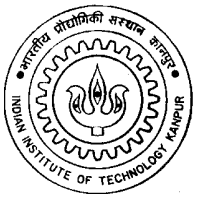
Instruction Cache Address Prediction for Superscalar Processors

A Thesis Submitted
in Partial Fulfillment of the Requirements
for the Degree of

Master of Technology

by

Shankar Seal



to the

Department of Computer Science & Engineering
Indian Institute of Technology, Kanpur

January, 2001

Certificate

This is to certify that the work contained in the thesis entitled "*Instruction Cache Address Prediction for Superscalar Processors*", by *Shankar Seal*, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

January, 2001

(Dr. Rajat Moona)
Department of Computer Science & Engineering,
Indian Institute of Technology,
Kanpur.

Acknowledgements

I would like to take this opportunity to express my gratitude to my thesis supervisor Dr. Rajat Moona. He has helped me at every stage of this work with his innovative ideas and his deep knowledge of the subject, without which this thesis could not have been possible. He has always been very understanding, compassionate, encouraging and helpful to me.

I am thankful to all the faculty members of the department of Computer Science and Engineering, for helping me have a better understanding of various subjects of computer science. I am also thankful to Atul Kumar, P. Suresh and Rajiv A. R. for helping me in various aspects of this work.

I would like to all my classmates of the Mtech'99 batch for being supportive and helpful friends.

I am grateful to my parents for their blessings and encouragement. Finally I would like to thank Sudeshna, who has been a constant source of inspiration to me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The recent processors like RS10000, Sun UltraSPARC etc. employ superscalar architecture. These processors issue multiple instructions per cycle and employ multiple functional units and hardware scheduling techniques to achieve maximum parallelism at the instruction level. To exploit maximum efficiency such multiple issue processors must be fed by high instruction fetch bandwidth. The instruction fetch unit must fetch enough instructions every cycle to keep the functional units busy. No clock cycle should go idle and thus several instructions need to be fetched at every clock cycle.

Meeting these performance requirements from the instruction fetch unit, depends among other factors on the I-cache performance and the branch prediction mechanism. In multi-way set–associative caches, data can reside in one of many cache *blocks* within a cache *set*. The data is stored along with a *tag* which is derived from the memory address where the data is actually stored. In conventional multi-way set-associative I-cache, an instruction is read in the following way. The address generated by the processor is divided into two parts — *tag* and *index*. The index selects the set of the I-cache to be accessed. The tag is compared simultaneously with the cache blocks tags of all the blocks in the set. The data is read from the block whose tag matches with the instruction tag. If none of the stored tags matches with the instruction tag, then a cache miss occurs and the instruction is read from the other levels of the memory hierarchy. With very fast clock cycles, this whole

procedure requires more than one clock cycle to complete. It is expected that the future processors, which are likely to have a very deep pipeline, will require several pipeline stages to fetch instructions.

Control hazards also play a crucial role in performance of the CPU. A conditional branch instruction can potentially change the flow of program. Before this instruction is executed, and the branch outcome is known, the address of the next instruction is not known. Speculative fetching of instructions (assuming the branch is not taken) can lead to the non-utilisation of the pipeline, if the branch is actually taken, thus wasting clock cycles. These unresolved branch instructions give rise to the *control hazard*. There are several methods of dealing with this problem and to reduce the associated penalties. The simplest method is to stall the pipeline once a branch instruction is detected. The other methods are to support *delayed branches*, or *branch prediction schemes*. In the latter technique, the likely outcome of the branch instruction and the target address, are predicted *before* the branch instructions are executed. For a multiple-issue processor, handling of control hazard is even more crucial for the performance. In the best case for an n-issue processor, branches will come up to n times faster into the pipeline. To reduce control hazard, correct prediction of the next instruction has to be made every clock cycle. In the next chapter we discuss various branch prediction schemes widely used in the modern processors.

1.1 Problem Statement

The two main factors that affect the performance of a multi-issue processor are: cache access time and fetching correct instructions every cycle. We address both these problems by predicting the address of the Instruction cache from where the next instruction has to be fetched. For a set associative cache, this address comprises of the selection of the set as well as the block within the set. We call this prediction scheme the next cache-line prediction scheme and we will use this term in the rest of this thesis. This kind of prediction will have two advantages. First, since we are also predicting the block within the cache-set from where the instruction has to be

fetched, the tag part of the instruction address need not be compared with the tags of the various other blocks in the set for selecting the block. This will make the cache access faster and also reduce power consumption of the I-cache. [4]

A misprediction is known by comparing the tag and index of the selected block with that of the instruction address. A misprediction however is associated with the penalty of killing wrongly fetched instructions. The next cache-line prediction reduces the need for sophisticated branch prediction scheme as well. At the fetch stage itself, we are predicting the next cacheline for fetching the next instruction, a correct prediction, can thus reduce control hazards.

1.2 Related Works

Koji Inoue et al. [4] proposed a new Way–predicting Set–Associative Cache. Experimental results showed that this cache gives high performance at low power consumption. In their scheme, known as the *way–predicting cache*, only one way in a cache-set is selected based on what they called the MRU (Most Recently Used) Algorithm. In an n-way set associative cache, each set has $\log_2$n bit MRU information. These bits are stored in a table which is accessed using the set-index address, and is used to speculatively select one way from the corresponding cache-set. Experimental results showed that, compared to conventional set–associative cache, the power consumption was reduced by 60–70% without any performance degradation.

Calder et al. [1] proposed a cache design called *Predictive Sequential Associative Cache* that provides same miss ratio as a two-way set associative cache and access time closer to a direct mapped cache. In this design, a conventional direct mapped cache is divided conceptually into two banks. This technique makes use of several *prediction sources* to select between the two banks in a set. The prediction sources can be adjusted according to pipeline constraints.

A similar work for Instruction Caches has been done by Calder and Grunwald [2]. They call their scheme as the *Next Cache Line and Set Prediction* or the NLS scheme. In this scheme, instructions following a branch instruction is fetched using an index into the Instruction Cache instead of a predicted branch target address.

Their work also investigates the use of NLS along with 2 bit correlating branch prediction table (see Chapter 2).

In our scheme, for every instruction fetched, we predict the address in instruction cache, from where the next instruction is to be fetched. We predict the cacheline, that is both the cache set, and the block within the cache where the desired instruction is expected to reside.

1.3 Goals Achieved

In this thesis we propose a new scheme for superscalar processors to predict the Instruction Cache address where the next instruction to be executed. The goals achieved in this work are

- We developed an address trace generator for UltraSPARC processor. The program instrumented a given input program at assembly language level and the instrumented code the address trace of the program. See chapter 3.
- A simulator was developed to test the performance of the newly proposed cacheline prediction scheme. The simulator models the fetch stage of a processor and its various parameters are configurable. See chapter 5.
- Using these tools several benchmarks were tested and studies of the scheme was done. The experimental results are given in Chapter 6.

1.4 Organization

The rest of the thesis is organized as follows. In the next chapter we briefly discuss the different branch prediction schemes, to compare them with our scheme. In Chapter 3, we discuss the trace generation mechanism of different benchmark programs. These address traces are the inputs to the simulation program used to test this new scheme. In Chapter 4 we provide and discuss in detail, the design issues and the performance issues of our scheme. In Chapter 5 we discuss the implementation of

the simulator. In Chapter 6 we provide the detailed experimental results. Finally we conclude this work and suggest some work that can be done to extend this work.

Chapter 2

Branch Prediction Mechanism

In this chapter, the various branch prediction schemes in modern processors are discussed in brief. Typically 15–20% of all instructions in a program are control transfer instructions. Predicting the right direction of control flow, early in the pipeline is vital in exploiting Instruction Level Parallelism (ILP) in a processor, especially in the superscalar ones. Various schemes have been proposed to predict the outcome of branch instructions.

2.1 What is Branch Prediction

Branch Prediction is a scheme predicts the outcome of a branch instruction. After a prediction is made, a processor can continue to speculatively fetch instructions from the predicted direction and thus maintain the supply of instructions to the pipeline. In absence of any prediction scheme, the processor must stall for unresolved branch instructions, which imposes heavy penalty on performance of a processor. A correct branch prediction can overcome this problem and can reduce control hazards and exploit more ILP. If the correct branch prediction rates are high enough to overshadow the misprediction penalties, then the overall performance of the processor is likely to improve.

2.2 Static vs. Dynamic Branch Prediction

Static Branch Prediction schemes define a static prediction used by the processor. For example, MIPS-X, predicts all branches as *taken* while Motorola MC88000 predicts all branches to be *not taken*. Static prediction schemes are utilized by the compilers where in they use the profiling information from previous runs of a program and generate appropriate kind of instructions.

Dynamic branch prediction techniques take information derived from the dynamic execution of the program and predict the outcome of a branch instruction. The hardware produces two outputs, the expected direction of the branch and the branch target. The predicted branch direction is signified by a bit (taken / not taken), while the target is the address of the next instruction in the predicted path. For our reference in this thesis, the table that provides the direction outcome of a branch instruction as a *branch prediction table*, or BPT [3]. Similarly the table that is used to predict the address of the target instruction is referred to as a *branch target buffer*, or BTB [3]. In this thesis we restrict ourselves to dynamic branch prediction schemes only.

2.3 Dynamic Branch Prediction

The dynamic branch prediction mechanisms can be classified into two groups. The *counter based* schemes, and the *correlation based* schemes. The predictors of the former type, also known as the One-level branch predictors, predict the outcome of a branch, depending on its recent behavior. The predictors of the second type, also known as the Two-level branch predictors, which predict the outcome of a branch instruction on the basis of recent outcomes of other branch instructions as well.

2.3.1 Counter Based Branch Prediction

The simplest form of this scheme is a *branch-prediction buffer* or *branch history table*. It is basically a small cache indexed by few least significant bits of the branch instruction address. In the simplest case, the entries of the cache are one bit wide.

When a bit corresponding to a branch instruction address is set, the branch is predicted to be taken. If this bit is zero, then the branch is predicted to be not taken. After the buffer is accessed, and the prediction bit is read, the instructions are fetched from the predicted direction. In case of a misprediction, the wrongly fetched instructions are killed and the prediction bit is toggled. This scheme is useful, only when the branch delay is longer than the time taken to compute possible target PC. This scheme however has some drawbacks. Consider a loop branch which is taken nine times in a row and not taken the last time. The above scheme will predict branch to be not taken, the first time the branch is encountered. The prediction bit is then toggled and it gives the correct prediction for the next eight times. The last time, when the branch is actually not taken, it is predicted to be taken, leading to another misprediction. So for a branch which is taken 90% of time, the prediction accuracy is only 80%. The situation can be improved, if instead of one prediction bit, we used n-prediction bits.

In an n-bit prediction buffer, each entry is an n-bit counter. The counter is incremented, every time the branch is resolved to be taken, and decremented every time it is not taken. A branch is predicted to be not taken, if the corresponding counter value is less than 2^{n-1}, and predicted taken otherwise.

A 2-bit counter branch predictor is most commonly used and is found sufficient in most branch applications.

The buffer can be implemented in many ways.

- **Direct Mapping** The cache entry is indexed directly by the last few bits of the branch instruction address. However, if the size of the buffer is too small, a large number of branch instruction will map to the same entry, causing **aliasing**.

- **Fully Associative** In this types of buffers, the entries, along with the prediction bits also contain tag information. The cache entry is found by comparing the tag with the few least significant bits of the branch instruction address. This scheme needs *cache replacement* strategy, which could be *LRU* or *FIFO*.

- **Set Associative** In this implementation, each address maps into a set of

entries, having different tags. The cache is indexed as usual using few least significant bits of the instruction address, and then tag comparison is done associatively to obtain the correct prediction bits.

2.3.2 Correlation Based Branch Prediction

Branch predictors, that uses the recent behaviors of other branches for predicting the outcome of a branch are known as *correlating predictors* or *two-level predictors*. This scheme was originally proposed by Yeh and Patt [8]. To understand the working of such a predictor, let us consider a correlating branch predictor which uses one correlation bit and one prediction bit. This type of predictor can be viewed as consisting of *two separate* prediction bits. One prediction if the branch instruction executed prior to the one being predicted was taken, and the other, if that branch was not taken. This predictor is called a (1,1) predictor since it uses the behavior of the last (one) branch to select from a pair of one bit predictors. In general a (m,n) correlating predictor, uses the behavior of the last m branch instructions to choose from 2^m branch predictors, each having n prediction bits.

This scheme can be implemented by a rather simplistic hardware. It consists of a *global history table* which holds the history of outcome of the last m branches. The global history table can be implemented using an m-bit shift register, where a 1 is shifted in every time a branch is taken. Similarly a 0 is shifted if a branch is not taken. The branch prediction buffer can be viewed as a two dimensional table of counters. It can be indexed using the instruction address and the global history table.

In the original scheme proposed by Yeh and Patt, known as the *Two-level adaptive prediction scheme* (also referred as the **Yeh algorithm**), history information of the previous branch outcomes was maintained. This information however was local and kept was in *correlation registers*, sometimes called as the *local history table*. For each branch, there is an associated correlation register, and the pattern of the branch history is stored there. Based on this pattern, a particular entry in another buffer called the *global pattern table* (GPT) is accessed. The corresponding correlation register and the entry in GPT are updated each time a branch is resolved. An

entry in GPT can be shared by several branches. In order to lookup, we index the table by the lower bits of the address of the branch instruction. The entry in the table provides the corresponding correlation register, whose value is used to index GPT to get the appropriate prediction bits.

Although a correlating predictor gives better performance than simple one-level branch predictors, it has a disadvantage of an expensive implementation and the fact that the so called **Warm Up Phase** (the time the table entries contain usable values) is much longer.

Besides the two major classes of branch predictors discussed above, there are another kinds of predictors such as the *hybrid branch predictors* [5], which use more than one sources of predictions, which are independent of each other. A selection mechanism is used to select from among different sources of prediction.

2.4 Branch Target Buffer

In all the schemes discussed in the previous section, the prediction occurs at the instruction decode stage when the fetched instruction is decoded and it is known to be a branch instruction. If the branch delay is longer than the time to compute the target PC, the above schemes can reduce the branch delay by predicting the outcome of the branches earlier. But in certain processors, the outcome of the branch and the target address are known at roughly the same time. In this cases, the above schemes cannot help much. To reduce the branch penalty in these cases, we have to know the branch target address by the end of the Instruction Fetch (IF) stage itself. Therefore the address to fetch the next instruction, is predicted even *before* we know whether the current instruction is a branch instruction or not. In such schemes we can possibly have a zero branch penalty. This type of branch prediction schemes, where we predict the address of the next instruction are called a *branch-target buffer* or *branch-target cache* prediction scheme. Throughout this text, we shall refer to these type of buffers as **Branch Target Buffer** or **BTB**.

In a BTB based prediction, the buffer is accessed during the IF stage, using the address of the currently fetched instruction. Thus the next PC is known by the end

of IF cycle and the instruction fetch can continue without any delay. The branch target buffer is a table, whose entries contain PC and the predicted next PC. After an instruction is fetched, it is looked up in the table. If a matching entry is found, then the corresponding predicted PC is taken to be the next PC. If at a later stage, the current instruction does not take the branch, then we have a misprediction. The wrongly fetched instruction is killed and the entry is deleted from the buffer. If on the other hand, the current PC is not found in the buffer, then the next sequential instruction is fetched. Many processors improve the branch prediction by using both BPT based and BTB based schemes simultaneously. Examples are PowerPC 620, where the two buffers are decoupled, and the Pentium, where the BTB and the Branch Prediction Buffer are coupled together.

2.5 Branch Prediction schemes in recent Processors

- **PowerPC604** [6] has a 64 entry fully associative Branch Target Buffer for predicting the Branch Target Address and a decoupled direct mapped 512 entry Pattern History Table.

- **PowerPC620** has a 256 entry two-way set associative Branch Target Buffer for predicting the Branch Target Address and a decoupled direct mapped branch prediction buffer.

- **UltraSPARC** uses a 2-bit branch prediction scheme.

- **Intel Pentium** contains a 256 entry 4-way set associative Branch Target Buffer. Coupled with each Branch Target Buffer entry is a simple one-level 2-bit branch predictor that is responsible for the branch prediction.

- **Intel Pentium Pro** works with a 512 entry 4-way set associative Branch Target Buffer. Coupled with each Branch Target Buffer entry is in this case a 4-bit local branch history. This is mapped in a second level onto a Global Pattern History Table, thus implementing Yeh's Algorithm mentioned earlier.

2.6 Simulation

In our simulation we compared the performance of our newly proposed scheme with that of some existing prediction schemes. For this, we modeled in our simulator, our next cacheline prediction scheme, a BTB based scheme and a (2,2) correlation based Branch Prediction scheme.

Chapter 3

Address Trace Generation

Address trace of a program is the sequence of instruction addresses, through which the control of a program flows during its execution.

We needed the address traces of the benchmark programs that we tested with our new scheme. The sequence of instructions that were predicted by our scheme were compared with the address trace generated. In this chapter we discuss the process by which these address traces were generated.

3.1 Overview

We developed a code instrumentation program which changed the program to be tested, so that it generates the address trace when it is run. The instrumentation was not done on the source code, but on the compiled *assembly code*. The instrumentation was done in a way so that the original address trace is not affected. We chose Sun UltraSPARC assembly language code for instrumentation. The reason of this choice is that UltraSPARC is a RISC processor with fixed instruction length of 4 bytes. It has got many other features which made code instrumentation easier.

We first compiled a C source code using – S option of gcc to get the assembly code. Then the instrumentation program was run on the assembly program to generate the instrumented code. In the instrumented code all occurrences of control transfer instructions, which included all the **branch** instructions the function **call**

instruction, and the **ret** from instruction; were replaced by a instruction to transfer the control to a function in instrumentation code. We called this function “foo” In this newly introduced function we calculated the addresses of the source and destination instructions of the control transfer instruction involved, and wrote them into a trace file. Then the actual control transfer instruction was executed from within the "foo" function. This way the original code trace was not modified and the instrumentation code was not traced. We now discuss the instrumentation mechanism in more detail.

3.2 Features of SPARC Architecture

In this section some features of the SPARC processor architecture [7] are discussed, which are necessary to understand the code instrumentation procedure.

3.2.1 Registers

The SPARC processor has two types of registers. The *general purpose* registers and *control/status* registers. The general purpose register set consists of 8 **global** registers (g0 to g7), and several register windows which are changed upon function calls and restored upon return. The register windows have 8 *in* registers (i0 to i7), 8 *local* registers (l0 to l7) and 8 *out* registers (o0 to 07).

Upon executing the *save* instruction, a new register window is allocated such that its *in* registers i0 to i7 are same as the *out* registers o0 to o7 registers of the old window. Thus parameters can be passed and values returned between functions using registers. So the save instruction is typically associated with a function call. The *restore* restores the register window. See 3.2.3 for details.

The following control registers, are important for instrumentation purposes.

- **Program Counter (PC)** contains the address of the instruction currently being executed by the Instruction Unit (IU).
- **The next Program Counter (nPC)** register contains the address of the next instruction to be executed.

- **Condition Code Register (CCR)** holds the integer condition codes.

3.2.2 Delayed Control Statement

In SPARC processors, control transfer instructions are *delayed*, i.e. the effect of the control transfer is delayed by one instruction. In case of these delayed control transfer instructions, after the execution the value of the nPC register is changed. The PC is changed to PC+4 and the next instruction in the program storage order is fetched into the pipeline. As a result the control transfer takes place after a delay of one cycle. The instruction following a delayed control transfer instruction is said to be in the **delay slot**. Usually after a branch instruction or a call instruction the compiler introduces the **nop** instruction. Sometimes, optimized codes can have other meaningful instructions in the delay slot which does not affect the control flow and maintains the program semantics.

3.2.3 SAVE and RESTORE Instructions

The **SAVE** instruction provides the routine executing it, with a new register window. The *out* registers from the old window are visible as the *in* registers of the new window. The contents of the newly made available registers (*out* and *local*) in the new window are zero. The instruction syntax is:

`save` reg_{rs1},*reg_ or_ imm*,reg_{rd}

Additionally the instruction behaves as a normal ADD instruction. It takes the source operands from the previous window and the writes into the destination register in the new window. Typically the save instruction is used to generate a new stack pointer (denoted in program by %sp which is an alias to o6). This is done along with allocating new register window in one *atomic* operations. For example, upon execution of the instruction

```
save        %sp,-120,%sp
```

a new register window is allocated and the stack pointer is moved by 120 bytes. (The old stack pointer remains in the old windows)

The **RESTORE** instruction restores the register window saved by the last SAVE instruction executed by the current process. The *in* registers of the old window now becomes the *out* registers of the new window. The *in* and *local* registers of the new window retain their previous values.

3.2.4 Function Call and Parameter Passing

The **Call** instruction causes a control transfer to the desired subroutine. It also saves the value of PC, into *out* register o7. When the called routine executes the save instruction, the register o7 of the calling routine is known as *in* register i7 in the new window, which now holds the address to the call instruction.

The **Ret** instruction, like the call instruction is also synthetic instruction and is equivalent to

```
jmpl      %i7 + 8, %g0
```

It causes a register–indirect delayed control transfer to the specified address `[i7]+8` As i7 contains the address of the call instruction, the control will be transfered to the instruction immediately after delay slot of the call instruction (offset 8). Typically, after a ret statement, a restore instruction is kept in the delay slot, which restores the register window to that of the calling function.

Up to six parameters can be passed to a subroutine, by placing them in the *out* registers o0 to o5. Additional parameters are passed through memory stack. Thus in the called routine, after executing save instructions, parameters are available in registers i0 to i5. The stack pointer is implicitly passed in o6 which becomes the current procedure's frame pointer (i6).

A function can return several integer values using the *in* registers starting at i0 which are visible in the parent routine register window starting at at o0. In addition, SPARC also has several floating point registers which are used for passing/returning floating point values.

The *local* registers are used for *automatic* variables and other temporary values.

3.3 Code Instrumentation

In this section we shall discuss the details of instrumenting the various branch instructions, call and ret instruction.

3.3.1 Instrumentation of the Branch instruction

Supposing we have the following piece of assembly code:

```
          .
          .
          .
        cmp %l0,10
        ble LL1             ;branch to LL1 if l0 is less or equal to 10
        nop
        ...
          .
          .
          .
  LL1:  ...
```

Now we replace the `ble` instruction by a call to a special function called "foo". So the instrumented code will look like the following.

```
          .
          .
          .
        cmp %l0,10
        call foo,0          ;call function foo
        nop
        ...
          .
          .
  LL1:  ...
```

The corresponding foo function is shown in figure 3.1

Let us now analyze the instrumented code. The routine foo first executes a save instruction to get a new register window and moves the stack pointer by 120 bytes. The *rd* instruction saves condition code register (CCR) into local register l0. We have to do this because the succeeding instructions can change the value of CCR and thus change the condition under which the original branch would have been taken. The address of the original branch instruction is the address of `call foo,0` instruction and is available in register i7 of the current window. The original target address of the branch is LL1. The "myfprintf" routine is used to write data into the trace file. Using this function, we first write the source and target addresses of the original branch instruction. After this, we execute the original branch instruction. Note that, if the branch is not taken then foo should return to the instruction after the delay slot. For this, the fall through address is saved in the global variable "bretaddress".

Then the register window is restored; the CCR register is restored using *wr* instruction and the actual branch is executed. If the branch is not taken, it is noted in the trace file and the control should be transferred back to the correct instruction.

Finally the saved branch return address is written back to i7 and ret instruction is executed. In the delay slot the restore instruction is put.

3.3.2 Instrumentation of the Call Instruction

Call instructions are essentially jump and link instructions, upon execution of which the address to the call instruction is stored in the link register o7. Function calls can be nested. Therefore unlike the handling of branch instructions, the return address cannot be stored in a single global variable. We therefore use a different way of handling call instruction. We label the instruction following the delay slot instruction of a call instruction by a special *return label*. Finally the function call is replaced by a call to the foo function. In the foo function (figure 3.2), after the source and target addresses are noted, the actual function is called. Finally upon return from the original function to the foo function, an unconditional branch is made to the corresponding return label. There is one more issue involved. A function

```
.section        ".text"
        .align 4
        .global foo
        .type   foo,#function
        .proc   020
foo:
        !#PROLOGUE# 0
        save %sp,-120,%sp               ;save register window
        !#PROLOGUE# 1
        rd %ccr,%l0                     ;save ccr
        mov %i7,%o0                     ;branch source address
        sethi %hi(.LL1),%l1
        or %l1,%lo(.LL1),%l1
        mov %l1,%o1                     ;branch target address
        call myfprintf,0                ;write into trace file
        nop
        sethi %hi(bretaddr),%l1         ;save the fall-through instruction address
        st %i7,[%l1+%lo(bretaddr)]
        nop
        wr %l0,%g0,%ccr                 ;restore ccr
        restore                         ;restore register window
        ble .LL1                        ;take actual branch
        nop
        save %sp,-120,%sp
        mov 0,%o0
        mov 0,%o1
        call myfprintf,0                ;write 0 0 to trace file to denote
                                        ;branch not taken
        nop
        sethi %hi(bretaddr),%l0
        ld [%l0+%lo(bretaddr)],%i7
        sethi %hi(bretaddr),%l0
        st %g0,[%l0+%lo(bretaddr)]      ;restore fall through address to i7
        ret                             ;fall through
        restore
.LLfoo:
        .size   foo,.LLfoo-foo
```

Figure 3.1: The foo function replacing a branch instruction

call introduces two control transfers. Namely, from the caller to callee routine and back. To store the second control transfer information into the trace file, we need to know, from where in the called subroutine, the control was finally returned. For this, we use another global variable called "retaddr". The instrumented code for ret instruction (see 3.3.3) stores the appropriate instruction address into retaddr. If however, it is a call to a library routine, we cannot know from where the control was returned (see 3.5) and thus a zero is stored in place of retaddr.

Supposing we have assembly instructions as follows

```
        .
        .
        .
        call fun,0              ;call function fun
        nop
        ...
        .
        .
        .
```

After instrumentation it is changed to the following code

```
        .
        .
        .
        call foo,0              ;call function foo
        nop
.global  .LLret                 ;special label to denote return instruction
LLret:
         ...
          .
          .
```

```
.section        ".text"
        .align 4
        .global foo
        .type   foo,#function
        .proc   020
foo:
        !#PROLOGUE# 0
        save %sp,-120,%sp               ;save register window
        !#PROLOGUE# 1
        mov %i7,%o0                     ;source address
        sethi %hi(fun),%l1
        or %l1,%lo(fun),%l1
        mov %l1,%o1                     ;target address
        call myfprintf,0                ;write into trace file
        nop
        restore
        call fun,0                      ;actual function call
        nop
        save %sp,-120,%sp
        sethi %hi(.LLret),%l0           ;return address
        or %l0,%lo(.LLret),%o1
        sethi %hi(retaddr),%l0          ;return from where?
        ld [%l0+%lo(retaddr)],%l1
        cmp %l1,0                       ;if retaddr=0 then it is not known
                                        ;from where control is returned
        bne .LLfoor1
        mov 0,%o0
.LLfoor1:
        mov %l1,%o0
        call myfprintf,0                ;write into trace file
        nop
        sethi %hi(retaddr),%l0          ;reset retaddr
        st %g0,[%l0+%lo(retaddr)]
        b .LLret                        ;unconditional branch to return address
        restore
.LLfoo:
        .size   foo,.LLfoo-foo
```

Figure 3.2: The foo function replacing a call instruction

3.3.3 Instrumentation of the Return Instruction

The foo function replacing a ret instruction is shown in Figure 3.3. The function simply stores the address of the ret statement (stored in o7) into the global variable "retaddr". Note that, here no new register window is saved, and it uses the register window of the routine from which it was called.

Therefore, if we have assembly instructions as follows

```
   .
   .
   .
ret                  ;call function fun
restore
...
   .
   .
   .
```

instrumentation will change the code to

```
   .
   .
   .
call foo,0            ;call function foo
nop                   ;restore is replaced by nop
 ...
   .
   .
   .
```

3.4 Other Details

The *myfprintf* routine accepts two addresses as arguments and stores them into the trace file. The routine uses a buffer to store the addresses temporarily. When the

buffer is full, it is emptied into the trace file. At the time of program termination, the remaining part of the buffer was flushed to the trace file.

Since for each of the control transfer instructions, we need a separate "foo" function, we have to give them unique names. We used the following naming convention for the foo functions: <fname>foo<num>, where fname is the name of the `.s` and num is a unique sequence number. For example the foo function for the 10^{th} control transfer statement in a file a.s will have the name `afoo10`.

After all the foo routines are generated they are put together into a file called foo.s. All the instrumented assembly programs are stored in files with name T<filename>.s , where filename was the name of the original .s file. Since there can be more than one such instrumented .s files, and that foo.s has code that uses labels within them, all the labels and function names are made *global*.

3.5 Drawbacks

There are certain flaws in this address trace generator. First of all, to preserve the address space of the code to be instrumented, we could not add any extra instruction to the individual .s files. All we could do is to replace a control transfer statement with a function call. Now consider the following branch statement:

```
ble LL1
```

Here, in order to write the corresponding "foo" function, we need the branch instruction (ble) and the target label (LL1). We cannot pass them as parameters to foo, as it would require several instructions to be inserted into the original program. We have similar problems with the call instructions too. Therefore, several foo functions are generated, one for each control transfer instruction. As a result the generated foo.s file is large in size, and after assembling, the generated binary code is huge.

In our approach, all register indirect jumps cannot be instrumented. Since in the "foo" function, we have to save a new register window, instrumentation is not possible if the argument register is a *local* or a *in* register, as these are not visible in the register window of the foo function. Only if it is an *out* register, we change it to

a *in* register in the instrumented code. An example will clarify the issue. Suppose we have to instrument the instruction.

```
jmp   %o0
```

The corresponding foo function is shown in figure 3.4.

Obviously an instruction like `jmp %i7` cannot be instrumented by our program.

There is another shortcoming in our approach. Since we are instrumenting at the assembly level we cannot trace the execution of the different library routines. For this reason, the trace file only holds the jump from the user code to the library routine, but not the trace of the library routine itself. Since we do not know from where is the library routine the control was returned, the corresponding entry in the trace file returned as a zero.

```
.section        ".text"
        .align 4
        .global foo
        .type   foo,#function
        .proc   020
foo:
        !#PROLOGUE# 0
        mov %o7,%l1
        sethi %hi(retaddr),%l0
        st %l1,[%l0+%lo(retaddr)]          ;store o7 into retaddr
        ret                                ;return control
        restore
.LLfoo:
        .size   foo,.LLfoo-foo
```

Figure 3.3: The foo function replacing a ret instruction

```
foo:
        !#PROLOGUE# 0
        save %sp,-120,%sp                       ;new register window
        !#PROLOGUE# 1
        rd %ccr,%l0
        mov %i7,%o0
        mov %i0,%o1                             ;The o0 register becomes i0
                                                ;pass i0 as parameter to myfprintf
        call myfprintf,0
        nop
        sethi %hi(bretaddr),%l1
        st %i7,[%l1+%lo(bretaddr)]
        nop
        wr %l0,%g0,%ccr
        restore
        jmp %o0                                 ;the original jmp instruction
        nop
        save %sp,-120,%sp
        mov %o0,%i7
        mov 0,%o0
        mov 0,%o1
        call myfprintf,0
        nop
        sethi %hi(bretaddr),%l0
        ld [%l0+%lo(bretaddr)],%i7
        sethi %hi(bretaddr),%l0
        st %g0,[%l0+%lo(bretaddr)]
        ret
        restore
```

Figure 3.4: foo function for *jmp %o0*

Chapter 4

Next Cacheline Prediction

In this chapter we discuss in detail the new *Next Cacheline Prediction* scheme that is proposed in this thesis. As mentioned in the first chapter, the factors that affect the performance of a multi-issue processor are the I-cache access time and prediction of the next set of instructions to be fetched into the pipeline. Our scheme predicts the location of the cacheline, i.e. the set and way, where the next instruction(s) reside. Since, in this scheme, we need not compare the tag of an instruction address with the tag stored in cache blocks, in a particular set, it makes cache access faster and reduces the I-cache power consumption. In the remaining part of the chapter, we shall address the I-cache simply as cache.

4.1 The Design

The main feature of this scheme is a **Set Way Prediction Table** which is a small cache memory whose entries contain the following information.

current set, current way $\rightarrow$ next set, next way

As the instructions are fetched from an address, an entry in the table is searched for the next set and next way. The current set and way for this entry should be same as the set and way for the current instructions. The next set and way is the predicted set and way in the cache from where the next instruction is to be fetched. Whenever an instruction is fetched, from a given set, way; the table is

looked up for a corresponding entry. If the current set and way is found in the table, then the next instruction is fetched from the predicted set and way. If no entry is found corresponding to the current set and way, then the default prediction is used. The default prediction is based on the spatial locality principle. instructions that are located in neighboring positions in memory, when fetched to cache, will tend to fill up the cache column by column. Hence, if the current instruction comes from set s, way w; then the next instruction is likely to be in set (s+1) and way w. This is the default prediction of this scheme.

4.1.1 Working Principle

Initially the prediction buffer is empty. The cache is accessed in the traditional way to translate the address in the PC to correct cache address (set and way) to read the instruction. After this, the current set and way is used to predict the next set and way and the instruction is speculatively read from the predicted cacheline. If the prediction is not correct, then the speculatively fetched instructions are killed and the prediction table is updated. The correct PC is then used to access the cache in traditional way and fetch the correct instruction.

4.1.2 Cold Start

At the beginning of the program execution, most predicted cache addresses correspond to invalid cachelines. Accessing these cachelines result in cache miss. In this case, the address of the instruction is constructed using the address of the previous instruction read, and the instruction is read from this address.

Consider a 2 way associative cache of size 1 Kilobytes (128 sets, each with 2 blocks), 32 bit wide cachelines. The instruction addresses are 20 bit wide. Assume that, the last cacheline read was from set `0x44` and way `0` of the cache. The corresponding tag was `0x83`. As there are no entries corresponding to this set/way in the prediction table, the next set and way are predicted as `0x45` and `0`. As this cacheline is invalid, the instruction has to be read from memory. From the tag and index of the previously fetched instruction we can easily reconstruct the address of

this fetched instruction to be `0x10710`. Assuming instructions are 4 bytes long, the address of the next instruction is `0x10714` (which indeed belongs to set `0x45`). This instruction is then read from the memory and placed in way 0 of set `0x45`.

Note that, once the entire cache is filled up by some instructions, all the speculative cache reads will result in a hit. However, as we shall see later, there is no guarantee that all these speculative reads fetch the desired instruction.

4.1.3 A Modified Design

As a modification of this scheme, we can use the Branch Target Buffer(BTB) as an additional source for prediction the address to the next instruction to be fetched. As in the earlier scheme, the current set and way is used to predict a new pair of set and way. Simultaneously, the address of the current instruction is used to refer to the BTB as usual, and predict the PC of the next instruction. The tag and index of the predicted PC are compared with the tag and index of the speculatively fetched instruction. If the tag and index of the fetched instruction do not match that of the predicted PC, the speculatively fetched instruction is killed, and the cache is accessed in the traditional way, using the predicted PC. In such a case, the set way prediction table is also updated. It is seen that the next cacheline scheme gives better performance when coupled with the BTB (see Chapter 2).

Using the next cacheline prediction scheme, the focus shifts from address prediction, to predicting the exact location in the cache where the instruction resides. A cacheline replacement (due to cache miss) can have serious implications on the performance.

4.2 Different Performance Issues

As discussed in Chapter 2, in a processor using a branch prediction table, the outcome of a instruction branch is predicted. Similarly in a Branch Target Buffer based prediction, the target address of a branch instruction is predicted. Essentially, in these branch prediction schemes, the address to the next instruction is predicted. In our scheme, we do not predict the next memory address from where the instruction

is to be fetched, but rather predict the position in cache, where the next instruction is likely to be present is predicted. Thus in our scheme there is a shift of focus from instruction address, to the actual location of the instruction in the cache.

It is this distinction, which affects many performance issues of this scheme. If a BTB based scheme makes a prediction of control transfer from an instruction address PC1 to another instruction address PC2, then it signifies that there *is* a control transfer instruction at address PC1. This control transfer may not always take place (as in the case of a conditional branch). However the control transfer instruction will be present throughout the lifetime of the program. So an entry in the BTB always convey some *relevant* piece of information. Now, consider the set-way prediction table of our scheme. Suppose, due to a cache miss, the instruction in a particular cacheline of the cache gets replaced, then all information stored about that set and way of the cache loses all relevance. This unfortunately gives rise to certain drawbacks in this scheme. Now, we discuss the various problems in such a system.

4.2.1 Problem 1

The most common problem occurs when a instruction that is the source or target of a control transfer instruction gets replaced due to cache replacement policy. Let us explain it with an example (fig. 4.1). Suppose there is a jump from address `0x106d4` to `0x10700`. Suppose the first instruction is in way 0 of set 53 and the second instruction is in way 0 of set 0. The set way prediction table will contain an entry (0,53) $\rightarrow$ (0,0), corresponding to this jump. Now, suppose, the instruction `0x10700` gets replaced due to cache miss. Now, the corresponding entry in the prediction table loses relevance. If at some point later the control reaches at instruction `0x106d4`, then the next set and way will be wrongly predicted as (0,0) where the instruction has been replaced. One solution to this problem could be to invalidate all the entries in the set way prediction table corresponding to a set-way which just got replaced. But, even then this will not substantially improve the situation. Supposing, the entry in the prediction table is invalidated. When control will be transferred to `0x106d4` at set 53, way 0, the prediction table will be looked up. When no corresponding

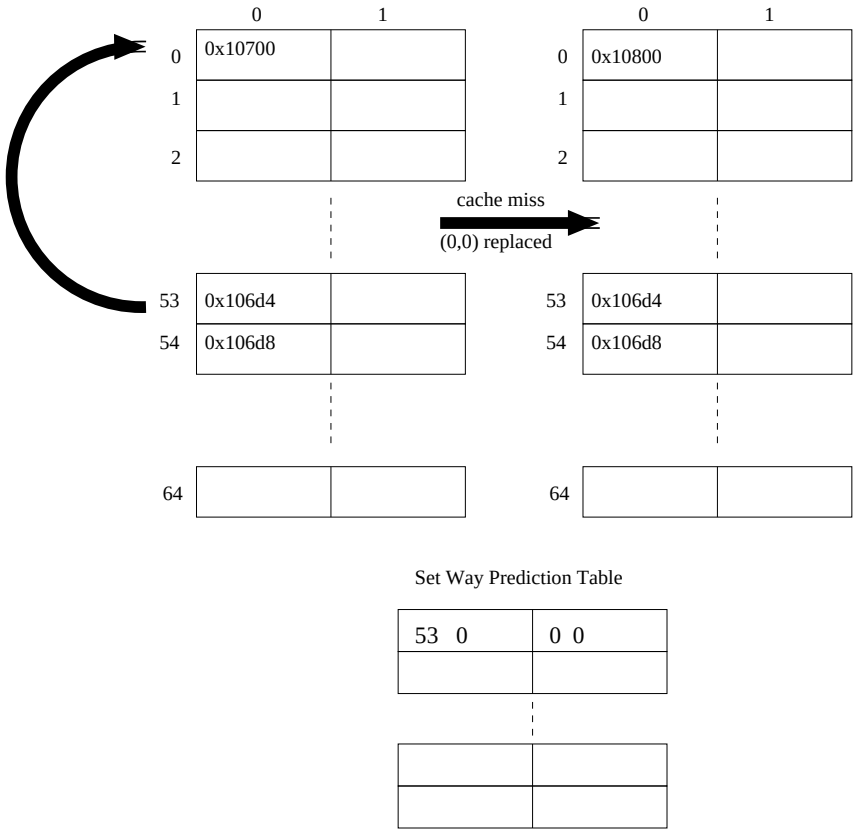


Figure 4.1: Wrong Prediction due to Cache Replacement

entry will be found, it will predict by default to fetch instruction from set 54, way 0, which in this case contains `0x106d8`. Now suppose the branch instruction at `0x106d4` actually takes the branch. Then also a misprediction occurs. In both the cases the performance penalty is the same. However if the branch is *not taken* after the cacheline gets replaced, then there will be some advantage.

4.2.2 Problem 2

This problem is explained with an example. Supposing, the instructions `0x106d4`, `0x106d8` and `0x106dc` are stored in way 0 of the adjacent sets 53, 54 and 55 respectively (fig. 4.2). Initially there are no entries corresponding to any of these

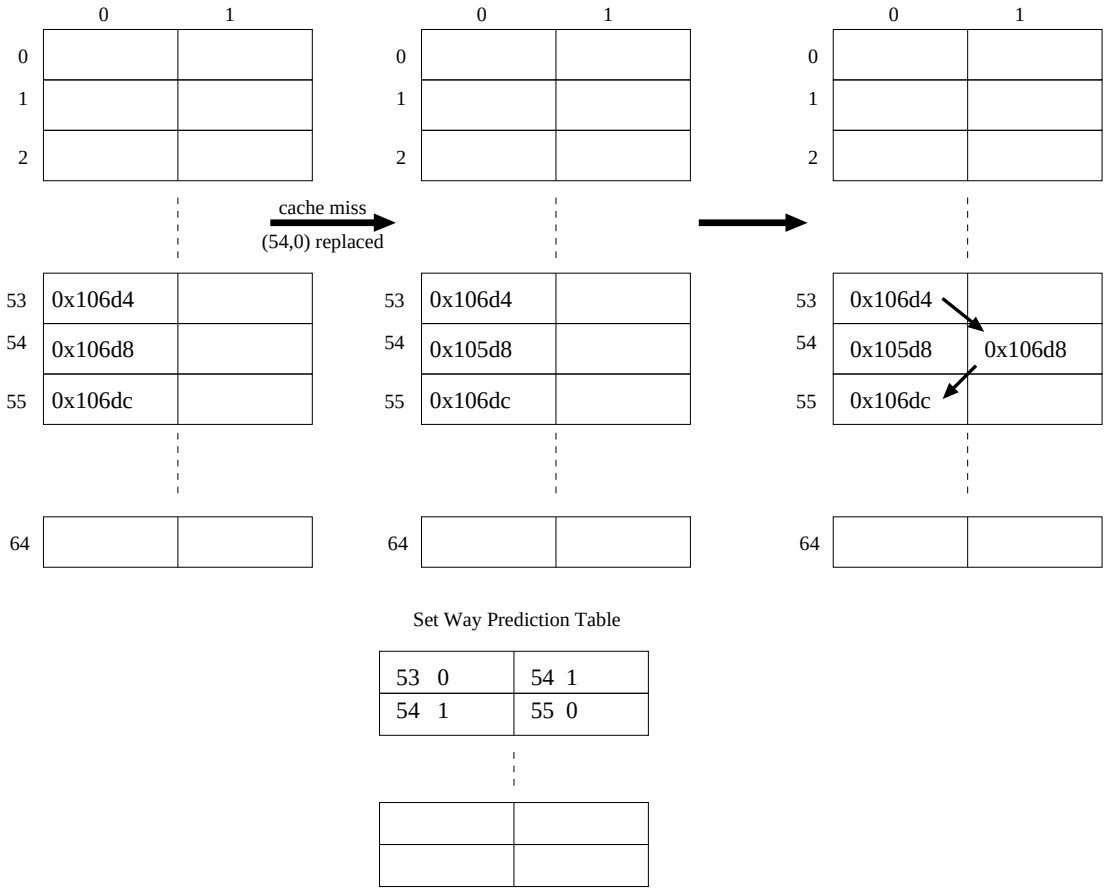


Figure 4.2: Pseudo Cache Miss

instructions in the prediction table. Now suppose the instruction `0x106d8` gets replaced by instruction `0x105d8`. Then when the control is transferred to `0x106d4` at set 53, way 0, the prediction scheme predicts the next instruction to be fetched from set 54, way 0. It is not a cache miss, but the wrong instruction is fetched. A normal cache read will result in a cache miss. After the cache penalty the correct instruction will be read from the memory and placed in the cache at way 1 of set 54. The mistake will be discovered after a few clock cycles. By that time some other wrong instructions have been fetched and all these instructions have to be killed. The subsequent next prediction will try to fetch an instruction from way 1 of set 55. This prediction will meet a similar fate, since the expected instruction is in way 0

of set 55. So again there will be a penalty associated with it, although the desired instruction is *already* present in the cache. This sort of *pseudo cache miss* degrade the performance of the system.

As an added penalty, the set way prediction table will be updated with two entries (53,0) $\rightarrow$ (54,1) and (54,1) $\rightarrow$ (55,0). These two entries signify that after set 53, way 0 is accessed, the next instruction is to be fetched from set 54, way 1 and then from set 55 way 0. Although there is no jump from `0x106d4` to `0x106d8` or from `0x106d8` to `0x106dc`, the entries corresponding to these sequential flow of control, will remain in the prediction table. Thus the prediction table can have entries, which actually depicts sequential flow of control, rather than a jump.

4.2.3 Branch Target Buffer and Cacheline Prediction

The penalties associated with the problems mentioned above, can be reduced if the mispredictions are detected earlier. We incorporated BTB based prediction also in our scheme (see 4.1.3) assuming that BTB gives a more accurate prediction for the addresses. Experimental results confirms it. Note that, even in the modified scheme, the above mentioned problems will not be eliminated, since the primary source of prediction is still the set way prediction buffer. But, in the modified scheme, whenever we speculatively fetch a wrong instruction, the following comparison with the predicted PC, will result in a mismatch and the error will be detected at a much earlier stage of the pipeline, thus reducing the penalty.

4.2.4 Problem 3

Association of BTB along with the cache prediction table brings in some more problems.

As we saw that the set way prediction has to store information about instructions which are not of control transfer type. Since the set way prediction table has a limited number of entries, entries in the table can get replaced, which may be entries for control transfer instructions. However the BTB may still have the information about the branch instruction.

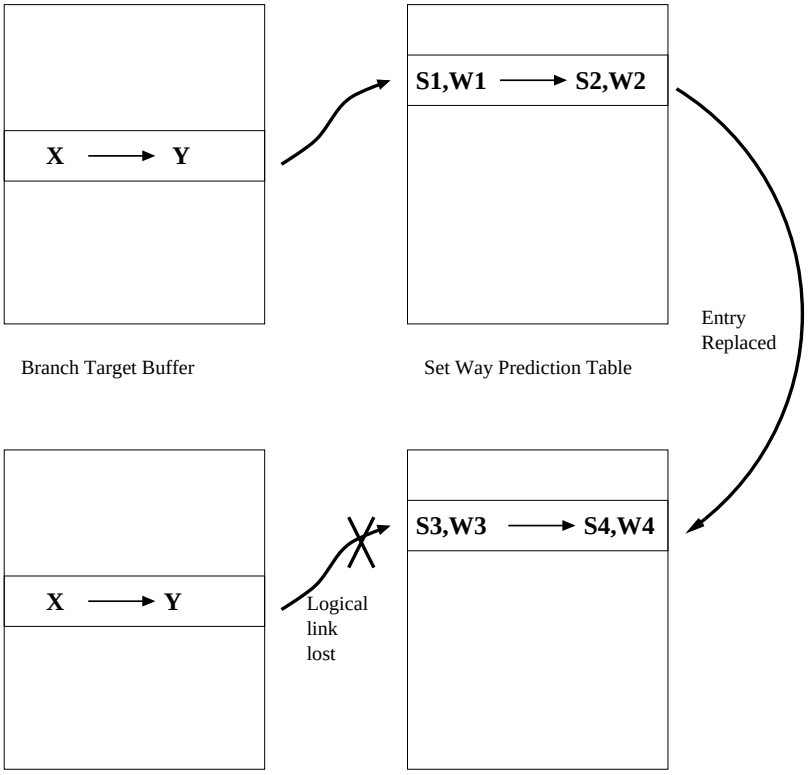


Figure 4.3: Mismatch Due to Decoupling of BTB and Set Way Prediction Table: Case 1

Sometimes, such inconsistencies between the two tables can lead to degradation in performance as explained in the following example (fig 4.3). Suppose we have a jump from address X to address Y. The instructions were stored in the cache at (s_1,w_1) and (s_2,w_2) respectively. The BTB contains an entry X $\rightarrow$ Y. Similarly the set way prediction table contains the entry (s_1 w_1) $\rightarrow$ (s_2 w_2). Now assume that the entry (s_1 w_1) $\rightarrow$ (s_2 w_2) is replaced by some other unrelated entry (s_3 w_3) $\rightarrow$ (s_4 w_4). Now suppose the control is transferred to the instruction X, stored at set s_1 way w_1. The set way prediction table has no entry for this cacheline and will mispredict the next instruction to be fetched from set ($s_1 + 1$ way w_1), The BTB based prediction provides the next predicted PC to be Y, which is a correct prediction if the branch is taken. A misprediction is detected and all the fetched instructions are killed. After this the correct instruction will be fetched, but after a considerable penalty.

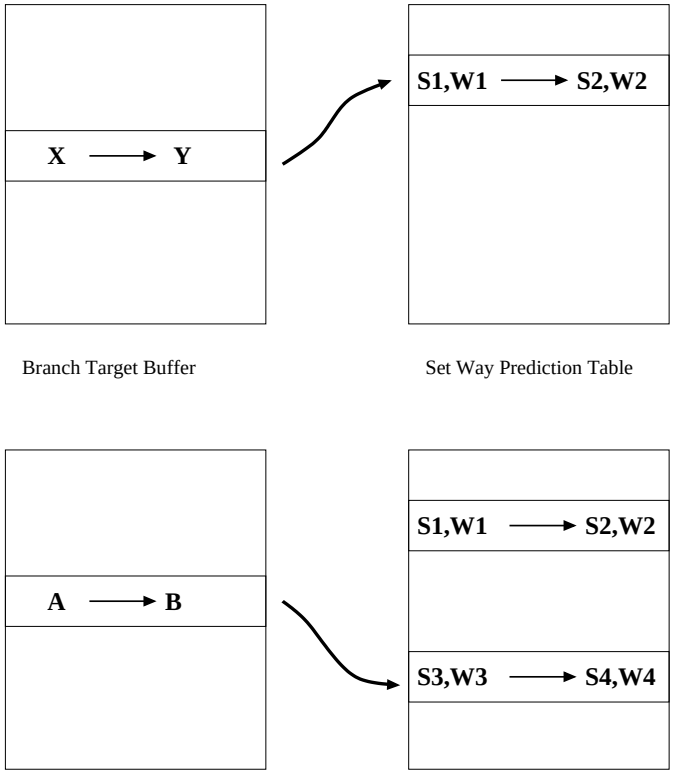


Figure 4.4: Mismatch Due to Decoupling of BTB and Set Way Prediction Table: Case 2

Just the reverse case can happen (see fig 4.4). Instead of the set way prediction entry being replaced, it may be the case that the entry X → Y in BTB got replaced by another entry A → B. Now, if the control goes to the instruction X, the set way prediction table will predict the next instruction to be in set s_2 way w_2, which is correct prediction. Later, when the BTB is consulted , and no entry is found for address X, it will predict the next PC to be X+4 (assume that the instructions are all 4 bytes long), which is a misprediction. A mismatch will be detected and the correctly fetched instruction will be killed. A wrong instruction will be fetched as a result of BTB misprediction and the set way prediction table will be updated as a result of which the correct entry of (s_1 w_1) → (s_2 w_2) will be removed. This mistake will be detected at the later stage in the pipeline and the correct instruction will be fetched, but after a considerable penalty. Note that although, it was a misprediction

of the BTB and there would have been a penalty even if we had BTB only; the penalty would have been lesser than the penalty in this case.

4.3 Advantages

In this scheme, whenever the instruction cache is accessed, associative tag comparison need not be done. This makes the instruction fetch operation faster. Since we are bypassing the tag comparison step, we are reducing the length of the fetch pipeline. Essentially, there are two parallel instruction fetch paths. The shorter one is taken most of the time. The longer traditional path is taken, only when we detect a misprediction.

As observed in [4], eliminating tag comparison reduces power consumption, our scheme has the added advantage of saving power.

The performace of the scheme for predicting the next instruction is comparable to the other existing schemes. The details are given in Chapter 6.

Chapter 5

Simulator

A simulator was built to simulate the proposed cacheline prediction scheme, and compare its performance with other prediction schemes. The simulator is highly configurable. The simulator models *only the fetch stage* of a processor and is highly configurable. It simulates instruction fetching mechanism and prediction of the next instruction of a processor.

5.1 Overview of Simulator

The inputs to the simulator are the trace file (chapter 3) for the program to be tested and the configuration file which contains the values of the different parameters used to model the architecture.

It can simulate three different type of prediction schemes. It can simulate the BTB based scheme, the BPT based scheme and the proposed next cacheline prediction scheme. These schemes can be simulated independently or in combination with another scheme.

It models the instruction cache and various events like cache miss associated with it. It also models the various prediction buffers. It can simulate stalls in pipeline and flushing of the pipeline.

The simulator does not take the instructions of the program into consideration and does not execute the program. It only simulates the instruction fetch and

prediction mechanisms.

The address of the first instruction of the program under test is supplied to the simulator at the beginning of the simulation The simulator compares the predicted instruction address with the trace file. If the predicted address does not match with the trace file, then a misprediction occurs, and the simulator updates its tables and the correct instruction address is taken from the trace file. The simulation continues until the entire trace file is read.

It records the number of mispredictions occurred, total time taken in clock cycles etc. It also models the instruction cache and simulates various events like cache miss associated with it. These statistical data is used to compare the performance of the various instruction address prediction schemes.

5.2 Configuration File

The simulator is designed to be as much configurable as follows. It uses a configuration file to read in the various parameters, used to model the processor architecture and the instruction fetch prediction schemes. An example configuration file is shown in fig 5.2. The syntax of the lines in the configuration file is *parameter: value* In the configuration file, a line beginning with a hash symbol is ignored and can be used as a comment line.

5.3 Event Queue Modeling

To simulate the pipeline, we used *Event Queue Modeling* technique. The different stages of the pipeline can be viewed to perform some operations as responses to certain *events*. For example, cache miss is an event, and as a response to it, the instruction is fetched from the next level of memory. An event can trigger one or more events as a consequence. Depending on outcome of some operation, the consequent events can be different. For example, as a response to the cache read event, the I-cache is accessed. If the access results in a hit, some events are generated to dispatch the fetched instruction and initialize fetch of next instruction. However,

```
# configuration file
# Abbreviations used
# BTB Branch Target Buffer
# BPT Branch Prediction Table

# length of instruction address in bits
address: 24

# I-cache size in KiloBytes
size: 32

# cache associativity
assoc: 2

# cacheline size in bytes; a cache line can hold one or more words
line: 8

# cache word size in bytes
word: 4

# cache replacement policy: 0 FIFO 1 RANDOM 2 LRU
replace: 0

# cycle after fetch when BTB predicts next PC
btbd: 1

# cycle after fetch when BPT predicts outcome of branch instruction
bptd: 2

# is set way prediction present
swpt: 0

# is branch prediction table present
isbpt: 0

# is branch target buffer present
isbtb: 1
```

Figure 5.1: configuration file

```
# number of entries in BPT
bptsize: 1024

# history bits in BPT : number of branches whose history is recorded
m: 2

# prediction bits per entry in the BPT
n: 2

# cycles after cache access when outcome of branch is known
branch: 5

# Instruction Buffer size (number of instructions)
ibufsize: 32

# set way prediction table size (number of entries)
swptsize: 256

# BTB size (number of entries)
pcptsize: 256

# miss penalty (in cycles)
missp: 4

# instruction issue rate
irate: 2
```

Figure 5.2: configuration file continued

a miss generates the cache miss event. Some events like the cache miss can generate stalls in the pipeline. Some other events (like a branch misprediction) can lead to killing of instructions, which means all the existing events are killed.

Thus the pipeline is modeled as a series of events or an *event queue*. This event queue is implemented as a linked list in our implementation. The nodes of this linked list are the actual events, and contain the following fields.

```
/* structure definitions for a node in the event queue */
typedef struct entag {
   int etype;           /* event type */
   int time;            /* time at which the event is scheduled to occur */
   int param1;          /* parameter */
   int param2;          /* parameter */
   int param3;          /* parameter */
   struct entag *next;  /* link to the next event */
}enode;
```

The `etype` field distinguishes one event from other. The different event types used in the simulator are as follows.

```
loadPC                      /* load PC with new value */
predictNextPC               /* predict next PC */
predictSetWay               /* predict next set and way of cache
                               (cacheline prediction)*/
compareTag                  /* compare address of speculatively
                               read instruction with predicted PC */
predictBranch               /* Branch Prediction Table predicts branch */
speculativeCacheRead        /* read instruction on basis from
                               predicted cacheline */
cacheMiss                   /* cache miss */
```

```
chkBranch                       /* Resolve branch instructions,
                                   check whether next instruction
                                   matches branch target */
updateSetWayPredictionTable     /* update the Set Way (next cacheline)
                                   Prediction Table */
updatePCPredictionTable         /* update the BTB */
selectWay                       /* select the cache block in a set */
readWay                         /* read from the way */
```

Every event is associated with a time at which they occur. This time is stored in the `time` field. The simulator executes in a loop. At the beginning of each iteration, events that are supposed to be handled at the current time, are taken out from the event queue, and handled sequentially. The handling of an event often depends on some external factors. These can be passed to the event handlers through three parameters. For example, the update Branch Target Buffer has to create a new entry or delete an entry depending on the outcome of the previous branch instruction. This information can be passed by the `param` fields.

After events are handled, they may generate some more events, which are then placed at the appropriate place in the event queue in sorted order. Stall in a pipeline can be modeled by delaying the time at which the events in the queue are scheduled to occur. Flushing the pipeline is equivalent to deleting all the future events in the queue at that time.

5.4 Modeling

The simulator models the instruction cache, the various prediction tables etc. The implementations of these are discussed in brief.

5.4.1 Instruction Cache

The Instruction Cache is implemented as a two dimensional array of cache blocks. The number of columns in the array is equal to the cache associativity. The number of rows in the array is equal to the number of sets in the cache. The size and other parameters of the cache are configurable (as given in section 5.2). The contents of the cache block, i.e. the instruction itself was not stored since it would not serve any purpose to us. Only the tag for each cache block was stored along with the valid bit information. The structure definition is given below.

```
typedef struct {
  int tag;          /* cache line tag */
  int valid;        /* whether cache entry is valid */
  int age;          /* used for cache replacement */
} ictype;
```

5.4.2 Prediction Tables

The different prediction buffers are also caches and are implemented as array of structures.

The Set Way Prediction Buffer or the next cacheline prediction buffer is a fully associative cache. Each entry contains four fields. Current cache set and way (the block within the cache set) and predicted cache set and way. Each entry has a count associated with it, which is incremented every time the buffer is accessed. During replacement, the entry with maximum count (i.e. the oldest entry) is replaced and the count is reset.

```
struct {
  int set,way;    /* current */
  int nset,nway;   /* next */
  int count;  /* for replacing */
}
```

The Branch Target Buffer (BTB) is also modeled as a fully associative cache and implemented as an array of the following structure.

```
struct {
  int PC;
  int nPC;
  int count; /* for replacing */
}
```

The Branch Prediction Table (BPT) is modeled as a direct mapped cache. Each entry has a tag (to prevent aliasing), and a set of prediction bits, depending on the depth of history of other branches maintained. The cache replacement policy is FIFO. In our implementation we also keep a field called PC per entry which gives the predicted branch target. In reality, the BPT does not predict branch target, but by the time it is accessed, the branch target of the branch instruction is already resolved.

```
/* Branch Prediction Table: Directly Mapped Cache */
struct {
  int tag;
  int *pbits;          /* pointer to set of prediction bits */
  int PC;              /* target PC */
}
```

5.4.3 Instruction Address Queue

In processor employing some kind of instruction address prediction, the validity of a prediction is not known immediately after the instruction is fetched speculatively. After a branch instruction is fully resolved, the target instruction address is compared with the address of the next instruction fetched speculatively. If the instrcution is not of control transfer type, the address of the next instruction should be the address of the current instruction plus the fetch offset. To model this, in our simulator too, the predicted instruction addresses are not compared with the trace file immediately. Instead, they are stored in a queue and after a number of cycle (as specified in the configuration) the address in the front of the queue is compared with the trace file. If no misprediction is detected, they are dequeued. Otherwise the event queue is flushed, the queue storing instruction address is emptied and the prediction tables are updated, and the correct instructions are fetched. The queue is implemented using circular array.

Chapter 6

Experimental Results

The simulator was run on address trace of a few benchmark programs. Table 6.1 shows the benchmark programs tested and their brief description

All the Benchmark programs are written in C.

Some abbreviations used in this chapter are:

- BTB – Branch Target Buffer

- SWPT – Set Way Prediction Table, i.e. next cacheline prediction table

- BPT – Branch Prediction Table

Table 6.2 gives the total number of instructions in the different benchmark programs that we tested.

Compress	SPEC 95 Integer Benchmark data compression
Perl Interpreter	SPEC 95 Integer Benchmark program
Banner	Simple program to print banners on printer
Lisp Interpreter	CPU intensive SPEC 95 Integer Benchmark program
Whetstone	Netlib Double Precision Benchmark program
Matmul	Matrix Multiplication Program

Table 6.1: Benchmark Programs

Benchmark	Number of Instructions
Compress	5831442
Perl Interpreter	1682454
Banner	3806932
Lisp Interpreter	458974
Whetstone	1393724
Matmul	1808794

Table 6.2: Number of Instructions in different Benchmark Programs

6.1 Architecture Configuration

In this chapter we shall use the terms next cacheline prediction scheme and Set Way Prediction Table based scheme interchangeably. The benchmark was tested for the following prediction schemes:

1. **BTB and BPT:** Branch Target Buffer based prediction along with Branch Prediction Table
2. **BTB:** Branch Target Buffer
3. **BTB and SWPT:** Branch Target Buffer based prediction along with Set Way Prediction table based scheme (next cacheline prediction scheme)
4. **SWPT:** Set Way Prediction Table (next cacheline prediction scheme)

In all the experimental results shown later in this chapter we have used the following architecture configuration. The size of BPT was kept at 1024 entries. Both the BTB and SWPT were tested with 32 and 256 entries. The benchmark programs were tested for 16 KB and 32 KB 2-way set associative instruction caches, with FIFO replacement policy. The cache miss penalty was kept at 4 cycles. The instruction issue rate was 2 instructions/cycle. Branch outcome is taken to be resolved after 5 cycles of instruction fetching.

6.2 experimental Results

The following figures show the total time (in clock cycles) needed by the various benchmark programs for the various cache and prediction buffer configurations. The figures show our scheme to give comparable results with other schemes for most benchmark programs except **Perl Interpreter** and **Lisp Interpreter**. The performance improves drastically by increasing the prediction table size.

6.3 Figures

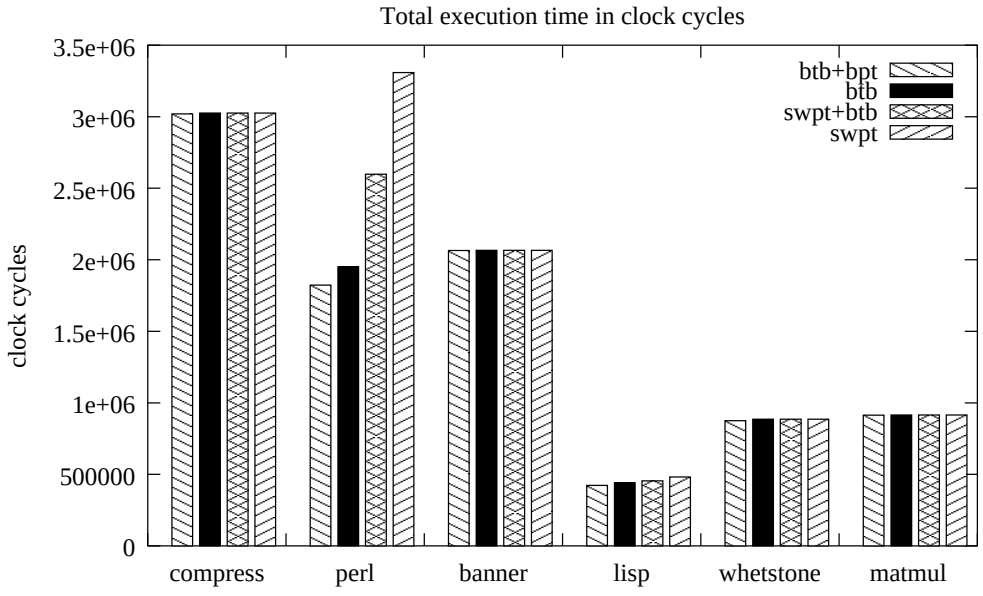


Figure 6.1: Clock cycles taken for executinmg different benchmark programs under different prediction schemes. I-Cache: 16 KB BTB: 32 entries SWPT: 32 entries BPT: 1024 entries

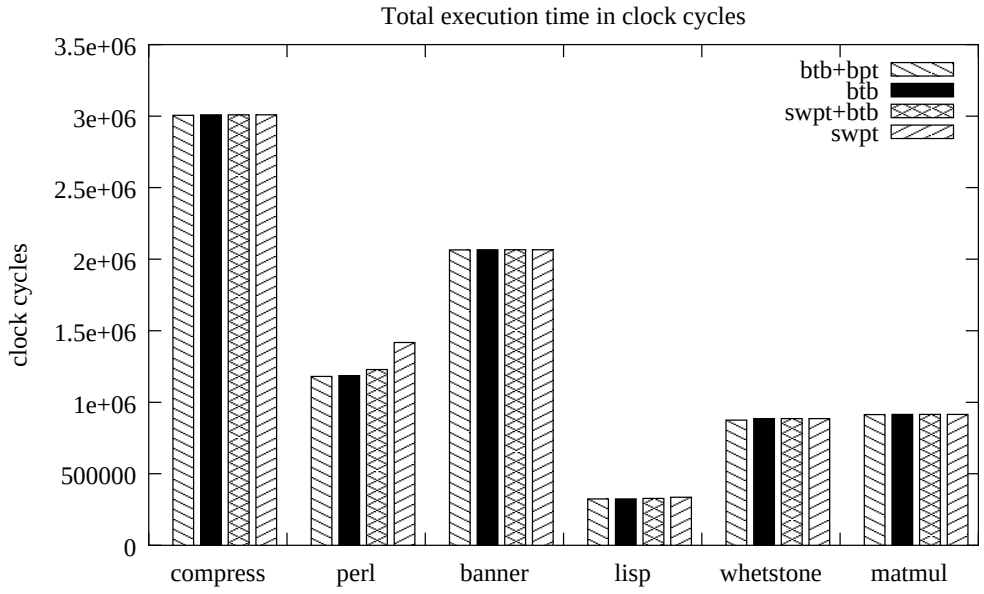


Figure 6.2: I-Cache: 16 KB BTB: 256 entries SWPT: 256 entries BPT: 1024 entries

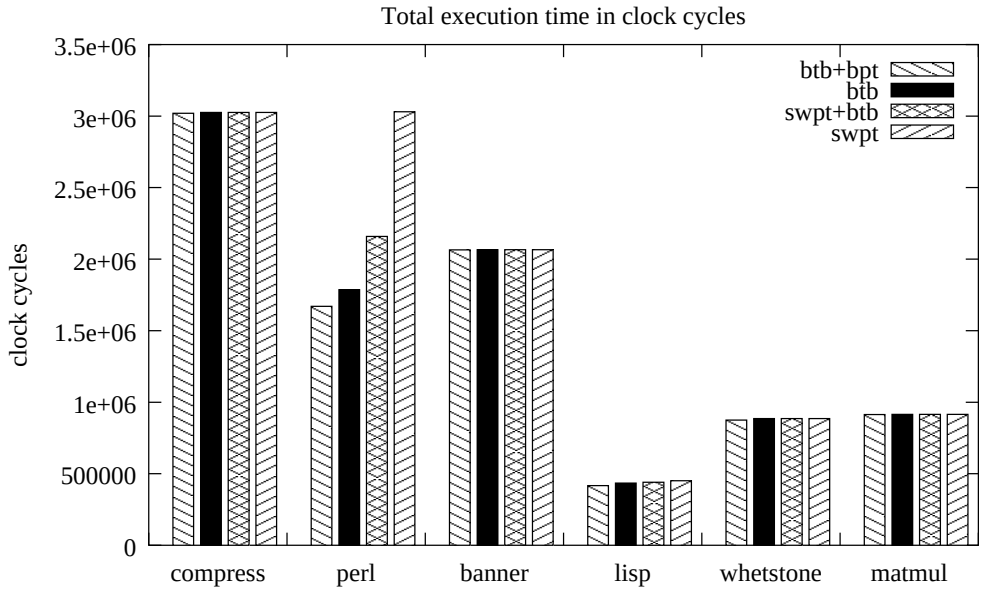


Figure 6.3: I-Cache: 32 KB BTB: 32 entries SWPT: 32 entries BPT: 1024 entries

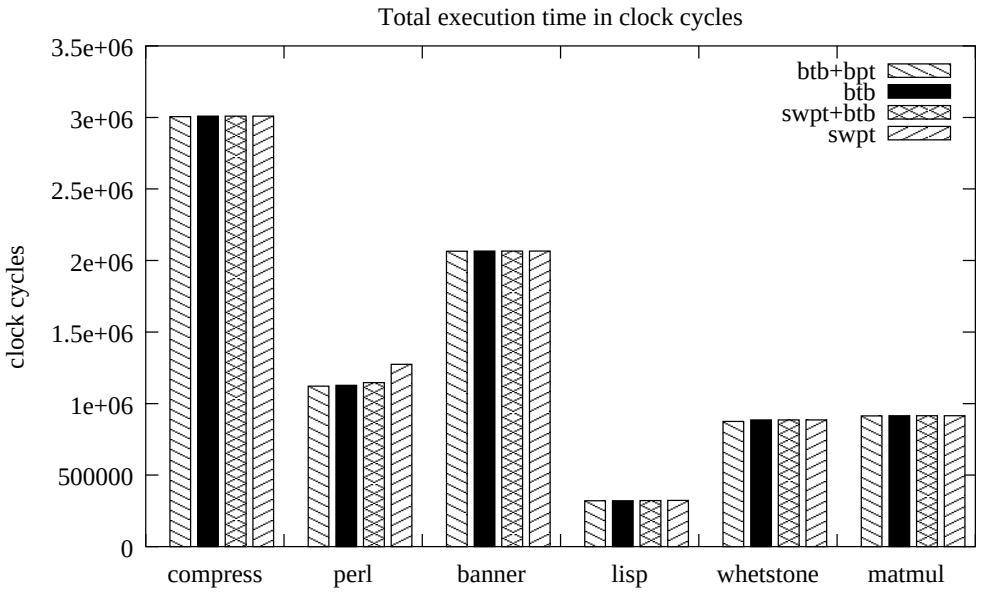


Figure 6.4: I-Cache: 32 KB BTB: 256 entries SWPT: 256 entries BPT: 1024 entries

In the combined Set Way Prediction Table and BTB based scheme, whenever the address of the instruction fetched on the basis of cacheline prediction does not match with the instruction address predicted by BTB, we say an instruction address *mismatch* to have occurred. In the next two figures, we have shown the variation of instruction address mismatches with change in associativity of the Instruction Cache. It is observed that the number of mismatches increases drastically with the increase in associativity. We have shown the bar charts for the benchmark programs **Perl Interpreter** and **Lisp Interpreter**. This observation can be related to the problem discussed in 4.2.2

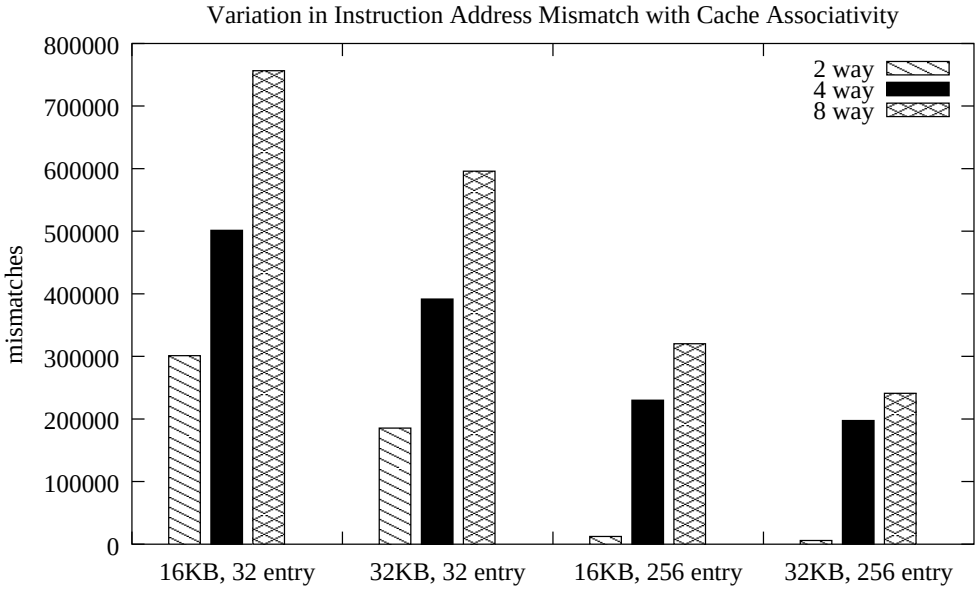


Figure 6.5: Variation in mismatch with cache associativity for Perl Interpreter

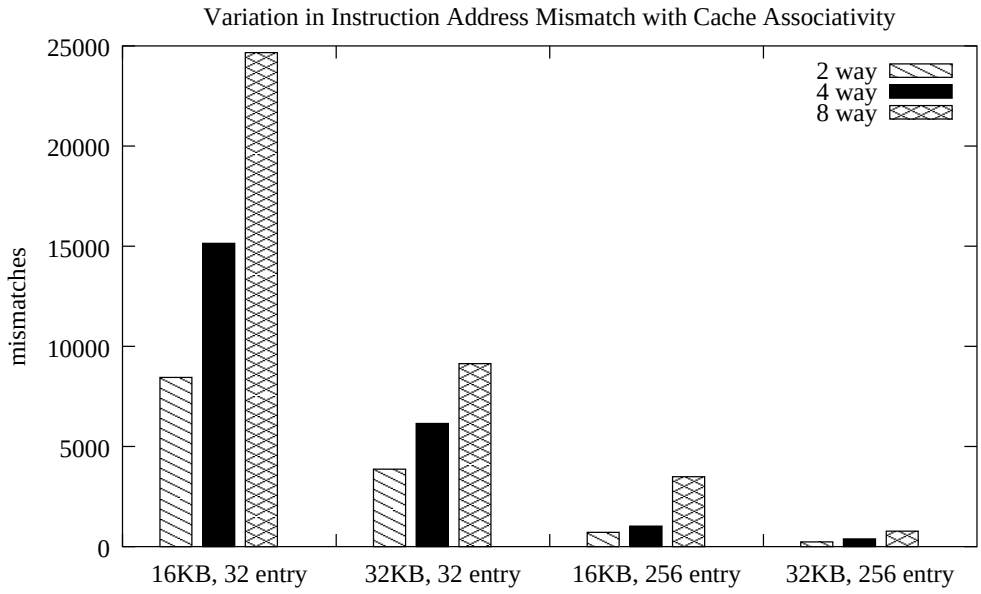


Figure 6.6: Variation in mismatch with cache associativity for Lisp Interpreter

Chapter 7

Conclusion and Future Work

From the statistical data in Chapter 6, it can be concluded that generally the next cacheline prediction scheme gives comparable performance as the other branch prediction based schemes. Only in the cases of **Perl Interpreter** and **Lisp Interpreter** program, did the scheme perform poorly. For loop intensive programs, like **Whetstone** and **Matrix Multiplication**, the next cacheline prediction scheme gave good performance. So this scheme can act as an alternative to the different existing sophisticated branch prediction mechanism.

Since tag comparison is not performed everytime the cache is accessed, the fetch pipeline gets shorter. There are actually two parallel fetch paths in our scheme. The shorter path (which is taken more often) bypasses the tag comparison step. The other longer path accesses the cache in traditional way, whenever a misprediction occurs. So, in general, our scheme will make cache access faster without compromising with the hit rate.

The work done by Koji Inoue et al. [4] shows that Way Prediction Caches can save up to 70% of cache power consumption. Since I-caches are accessed in almost every clock cycle, reducing power consumption of I-cache can save a considerable amount of total power consumed by the processor. The tag comparison of all the blocks in a particular cache set can be almost eliminated using next cacheline prediction, thus reducing power consumption. So there could be a power-performance trade off. This fact also increases the potential of the scheme. Next, we propose

some future directions for improving the design of the scheme and building a more efficient simulator for better testing purpose.

It was also seen that with increase in associativity of cache, the misprediction rate increased, resulting in performance degradation. The scheme will produce best results for direct mapped Instruction caches and 2-way set associative instruction caches.

7.1 Future Directions

1. We have seen in Chapter 4 that how cache replacement can degrade the performance of our prediction scheme. Especially, when a cache block containing a branch instruction, or a target instruction of a control transfer instruction, is replaced, then the prediction scheme fails. From this information we can suggest that if we can mark these instructions as *privileged* and design a cache replacement policy such that the privileged instructions are never replaced, or at the most replaced, when there is no other way, then our scheme may show better results. It is apparent that, the design of the cache and the replacement policy will have a tremendous impact on the performance of our scheme, a new kind of cache design is required which will enhance the prediction performance.

2. It is seen that increasing the prediction buffer size improves performance. But since we intend to implement the buffer as a fully associative cache, we cannot have a very large buffer. So an alternative design for the prediction buffer, so as to increase its size, can also help. We can implement the prediction buffer as a directly mapped cache indexed by the index of the current instruction.

3. The scheme stores information for non-control transfer statements in the prediction buffer along with branch information. This takes up space in the prediction buffer which could otherwise be used for storing other branch prediction informations. The next cache-block to be accessed in case of non-branch instructions can be stored in a special field in the cache block itself [2].

4. In our scheme, the prediction buffer stores fewer bits than a Branch Target

Buffer. The latter has to store the full instruction address, while in our scheme, only a few bits are required to store the cache block location. So with the same cost of that of a BTB, we can have a much larger set-way prediction table. So the latter can store more branch information at reduced cost. In our scheme, we have proposed the set-way prediction to be a fully associative cache memory. Instead, we could have a tagged direct mapped cache, indexed by the cache set number. This way, we could have larger tables yielding better results.

5. The performance degradation with increase in associativity suggests that the scheme in its present format is not suitable for more that 2-way associative caches. A new variation to this scheme for highly associative cache need to be investigated.

6. The simulation should be carried out using standard tools like simplescalar which can fully simulate various features of a superscalar architecture.

7. Finally, a word must be said about the trace generation program too. As mentioned in Chapter 3, there are many limitations to the program. Since we could not get the trace of a library routine, the generated trace was not complete. In fact, any call to a library routine was visualized as a combination of two unconditional jumps. From point of invocation to the library routine, and back. This assumption might have affected the performance of the prediction scheme. The generated trace files were too large. The program fails to generate traces of programs which uses *local*and *in* registers as jump targets. These problems can be solved if address traces are generated using some profiling tools.

Bibliography

[1] Brad Calder, D. G., and Emer, J. Predictive Sequential Associative Cache. *2nd International Symposium on High Performance Computer Architecture* (February 1996), 244–253. http://www.cs.colorado.edu/~grunwald/Papers/HPCA96-SeqAssocCache/paper.ps.

[2] Calder, B., and Grunwald, D. Next Cache Line and Set Prediction. *Proceedings of the 22nd annual international symposium on Computer architecture* (June 1995), 287–295. http://dev.acm.org/pubs/articles/proceedings/isca/223982/p287-calder/p287-calder.pdf.

[3] Hennessey, J. L., and Patterson, D. *Computer Architecture: A Quantitative Approach*, 2 ed. Harcourt Asia PTE LTD.

[4] Koji Inoue, T. I., and Murakami, K. Way-Predicting Set-Associative Cache for High Performance and Low Energy Consumption. *Proceedings of 1999 International Symposium on Low Power Electronics and Design (ISLPED'99)* (August 1999), 273–275. http://www.kasuga.csce.kyushu-u.ac.jp/~pparam/paper/PPRAM-TR-42.ps.gz/.

[5] Lee, B. Dynamic Branch Prediction. www.ece.orst.edu/~benl/Projects/branch_pred/ .

[6] S. Peter Song, M. D., and Chang, J. The PowerPC 604 RISC microprocessor. *IEEE Micro* (October 1994).

[7] Weaver, D. L., and Germond, T. *The SPARC Architecture Manual, Version 9*. Prentice Hall.

[8] Yeh, T., and Patt, Y. Two-level Adaptive Branch Prediction. *24th ACM/IEEE International Symposium on Microarchitecture* (November 1991).