Abstract

PANIRWALA, CHINTAN DIPAKCHANDRA. Exploring correlation for Indirect Branch Prediction. (Under the direction of Dr. Huiyang Zhou.)

History-based branch direction predictors for conditional branches are shown to be highly accurate. Indirect branches however, are hard to predict as they may have multiple targets corresponding to a single indirect branch instruction. With the state of the art predictors like Indirect Target Tagged Geometric length predictor (ITTAGE) which uses long history buffers to predict indirect branches, we still have some branches that are very hard to predict. We observed that in case of some of these branches, the branch target directly depends on the load address which is then used by the branch instruction. Based on this observation we propose Address Target Correlation for prediction of indirect branches.

This work explores two implementations. The first is a hardware only solution and the second is the combined hardware-software solution. In both cases we use ITTAGE as our baseline predictor. In hardware only solution we use a small table termed Address Target Table to predict the targets of hard to predict indirect branches during address generation stage of the load instruction. During the Address Generation (AGEN) stage of load we check if there is some dependent indirect branch and if there is one we provide prediction for that branch. During the execution stage of the indirect branch, we update the address target table based on the actual outcome of the branch. We showed that though hardware only solution is useful, we cannot get much improvements in terms of Misprediction Penalty per Kilo Instruction (MPPKI).

This limitation gave us motivation for looking beyond immediate load instruction and using compiler assisted address target correlation. We used ITTAGE as baseline predictor. We developed an algorithm that looks for the producer load instruction in the instruction trace as far as possible from the consumer branch instruction. It also uses address target table to cache load address and associated branch targets. It then takes prediction based on this load address.

Exploring Correlation for Indirect Branch Prediction

by
Chintan Dipakchandra Panirwala

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Master of Science

Computer Engineering

Raleigh, North Carolina

2012

APPROVED BY:

Dr. Huiyang Zhou
(Committee Chair)

Dr. Eric Rotenberg

Dr. Gregory T. Byrd

Dedication

To my beloved parents and sisters

To all my family members

Biography

Chintan Panirwala was born in Surat, India in 1988 as first son to Dipakchandra and Rita Panirwala. He did his high school at V. D. Desai (Vadiwala) High School in 2006. He started his undergraduate studies in Electronics Engineering at Sardar Vallabhbhai National Institute of Technology (SVNIT), Surat. Chintan finished his Bachelor's Degree in 2010. In 2010, he got accepted in the Computer Engineering graduate program at North Carolina State University. Here he started his research in branch prediction under the direction of Dr. Huiyang Zhou. He along with Nikunj Bhansali presented a paper on Indirect Branch Prediction at Championship Branch Prediction – 3 (CBP-3). In his thesis, he studied correlation between produce load addresses and consumer indirect branches and their effectiveness for Indirect Branch Prediction. During his studies, Chintan got an opportunity to intern at IBM where he got a chance to work with some talented people on Hardware Based Compression Algorithm. Upon the completion of his Master's degree he will be starting his first job at Intel Corporation in Hillsboro, Oregon.

Acknowledgements

First of all, I would like to thank my parents Dipakchandra Panirwala and Rita Panirwala. Through my journey they were a great support and motive for me to succeed. Many thanks to my two twin sisters Hunny and Dimpy for their support.

I would like to have special thanks to my Advisor Prof. Huiyang Zhou. His non-stopping help, attention, and guidance kept me going forward even during the rough times. During my two years at graduate school I learned a lot from him. My grad school journey was not an easy one, but sure after finishing now, I am proud of what we have achieved.

I would like to acknowledge my committee members: Dr. Greg Byrd and Dr. Eric Rotenberg. Thanks for your comments during my defense, they helped to improve the quality of my final thesis.

Special thanks to my friend Nikunj Bhansali with whom I worked on the same subject during my initial time of the research. Without him we would not have come this far. I would also like to thank Mihir Shah and Saurabh Gupta who were always ready to solve my doubts and who were always ready to help.

Table of Contents

LIST OF FIGURES viii

LIST OF TABLES x

1 INTRODUCTION 1

1.1 Indirect Branch 1

1.2 Indirect Branch Code Example 2

1.2.1 Indirect Branch generated by "switch" statement 2

1.2.2 Indirect Branch generated by a pointer to a function 5

1.2.3 Indirect Branch generated by a Virtual Function call 7

1.3 Indirect Branch Prediction 8

1.3.1 Value Based BTB Indexing 11

1.3.2 Compiler Guided Value Pattern 11

1.3.3 ITTAGE 12

2 ADDRESS TARGET CORRELATION 15

2.1 Address Target Correlation Concept 15

2.1.1 Prediction using ATT 16

2.1.2 ATT Update 18

2.1.3 Storage Cost 18

2.2 Evaluating Address Target Correlation 19

2.2.1 Methodology 19

2.2.2 ITTAGE Results 20

2.2.3 ATC Results 22

2.2.3.1 MPKI Results 22
2.2.3.2 MPPKI Results 24
2.2.4 ATC Results for INT05, INT06 and SERVER01 benchmarks 25

3 MULTI-LEVEL ADDRESS TARGET CORRELATION 28

3.1 Multi-Level Address Target Correlation Concept 29
3.1.1 First Level of Address-Target Correlation 29
3.1.2 Second Level of Address-Target Correlation 30
3.1.3 Third Level of Address-Target Correlation 31

3.2 Evaluating Multi-Level Address Target Correlation 32

4 COMPILER ASSISTED ADDRESS TARGET CORRELATION 35

4.1 Instrumentation 36
4.1.1 Instruction trace analysis 36
4.1.1.1 Indirect Branch 0x404750 36
4.1.1.2 Indirect Branch 0x40894b 37

4.2 Algorithm 41
4.2.1 Algorithm to trace address target correlation 41
4.2.2 Algorithm to determine stop_level 44

4.3 Case Study 47

5 EVALUATING COMPILER ASSISTED ATC 51

5.1 Methodology 51

5.2 ITTAGE MPKI results and analysis 53

5.3 Benchmark Analysis **55**
5.3.1 Omnetpp 55
5.3.1.1 Indirect Branch 0x40e9e6 56
5.3.2 Povray 57
5.3.2.1 Indirect Branch 0x4735aa 58
5.3.3 H264ref 60
5.3.3.1 Indirect Branch 0x43fa34 60

5.4 Performance of Address Target Correlation **62**
5.4.1 Case Study 65

5.5 Summary **69**

6 CONCLUSIONS **70**

7 BIBLIOGRAPHY **71**

List of Figures

Figure 1 Indirect Branch Generated by "switch" Statement ____ 3
Figure 2 Assembly code for switch-case statement ____ 4
Figure 3 Indirect Branch Generated by Pointer to a Function ____ 5
Figure 4 Assembly code for pointer to a function call ____ 6
Figure 5 Indirect Branch Generated by Virtual Function Call ____ 7
Figure 6 an entry in ITTAGE predictor ____ 12
Figure 7 ITTAGE Predictor ____ 13
Figure 8 Code example of address-target correlation ____ 15
Figure 9 Address Target Table to exploit ATC ____ 16
Figure 10 Trace for SERVER01 when target = 0x409080 ____ 17
Figure 11 Trace for SERVER01 when target = 0x40908f ____ 17
Figure 12 MPKI for ITTAGE ____ 21
Figure 13 MPPKI for ITTAGE ____ 21
Figure 14 Total MPKI for ITTAGE + ATT configuration ____ 23
Figure 15 Percentage Improvements in MPPKI for ITTAGE + ATT configuration ____ 24
Figure 16 Comparison of MPKI between ITTAGE and ITTAGE + 8*8 ATT for INT05, INT06 and SERVER01 benchmarks ____ 26
Figure 17 Comparison of MPPKI between ITTAGE and ITTAGE + 8*8 ATT for INT05, INT06 and SERVER01 benchmarks ____ 27
Figure 18 MPPKI gain for immediate Address Target Correlation ____ 28
Figure 19 Instruction trace for First Level of ATC ____ 29
Figure 20 Instruction trace for Second Level of ATC ____ 30
Figure 21 Instruction trace for Third Level of ATC ____ 31
Figure 22 Instruction trace for SERVER01 benchmark ____ 32
Figure 23 Improvements in MPPKI for Multi-level ATC ____ 33
Figure 24 Source code for indirect branch 0x404750 ____ 37

Figure 25 Assembly code for indirect branch 0x404750 ____ 39
Figure 26 Source code for indirect branch 0x40894b ____ 40
Figure 27 Assembly code for indirect branch 0x40894b ____ 40
Figure 28 Algorithm to trace Address Target Correlation ____ 43
Figure 29 Algorithm to detect stop_level ____ 46
Figure 30 instruction trace for Indirect Branch 0x404750 ____ 47
Figure 31 Address target correlation at each level ____ 49
Figure 32 most hard to predict benchmarks for ITTAGE ____ 54
Figure 33 Source code for indirect branch 0x40e9e6 ____ 56
Figure 34 Assembly code for indirect branch 0x40e9e6 ____ 57
Figure 35 Source code for indirect branch 0x4735aa ____ 58
Figure 36 Assembly code for indirect branch 0x4735aa ____ 59
Figure 37 Source code for indirect branch 0x43fa34 ____ 60
Figure 38 Assembly code for indirect branch 0x42a054 ____ 61
Figure 39 Cycles saved for branches 0x404750 and 0x4048df ____ 62
Figure 40 Cycles saved for branch 0x40894b ____ 62
Figure 41 Cycles saved for branch 0x40324e ____ 64
Figure 42 Instruction trace of branch 0x404750 showing cause of mispredictions ____ 68

List of Tables

Table 1 Storage Cost for Address Target Table 18
Table 2 MPKI and MPPKI for INT05, INT06 and SERVER01 22
Table 3 Improvements in MPKI after using ATC 23
Table 4 Improvements in MPPKI after using ATC 24
Table 5 ATC results for INT05 benchmark 25
Table 6 ATC results for INT06 benchmark 25
Table 7 ATC results for SERVER01 benchmark 25
Table 8 Improvements in MPPKI with Multi-level ATC for different predictor sizes 33
Table 9 Cache Parameters used for PIN Instrumentation 52
Table 10 MPKI for SPEC2006 benchmarks with ITTAGE 54
Table 11 Hard to predict branches for omnetpp benchmark 55
Table 12 Hard to predict branches for povray benchmark 58
Table 13 Hard to predict branches for h264ref benchmark 60
Table 14 Cycles saved for branches 0x404750, 0x4048df and 0x40894b with 3-bits confidence counter 63
Table 15 Cycles saved for branch 0x40324e with 3-bits confidence counter 64
Table 16 Cycles saved for branches 0x404750, 0x4048df and 0x40894b with 2-bits confidence counter 65
Table 17 Analysis of branch 0x404750 for 3-bits confidence counter 66
Table 18 Address target correlation level 2 and 3 for profiling period of 200 million instructions 67

1 Introduction

Modern superscalar processors use pipelining to overlap the execution of instructions and improve performance. This potential overlap of instructions is called Instruction Level Parallelism (ILP). But the control hazards prevent them from taking advantage of all the available ILP. One of the most used mechanisms to overcome control hazards is the branch prediction. Rather than stalling when a branch is encountered, a pipelined processor uses branch prediction to speculatively fetch and execute instructions along the predicted path. As pipeline deepens and number of instructions issued per cycle increases, the penalty for misprediction increases, as does the benefit of accurate branch prediction.

1.1 Indirect Branch

Control hazards are due to branch instructions. Branches can be classified into conditional or unconditional and direct or indirect branches. In case of conditional branches there are only two possibilities. If it is taken then we jump to the specified target and if it is not taken then we go to next Program Counter (PC). Branch prediction research has shown that for a conditional branch instruction, its direction can be predicted with high accuracy. Branch Target Buffer (BTB) can be used to predict the target. Furthermore, a direct branch has a fixed target encoded in the instruction, thus making conditional or unconditional direct branches easier to predict. Unlike direct branches, indirect branch instructions are hard to predict as they may have multiple targets corresponding to a single indirect branch instruction. Indirect branch has the following format:

```
JMP *%eax
```

This can be interpreted as “jump indirect on the eax register”, which would mean that the next instruction to be executed would be at the address whose value is in register eax. Thus, rather than specifying an address of the next instruction to execute, as in a direct branch, the argument specifies where the address is located [1]. We have focused on indirect branches for the x86-64 instruction set architecture.

1.2 Indirect Branch Code Example

Virtual Function call, function call by a pointer to a function and switch statements are the three major sources of indirect branches.

1.2.1 Indirect Branch generated by “switch” statement

A switch statement uses an indirect branch instruction to continue processing from the case label that matches the input argument. Example code is shown in Figure 1. Assembly code for the corresponding example code is shown in Figure 2. Source instruction and mnemonics for Indirect Branch are shown in bold red.

```
void main (void)    {
        int ret;
        while (1)    {
                switch (ret)  {
                        case 0:
                                ret++;
                                break;
                        case 1:
                                ret++;
                                break;
                        case 2:
                                ret++;
                                break;
                        case 3:
                                ret++;
                                break;
                        case 4:
                                ret++;
                                break;
                        case 5:
                                ret++;
                                break;
                        default:
                                ret = 0;
                                break;
                } //end switch
        } //end while
} //end main
```

Figure 1 Indirect Branch Generated by "switch" Statement

```
400448:   push    %rbp
400449:   mov     %rsp, %rbp
40044c:   jmp     400490 <main+0x48>
40044e:   addl    $0x1, 0xfffffffffffffff8 (%rbp)
400452:   cmpl    $0x5, 0xfffffffffffffffc (%rbp)
400456:   ja      400489 <main+0x41>
400458:   mov     0xfffffffffffffffc (%rbp), %eax
40045b:   mov     0x400598(, %rax, 8), %rax
400463:   jmpq    *%rax
400465:   addl    $0x1, 0xfffffffffffffffc (%rbp)
400469:   jmp     400490 <main+0x48>
40046b:   addl    $0x1, 0xfffffffffffffffc (%rbp)
40046f:   jmp     400490 <main+0x48>
400471:   addl    $0x1, 0xfffffffffffffffc (%rbp)
400475:   jmp     400490 <main+0x48>
400477:   addl    $0x1, 0xfffffffffffffffc (%rbp)
40047b:   jmp     400490 <main+0x48>
40047d:   addl    $0x1, 0xfffffffffffffffc (%rbp)
400481:   jmp     400490 <main+0x48>
400483:   addl    $0x1, 0xfffffffffffffffc (%rbp)
400487:   jmp     400490 <main+0x48>
400489:   movl    $0x0, 0xfffffffffffffffc (%rbp)
400490:   cmpl    $0xc7, 0xfffffffffffffff8 (%rbp)
400497:   jle     40044e <main+0x6>
400499:   leaveq
40049a:   retq
```

Figure 2 Assembly code for switch-case statement

1.2.2 Indirect Branch generated by a pointer to a function

Calling a function pointed by a function pointer will result in an Indirect Branch. Example code is shown in Figure 3. Assembly code for the corresponding example code is shown in Figure 4. Source instruction and mnemonics for Indirect Branch are shown in bold red.

```
unsigned char loop_inc ( unsigned char loop_count )
     {
     return ++loop_count;
} //end loop_inc


void main(void)     {
     unsigned char loop = 0;
     unsigned char (*pf) (unsigned char);    //Define a
pointer to a function
     pf = loop_inc;                     //Assign address of
loop_inc() to pf
     while( loop <= 100 )     {
          loop = pf ( loop );
     } //end while
} //end main
```

Figure 3 Indirect Branch Generated by Pointer to a Function

```
40045a: push   %rbp
40045b: mov    %rsp, %rbp
40045e: sub    $0x10, %rsp
400462: movb   $0x0, 0xfffffffffffffff7 (%rbp)
400466: movq   $0x400448, 0xfffffffffffffff8 (%rbp)
40046e: jmp    40047d <main+0x23>
400470: movzbl 0xfffffffffffffff7 (%rbp),%edi
400474: mov    0xfffffffffffffff8 (%rbp), %rax
400478: callq  *%rax
40047a: mov    %al, 0xfffffffffffffff7 (%rbp)
40047d: cmpb   $0x64, 0xfffffffffffffff7 (%rbp)
400481: jbe    400470 <main+0x16>
400483: leaveq
400484: retq
```

Figure 4 Assembly code for pointer to a function call

1.2.3 Indirect Branch generated by a Virtual Function call

Object oriented programs are becoming more common as more programs are written in modern high-level languages such as Java, C++ and C#. These languages support polymorphism [2], which significantly eases the development and maintenance of large modular software projects. To support polymorphism, modern languages include dynamically dispatched function calls (i.e. virtual functions) whose targets are not known until run-time because they depend on the dynamic type of the object on which the function is called. Virtual function calls are usually implemented using indirect branch/call instructions in the instruction set architecture. Example code for virtual function call is shown in Figure 5.

```
class Base {
      public:
      virtual const char* GetName() { return "Base";}
};
class Derived: public Base {
      public:
      virtual const char* GetName() { return "Derived"; }
};
int main() {
      Derived cDerived;
      Base &rBase = &cDerived;
      cout << "rBase is a " << rBase.GetName() << endl;
return 0;
}
```

Figure 5 Indirect Branch Generated by Virtual Function Call

1.3 Indirect Branch Prediction

Unlike direct branches, indirect branch instructions are hard to predict as they may have multiple targets corresponding to a single indirect branch instruction. Function call returns are a special type of indirect branch predicted accurately using a return address stack [3]. Lee and Smith [4] used branch target buffer (BTB) to predict indirect branches. A BTB predicts the last taken target of the branch as the current target. This scheme works well for direct conditional or unconditional branches since they have only one taken target. Indirect branches often have multiple targets for the same static instruction. Therefore, a BTB-based predictor, though simple in design, is inaccurate for predicting indirect branches with multiple targets. Chang et al. [5] proposed a branch history based two-level predictor known as the “target cache” for predicting the target address of indirect jumps. It uses the same concept as a two-level branch direction predictor. Target addresses from recently executed indirect jump instructions are recorded in a target history register. When an indirect jump is fetched, the fetch address and the target history register are used to index the target cache to predict the next target address. Upon retiring the indirect jump, the target cache entry and the target his-tory register is updated with the actual target address. Several configuration parameters such as size of target history register, number of target address bits shifted in the target history register, bit position from where these bits are taken etc. have direct impact on the performance of target cache predictor. Li et al. [6] proposed rehashable BTB (R-BTB) scheme that dynamically identifies hard-to-predict indirect branches and stores their targets in the traditional BTB using a different indexing function. This new index is computed by XORing the target history register with the jump PC. Other branches including the easy-to-predict indirect branches continue to index the BTB using their PC value. Driesen et al. [7] [8] combined multiple target predictors using a cascaded predictor. Cascaded predictor is a hybrid predictor consisting of a simple predictor for easy-to-predict indirect branches, and a more complex predictor for hard-to-predict indirect branches. Kalamatianos et al. [9]

proposed predicting indirect branches via data compression. Their predictor uses the prediction by partial matching (PPM) algorithm of order three, which is a set of four Markov predictors of decreasing size, indexed by an indexing function formed by a decreasing number of bits from previous targets in the target history register. Kim et al. [10] proposed Virtual Prediction Counter (VPC) prediction which uses the existing conditional branch predictor for predicting the indirect branches. Conceptually, VPC treats an indirect branch instruction with number of targets as direct branches, each with its own unique target address. When an indirect jump instruction is fetched, the VPC prediction algorithm accesses the conditional branch predictor for MAXITER times, each time as a different "virtual direct branch' of the same indirect branch. This iterative process stops either when a "virtual direct branch" is predicted to be taken, or MAXITER number is reached, in which case the processor is stalled until the indirect branch is resolved. Here, MAXITER determines the number of attempts made to predict an indirect branch. Each attempt takes one cycle during which no new instruction is fetched.

Precomputation-based target prediction tries to calculate the target address in anticipation of having to make a prediction. Roth et al. [11] proposed a precomputation-based prediction scheme specifically for virtual function calls. It dynamically captures the sequence of instructions involved in the target generation process. Whenever the first instruction in the sequence completes, it quickly executes the rest of the instruction sequence using a separate execution engine, and computes the target before the actual call instruction is encountered. Although this technique avoids using specialized jump predictor, it requires significant hardware for capturing the target generation instructions along with a fast execution engine to pre-compute the target. Furthermore, this technique is very specific to virtual function call's target prediction, as their target generation process consists of a fixed pattern of three dependent loads followed by an indirect call. Kaeli et al. [12] proposed the case block table (CBT), a hardware table that stores case addresses of a switch-case statement. When a switch-case statement is encountered, instead of executing a series of conditional branches it uses the value of the case block variable to fetch the case address from the CBT. By jumping

directly to the appropriate case block instead of executing a series of conditional branches, the dynamic instruction count is reduced by 9%. However, modern compilers already generate the jump table corresponding to the switch-case statement if the number of cases exceeds certain threshold [5]. If the value of the case variable is not known, CBT redirection is delayed until it is available; resulting in performance degradation for deeply pipelined superscalar processors. Finally, Joao et al. [13] proposed dynamic predication of hard-to-predict indirect jump instructions. When a hard-to-predict indirect jump instruction is fetched, the processor starts fetching from N different targets of the jump instruction, thereby increasing the probability of fetching from the correct target path at the expense of executing more instructions. They showed that N=2 is a good trade-off between performance and complexity.

ITTAGE predictor [14] uses long history information to predict target of indirect branch. In [15] Farooq et al. propose Value Based BTB Indexing (VBBI) for indirect branch prediction. In VBBI, for each static hard to predict indirect branch instruction, the compiler identifies a "hint instruction", whose output value strongly correlates with the target address of the indirect jump instruction. At run time, multiple target addresses of the indirect jump instruction are stored and subsequently accessed from the BTB at different indices computed using the jump instruction PC and the hint instruction output values. Compiler guided Value Pattern (CVP) prediction [16] uses the compiler-guided value pattern as the correlated information to hint the dynamic predictor. The value pattern reflects the pattern regularity of the value correlation, and thus significantly improves the prediction accuracy.

1.3.1 Value Based BTB Indexing

In VBBI for every hard-to-predict static indirect jump instruction, the compiler identifies an instruction whose output value strongly correlates with the target address taken by the jump instruction. When this correlated instruction (hint instruction) produces a value (hint value), the calculated target address of the indirect jump instruction is stored in the BTB at an index computed by hashing the PC of the jump instruction with the hint value. Next time when the jump instruction is fetched, BTB is indexed using its PC and the new hint value to get the predicted target address. In order to maintain strong correlation between the target and the hint value, the current hint value is used. In cases where the latest hint value is not available when the jump instruction is fetched, target prediction is made using old hint value. However, in these cases a second and more accurate target prediction is made using the new hint value when it becomes available. A more accurate second prediction effectively reduces the impact of jump mispredictions on performance by decreasing the jump misprediction penalty.

1.3.2 Compiler Guided Value Pattern

CVP combines multiple correlated data values to form a value pattern and then uses the value pattern to distinguish dynamic occurrences of each indirect branch. So even if the latest single data value is unavailable at the prediction time, it still achieves high prediction accuracy. At compile time, the compiler identifies the correlated data values based on three program substructures: virtual function calls, switch-case statements and function pointer calls. The compiler-assisted information is passed to the dynamic predictor using a special NOP instruction, called guiding instruction. At runtime, the processor combines those data values to form the value pattern according to the compiler-assisted information. When an

indirect branch instruction is fetched, the predictor uses the hash value of the branch address and the value pattern to get the final predicted target address from the BTB.

1.3.3 ITTAGE

As shown in Figure 7, ITTAGE indexes several predictor tables through independent functions of the global branch/path history and the branch address. A tag-less base predictor provides default prediction. The tagged predictors T_i, $1<i<M$, are indexed using history lengths $L(i) = (int)(a^{i-1} * L(1) + 0.5)$. History lengths form a geometric series which allows to efficiently capturing correlation on recent branch outcomes as well as on very old branches. A predictor table entry (Figure 6) features target address, a tag, a 2-bit confidence counter "Ctr" allowing some hysteresis on the predictor and a useful bit U for controlling the update policy. At prediction time, the base predictor and the tagged components are accessed simultaneously. The prediction is provided by the hitting tagged predictor component that uses the longest history. In case of no matching tagged predictor component, the default prediction is used. However when the confidence counter of the matching entry is null, the alternate prediction "altpred" is preferred. Generally the predictors tables are updated at retire time to avoid pollution of the predictor by the wrong path. A single predictor table entry may provide several mispredictions in a row due to this late update. In order to reduce this impact, ITTAGE uses the immediate update mimicker (IUM).

Target	Tag	Ctr	U

Figure 6 an entry in ITTAGE predictor

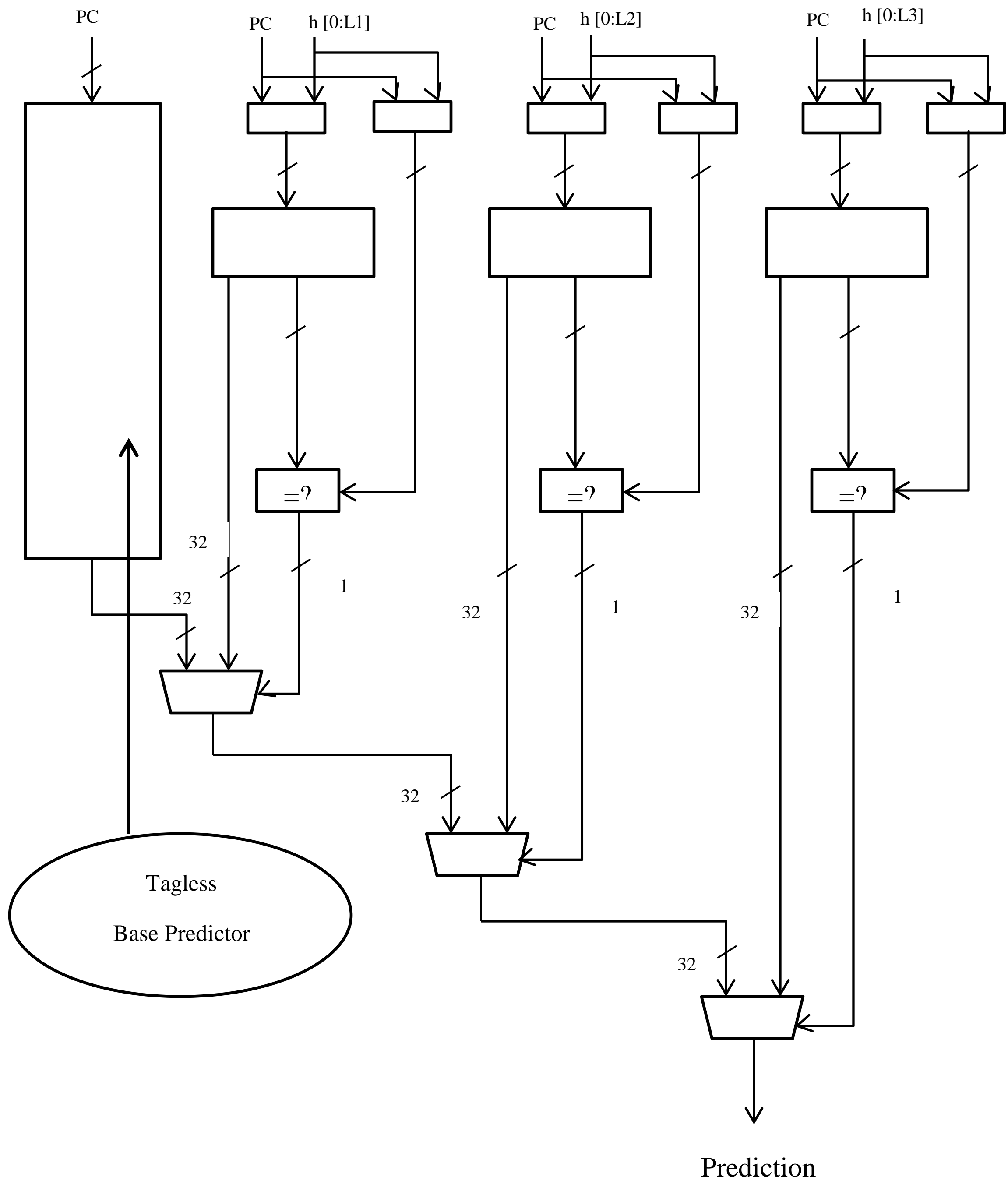

Figure 7 ITTAGE Predictor

Our approach is different from the above mentioned history based, precomputation based or value based schemes in that we try to exploit correlation between load addresses and branch targets. We found that there exists a very strong correlation between the producer load addresses and the consumer branch targets, similar to address-branch correlation explored for conditional branches [17]. We exploit such correlation at the address generation (AGEN) stage of a load instruction to make early correction of mispredicted indirect branches. Our approach is similar to VBBI but the fundamental difference is that we exploit the stable relationship between load addresses and indirect branch targets, which may span multiple load instructions, while VBBI focuses on data dependence flow. For example, in the case of "switch (p->num) {…}; p= p->next", VBBI uses the last definition of "p->num" while our approach uses "& (p->num)" or "p" to predict/resolve the indirect branch "switch (p->num)".

2 Address Target Correlation

2.1 Address Target Correlation Concept

From our trace analysis of the benchmarks, we found that the targets of many hard to predict indirect branches are dependent upon load values. Since the loaded value will be immediately used by the dependent indirect branches, such execution results are available too late to be useful to reduce misprediction penalty. Therefore, we examine the correlation between the producer load addresses and the consumer branch targets, similar to the address-branch correlation observed for conditional branches [17]. Figure shows example code for address-target correlation.

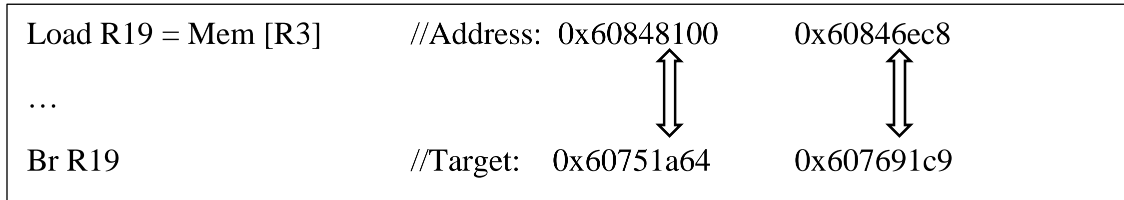

Figure 8 Code example of address-target correlation

From Figure 8, we can see that the producer load accesses two primary addresses and either address contains a different branch target. As long as the data structure (e.g., a virtual function table) at these addresses is not frequently updated, the addresses of the producer loads are sufficient to determine the targets of their consumer indirect branches. We refer to such correlation as address-target correlation (ATC). In our design, we capture ATC in a small cache-like set associative structure, called Address Target Table (ATT) as shown in

Figure 9. Each entry in ATT contains a tag and multiple pairs of hashed addresses and the corresponding targets.

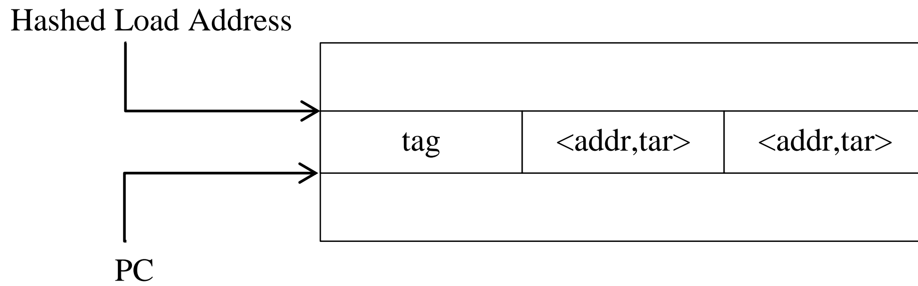

Figure 9 Address Target Table to exploit ATC

2.1.1 Prediction using ATT

We access the ATT to take a prediction during the Address Generation (AGEN) stage of load instruction. During AGEN stage of a load instruction we check if there is an indirect branch which depends on the load address of this instruction. If we find the correlation, the PC of the consumer indirect branch is used for tag match to see whether an entry in ATT has been allocated for it. If so, the hashed address of the producer load will be used to compare with multiple address-target pairs in the entry to find a matching pair to provide the prediction for the consumer branch. After taking the prediction we compare it with the prediction provided by ITTAGE at fetch stage. If the prediction differs from the one made at the fetch stage of the indirect branch, an early misprediction recovery is initiated to reduce the misprediction penalty.

Type	Src1	Src2	Src3	Dest	Target	SRCdata1	SRCdata2	SRCdata3	LDST1	LD_Addr
Load	32	2	18	18	0	0	4c4540	1	409080	4c4548
MOV	18	0	0	19	0	409080	0	0	0	0
MOV	30	19	0	19	0	33	409080	0	0	0
JMP	30	19	0	0	409080	33	409080	0	0	0

Figure 10 Trace for SERVER01 when target = 0x409080

Type	Src1	Src2	Src3	Dest	Target	SRCdata1	SRCdata2	SRCdata3	LDST1	LD_Addr
Load	32	2	18	18	0	0	4c4540	0	40908f	4c4540
MOV	18	0	0	19	0	40908f	0	0	0	0
MOV	30	19	0	19	0	33	40908f	0	0	0
JMP	30	19	0	0	40908f	33	40908f	0	0	0

Figure 11 Trace for SERVER01 when target = 0x40908f

Figure 10 and Figure 11 represents the typical example of address target correlation. This is taken from the SERVER01 trace. When the load instruction is in the AGEN stage, we check if there is any indirect branch that will directly use this load instruction's results. If we find such instruction we use load address for predicting target of an indirect branch. In our example we found that the destination register of LOAD is same as the source register of the next MOV instruction. Similarly, destination register of this MOV is same as the source register of the MOV after it. The indirect branch after it directly uses the destination of the MOV as the source register. Thus, in this case there is one to one relationship between load address and branch target as in when the load address is 0x4c4548 indirect branch jumps to 0x409080 and when the load address is 0x4c4540, indirect branch jumps to 0x40908f. As a result, if we can determine the load address during the AGEN stage, we can accurately predict the outcome of the immediate indirect branch.

2.1.2 ATT Update

ATT is updated at the execution stage of an indirect branch. At the execution stage of an indirect branch we check if the branch was mispredicted at the fetch stage. Only if it is mispredicted at the fetch stage, we search for its producer load address and then update ATT with the actual branch target. The least-recently-used (LRU) replacement policy is used to select a victim in ATT. Random replacement is used if there are more address-target pairs than what each entry in ATT can maintain.

2.1.3 Storage Cost

Each entry in address target table contains a tag and multiple address-target pairs. We hash load address and branch PC to compute the tag. We use 32 bits for tag storage. We use 32 bits for storing targets and 10 bits for storing hashed load addresses. Storage requirement for LRU field depends on number of entries (N) in the address target table and is $\log_2 N$. Storage requirements are shown in Table 1.

Table 1 Storage Cost for Address Target Table

<Number of Entries, Address-Target Pairs>	Storage Cost (Bytes)
<26,10>	1485
<8,8>	371
<8,4>	203

2.2 Evaluating Address Target Correlation

2.2.1 Methodology

For the experiments in this section we used simulation infrastructure provided in Championship Branch Prediction (CBP). The framework models a simple out-of-order core with following parameters.

- 256-entry reorder buffer, and three schedulers: an integer scheduler with 64 entries and a floating point and load/store schedulers with 32-entries each.
- The processor has a 14-stage, 4-wide pipeline except in the execution stage where it has a 12-wide execution scheduler (6 integer, 4 FP and 2 load-store).
- The memory model consists of a 2-level cache hierarchy, consisting of an L1 split instruction and data caches, and an L2 last level cache. All caches support 64-byte lines. The L1 instruction cache is 32KB 8-way set associative cache. The L1 data cache is 32KB 8-way set-associative. The L2 data cache is a 4 MB, 8-way set-associative cache.

We also used traces provided by the CBP for experiments. We used 64 KB ITTAGE (which was adjudged best predictor in CBP) as our base predictor. With ITTAGE as base predictor, for each trace we got the Misprediction per Kilo Instruction (MPKI) and MPPKI results. Based on these results we determined most hard to predict traces for ITTAGE predictor. We also short listed most hard to predict branches in those traces. For these hard to predict branches we obtained the instruction trace containing information such as program counter, opcode, source registers and their values, destination registers and their values, load-store addresses and data, branch target and the cycle number when they were in different stages of

the pipeline. We used this information to check for the address-target correlation and how much cycles we can save by correcting the misprediction.

To this infrastructure, we added our modification to detect address target correlation and a small cache like set associative structure to take prediction. We look six instructions back to trace address target prediction. We took prediction in the address generation stage of the load instruction. However, we only took prediction if it was different from the one provided by ITTAGE in fetch stage. We only updated our predictor if there was misprediction in the fetch stage by ITTAGE.

2.2.2 ITTAGE Results

First we evaluated the 64KB ITTAGE predictor which was the best indirect branch predictor in the most recent Championship Branch Prediction (CBP). For 40 traces provided in the CBP, performance of ITTAGE in terms of MPKI and MPPKI is shown in Figure 12 and Figure 13. The total MPPKI for ITTAGE is 1364.11 and total MPKI is 10.4836. However if go in detail, we observe that INT05, INT06 and SERVER01 are the most important benchmarks as they comprises of about half of the total MPKI and MPPKI. This data is shown in Table 2.

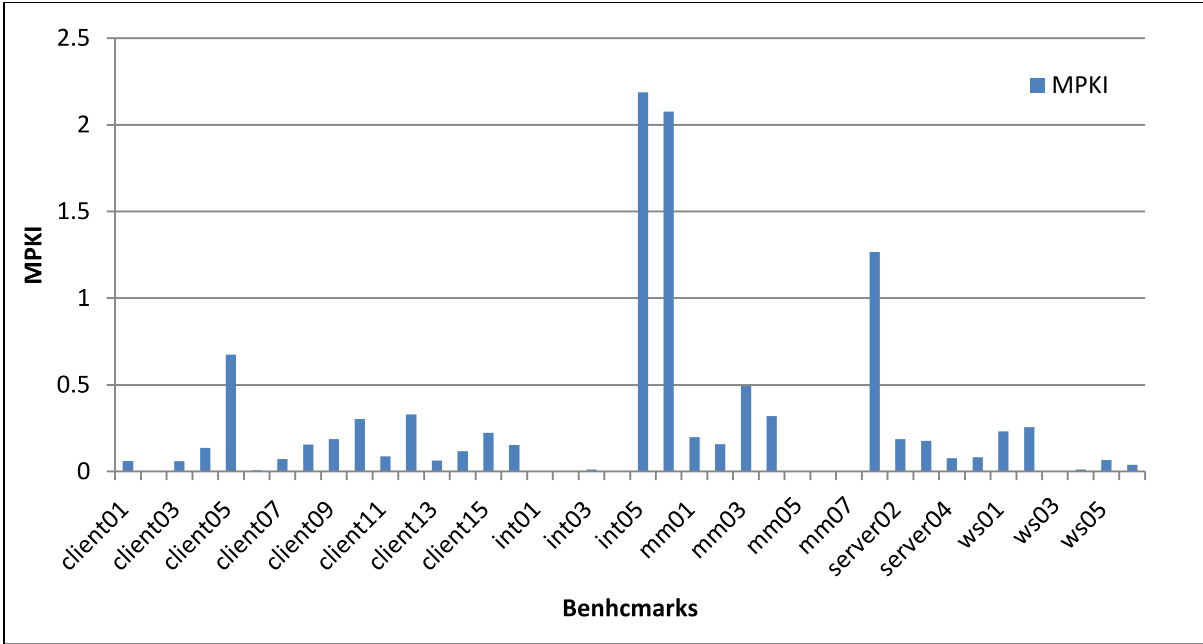

Figure 12 MPKI for ITTAGE

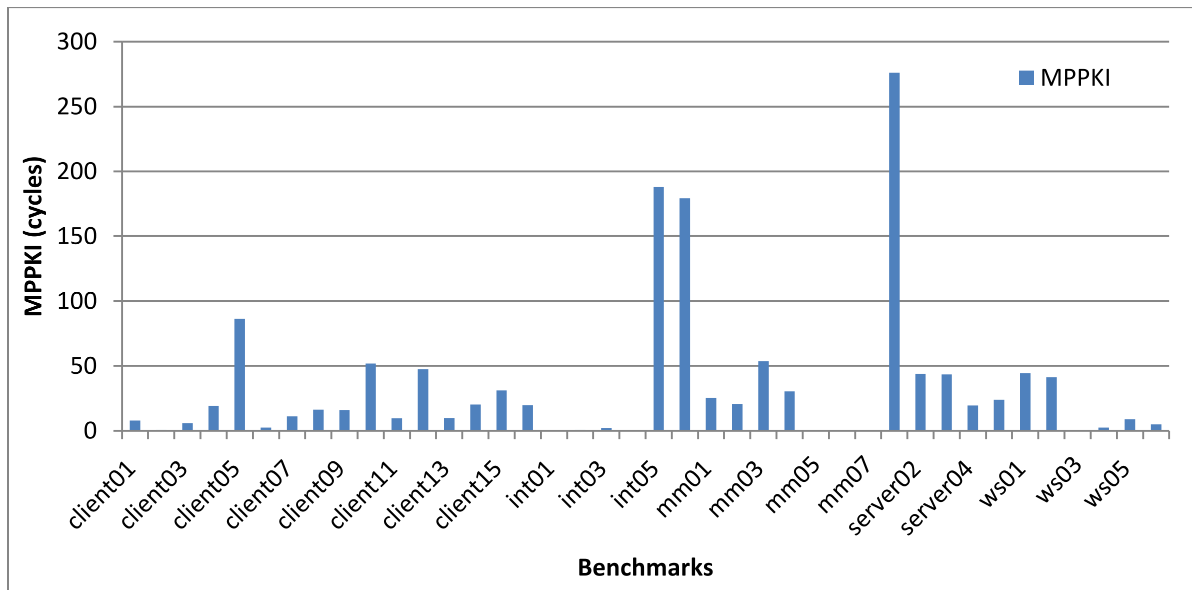

Figure 13 MPPKI for ITTAGE

Table 2 MPKI and MPPKI for INT05, INT06 and SERVER01

Benchmark	MPKI	MPPKI
INT05	2.1874	187.99
INT06	2.0773	179.34
SERVER01	1.2663	276.14
Total	**5.5310**	**643.47**

2.2.3 ATC Results

We evaluated ATC on the same CBP benchmarks on which results for ITTAGE are shown. We used 64 KB ITTAGE as our baseline predictor which would provide prediction at fetch stage. During AGEN stage, we only take prediction from ATT if its prediction is different from the one provided by ITTAGE at fetch stage. Also during execute stage; we only update ATT if the branch was mispredicted at the fetch stage.

2.2.3.1 MPKI Results

Table 3 shows improvements we get in MPKI after using Address Target Correlation at the AGEN stage in co-operation with the ITTAGE at fetch stage. It is clear from the results shown that, just by using a small (8 entries, 4 address-target pairs) ATT with ITTAGE we can get as much as 50% reduction in the MPKI. If we increase the number of address-target pairs from 4 to 8 we can reduce total MPKI by 60%. Using a large table with 26 entries and 10 address-target pairs does not give much benefit. Same data is shown in Figure 14. These results were obtained while looking 6 instructions back for tracing address-target correlation.

Table 3 Improvements in MPKI after using ATC

Predictor	Total MPKI	Percentage Improvements
ITTAGE	10.4836	00.00
ITTAGE + 8*4 ATT	5.1664	50.72
ITTAGE + 8*8 ATT	4.2949	59.03
ITTAGE + 26*10 ATT	4.2041	59.89

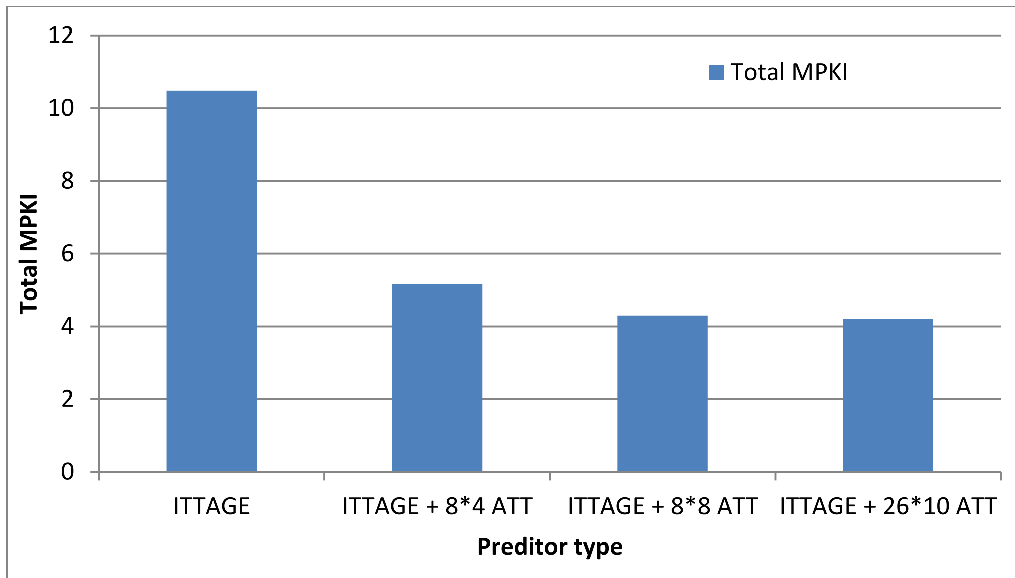

Figure 14 Total MPKI for ITTAGE + ATT configuration

2.2.3.2 MPPKI Results

Table 4 shows percentage improvements we get in MPPKI after using Address Target Correlation at the AGEN stage in co-operation with the ITTAGE at fetch stage. The results were obtained while looking back 6 instructions for address-target correlation.

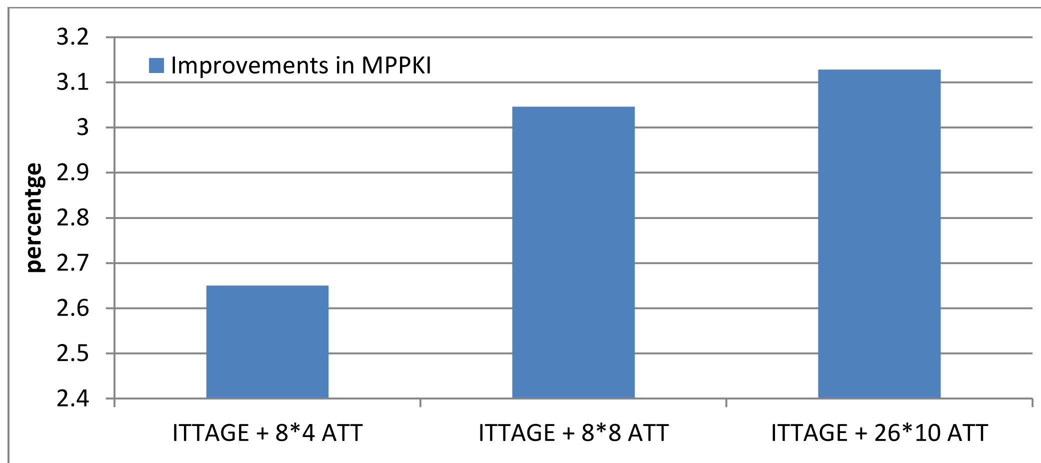

Figure 15 Percentage Improvements in MPPKI for ITTAGE + ATT configuration

Table 4 Improvements in MPPKI after using ATC

Predictor	Total MPPKI	Percentage Improvements
ITTAGE	1364.11	0.00
ITTAGE + 8*4 ATT	1327.96	2.65
ITTAGE + 8*8 ATT	1322.56	3.05
ITTAGE + 26*10 ATT	1321.43	3.11

Early misprediction recovery using ATT with 8 entries and 4 address-target pairs provides a 2.65% improvement in MPPKI compared to ITTAGE. Using ATT with 8 entries and 8 address-target pairs gives an improvement of 3.05%. However, large ATT with 26 entries and 10 address-target pairs does not give much benefit.

2.2.4 ATC Results for INT05, INT06 and SERVER01 benchmarks

Earlier we observed that for ITTAGE predictor, INT05, INT06 and SERVER01 accounts for about half of the total MPKI and MPPKI of all the CBP benchmarks. The results below show how our predictor fairs on these benchmarks.

Table 5 ATC results for INT05 benchmark

Predictor	Coverage (%)	Accuracy (%)	Improvements in MPKI (%)	Improvements in MPPKI (%)
ITTAGE+8*4 ATT	84.04	100	84.02	6.07 (11 cycles)
ITTAGE+8*8 ATT	99.39	100	99.39	7.17 (13 cycles)
ITTAGE+26*10 ATT	99.44	100	99.44	7.18 (13 cycles)

Table 6 ATC results for INT06 benchmark

Predictor	Coverage (%)	Accuracy (%)	Improvements in MPKI (%)	Improvements in MPPKI (%)
ITTAGE+8*4 ATT	79.92	100	79.92	5.75 (10 cycles)
ITTAGE+8*8 ATT	99.41	100	99.41	7.15 (13 cycles)
ITTAGE+26*10 ATT	99.48	100	99.48	7.15 (13 cycles)

Table 7 ATC results for SERVER01 benchmark

Predictor	Coverage (%)	Accuracy (%)	Improvements in MPKI (%)	Improvements in MPPKI (%)
ITTAGE+8*4 ATT	99.39	100	99.39	3.68 (10 cycles)
ITTAGE+8*8 ATT	99.39	100	99.39	3.68 (10 cycles)
ITTAGE+26*10 ATT	99.39	100	99.39	3.68 (10 cycles)

Here coverage is defined as the branches that were earlier mispredicted by ITTAGE and for which prediction is provided by the ATT at the AGEN stage. Accuracy is measured by the correct prediction provided by the ATT. Improvements in MPKI and MPPKI are compared to ITTAGE predictor.

As shown in Table 5 for INT05 benchmark, 8*4 ATT is able to provide prediction for about 84% of the mispredicted branches with 100 % accuracy which reduces MPKI from 2.1874 to 0.3491 and MPPKI from 187.99 to 176.50. Increasing the predictor size we can achieve almost 100% coverage and as a result MPKI drops to 0.0133.

As shown in Table 6 for INT06 benchmark, 8*4 ATT is able to provide prediction for about 80% of the mispredicted branches with 100 % accuracy which reduces MPKI from 2.0773 to 0.4172 and MPPKI from 179.34 to 169.02. As is the case with INT05 benchmark, increasing predictor size achieves 100% coverage reducing MPKI to 0.0123.

However SERVER01 represents an interesting case where size of the predictor does not make any difference to the performance. As shown in Table 7 Even with 8*4 ATT we can achieve 100% coverage which reduces MPKI from 1.2663 to 0.0077 and MPPKI from 276.14 to 265.98.

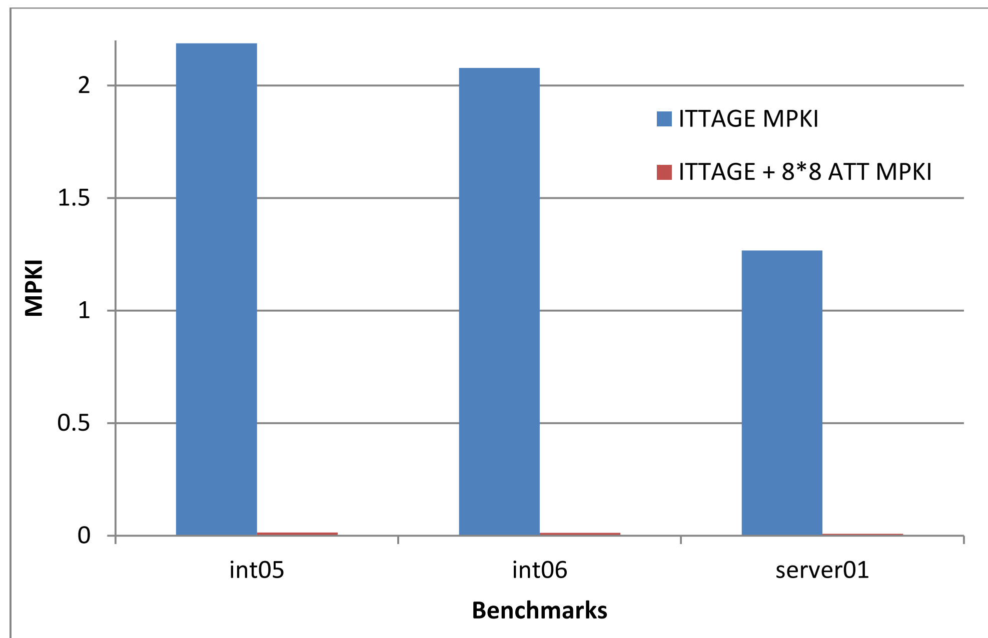

Figure 16 Comparison of MPKI between ITTAGE and ITTAGE + 8*8 ATT for INT05, INT06 and SERVER01 benchmarks

This may be due to the fact that SERVER01 has a very few hard to predict branches with a very few disparate targets. But these branches occur highly frequently and are so random in behavior that ITTAGE is not able to capture the history information properly.

As shown in Figure 16, by using 8*8 ATT at the AGEN stage to exploit address target correlation while looking back for 6 instructions with ITTAGE as main predictor we are able to bring down MPKI to almost zero for these three benchmarks. Figure 17 shows that Address-target correlation is able to achieve on average reduction of 11 cycles for these three benchmarks. This relatively limited performance enhancement is due to the fact that the latency between the AGEN stage of a producer load and the execution (EXE) stage of its consumer indirect branch is often a small portion of the misprediction penalty, which starts from the fetch stage of the indirect branch. For higher performance gains we need to explore address-target correlation beyond the immediate producer-consumer pairs.

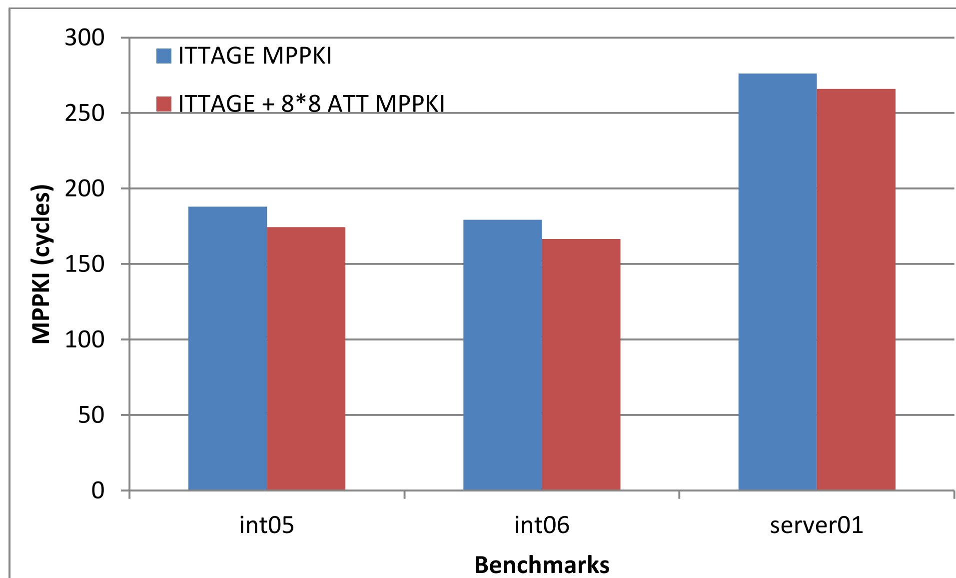

Figure 17 Comparison of MPPKI between ITTAGE and ITTAGE + 8*8 ATT for INT05, INT06 and SERVER01 benchmarks

3 Multi-Level Address Target Correlation

In previous chapter, we concluded that using one level of address-target correlation and looking at immediate load address to predict indirect branch outcome does reduce MPKI to almost zero but it does not give much improvements in terms of MPPKI. This phenomenon is shown in Figure 18. As shown in Figure 18, indirect branch is fetched at cycle 3734276 and is resolved in execution stage at 3734362. Thus if this branch was mispredicted, it will incur a penalty of 86 cycles. However with address-target correlation we can accurately predict the branch outcome during the address generation stage of the load instruction. In this case also we will not be able to predict target before cycle 3734356 which is just 6 cycles before the indirect branch finally gets resolved in execution stage.

Type	src1	src2	src3	dst	Target	srcval1	srcval2	srcval3	ld_addr	fetch cycle	execute cycle
Load	32	2	18	18	0	0	4c4540	1	4c4548	3734275	3734356
MOV	18	0	0	19	0	409080	0	0	0	3734276	3734358
MOV	30	19	0	19	0	33	409080	0	0	3734276	3734360
JMP	30	19	0	0	409080	33	409080	0	0	3734276	3734362

Figure 18 MPPKI gain for immediate Address Target Correlation

Thus, a major problem with single level of address-target correlation is that the correction for the mispredicted indirect branch arrives very late during the address generation stage of the load instruction. Latency between the address generation of the load instruction and execute stage of the indirect branch is only a small portion of the total penalty we will incur in case of the misprediction i.e. the latency between the fetch stage of the indirect branch and execute stage of the indirect branch. In above example, using address-target correlation we saved about 6 cycles which is only 7% of the total misprediction penalty.

3.1 Multi-Level Address Target Correlation Concept

As shown previously, looking at immediate load instruction for address-target correlation does not give much benefit in terms of MPPKI as the branch is resolved too late in the pipeline stage. However if we can find address target correlation beyond the immediate load instruction we may be able to get more benefits in terms of MPPKI. This gives motivation for the multi-level Address Target Correlation.

3.1.1 First Level of Address-Target Correlation

Figure 19 Instruction trace for First Level of ATC

As explained previously, we only look for correlation between load address and immediate indirect branch. In Figure 19, there is an address-target correlation between Load* R19 and

Indirect branch R19 instruction. Load instruction always reads from two different addresses which results in two different targets. There is very high probability that this relationship will be one-to-one and we will be able to predict branch target accurately. However Load* and indirect branch may not be apart by more than 5-10 cycles and hence there is not much benefit in terms of MPPKI.

3.1.2 Second Level of Address-Target Correlation

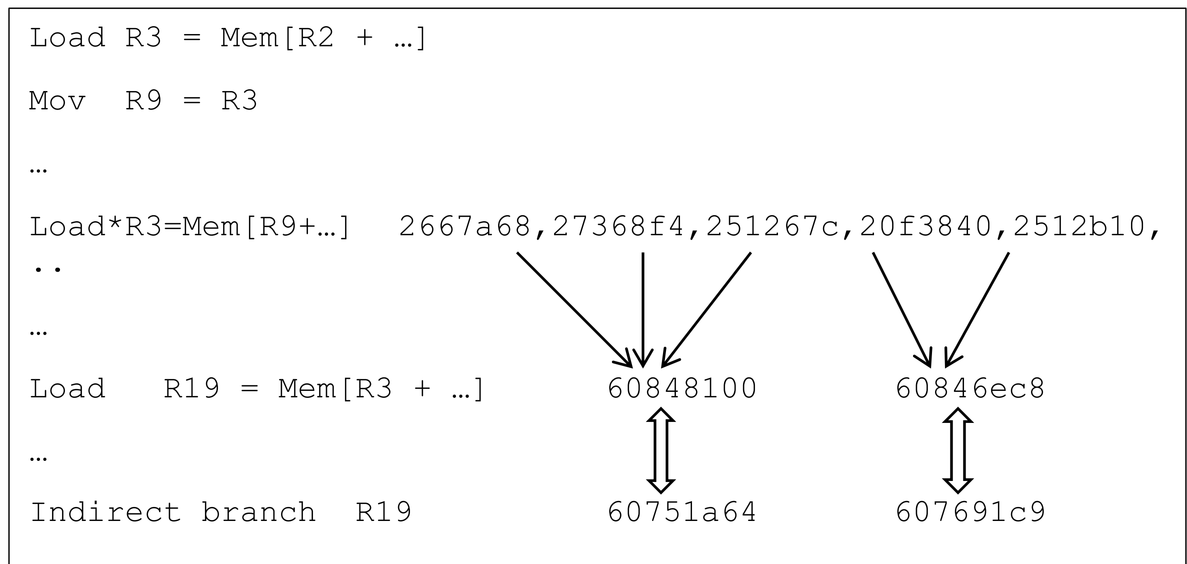

Figure 20 Instruction trace for Second Level of ATC

In Figure 20, we have address-target correlation between immediate load-indirect branch pair. But looking beyond we find that the Load R19 gets its address form the Load* R3 instruction. Load R3* reads from multiple addresses and based on that address, the Load R19 will get the address from which it will be reading. We can also see that in this example there is many-to-one relationship which is still useful for accurately predicting branch target. In

this case Load* instruction may be 10-20 cycles away from consumer branch instruction and hence if we can accurately predict branch target from this load address we can get more benefit compared to the first case. Also we may start observing many-to-one, many-to-many or one-to-many relationship at this level. Many-to-many or one-to-many relationships are not useful for branch prediction.

3.1.3 Third Level of Address-Target Correlation

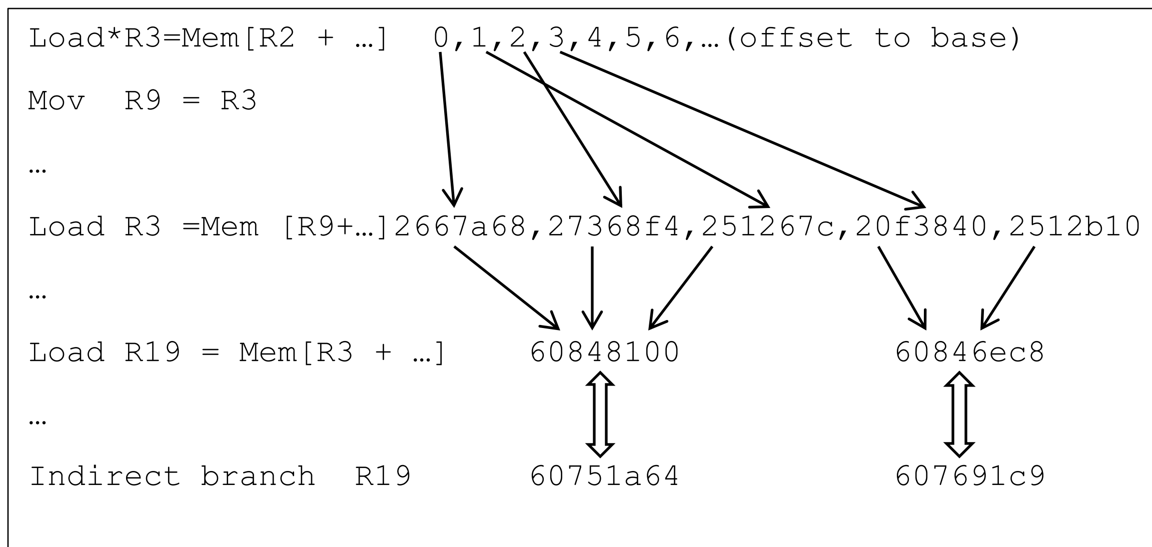

Figure 21 Instruction trace for Third Level of ATC

In Figure 21, we are looking beyond second level. At this level we see that Load* R3 provides values for address calculation for second level load. At this level Load* instruction may be 30-50 cycles apart and thus if we can use this load address to predict branch target, we may be able to get more performance benefit in terms of MPPKI. However, as we go away from the indirect branch for tracing address-target correlation we may not get one-to-one or many-to-one relationships which are required for accurate prediction.

3.2 Evaluating Multi-Level Address Target Correlation

We evaluated multi-level address target correlation for INT05, INT06 and SERVER01 benchmarks from CBP benchmarks. We observed that for INT05 and INT06 benchmarks, there is no address target correlation beyond immediate level. However SERVER01 benchmark has the potential for multi-level address target correlation.

PC	Type	src1	src2	src3	src4	dst	Target	LDST1	ld_addr	fetch cycle	execute cycle
408fcf	Load	32	2	0	18	14	0	2	5e5544	3734271	3734345
408fd7	MOV	14	0	0	0	13	0	0	0	3734271	3734347
408fda	MOV	13	16	0	0	13	0	0	0	3734271	3734349
408fdd	LOAD	32	2	17	0	19	0	adf49353	4da464c0	3734272	3734323
408fdf	MOV	13	19	0	0	19	0	0	0	3734272	3734351
408fe1	CB	95	30	2	0	0	408ff8	0	0	3734272	3734353
408ff8	Load	32	2	0	18	19	0	1	5e5944	3734273	3734345
408ffc	MOV	19	0	0	0	13	0	0	0	3734273	3734347
409000	MOV	14	0	0	0	18	0	0	0	3734273	3734347
409003	MOV	18	2	0	0	18	0	0	0	3734273	3734349
409007	MOV	18	2	0	0	19	0	0	0	3734274	3734351
40900b	CB	95	30	2	0	0	4090a0	0	0	3734274	3734354
409014	Load	32	2	0	18	19	0	409080	4c4548	3734275	3734356
40901e	JMP	30	19	0	0	0	409080	0	0	3734276	3734362

Figure 22 Instruction trace for SERVER01 benchmark

Figure 22 shows the queue of instructions which are available in address generation unit when load at 0x408fcf is encountered. We know from our previous observations that there is direct correlation between load at 0x409014 and branch at 0x40901e. Now we can see in the instruction trace that source 18 of load is the destination register of MOV instruction at 0x409003.

Now source register of this MOV gets its value from MOV at 0x409000. Source register of this MOV is the destination register of load instruction at 0x408fcf. Now we found that there is many-to-one correlation between load address of this instruction and target of the indirect branch at 0x40901e. Hence we can use the address of this load to predict the outcome of the indirect branch. Since this address will be available in the address generation stage of the load instruction at 0x408fcf, we can make a prediction at cycle 3734345. Now indirect branch at 0x40901e will get resolved in execution stage at cycle 3734362. As a result we can save almost 17 cycles in this case. As shown in Figure 23, this gives overall improvement of about 16 cycles for SERVER01 benchmark with 8*8 ATT.

Table 8 Improvements in MPPKI with Multi-level ATC for different predictor sizes

Predictor	MPPKI improvement with Immediate ATC	MPPKI improvement with Multi-level ATC
ITTAGE + 8*4 ATT	3.68% (10 cycles)	4.48% (12 cycles)
ITTAGE + 8*8 ATT	3.68% (10 cycles)	5.70% (16 cycles)
ITTAGE + 26*10 ATT	3.68% (10 cycles)	5.70% (16 cycles)

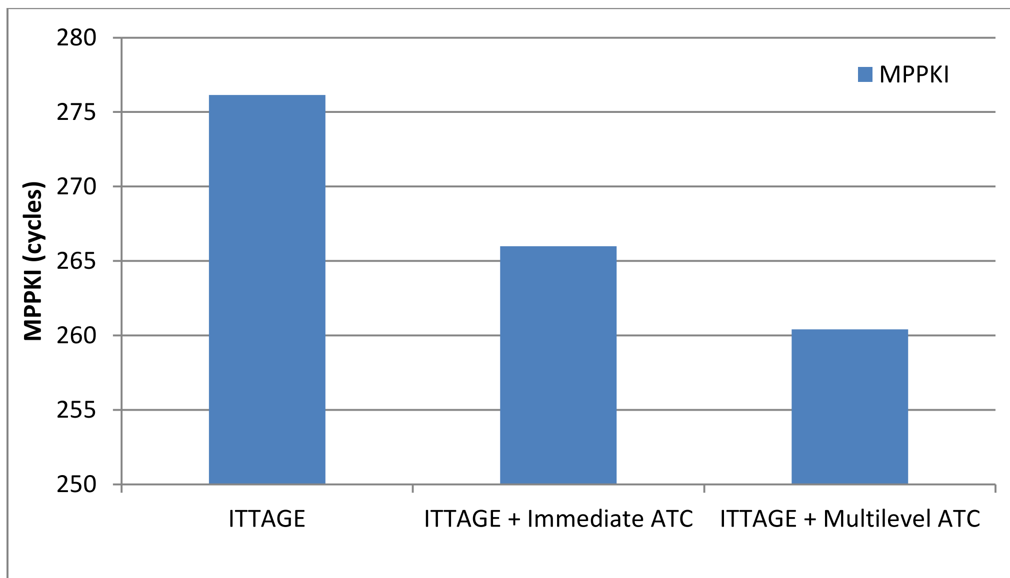

Figure 23 Improvements in MPPKI for Multi-level ATC

As shown in Table 8, with multi-level address target correlation we may be able to save 6 more cycles compared to immediate address target correlation for SERVER01 benchmark.

4 Compiler Assisted Address Target Correlation

Thus address-target correlation beyond immediate level has significant potential for performance improvement. We may be able to get more benefits if we can look further for the addresses that can give us hint about the outcome of the current indirect branch. Such correlation can be easily and effectively identified by the compiler based on high level dataflow information. Based on this observation, we argue that a compiler-microarchitecture co-operative approach can be more accurate and more efficient to trace address target correlation. Recent research [15] has shown value correlation is more effective than traditional history correlation for indirect branch prediction. However, there is a limitation for the value correlation in that it is very difficult to find proper data values that have strong correlation with branch targets. Previous predictors generally rely on large storage buffer [18], complex control logics [19] or expensive manual profiling [15] to identify proper value correlation, and they are usually ineffective even with aggressive hardware support. However, we propose to identify effective address target correlation based on high level dataflow information at compile time.

Compiler assisted address target correlation relies on the co-operation between the compiler and microarchitecture. First it relies on compiler to automatically identify the correlation between producer load addresses and consumer indirect branch. Second, it also relies on address target table to make a prediction. This extra hardware's storage requirement is small and control logic is also very simple. We will use PIN tool for trace generation and measurements of parameters such as MPKI. We will use SPEC2006 benchmarks to evaluate the compiler assisted address target correlation. In this case also we will use 64KB ITTAGE as our baseline predictor.

4.1 Instrumentation

4.1.1 Instruction trace analysis

After analyzing branch prediction behavior of ITTAGE for each benchmark, we have a list of indirect branches with a very high dynamic count and are very hard to predict. For these hard to predict indirect branches we generated instruction trace containing information such as program counter, opcode, source registers and their values, destination register and their values, if the instruction is load or store, load or store addresses and if there was a hit or a miss in I-cache or D-cache for this instruction. We used information from this dynamic instruction trace to get the understanding of the code that generates hard to predict branches. We also used the same instruction trace to track address target correlation between producer load addresses and consumer indirect branch. We also analyzed if there is one-to-one, many-to-one or one-to-many relationships between producer load address and consumer indirect branch. Section below shows code behavior of some of the hard to predict indirect branches.

4.1.1.1 Indirect Branch 0x404750

Indirect branch 0x404750 occurs in sjeng benchmark with a dynamic count of 129521508. The branch has misprediction rate of 4.17%. As shown in Figure 24 , the source code that generates the indirect branch 0x404750. As shown in the source code, this is an example of the indirect branch generated by “switch-case” statement. As shown in Figure 24, the indirect branch has strong correlation with the variable ‘j’. Array “pieces[j]” gets its value from different load addresses which are determined by variable ‘j’. These values are then assigned to variable ‘from’ which is then used to generate different load addresses for “board [from]”.

Thus there is address target correlation between load addresses for "pieces [j]' and the target of the indirect branch 0x404750.

```
if (white_to_move)  {
     for (a = 1, j = 1; (a <= piece_count) &&
          (((Variant != Suicide) && ! kingcap) ||
          ((Variant == Suicide) &&
          (fcaptures  == captures))); j++) {
               i = pieces[j];
               if (! i)
                    continue;
               else
                    a++;
               from = i;
               gfrom = i;
               switch (board[from]) {
                    case (wpawn):
```

4.1.1.2 Indirect Branch 0x40894b

Figure 26 shows a source code for the indirect branch 0x40894b which is example of the indirect branch generated by a pointer to function call. The branch has a dynamic count of 263142805 and has a misprediction rate of 4.6%. Different function pointer will be called based on the return value of function "piecet (i)". Variable 'i' gets its value from "pieces[j]".

These values are loaded from different load addresses which are determined by the variable ‘j’. Thus there is address target correlation between load addresses for array “pieces [j]” and function pointer calls at * (evalRoutines [piecet (i)]).

```
4046de: mov 4758268(%rip), %eax #88e1e0 <white_to_move>
4046e4: test %eax, %eax
4046e6: je   40487b <gen+0x1cb>
4046ec: mov 4767354(%rip), %eax # 89056c <piece_count>
4046f2: xor    %ebp, %ebp
4046f4: mov    $0x1, %r13d
4046fa: test   %eax, %eax
4046fc: jle    404822 <gen+0x172>
404702: mov 4759496(%rip), %ecx # 88e6d0 <Variant>
404708: $0x3, %ecx
40470b: je     404ba8 <gen+0x4f8>
404711: mov  2314921(%rip), %eax # 6399c0 <kingcap>
404717: test   %eax, %eax
404719: jne    404822 <gen+0x172>
40471f: mov    0x890b64 (, %rbp, 4), %ebx
404726: test   %ebx, %ebx
404728: je     404811 <gen+0x161>
40472e: movslq %ebx, %rdx
404731: add    $0x1, %r13d
404735: mov    %ebx, 2314893(%rip) # 6399c8 <gfrom>
40473b: cmpl   $0xb, 0x875bc0 (, %rdx, 4)
404743: ja     404811 <gen+0x161>
404749: mov    0x875bc0 (, %rdx, 4), %eax
404750: jmpq   *0x419f20 (, %rax, 8)
```

Figure 25 Assembly code for indirect branch 0x404750

```
for (j = 1, a = 1; (a <= piece_count); j++) {
      i = pieces[j];
      if (! i)
            continue;
      else
            a++;
      score+= (*(evalRoutines[piecet(i)]))(i,pieceside(i));
```

Figure 26 Source code for indirect branch 0x40894b

```
   408915:  add   $0x1, %rbp
   408919:  cmp   4750412(%rip),%r12d # 89056c <piece_count>
   408920:  jg 4090f9 <std_eval+0xaa9>
   408926:  mov 0x890b64 (, %rbp, 4),%edi
   40892d:  test %edi, %edi
   40892f:  je   408915 <std_eval+0x2c5>
   408931:  movslq %edi, %rbx
   408934:  add $0x1, %r12d
   408938:  mov 0x875bc0 (, %rbx, 4), %esi
   40893f:  add   $0x1, %esi
   408942:  mov %esi, %eax
   408944:  and $0x1, %esi
   408947:  sar %eax
   408949:  cltq
   40894b:  callq *0x41a320 (, %rax, 8)
```

Figure 27 Assembly code for indirect branch 0x40894b

4.2 Algorithm

Usually target of indirect branch is written to the register via some load instruction. If there is one to one or many to one correlation between this producer load instruction and consumer indirect branch we can exploit it to accurately predict the outcome of the branch instruction. The earlier in the dynamic instruction trace we identify such load instructions, the more benefits we can get in terms of MPPKI savings. We have developed such compiler algorithm that traces the address target correlation in the dynamic instruction trace and tries to correct the target prediction of indirect branch that was mispredicted by ITTAGE. The algorithm also tries to predict the address target correlation level from which to take a prediction based on one-to-one, many-to-one or one-to-many relationships that exist at that level between producer load addresses and consumer indirect branch.

4.2.1 Algorithm to trace address target correlation

The key steps in tracing algorithm are as followed:

- Fill in the instruction queue and start tracing algorithm if the last instruction is hard to predict indirect branch.
- Check if the source registers of the instruction pointed by cnt1 matches with the destination register of the instruction pointed by cnt2.
- If there is a match, use the source registers or the load address of the instruction pointed by cnt2 for prediction.
- Repeat the process till we reach the end of an instruction queue.

Figure 28 shows an algorithm to trace address target correlation for the instruction trace of given hard to predict indirect branch. We start filling in the instruction queue with the instructions that are executed. Also we keep on checking if the last instruction to be filled in was hard to predict branch. If the last instruction was hard to predict indirect branch we start tracing correlation. For that we maintain two pointers cnt1 and cnt2. Initially cnt1 points to indirect branch instruction and cnt2 points to instruction just preceding it. Then we check if the source registers of instruction pointed by cnt1 matches with the destination register of the instruction pointed by cnt2. If there is no match then we decrement cnt2 to point to the instruction preceding it whereas cnt1 will keep pointing to the same instruction. We keep on doing this until we find a match or we reach the end of queue or we reach the stop_level limit. Whenever there is a match, we store the load address or the source value of the instruction pointed by cnt2. We also store the branch address it points to. We will use this information for prediction. Now we assign cnt2 to cnt1 so that cnt1 points to that instruction and decrement cnt2 so that it points to instruction just preceding it. We repeat the same process till we reach the end of queue or we reach the stop_level. When we reach the end of queue or stop_level, we check if there was a match. If there was a match, we must have stored load address or the source value to be used for prediction. We use this value and the branch pc to access the address target table which provides the prediction. This is the same structure as shown in Figure 9 and the prediction mechanism works in the same way as explained in 2.1.1. We take prediction only when it is different from the one provided by ITTAGE. After taking prediction we also update the address target table using branch pc, producer load pc and actual branch outcome. The update mechanism is same as explained in 2.1.2.

```
set queue_size;

Fill in the instruction queue

If(current_instruction==hard_to_predict_indirect_branch)  {

     set cnt1 to queue_size - 1;

     for (cnt2 = cnt1 - 1;cnt2 >= stop_level;)  {

          if(instruction[cnt1].src==instruction[cnt2].dst)

               {

               prediction_src = instruction[cnt2].src ;

               set match1;

               set match2;

               }

          decrement cnt2;

          if match2 is set

               cnt1 = cnt2;

          else

               cnt1 = cnt1;

     }

     if match1 is set

          take prediction;

          update predictor based on actual branch outcome;

}
```

Figure 28 Algorithm to trace Address Target Correlation

4.2.2 Algorithm to determine stop_level

The key steps in stopping algorithm are as followed:

- Maintain a load table where each entry is the load address and the branch targets it points to. We maintain this information for each level of correlation.
- Check if at particular correlation level, there is one-to-one or many-to-one correlation between load address and branch targets.
- We start checking this information from the farthest level. If we find that there is one-to-many or many-to-many relation, we do not take prediction from that level and we come down one level.
- We repeat above steps till we find appropriate level where we can get one-to-one or many-to-one relations.

The parameter stop_level indicates the instruction beyond which we do not look for address target correlation. This may be because we have not found any useful load instruction beyond this level or the relationship between producer load addresses and consumer indirect branch is one-to-many or many-to-many which is not useful for address target correlation. We initially set this parameter to 1 indicating that we will look for address target correlation till the end of instruction queue. We also maintain table "ld_table []". Each entry in this table corresponds to one correlation level and contains a map whose key is the load address and the value is a vector containing all possible targets that a load address points to. We start tracing for address target correlation as explained in 4.2.1. Each time we find a match, we store the pointer cnt2 into the array level [] at the location pointed by cnt3. The address target correlation level is pointed by cnt3. So the statement level [cnt3] = cnt2 indicates that at the correlation level we found producer load address from an instruction pointed by cnt2. When there is a match, we also store the target into the vector pointed by current load address at current correlation level. Then we increment cnt3 to point to next level of correlation. Next time when the hard to predict indirect branch is encountered, we check if at each level the

load address was pointing to single target. If we find that there is no one-to-many relationship, we assign the instruction count stored at this level to the parameter stop_level. Thus stop_level now always points to the farthest instruction which provided the useful address-target correlation.

```
set queue_size;     Fill in the instruction queue;
if(current_instruction==hard_to_predict_indirect_branch)  {
        set stop_level = 1;
        for (i = 0; i < queue_size; i++)        {
                if (ld_table[i].load_address.size() < 2 &&
                        ld_table[i].load_address.size() != 0)
                                stop_level = level[i];
                }
        set cnt1 to queue_size - 1;
        cnt3 = 0;
        for (cnt2 = cnt1 - 1; cnt2 >= stop_level;)     {
                if(instruction[cnt1].src==instruction[cnt2].dst)      {
                        pred_src = instruction [cnt2].src;
                        level[cnt3] = cnt2;
                        ld_table[cnt3].load_address[pred_src].puch_back (target);
                        set match1;  set match2;
                        cnt3++;
                        }
                decrement cnt2;
                if match2 is set
                        cnt1 = cnt2;
                else
                        cnt1 = cnt1;
                }
        if match1 is set
                take prediction;
                update predictor based on actual branch outcome;
        }
```

Figure 29 Algorithm to detect stop_level

4.3 Case Study

PC	OPCODE	Source	Source Value	Dest.	Dest Value	Load Address
404a8c	ADD	rbp	1	**rbp**	2	0
404a90	CMP	r13d	3	rflags	297	89056c
404702	MOV	0	0	rcx	2	88e6d0
404708	CMP	rcx	2	rflags	297	0
40470b	JZ	rip	40470b	rip	404711	0
404711	MOV	0	0	rax	0	6399c0
404717	TEST	rax	0	rflags	246	0
404719	JNZ	rip	404719	rip	40471f	0
40471f	MOV	**rbp**	2	**rbx**	26	**890b6c**
404726	TEST	rbx	26	rflags	202	0
404728	JZ	rip	404728	rip	40472e	0
40472e	MOVSXD	**rbx**	26	rdx	26	0
404731	ADD	r13d	3	rflags	4	0
404735	MOV	rbx	26	rip	0	0
40473b	CMP	rdx	26	rflags	297	875c58
404743	JNBE	rip	404743	rip	404749	0
404749	MOV	rdx	26	**rax**	1	**875c58**
404750	JMP	**rax**	1	rip	0	419f28

Figure 30 instruction trace for Indirect Branch 0x404750

As mentioned earlier in 4.2.1 we maintain an instruction queue to trace address target correlation. We start looking for address target correlation when the last instruction filled into the instruction queue was hard to predict indirect branch. Figure 30 shows the state of the instruction queue at such moment. In this case hard to predict indirect branch is 0x404750. Source code corresponding to this instruction is shown in Figure 24. For the purpose of this example assume that queue_size is 18. So cnt1 is initially set to 17, cnt2 is set to 16 and cnt3 is set to 0. At the moment when the instruction 0x404750 is filled into the queue, cnt1 is pointing to instruction 0x404750 and cnt2 is pointing to instruction 0x404749. We find that source register of 0x404750 is same as the destination register of the 0x404749. Therefore we can use the load address of instruction 0x404749 for branch prediction. Since there is a match we store the load address and the target it is pointing at correlation level zero. Then we increment cnt3 to indicate correlation level one. We set cnt1 to point to 0x404749 and we decrement cnt2. At this point we do not find match between source register of 0x404749 and destination register of 0x404743. So in this case cnt1 and cnt3 will remain same and we will only decrement cnt2 so that it points to 0x40473b.We keep repeating the same process until we find a match. When the cnt2 is pointing to 0x40472e, there is a match between the source register of 0x404749 and the destination register of the 0x40472e. So we can use source value of 0x40472e for prediction. We also store the source value and the target it is pointing at correlation level one and then we increment cnt3 to point to correlation level two. Since there was a match, cnt1 will now point to 0x40472e and cnt2 will now point to 0x404728. The next match occurs when the cnt2 is pointing to 0x40471f and cnt1 is pointing to 0x40472e. We can use the load address of the instruction 0x40471f to make prediction. We also store the load address and the target it is pointing at correlation level two. Then we increment cnt3. We set cnt1 to point to 0x40471f and we decrement cnt2 by one. We repeat the same process again till we find a match when cnt2 is pointing to 0x404a8c. We can use the source value of 0x404a8c to make a prediction. We also store this source value and the target it is pointing at correlation level three. As a result we will have information as shown in Figure 31 at the end of trace.

Back trace level 0			
Load Address	Target		
875c30	404757		
875c34	404a33		
875c58	4048e6		
875c5c	4048e6		
875c60	4048e6	404811	404757
875c74	4048e6		
875c8c	404811	404aa2	
875c90	404811		
875ca0	404aa2		
875ca4	404b66		

Back Trace Level 1			
Source value	Target		
1c	404757		
1d	404a33		
26	4048e6		
27	4048e6		
28	4048e6	404811	404757
2d	4048e6		
33	404811	404aa2	
34	404811		
38	404aa2		
39	404b66		

Back Trace Level 2		Back Trace Level 3	
Load Address	Target	Source value	Target
890b64	404757	0	404a33
890b68	404a33	1	4048e6
890b6c	4048e6	2	4048e6
890b70	4048e6	3	4048e6
890b74	4048e6	4	4048e6
890b78	4048e6	5	404aa2
890b7c	404aa2	6	404b66
890b80	404b66	7	404811
890b84	404811	8	4048e6
890b88	4048e6	9	404811
890b8c	404811	a	404811
890b90	404811	b	404811
890b94	404811	c	4048e6
890b98	4048e6	e	404811
890ba0	404811	f	404811
890ba4	404811	10	404a33
890ba8	404a33	11	404811
890bac	404811	12	404811
890bb0	404811	14	404811

Figure 31 Address target correlation at each level

As shown in Figure 31, at back trace level zero load address 0x875c30 only points to target 0x404757. However, load address 0x875c60 points to multiple targets. So is the case with load address 0x875c8c. Hence there exists one-to-many relationship at this level. Similarly at back trace level 1, there are one-to-many relationships between some of the source values and branch targets. However, at back trace level 2 there is one-to-one relationship between produce load address and consumer branch targets and we can accurately predict branch outcome from this level. Same is the case with back trace level 3, where there is one-to-one relationship between source values and branch targets.

We will use this information to determine the stop_level for our algorithm. Since we always initiate stop_level to one, we tend to search entire queue for address target correlation. In this case as explained earlier array level [] will have values {16, 11, 8, 0}. So in algorithm 4.2.2 we will find that at correlation level zero and one there is one-to-many relationship and hence stop_level will not be assigned values 16 or 11. At correlation level 2 and 3, there is one-to-one relationship and stop_level will be assigned values 8 and 0. Since zero is the last value assigned to stop_level, in this case we will search for entire queue.

5 Evaluating Compiler Assisted ATC

5.1 Methodology

For the experiments in this section we used SPEC2006 benchmarks. We used PIN tool from Intel for instrumentation. Pin is a dynamic binary instrumentation framework for the IA-32 and x86-64 instruction-set architectures that enables the creation of dynamic program analysis tools. We used PIN:

- To get the branch prediction results for SPEC2006 benchmarks using 64KB ITTAGE as an indirect branch predictor. We used these results to identify hard to predict branches.
- To generate the instruction trace for the given benchmark binary so that we can trace address target correlation using our algorithm.
- To get the final branch prediction results for SPEC2006 benchmarks using 64 KB ITTAGE with Address Target Table as an indirect branch Predictor.

After simulating all SPEC2006 benchmarks with ITTAGE as our base Indirect Branch Predictor, we have list of indirect branches with a very high dynamic count and are very hard to predict. For these hard to predict indirect branches we generated instruction trace containing information such as program counter, opcode, source registers and their values, destination register and their values, if the instruction is load or store, load or store addresses and if there was hit or miss in I-cache or D-cache for this instruction. To get the average access time between producer load access and consumer indirect branch, we used three level cache structures with the parameters shown in Table 9. Average access time for main memory was considered 200 cycles. We used information from this dynamic instruction trace

to get the understanding of the code that generates hard to predict branches. We also used the same instruction trace to track address target correlation between producer load addresses and consumer indirect branch. We also analyzed if there is one-to-one, many-to-one or one-to-many relationships between produce load address and consumer indirect branch.

Table 9 Cache Parameters used for PIN Instrumentation

L1 Instruction Cache	
Cache Size	32 KB
Block Size	64
Associativity	2
Access Time	2 cycles
L1 Data Cache	
Cache Size	32 KB
Block Size	64
Associativity	2
Access Time	2 cycles
Unified L2 Cache	
Cache Size	2 MB
Block Size	64
Associativity	1
Access Time	10 cycles
Unified L3 Cache	
Cache Size	16 MB
Block Size	64
Associativity	1
Access Time	40 cycles

5.2 ITTAGE MPKI results and analysis

We replaced the default BTB predictor provided by the PIN tool with 64 KB ITTAGE predictor. We collected following data for SPEC2006 benchmarks:

- Total dynamic count of each indirect branch
- Misprediction count for that benchmark
- MPKI for that benchmark

Figure 32 shows benchmarks with highest MPKI with ITTAGE indirect predictor. MPKI for all benchmarks is shown in Table 10. There is some interesting behavior seen with some of the benchmarks.

- As shown in Table 10, gamess benchmark has the highest MPKI. However the benchmark has a very low dynamic count of indirect branch and even if we predict each branch correctly, there is not much scope for performance improvement.
- Sjeng benchmark has the second highest MPKI. Also it has a very high dynamic count of indirect branch and many of these branches are hard to predict. So it offers good opportunity for performance improvement. Same is the case with dealII, omnetpp, povray, gobmk and h264ref benchmarks.
- MPKI of benchmarks hmmer and xalanc is not as high but there are some hard to predict indirect branches with high dynamic count.
- Lbm and calculix benchmarks have relatively high MPKI but there are not any hard to predict indirect branch with high dynamic count and hence they do not offer much opportunity for performance improvement.
- Soplex and bzip have relatively low MPKI but there are some hard to predict indirect branches with high occurrence and thus they provide some opportunity for performance improvement.

- Benchmarks gromacs, astar, gems, mcf, zeusmp, milc, namd, leslie and bwaves all have very low MPKI and do not offer any opportunity for performance improvement.

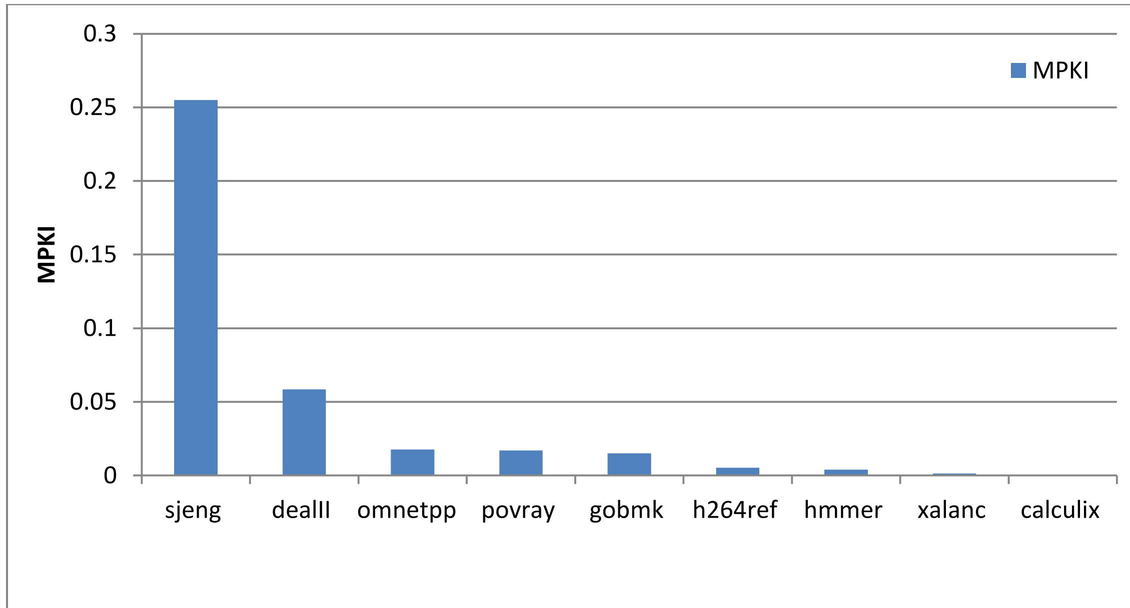

Figure 32 most hard to predict benchmarks for ITTAGE

Table 10 MPKI for SPEC2006 benchmarks with ITTAGE

Benchmark	MPKI	Benchmark	MPKI
gamess	0.557434	soplex	8.32E-05
sjeng	0.254975	bzip	5.16E-05
dealII	0.0584253	gromacs	4.15E-06
omnetpp	0.0175544	astar	9.60E-07
povray	0.0170141	gems	4.49E-07
gobmk	0.0150332	mcf	4.04E-07
h264ref	0.00534244	zeusmp	2.57E-07
hmmer	0.00394453	milc	2.38E-07
xalanc	0.00145833	namd	2.16E-07
lbm	0.000278143	leslie	1.72E-07
calculix	0.000177632	bwaves	9.53E-08

5.3 Benchmark Analysis

Though benchmarks dealII, omnetpp, povray and gobmk have high MPKI and some branches with very high misprediction rate, these branches do not exhibit address target correlation. Some of these branches are explained with code examples in the following sections. Benchmark sjeng has highest MPKI and many of its hard to predict branches exhibit address target correlation.

5.3.1 Omnetpp

Some of the hard to predict branches for omnetpp benchmark are as shown in Table 11. These branches have been classified as hard to predict based on their high misprediction count or high misprediction rate.

Table 11 Hard to predict branches for omnetpp benchmark

Branch PC	Dynamic Count	Misprediction Count
0x4057cb	3194694	97442
0x40e9e6	520435368	6070981
0x40f7a4	2574274	80384
0x410e9a	150937466	225799
0x45b9e6	276949108	3298492

As mentioned earlier, these branches do not exhibit address target correlation. This phenomenon is explained using one of the branches with its corresponding source code and assembly code in the next subsection.

5.3.1.1 Indirect Branch 0x40e9e6

```
void EtherMAC::printState()
{
#define CASE(x) case x: EV << #x; break
    EV << ", receiveState: ";
    switch (receiveState) {
        CASE (RX_IDLE_STATE);
        CASE (RECEIVING_STATE);
        CASE (RX_COLLISION_STATE);
    }
    EV << ", backoffs: " << backoffs << endl;
#undef CASE
}
```

Figure 33 Source code for indirect branch 0x40e9e6

Figure 33 shows source code for branch 0x40e9e6 and the corresponding assembly code are shown in Figure 34. In address target correlation we look for the correlation between producer load addresses and consumer indirect branches. However, in this case the outcome of the "switch" statement depends on the value of the variable "receiveState" which will be given that value somewhere in the code. So this is the concept used by Value Based BTB Indexing for indirect branch prediction as explained in 1.3.1 and not the address target correlation.

```
40e9c0:   push   %rbp
40e9c1:   mov    %rdi, %rbp
40e9c4:   push   %rbx
40e9c5:   sub    $0x8, %rsp
40e9c9:   mov    2674041(%rip), %eax # 69b748 <ev +0x8>
40e9cf:   test   %eax, %eax
40e9d1:   je     40ea80 <EtherMAC::printState () + 0xc0>
40e9d7:   cmpl   $0x6, 0x2cc (%rbp)
40e9de:   ja     40ea08 <EtherMAC::printState () + 0x48>
40e9e0:   mov    0x2cc (%rbp), %eax
40e9e6:   jmpq   *0x46d458(, %rax, 8)
```

Figure 34 Assembly code for indirect branch 0x40e9e6

5.3.2 Povray

Some of the hard to predict branches for povray benchmark are as shown in Table 12. These branches have been classified as hard to predict based on their high misprediction count or high misprediction rate. As mentioned earlier, these branches do not exhibit address target correlation. This phenomenon is explained using one of the branches with its corresponding source code and assembly code in the next subsection.

Table 12 Hard to predict branches for povray benchmark

Branch PC	Total Dynamic Count	Misprediction Count
0x411dc1	4136427432	4082519
0x442d52	18227834	1385094
0x45654e	2186191999	5871880
0x4566d5	514757454	5154863
0x4735aa	30703314	487893

5.3.2.1 Indirect Branch 0x4735aa

```
DBL   Evaluate_TPat   (TPATTERN   *TPat,   VECTOR   EPoint,
INTERSECTION *Isection)

{

    switch (TPat->Type)

    {

        case  AGATE_PATTERN:          value  =  agate_pattern
(EPoint, TPat);   break;

     ...

     }

}
```

Figure 35 Source code for indirect branch 0x4735aa

Figure 35 shows source code for branch 0x40e9e6 and the corresponding assembly code is shown in Figure 36. In address target correlation we look for the correlation between producer load addresses and consumer indirect branches. However, in this case the outcome of the “switch” statement depends on the value of the variable “Type” which will be given that value somewhere in the code. So this is the concept used by Value Based BTB Indexing

For indirect branch prediction as explained in 1.3.1 and not the address target correlation.

```
473570:   mov     %rbp, 0xffffffffffffffd8 (%rsp)
473575:   mov     %r12, 0xffffffffffffffe0 (%rsp)
47357a:   mov     %rsi, %rbp
47357d:   mov     %rbx, 0xffffffffffffffd0 (%rsp)
473582:   mov     %r13, 0xffffffffffffffe8 (%rsp)
473587:   mov     %rdi, %r12
47358a:   mov     %r14, 0xfffffffffffffff0 (%rsp)
47358f:   mov     %r15, 0xfffffffffffffff8 (%rsp)
473594:   sub     $0x288, %rsp
47359b:   movzwl  (%rdi), %eax
47359e:   sub     $0x5, %eax
4735a1:   cmp     $0x2d, %ax
4735a5:   ja      4735b1
4735a7:   movzwl  %ax, %eax
4735aa:   jmpq    *0x4d0a28 (, %rax, 8)
```

Figure 36 Assembly code for indirect branch 0x4735aa

5.3.3 H264ref

Some of the hard to predict branches for h264ref benchmark are as shown in Table 13. The branch has been classified as hard to predict based on its high misprediction count. As mentioned earlier, this branch does not exhibit address target correlation. This phenomenon is explained using corresponding source code and assembly code in the next subsection.

Table 13 Hard to predict branches for h264ref benchmark

Branch PC	Total Dynamic Count	Misprediction Count
0x43fa34	1975097520	2524634

5.3.3.1 Indirect Branch 0x43fa34

```
orgptr = orig_blocks;
bindex = 0;
for (blky = 0; blky < 4; blky++)
     {
     LineSadBlk0=LineSadBlk1=LineSadBlk2=LineSadBlk3 = 0;
     for (y = 0; y < 4; y++)
          {
          refptr = PelYline_11 (ref_pic, abs_y++, abs_x,
img_height, img_width);
```

Figure 37 Source code for indirect branch 0x43fa34

Figure 37 shows source code for branch 0x43fa34 and the corresponding assembly code is shown in Figure 38. In address target correlation we look for the correlation between producer load addresses and consumer indirect branches. However, we were not able to find any such correlation for this branch. We were also not able to find particular program statement which causes this branch to behave this way.

```
43fa11:   mov    %r9, 0x30 (%rsp)
43fa16:   mov    %r10, 0x28 (%rsp)
43fa1b:   mov    %r11d, 0x20 (%rsp)
43fa20:   mov    %eax, 0x9c (%rsp)
43fa27:   mov    0x54(%rsp), %r8d
43fa2c:   mov    0x88(%rsp), %rdi
43fa34:   callq  *2362742(%rip)     # 6807b0 <PelYline_11>
```

Figure 38 Assembly code for indirect branch 0x42a054

From above analysis it is clear that benchmarks omnetpp, povray, h264ref and gobmk do not get benefits from address target correlation. The benchmark that benefits most from address target correlation is sjeng. So we present the results for the sjeng benchmark in the following section.

5.4 Performance of Address Target Correlation

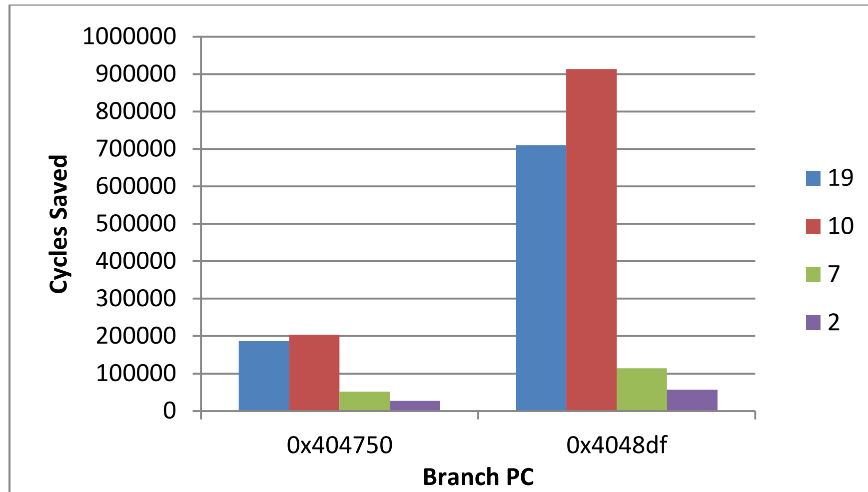

Figure 39 Cycles saved for branches 0x404750 and 0x4048df

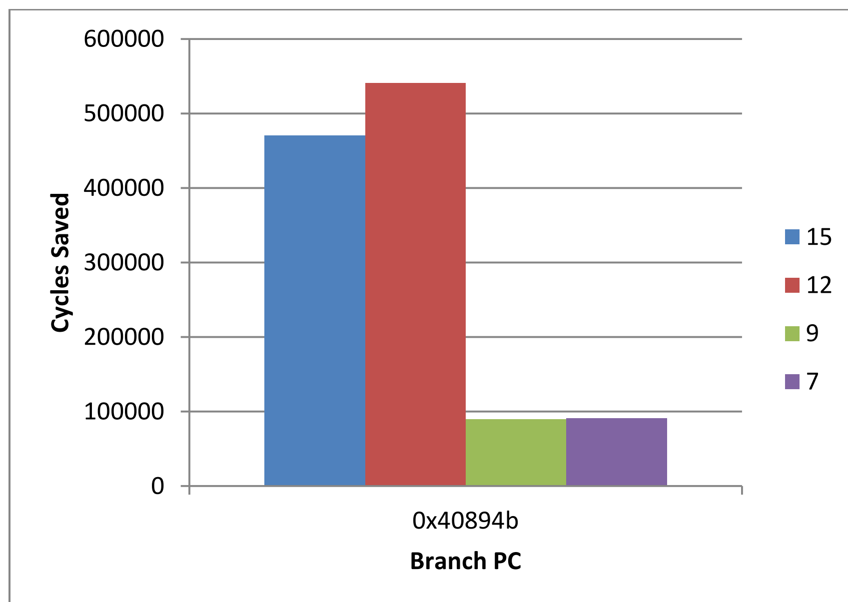

Figure 40 Cycles saved for branch 0x40894b

As discussed in 5.3, sjeng benchmark benefits most from Address Target Correlation. For this benchmark we identified hard to predict indirect branches and applied the ATC algorithm with 3 bit confidence counter to get the cycles saved in case of each branch. Figure 39 shows cycles saved for branches 0x404750 and 0x4048df while looking back 19, 10, 7 and 2 instructions. Same information is shown in Table 14. For both branches as mentioned in Figure 31, there is one-to-one correlation between load address and branch target at level 3 and level 2 (data structure at load addresses for these levels are relatively stable). As a result, we can save more cycles while looking 19 and 10 instructions back. However for level 1 and level 0 there is one-to-many relation between load addresses and branch targets. This explains huge drop in cycles saved from looking back 10 instructions to looking back 7 instructions. Same is the case when we look back for 2 instructions.

Figure 40 shows cycles saved for branch 0x40894b while looking back 15, 12, 9 and 7 instructions back.

Table 14 Cycles saved for branches 0x404750, 0x4048df and 0x40894b with 3-bits confidence counter

PC	Number of Instructions Traced Back	Cycles Saved
0x404750	19	187038
	10	204116
	7	51948
	2	26696
0x4048df	19	710298
	10	913812
	7	114136
	2	57294
0x40894b	15	470320
	12	540764
	9	89760
	7	90588
	6	0
	5	0
	3	0
	2	0

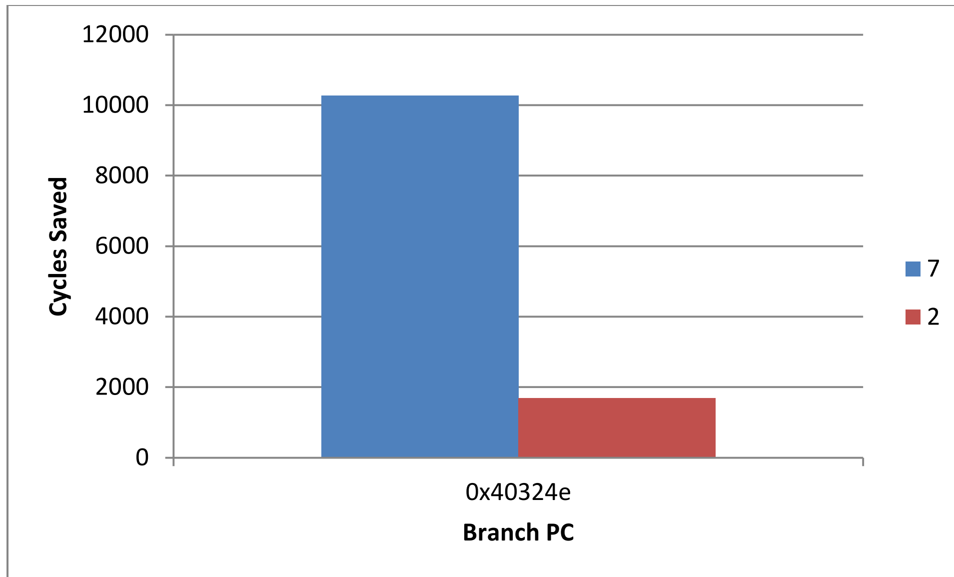

Figure 41 Cycles saved for branch 0x40324e

Table 15 Cycles saved for branch 0x40324e with 3-bits confidence counter

PC	Number of Instructions Traced Back	Cycles Saved
0x404750	12	0
	8	0
	7	10272
	2	1684

For hmmer benchmark we identified hard to predict indirect branches and applied the ATC algorithm with 3 bit confidence counter to get the cycles saved in case of each branch. Figure 41 shows cycles saved for branch 0x40324e while looking back 7 and 2 instructions. Same information is shown in Table 15.

5.4.1 Case Study

It is clear from above discussion that we are not getting the expected performance improvements in terms of cycles saved. In order to find the reason behind the limited gain we used 2 and 3 bits confidence counter. The results in Table 14 are obtained by using 3 bits confidence counter. We obtained the same results for 2 bits confidence counter. These results are shown in Table 16. Still we did not get any improvements. The table size cannot be a problem because we are using maps to implement the load table structure thus dynamically changing the table size according to requirements. In order to pinpoint the problem we did the detailed analysis for branch 0x404750 which is discussed in this section.

Table 16 Cycles saved for branches 0x404750, 0x4048df and 0x40894b with 2-bits confidence counter

PC	Number of Instructions Traced Back	Cycles Saved
0x404750	19	788874
	10	880534
	7	317298
	2	158538
0x4048df	19	1863984
	10	2288670
	7	520756
	2	261810
0x40894b	15	1562936
	12	1792800
	9	526960
	7	529292
	6	48
	5	72
	3	64
	2	8

Table 17 Analysis of branch 0x404750 for 3-bits confidence counter

Total Dynamic Count	129521508
Misprediction by ITTAGE	5397520
ATC Prediction when ITTAGE Mispredicts	46540
ATC Correct Prediction when ITTAGE Mispredicts	34011
ATC Prediction when ITTAGE Correct	612525
ATC Correct Prediction when ITTAGE Correct	592857
Wrong Corrections by ATC	19668
Cycles Saved	204116
Accuracy	0.951147

Table 17 shows analysis for branch 0x404750 for 3-bits confidence counter. The important entry to note is number of wrong corrections made by ATC. These are the target prediction changes made by ATC when ITTAGE has predicted correctly. These wrong corrections impact the performance adversely as the penalty of wrong correction is a lot more than the cycles saved by correcting mispredictions. Also we observed in Figure 31 that there is one to one correlation at level 3 and level 2 and as a result we should not have such a high number of mispredictions in the first place. In order to explain such behavior we extended the profiling period by 100 million instructions and obtained the same load tables we observed in Figure 31.

This table is shown in Table 18. We can see that now load address 0x890b98 points to two different targets. This behavior was not captured in the initial profiling period of 100 million instructions. As we went on increasing the profiling period we observed that the load addresses starts pointing to more and more branch targets which are the cause of mispredictions.

Table 18 Address target correlation level 2 and 3 for profiling period of 200 million instructions

<table>
<tr><th colspan="3">Back Trace Level 2</th><th colspan="3">Back Trace Level 3</th></tr>
<tr><th>Load Address</th><th colspan="2">Target</th><th>Source value</th><th colspan="2">Target</th></tr>
<tr><td>890b64</td><td colspan="2">404757</td><td>0</td><td colspan="2">404a33</td></tr>
<tr><td>890b68</td><td colspan="2">404a33</td><td>1</td><td colspan="2">4048e6</td></tr>
<tr><td>890b6c</td><td colspan="2">4048e6</td><td>2</td><td colspan="2">4048e6</td></tr>
<tr><td>890b70</td><td colspan="2">4048e6</td><td>3</td><td colspan="2">4048e6</td></tr>
<tr><td>890b74</td><td colspan="2">4048e6</td><td>4</td><td colspan="2">4048e6</td></tr>
<tr><td>890b78</td><td colspan="2">4048e6</td><td>5</td><td colspan="2">404aa2</td></tr>
<tr><td>890b7c</td><td colspan="2">404aa2</td><td>6</td><td colspan="2">404b66</td></tr>
<tr><td>890b80</td><td colspan="2">404b66</td><td>7</td><td colspan="2">404811</td></tr>
<tr><td>890b84</td><td colspan="2">404811</td><td>8</td><td colspan="2">4048e6</td></tr>
<tr><td>890b88</td><td colspan="2">4048e6</td><td>9</td><td colspan="2">404811</td></tr>
<tr><td>890b8c</td><td colspan="2">404811</td><td>a</td><td colspan="2">404811</td></tr>
<tr><td>890b90</td><td colspan="2">404811</td><td>b</td><td colspan="2">404811</td></tr>
<tr><td>890b94</td><td colspan="2">404811</td><td>c</td><td>4048e6</td><td>404aa2</td></tr>
<tr><td>890b98</td><td>4048e6</td><td>404aa2</td><td>e</td><td colspan="2">404811</td></tr>
<tr><td>890ba0</td><td colspan="2">404811</td><td>f</td><td colspan="2">404811</td></tr>
<tr><td>890ba4</td><td colspan="2">404811</td><td>10</td><td colspan="2">404a33</td></tr>
<tr><td>890ba8</td><td colspan="2">404a33</td><td>11</td><td colspan="2">404811</td></tr>
<tr><td>890bac</td><td colspan="2">404811</td><td>12</td><td colspan="2">404811</td></tr>
<tr><td>890bb0</td><td colspan="2">404811</td><td>14</td><td colspan="2">404811</td></tr>
</table>

In order to understand the cause of mispredictions we observed the instruction trace for an extended profiling period. This instruction trace is shown in Figure 42. As shown in trace, at first the value at load address 0x890b98 was 0x75. We were using load address with this value to make a prediction. After some instructions, there is a store to the same address which modifies the value at that address to 0x68. So next time when we use the same load address, the branch target is mispredicted.

PC	OPCODE	Source	Source Value	Destination	Destination Value	Load/Store Address
40471f	MOV	**rbp**	d	**rbx**	75	**890b98**
.		.		.		.
.		.		.		.
405725	MOV	**rbx**	68	**rax**	e	**890b98**
.		.		.		.
.		.		.		.
40471f	MOV	**rbp**	d	**rbx**	68	**890b98**
404726	TEST	rbx	68	rflags	202	0
404728	JZ	rip	404728	rip	40472e	0
40472e	MOVSXD	**rbx**	68	**rdx**	68	0
404731	ADD	r13d	d	rflags	e	0
404735	MOV	rbx	68	rip	0	0
40473b	CMP	rdx	68	rflags	297	875d60
404743	JNBE	rip	404743	rip	404749	0
404749	MOV	**rdx**	68	**rax**	1	**875d60**
404750	JMP	**rax**	1	rip	0	419f28

Figure 42 Instruction trace of branch 0x404750 showing cause of mispredictions

Thus every time a store modifies the content of the memory location, branch target will be mispredicted next time we use same address for prediction. We can avoid these mispredictions if we use Explicit Dynamic Branch Prediction with Active Updates (EXACT) predictor where store to an address directly updates its prediction in the predictor.

5.5 Summary

In section 5, we evaluated the compiler assisted address target correlation using PIN tool and SPEC2006 benchmarks. First we obtained the results for 64 KB ITTAGE to determine the hard to predict indirect branches. We studied these hard to predict indirect branches to check if they exhibits address target correlation. We found that only hard to predict branches of benchmarks sjeng and hmmer exhibits this behavior. Then we obtained the results for hard to predict branches of sjeng and hmmer benchmarks and found that we are not getting the expected performance benefits. So we did the detailed analysis of these branches to pinpoint the problem limiting the performance gain. We found that the store at the address which we are using to predict the branch target modifies the content of that address which causes the misprediction.

Please note that all the results in section 5.4 were obtained for sjeng and hmmer benchmarks only. More results would be needed to make a final conclusion.

6 Conclusions

Indirect branches are the leading cause of mispredictions for the state-of-the-art branch predictors like ITTAGE. ITTAGE uses large global and local history information to accurately predict Indirect Branches. But there are still some branches that are very hard to predict. As studied in 2.1.1, there is correlation between producer load address and consumer indirect branches.

We proposed hardware based and compiler assisted address target correlation to exploit the relationship between producer load addresses and consumer indirect branches. In hardware based solution we checked for the load instruction immediately above indirect branch. We used load address of this instruction to predict the target of the indirect branch during the address generation stage of the load. We showed that though we can get great improvements in terms of MPKI, performance improvements in terms of MPPKI is very less because of low latency between the address generation stage of the load instruction and the execute stage of the indirect branch.

To remedy this problem we then suggested that if we look for the address target correlation beyond immediate load-indirect branch pair, we can get more benefits in terms of MPPKI. We then suggested compiler assisted address target correlation to detect multiple levels of correlation. We developed the algorithm to detect multi-level address target correlation. We evaluated compiler assisted address target correlation in terms of number of cycles it saved while looking back certain instructions.

Our results showed that for sjeng benchmark, performance of compiler assisted address target correlation is limited because of the store instructions modifying the load addresses which we are using for branch target prediction.

7 Bibliography

[1] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," Intel Corporation,[Online].Available: http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html/. [Accessed 8 8 2012].

[2] L. Cardelli and P. Wegner, "On understanding types, data abstraction, and polymorphism.," *ACM Computing Surveys,* vol. 4, no. 17, pp. 471-523, 1985.

[3] D. R. Kaeli and P. G. Emma, "Improving the Accuracy of History-Based Branch Prediction," *IEEE Transactions on Computers,* pp. 469-472, 1997.

[4] J. Lee and A. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *Computer,* pp. 6-22, 1984.

[5] P. Chang, E. Hao and Y. N. Patt, "Target Prediction for Indirect Jumps," *ISCA-24,* pp. 274-283, 1997.

[6] T. Li, R. Bhargava and L. K. John, "Rehashable BTB: An Adaptive Branch Target Buffer to Improve the Target Predictability of Java Code," in *HiPC-02*, 2002.

[7] K. Driesen and U. Holzle, "The Cascaded Predictor: Economical and Adaptive Branch Target Prediction," *MICRO-31,* pp. 249-258, 1998.

[8] K. Driesen and U. Holzle, "Multi-stage Cascaded Prediction," in *Euro-Par*, 1999.

[9] J. Kalamatianos and D. R. Kaeli, "Predicting Indirect Branches via Data Compression," *MICRO-31,* pp. 272-281, 1998.

[10] H. Kim, J. A. Joao, O. Mutlu, J. Lee, Y. N. Patt and R. Cohn, "VPC Prediction: Reducing the Cost of Indirect Branches via Hardware-Based Dynamic Devirtualization," *ISCA-34,* pp. 424-435, 2007.

[11] A. Roth, A. Moshovos and G. S. Sohi, "Improving Virtual Function Call Target Prediction via Dependence-Based PreComputation," *ICS-13,* pp. 356-364, 1999.

[12] D. R. Kaeli and P. G. Emma, "Improving the Accuracy of History-Based Branch Prediction," *IEEE Transactions on Computers,* pp. 469-472, 1997.

[13] J. A. Joao, O. Mutlu, H. Kim, R. Agarwal and Y. N. Patt, "Improving the Performance of Object-Oriented Languages with Dynamic Predication of Indirect Jumps," *ASPLOS-13,* pp. 80-90, 2008.

[14] A. Seznec, "A 64-kbytes ITTAGE indirect branch predictor," in *3rd Championship Branch Prediction*, 2011.

[15] M. U. Farooq, L. Chen and L. K. John, "Value Based BTB Indexing for indirect jump prediction," *HPCA-16,* pp. 1-11, 2010.

[16] M. Tan, X. Liu, D. Tong and X. Cheng, "An Energy-Efficient Indirect Branch Prediction with Compiler-Guided Value Pattern," *ICS-12,* 2012.

[17] H. Gao, Y. Ma, M. Dimitrov and H. Zhou, "Address-Branch Correlation: A Novel Locality for Long-Latency Hard-to-Predict Branches," *HPCA-14,* pp. 74-85, 2008.

[18] R. Thomas, M. Franklin, C. Wilkerson and J. Stark, "Improving branch prediction by dynamic dataflow-based identification of correlated branches from a large global history," *ISCA-30,* pp. 314-323, 2003.

[19] L. Chen, S. Dropsho and H. Albonesi, "Dynamic data dependence tracking and its application to branch prediction," *HPCA-9,* pp. 65-76, 2003.