Indicates loop predictor buffer capacity is 32.

### 7.3.3 Loop MSB analysis and results

As assumed we believe that part of the branch address bits are used to index the loop predictor buffer. In this analysis we try to decode the msb of the branch address that is used as an index to the loop predictor buffer.

Figure 7.8 shows the loop msb analysis test results. The micro benchmark that was used for te BTB msb analysis (section 6.3.3.1) is modified slightly. In order to detect the msb index bit, we set the number of loops in the test just greater than the number of ways of the loop predictor buffer (loop branches = 3). The misprediction rate is calculated for different tests by varying the distance between the three branches from 8 to 512.

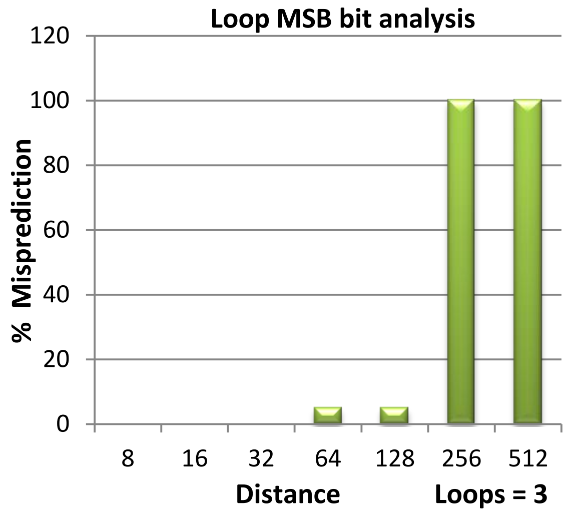


**Figure 7.8: loop MSB bit test results**

We observe that it results in significant misprediction when the distance = 256. This suggests that bit 8 (addr[7]) is the msb that is used to index the loop predictor buffer.

Test Summary: MSB index bit used to index loop predictor buffer is bit 8 (addr[7]).

### 7.3.4 Loop LSB analysis and results

In this analysis we try to determine the lsb that is used to index the loop predictor structure. In this analysis also we set the number of branches to 3 as explained in the previous section. The micro benchmark that was explained in section 6.3.3.2 is modified slightly. The distance between the second and third loop is varied as (1024+1), (1024+2), (1024+4), (1024+8) and (1024+16). By this we make sure that the three loops fall in the same set.

Figure 7.9 shows the results that are collected after performing the lsb analysis. It is clear from the analysis that it starts resulting in misprediction when (D-D') distance is greater than 8. This confirms that the lower 4 bits are not used to index the loop buffer. This is expected as the minimum distance that is practically possible between loops is 8.

Test Summary: LSB index bit used to index loop predictor buffer is bit 4 (addr[3]).

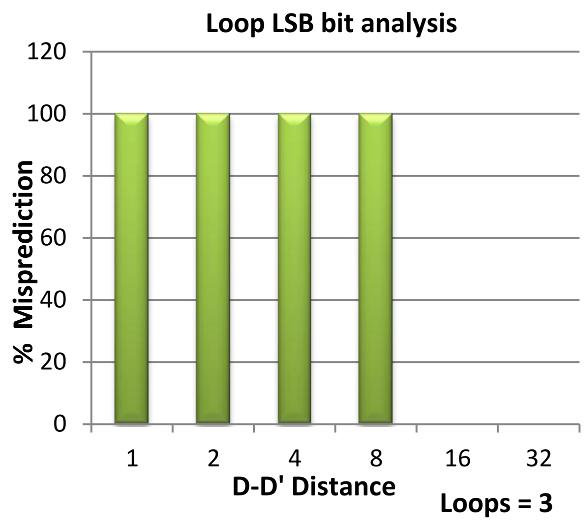


**Figure 7.9:loop LSB bit test results**

### 7.3.5 Loop set associativity analysis and results

In this test we decode the set associative structure of the loop predictor buffer. Like the BTB (section 6.3.2) this can also be 2-way or 4-way set associative. In this test, we keep the distance sufficiently higher than the msb index bit (> addr[7]). We set the distance between loops as 1024. The number of loops in the test is varied from the 2 to 5 (until mispredictions occur) and the results are collected.

From Figure 7.10 it can be observed that the test results in misprediction for the algorithm with 3 loops. This suggests that the loop structure is able to predict properly for 2 branches with the same index field. This concludes that the loop predictor buffer is organised as a two way associative structure with 16 entries in each set.

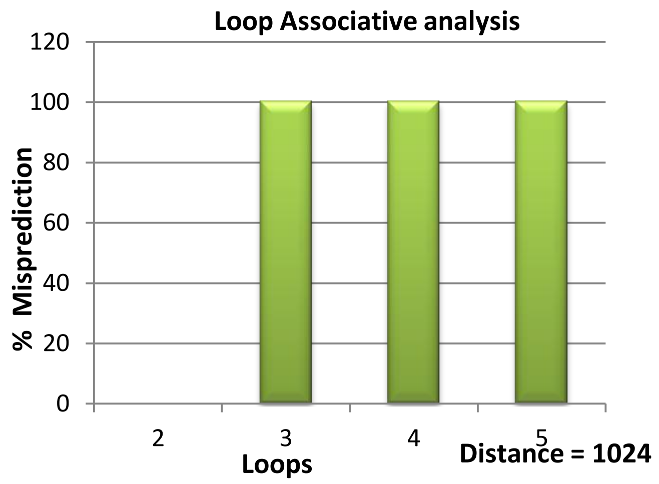


**Figure 7.10: loop set associative test results**

Test summary: concludes that the loop predictor buffer is a 2-way set associative structure.

### 7.3.6 Loop Tag MSB bit analysis and results

As seen from Figure 7.1 the loop buffer shows the tag field. In this analysis we intend to decode the number of tag bits (branch address bits > msb index bits) that are actually stored in the loop predictor buffer. Again we develop on the initial micro benchmark that was used in section 6.3.4 for the BTB structure. We place two loop branches in a test code and vary the distance between them from $2^{10}$ to $2^{15}$.

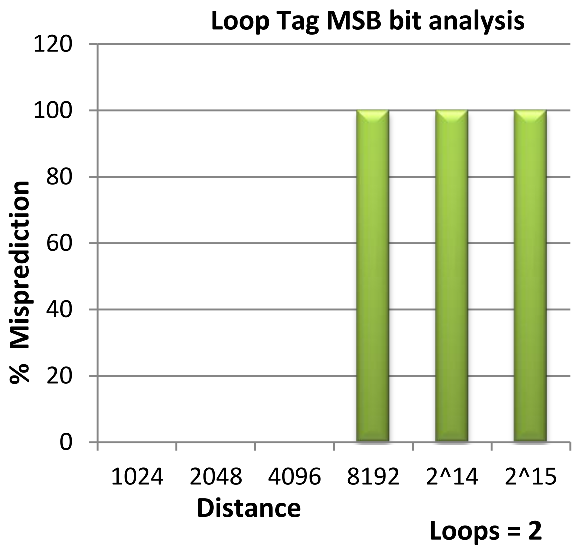


**Figure 7.11: loop tag msb bit test results**

It can be observed from the results collected, as shown in Figure 7.11, that the analysis results in misprediction when the distance = 8192 ($2^{13}$). This indicates that bit 12 (addr[11]) is the msb tag bit that is stored in the loop predictor buffer.

Test Summary: Concludes addr[11] (bit 12) is the msb tag bit that is stored in the loop buffer.

## 7.4 Relationship between Loop Predictor and BTB

In this sub-section, we try to determine the relationship between the loop predictor and the BTB. We try to determine if the loop predictor hit is dependent on the BTB miss.

Figure 7.12 shows the micro benchmark to find the relationship between the loop predictor and the BTB. We already know that the BTB is 8-way set associative. So normally we expect to have misprediction when we include nine conditional branches with distances of 8192 between them. In this analysis we include eight conditional taken branches and the ninth branch is the loop branch.

In this setup, it should result in any misprediction if the loop predictor hit is conditional dependent on the BTB miss. If instead this does not result in any misprediction then it is concluded that the loop predictor hit is not conditionally dependent on the BTB miss.

```
/* Address */             /* Code */
                          int main()
                         {
                                 unsigned int long i;
                                 for(i=0;i<1000000;i++)
                                 _asm {
/* @x */                          jnz loop0     // always taken 1st branch
                                          // dummy non-branch instruction
                        loop0:
/* @x+8192 */                     jnz loop1     // always taken 2nd branch
                                          // dummy non-branch instruction
                        loop1:
/* @x+(2*8192) */                 jnz loop2     // always taken 3rd branch
                                          // dummy non-branch instruction
                        loop2:     …
                                  …
/* @x+(7*8192) */            loop6:  jnz loop7  // always taken 8th branch
                                          // dummy non-branch instruction
                    loop7:
                                 sub ecx,1
                                 cmp ecx,0
/* @x+(8*8192) */                jnz loop7   // 9th branch(loop)
}
                                 return 0;
                         }
```

**Figure 7.12: micro benchmark to analyse relationship between BTB and loop predictor**

Test results: resulted in misprediction.

It was noted that this particular micro benchmark resulted in misprediction. This confirms that the loop predictor hit is conditionally dependent on the BTB hit. We observe the following behaviour:

- A loop predictor hit is conditionally dependent on the BTB hit. That is, a loop predictor predicts the branch only if the branch has an entry in the BTB. If a particular branch encounters a BTB miss then its entry in the loop predictor is not used.

This implementation is understandable as the BTB stores more tag bits than the loop predictor.

Test Summary: Loop predictor hit is conditionally dependent on a BTB hit.

## 7.5 Replacement policy

Since the loop predictor buffer is a 2-way associative structure, we are interested in decoding the replacement policy that is used to replace an existing entry in case of a conflict.

There are two possible implementations that are most commonly used: 1. the least recently used replacement policy or 2. First In First Out (FIFO) policy. Both require an extra bit to be stored in each entry of the buffer.

In order to decode what replacement policy is used we developed a micro benchmark which used three loop branches which are placed in memory such that all three target the same set. The three branches were made to have the occurrence pattern {A, B, A, C}.

If a LRU replacement policy is used then we expect the B and C branches to target the same entry in the loop predictor buffer. Hence this would result in a 50% misprediction rate. If a FIFO replacement policy is used then all the branches target both the entries in the branch predictor buffer. Thus this should result in a 100% misprediction rate.

The micro benchmark shown in Figure 7.6 is modified to include three branches and the above mentioned occurrence pattern. The results showed a misprediction of 50%, which concludes that the LRU replacement policy is incorporated.

Appendix B gives the details of this algorithm and the results obtained.

Test Results: resulted in 50% misprediction.

Test Summary: LRU replacement policy is incorporated.

# Chapter 8.

# Automatic Analyser

In this chapter we talk about the automatic analyser. The automatic analyser is a program that we developed to do all the analysis automatically. The main motivation to develop an automatic analyser is that it has an advantage of getting the details about the organisation of the underlying branch predictor without involving any manual analysis. This is particularly important in creating an architecture-aware compiler.

## 8.1 Block Diagram

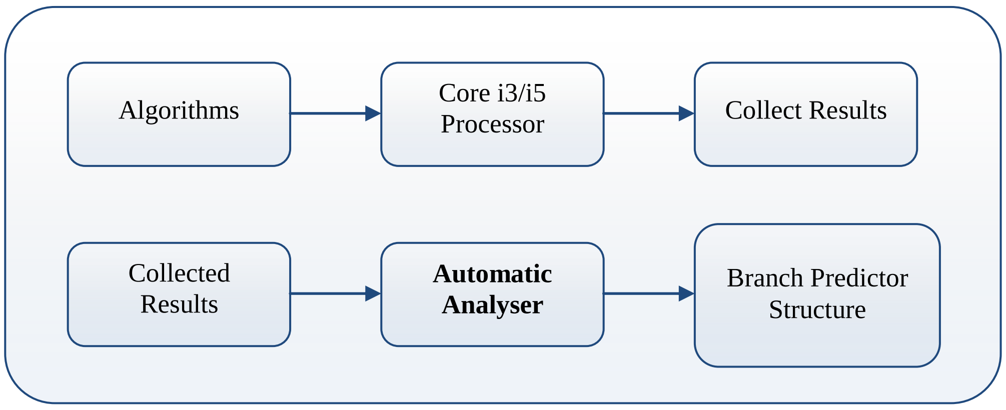


**Figure 8.1: Basic Implementation Block Diagram**

From the figure it is observed that the automatic analyser takes the collected results as input and outputs the underlying branch predictor structure. The advantage of having an automatic analyser is that it does not require any knowledge of the underlying architecture. It also helps in automating the process. The algorithms/micro-benchmarks developed are run on a processor and the collected results are just fed to the automatic analyser to get the branch predictor structure.

The automatic analyser is designed to give the structure of two types of branch predictors:

- Branch target buffer predictor.
- Loop predictor.

In the design of an automatic analyser it is assumed that these branch prediction buffers are indexed by part of the branch address. If the automatic analyser is not able to get any conclusive structure of the branch predictor then it outputs a message indicating this fact. In this case again the manual analysis should be used. Our automatic analyser is tested for the Nehalem architecture (core i3 and core i5 processors).

## 8.2 Automatic Analyser – Branch target buffer

Figure 8.2 shows the output of the automatic analyser indicating the structure of the BTB. The input is the misprediction results which were obtained after running each micro benchmark as explained in Chapter 6. Appendix A gives more details on the structure of the input that needs to be provided. Appendix A also includes the details of the output of the analyser.

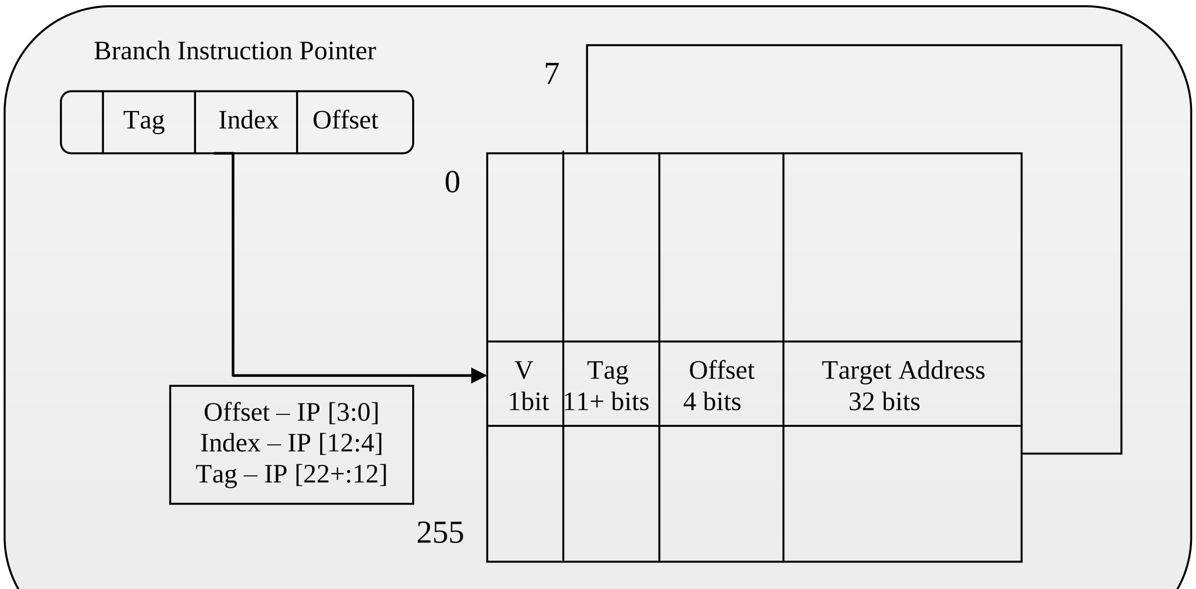


**Figure 8.2: Automatic Analyser output: BTB structure**

## 8.3 Automatic Analyser – Loop Predictor

Figure 8.3 shows the output structure showing the loop predictor structure. The inputs are the results obtained after running the micro benchmarks discussed in Chapter 7. The input and output of the analyser for loop predictor is shown in Appendix A.

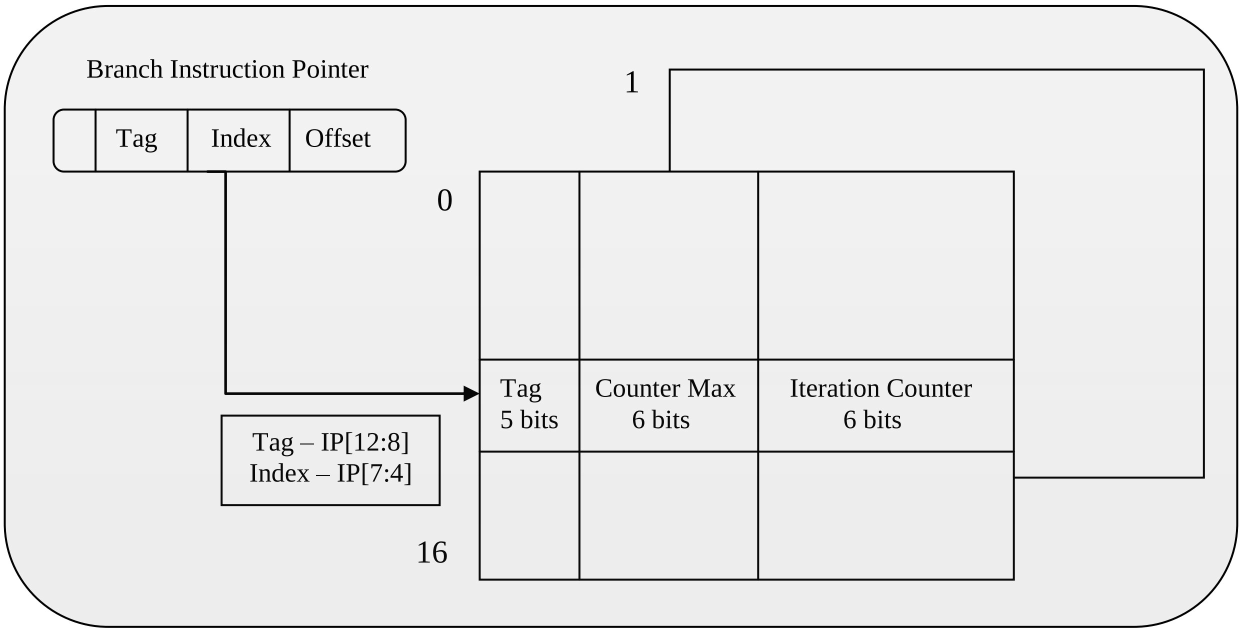


**Figure 8.3: Automatic Analyser output: Loop predictor structure**

## 8.4 Intel Core i5 Branch Prediction Analysis and Results

The analysis of the BTB branch predictor and the loop predictor were also performed on core i5 processor and similar results were obtained. Thus we have performed the analysis on core i3 and core i5 processors and we obtained similar results for both. This is understandable as both the processors have the same architecture and this also confirms the behaviour of the micro benchmarks developed.

## 8.5 Atom Processor Analysis

We had intended to perform the analysis on Intel's Atom processor as well. Unfortunately the Vtune performance analyser tool does not support the version of Atom processors which were available. In particular we tried to run our algorithms on an Atom NT535 and an Atom N450 processor. We monitored the following hardware events that were available in the Atom architecture but unfortunately they were not indicating the correct values.

- BR_INST_RETIRED.ANY
- BR_INST_RETIRED.MISPRED
- BR_INST_TYPE_RETIRED.COND
- BR_MISSP_TYPE_RETIRED.MISPRED

# Chapter 9.

# Conclusions and Future Work

## 9.1 Objectives Achieved

The following are the main achievements of this work:

- We have successfully carried out the study of branch predictors on ARM and Intel Nehalem architectures.
- The branch prediction structure of the ARM11 processor was studied and the structure of its branch predictor is verified by the set of algorithms developed.
- The branch prediction structure of the Intel Nehalem architecture was studied and we have successfully revealed the size, structure and access mechanism of the following types of branch predictors in Nehalem:
    - Branch Target Buffer Predictor
    - Loop Predictor
- The micro benchmarks developed were tested on two variants of the Nehalem architecture and the results obtained were similar, as expected.
- An automated analyser was developed to automate the whole process. It performs the analysis and reveals the structure of the underlying branch predictor.

## 9.2 Conclusions

Branch predictor units have become one of the critical blocks in the design of micro architecture for the modern processor, and they greatly influence the performance of the processor. To achieve efficient performance, the pipeline and the execution units of modern processors are to be kept filled with instructions, which greatly depends on the efficient implementation of the branch predictor unit.

In the thesis a set of micro benchmarks are presented targeting some of the important structures of a branch predictor unit, such as the branch target buffer and loop predictor. We have developed a systematic approach to determine the structure of the BTB and the loop predictor of the Intel Nehalem architecture. We have applied our algorithms and micro benchmarks on the 'Arrandale' codename of Intel processors: core i3 – 350M and core i5 – 450M. For each of them, we have determined the size as well as the organisation of their branch predictors.

We have developed an automated analyser that takes the result of our algorithms as an input and determines the structure of the branch predictor. This helps in automating the whole process and automatically detecting the branch predictor organisation.

## 9.3 Future Work

Insights on the branch predictor structure can be used in the design of architecture-aware compilers. The automatic analyser can be further modified for various different architectures and then can be incorporated into the compiler to automatically get the structure of the underlying branch predictor to improve the performance of the application.

It is predicted that the Nehalem architecture implements four types of branch predictors. The work done on two variants of Nehalem's branch predictor (BTB and loop prediction buffer) can be used as a starting point to try and decode the structure of the other two types of branch predictor (Indirect predictor and global predictor). Following which, an automatic analyser can be developed to reveal the entire structure of the Nehalem branch predictor.

The micro benchmarks/automated analyser developed can be used as a starting point in the design of a more robust analyser that is capable of analysing the complete branch prediction structure and this robust analyser can act as a valuable tool for the designers of branch predictors to test the performance of branch predictors as they are being designed.

# APPENDICES

# Appendix A

## Automatic Analyser Implementation

### Input

The results of the various micro benchmarks discussed in Chapter 6 and Chapter 7 are given as input to the automatic analyser. The figure below shows the contents of the header file that consists of all the results collected.

**BTB Algorithm's Results (input to analyser):**

```
int btb_table[BTB_ROWS][BTB_COLUMNS] = {
{0,2,4,8,16,32,64,128,256,512,1024,2048,4096},
{128,60,0,0,0,0,0,0,0,75,94,100,100},
{256,60,0,0,0,0,0,0,64,95,95,97,98},
{512,61,2,0,0,0,0,71,96,96,96,96,96},
{1024,62,5,0,0,0,78,93,96,96,96,100,100},
{2048,62,5,3,0,90,96,96,97,100,100,100,100},
{4096,62,70,70,87,90,97,97,97,100,100,100,100},
{8192,86,96,96,96,98,98,98,100,100,100,100,100}
};   //section 6.3.1

int btb_associativity_table[2][BTB_ENTRIES_ASSOCIATIVE] = {
{2,4,8,9,10,11,12,16},{0,0,0,10,20,70,70,50}}; // section 6.3.2

int btb_msb_table[2][BTB_ENTRIES_MSB] = {
{512,1024,2048,4096,8192,16384,32768,65536,262144},{0,0,0,100,100
,100,100,100,100} };  // section 6.3.3.1

int btb_lsb_table[2][BTB_ENTRIES_LSB] = {
{0,1,2,4,8,16,32},{100,100,100,100,100,0,0} };  //section 6.3.3.2

int btb_tag_msb_table[2][BTB_ENTRIES_TAG_MSB] = {
{20,21,22,23,24,25},{0,0,100,100,100,100} };   //section 6.3.4
```

**Automated Analyser output (BTB):**

```
********************* BTB ANALYSIS RESULTS ********************
             Number of entries in the BTB is = 2048
             BTB is 8 way set associative
             BTB msb index bit is 13
             BTB lsb index bit is 5
             BTB tag msb bit is 21
******************* BTB ANALYSIS RESULTS END *****************
```

**Loop Predictor Algorithm's Results (input to analyser):**

```
int loop_table[LOOP_ROWS][LOOP_COLUMNS] = { {0,8,16,32,64,128},
{8,0,0,0,25,100},
{16,0,0,0,100,100},
{32,0,0,100,100,100},
{48,75,75,100,100,100},
{64,100,100,100,100,100},
{128,100,100,100,100,100},
{192,100,100,100,100,100},
{256,100,100,100,100,100}
};    //section 7.3.2

int loop_count_table[2][LOOP_COUNT_ENTRIES] = {
{8,16,32,64,128},{0,0,0,0,100} };   //section 7.3.1

int loop_associativity_table[2][LOOP_ENTRIES_ASSOCIATIVE] = {
{2,3,4,5},{0,100,100,100}};    //section 7.3.5

int loop_msb_table[2][LOOP_ENTRIES_MSB] = {
{8,16,32,64,128,256,512},{0,0,0,0,0,100,100} };    //section 7.3.3

int loop_lsb_table[2][LOOP_ENTRIES_LSB] = {
{0,1,2,4,8,16,32,64,128},{100,100,100,100,100,0,0,0,0} };
//section 7.3.4

int loop_tag_msb_table[2][LOOP_ENTRIES_TAG_MSB] = {
{128,256,512,1024,2048,4096,8192,16384},{0,0,0,0,0,0,100,100} };
// section 7.3.6
```

**Automated Analyser output (Loop Predictor):**

```
********************** LOOP ANALYSIS RESULTS **********************
           Number of entries in the loop BPB is = 32
           Maximum loop counter in the loop BPB is = 64
           loop predictor buffer is 2 way set associative
           loop predictor buffer msb index bit is 8
           loop predictor buffer lsb index bit is 5
           loop predictor buffer tag msb bit is 13
******************* LOOP ANALYSIS RESULTS END ********************
```

# Appendix B

## Loop Entry Replacement Policy

The code snippet below shows the algorithm used to determine the replacement policy used in a loop predictor. Three loop branches are included: ‘A’, ‘B’ and ‘C’. The execution pattern is A, B, A, C.

```
/*Address*/         /* CODE */
                    int main(){
                          unsigned int long i;
                          for(i=0;i<1000000;i++){
                          _asm {
                                mov edx, 2
                                mov ecx,60
                    loop0:
                                sub ecx,1
                                cmp ecx,0
/*@x*/                          jnz loop0   // 'A' loop branch
                                ...   // non branch instructions
                                cmp edx,1
                                je loop3
                                mov ecx,60
                    loop1:
                                sub ecx,1
                                cmp ecx,0
/*@x+256*/                      jnz loop1   // 'B' loop branch
                                ...   // non branch instructions
                                mov ecx,60
                                sub edx,1
                                cmp edx,0   //to get ABAC pattern
                                jnz loop0
                    loop3:
                                mov ecx,60
                    loop2:
                                sub ecx,1
                                cmp ecx,0
/*@x+512*/                      jnz loop2  // 'C' loop branch
                                nop
                                }
                          }
                                return 0;
                          }
```

**Result:**

Percentage misprediction is calculated as follows:

$$Misprediction\ \% = \left( \frac{Total\ mispredicted\ branches}{\frac{\text{total branches}}{count}} \right)$$

| Count | Branches any | Branches conditional | Mispredicted branches any | Mispredicted conditional | Misprediction % |
|---|---|---|---|---|---|
| **60** | 244,000,000 | 243,000,000 | 2,520,000 | 2,600,000 | 50 |
| **30** | 122,000,000 | 121,000,000 | 2,450,000 | 2,500,000 | 50 |

# Bibliography

1. [book auth.] David A Patterson John L Hennessy. *Computer Architecture: A Quantitative Approach.*

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

3. *Experiment Flows and Microbenchmarks for Reverse Engineering of Branch Predictor Structures.* **Vladimir Uzelac, Aleksandar Milenkovic.** s.l. : IEEE, 2009. 978-1-4244-4184-6/09.

4. *The relative impact of memory latency, bandwidth and branch limit to microprocessor performance.* **P.Ranganathan, N Jouppi.** June 1997. Proceedings of the 1st workshop on mixing logic and DRAM: Chips that compute and Remember.

5. **Dezo Sima, Terence Fountain, Peter Kacsuk.** *A design based approach, advanced computer architecture.* p. page 104 and page 103.

6. *The microarchitecture of Intel, AMD and VIA CPUs.* **Agner, Fog.** s.l. : http://www.agner.org/optimize/#manuals, 2011.

7. *Two Level Adaptive Training Branch Prediction.* **Yeh, Patt Y N.** Albuquerque, New Mexico : Proceedings of the 24th annual international symposium on Microarchitecture, 1991. ACM pp51-61.

8. *Of Limits and Myths in Branch Prediction.* **Eden, Avinoam Nomik.** 2001.

9. *The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference.* **Sprangle, Literature: E.** Denver : s.n., June 1997. Proceedings of the 24th International Symbosium on Computer Architecture .

10. *Accurate Indirect Branch Prediction.* **Karel Driesen, Urs Holzle.** Barcelona : s.n., July 1998. ISCA '98 Conference Proceedings. pp. pp. 167-178.

11. *Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches.* **Po-Yung Chang, Patt Y N, Evers M,.** s.l. : 23rd Annual International Symposium, May 1996. pp. 22-24.

12. *Mechanism for Return Stack and Branch History Corrections under Misprediction in Deep Pipeline Design.* **Guan-Ying Chiu, Hui-Chin Yang, Walter Yuan-Hwa Li, Chung-Ping Chung.** 2008. Computer Systems Architecture Conference .

13. *Branch prediction using both global and local branch history.* **Chang M C, Chou Y-W.** s.l. : Computers and Digital Techniques, March 2002. IEEE proceedings. pp. Vol-149.

14. *Selective branch prediction reversal by correlating with data values and control flow.* **Aragon J.L, Gonzalez, J. Garcia J.M, Gonzalez A.** 2001. Vol. Identifier: 10.1109/ICCD.2001.955033, pp. pp 228 - 233.

15. *Demystifying Intel Branch Predictors.* **Milena Milenkovic, Aleksandar Milenkovic.** 2005. pp. 210-213. 10.1109/NORCHP.2005.1597026.

16. *Technical Reference Manual, 'ARM1136JF-S and ARM1136J-S'.* s.l. : ARM, revision: r1p5.

17. Inside Nehalem, Intel's future processor and system, 2011. *Real World Tech.* [Online] http://www.realworldtech.com/page.cfm?ArticleID=RWT040208182719&p=10.

18. Review Intel Nehalem. *Tom's hardware.* [Online] 2011 йил. http://www.tomshardware.com/reviews/intel-i7-nehalem-cpu.2041-3.html.

19. ARM Profiler Non-Intrusive Perfromance Analysis. *ARM.* [Online] 2011 йил. http://www.arm.com/products/tools/software-tools/rvds/arm-profiler.php.

20. Intel Vtune Amplifier XE 2011. *Intel Corporation.* [Online] http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

21. First the tick, Now the Tock: Next Generation Intel Micro-architecture(Nehalem). *Intel.* [Online] http://www.intel.com/pressroom/archive/reference/whitepaper_Nehalem.pdf.

22. *Branch Prediction using both global and local branch history information.* **Chang, M, Chou, Y-W.** Mar-2002. Computers and Digital Techniques, IEEE proceedings. Vols. vol-149.

23. *A Method and apparatus for branch prediction using a second level branch prediction table.* Literature: WO Patent 2000/014628.

24. *method of branch prediction using loop counters.* Literature: US Patent 5909573.