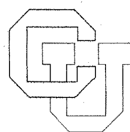
Hardware and Software Mechanisms for Instruction Fetch Prediction

Bradley Gene Calder

CU-CS-781-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

ANY OPINIONS, FINDINGS, AND CONCLUSIONS OR RECOMMENDATIONS EXPRESSED IN THIS PUBLICATION ARE THOSE OF THE AUTHOR(S) AND DO NOT NECESSARILY REFLECT THE VIEWS OF THE AGENCIES NAMED IN THE ACKNOWLEDGMENTS SECTION.

HARDWARE AND SOFTWARE MECHANISMS FOR INSTRUCTION FETCH PREDICTION

by

BRADLEY GENE CALDER

B.S. Computer Science, University of Washington, 1991

B.S. Mathematics, University of Washington, 1991

M.S. Computer Science, University of Colorado, 1993

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

1995

This thesis for the Doctor of Philosophy degree by

Bradley Gene Calder

has been approved for the

Department of

Computer Science

by

____

Dirk Grunwald

____

Andrew Pleszkun

Date _______________________

Calder, Bradley Gene (Ph. D., Computer Science)

Hardware and Software Mechanisms for Instruction Fetch Prediction

Thesis directed by Assistant Professor Dirk Grunwald

Accurate instruction fetch prediction and branch prediction is increasingly important on today's wide-issue architectures. Instruction fetch prediction is the process of determining the next instruction to request from the memory subsystem. Branch prediction is the process of predicting the likely out-come of branch instructions. Many branch prediction and instruction fetch prediction architectures have been proposed, from simple static techniques to more sophisticated hardware designs. This dissertation concentrates on the problem of instruction fetch prediction.

If the current instruction fetch contains a branch instruction and the branch is a taken PC-relative branch, the processor does not know which instructions to fetch next from the instruction cache if the target address has not yet been calculated. This is called the instruction fetch problem. Not knowing the target address, even if the direction of the conditional branch is correctly predicted, can cause the wrong instructions to be fetched from the instruction cache. Instruction fetch prediction solves this problem by supplying a mechanism to predict the target address enabling the instruction cache to immediately start fetching at the predicted destination.

This dissertation examines three techniques to improve instruction fetch prediction. The first technique, called Branch Alignment, is a software optimization that reorders basic blocks, taking into consideration the architectural cost model and the branch prediction architecture in order to improve instruction fetch prediction. The second mechanism is a hardware design called the Next Cache Line and Set prediction architecture which uses cache indices instead of target addresses to predict which instructions to fetch from the instruction cache. Lastly, a hardware/software architecture design, called the Precomputed-Branch architecture, uses precomputed branch addresses instead of PC-relative addresses in order to eliminate all instruction fetch penalties. Another contribution of this thesis is a system-level performance comparison of several current branch and instruction fetch prediction architectures using a full pipeline-level architectural simulator. This thesis also provides instruction fetch and branch prediction performance results for object-oriented C++ applications.

DEDICATION

To my wife
Jena
and
my parents,
Cindy and Evan.

ACKNOWLEDGMENTS

I would like to give special thanks to my advisor, Dirk Grunwald. This work would not have been possible without his advice, support, and the computing resources he provided to perform the thousands of simulations that went into this dissertation. I would also like to thank my thesis committee: Andy Pleszkun, Ben Zorn, Gary Nutt and Bill Waite.

I would like to thank Amitabh Srivastava and Alan Eustace for providing ATOM and OM, Joel Emer, Mike McCallig, and Dave Webb for providing Zippy, and especially Joel Emer for answering questions and providing help pertaining to Zippy for the system level branch architecture study. I would like to also thank Amitabh and Digital Equipment Corporation's Western Research Lab for supporting me for two summer internships and for providing software grants, and Bruce Foster for providing the equipment grants needed to perform much of this work. I would also like to thank ARPA for funding me with a Fellowship in High Performance Computing, administered by the Institute for Advanced Computer Studies, University of Maryland. I would also like to thank all the other people I have worked with or discussed ideas with over the years: Keith Farkas, Dennis Lee, Alan Eustace, Basim Kadhim, Rich Neves, Harry Jordan, Dave Wagner and Jean-Loup Baer.

I would like to thank all my friends at University of Colorado for making the past few years excellent ones: Basim and Sarah Kadhim, Rich and Michelle Neves, James Brunner, Suvas Vajracharya, Anshu Aggarwal, Jon Cook, Adam Griff, Harini Srinivasan, and many others.

Finally, I would like to thank my parents, Cindy and Evan, for their support, and especially my loving wife Jena for her patience, support, and encouragement through these years.

CONTENTS

CHAPTER
1 INTRODUCTION 1
1.1 Contributions 2
1.2 Organization 3
2 BACKGROUND 4
2.1 The Branch Problem and The Instruction Fetch Problem 4
2.2 Branch and Instruction Fetch Prediction 5
2.2.1 Static Prediction 7
2.2.2 Dynamic Prediction 8
2.3 Profile Code Transformation Optimizations 14
2.3.1 Optimization for Memory Hierarchies 14
2.3.2 Optimizations for Control Flow 14
2.4 Reducing Branch Penalties in C++ 15
3 EXPERIMENTAL METHODOLOGY 16
3.1 Experimental Tools 16
3.1.1 ATOM 16
3.1.2 OM 16
3.1.3 Zippy 16
3.2 Program Statistics 17
3.3 Metrics 17
3.3.1 Percent Increase in Execution Time (%IET) 17
3.3.2 Cycles Per Instruction (CPI) 17
3.3.3 Branch Execution Penalty (BEP) 17
4 SYSTEM LEVEL PERSPECTIVE 24
4.1 Introduction 24
4.2 Verification of Zippy Simulation Methodology 24
4.3 Processor Performance with Perfect Branch and Fetch Prediction 26
4.4 Metrics 26
4.5 Branch Architectures Simulated 28
4.6 Branch Architecture Performance 30
4.7 Validating the Branch Execution Penalty Metric 33
4.8 Implications of System Level Study 36
4.9 Summary 36
5 BRANCH ALIGNMENT 39
5.1 Introduction 39
5.2 Related Branch Alignment Work 39
5.3 Branch Prediction Architectures 40
5.3.1 Static Branch Prediction Architectures 40
5.3.2 Dynamic Branch Prediction Methods 41
5.4 Branch Alignment Algorithms 44
5.4.1 Greedy 44
5.4.2 Adding a Branch Cost Model 44
5.4.3 Try15 45
5.5 Methodology 46

5.6 Greedy and Try15 Branch Execution Penalty Results 46
5.7 Breakdown of Greedy Alignment Results 52
5.7.1 Link-Time Performance of Branch Alignment 57
5.8 Summary 59
6 NEXT CACHE LINE SET PREDICTION 60
6.1 Introduction 60
6.2 The BTB Instruction Fetch Prediction Architecture 62
6.3 Next Cache Line and Set Prediction Architecture 62
6.3.1 NLS-Table Versus NLS-Cache 65
6.3.2 Using Next Line Addresses with the Instruction Cache 66
6.3.3 Identifying Instructions as Branch Instructions 67
6.4 Simulation Methodology 67
6.5 Calculating Register Bit Equivalent Costs 68
6.6 NLS Architecture Results 71
6.6.1 Performance of the NLS-Cache Architecture 71
6.6.2 Performance of the NLS-Table Architecture 71
6.6.3 Increasing the Performance of the NLS-Cache Architecture 71
6.6.4 Related Work 79
6.7 Performance of the BTB Architecture 80
6.8 Comparison of NLS-Table and BTB Architectures 82
6.9 Summary 90
7 PRECOMPUTED BRANCH ARCHITECTURE 91
7.1 Introduction 91
7.2 The Design of Two Branch Architectures 91
7.2.1 A BTB-based Instruction Fetch Architecture 92
7.2.2 The Precomputed-Branch Architecture 92
7.2.3 Computing the Branch Target 94
7.2.4 Other Non-Relative Branch Architectures 94
7.3 Methodology 94
7.4 Partitioning Programs Into Branch Spaces 96
7.4.1 Performance of Program Partitioning Algorithms 98
7.5 Precomputed-Branch Architecture Performance 103
7.5.1 Comparing the BTB and Precomputed-Branch Architectures 103
7.6 Practical Concerns 109
7.6.1 Non-relative Branches in a Relative World 109
7.7 Design Issues and Comparison of Precomputed-Branch and NLS Architectures 111
7.8 Summary 111
8 CONCLUSIONS 113
9 FUTURE WORK 115
9.1 Implications for Branch Prediction Research 115
9.2 Static Prediction 115
9.3 Dynamic Prediction and Future Processor Designs 116
BIBLIOGRAPHY 117

FIGURES

FIGURE

2.1 Pattern History Table and State Diagram for 2-Bit Counter. 9
2.2 Correlated Conditional Branch Prediction. 10
2.3 A Schematic Representation of the Coupled PAs and Branch Target Buffer Architecture. . . 12
2.4 A Schematic Representation of a Decoupled GAg and Branch Target Buffer Architecture. . . 13
5.1 Benefits of Code Transformation for `elim_lowering` in `espresso` 42
5.2 Routine `input_hidden` from `alvinn` 43
5.3 Example Illustrating How Try15 Reduces Branch Costs 46
5.4 Branch Alignment Total Execution Time Improvement on a DEC 3000-600 Alpha AXP for the SPEC92 C Programs. 58
6.1 Register Bit Equivalent Costs for On-chip BTB and Instruction Cache Architectures. 61
6.2 A Schematic Representation of a Decoupled BTB Architecture 63
6.3 A Schematic Representation of the NLS-Table Architecture. 64
6.4 Register Bit Equivalent Costs for NLS and BTB Architectures 70
6.5 Average NLS-Cache Performance 72
6.6 Average NLS-Table Performance 76
6.7 Average Branch Execution Penalty for BTB and NLS-Table Architectures 81
6.8 Access Time for BTB Architecture 83
6.9 NLS-table and BTB Performance for Doduc 84
6.10 NLS-Table and BTB Performance for Cfront. 85
6.11 NLS-Table and BTB Performance for Espresso. 86
6.12 NLS-Table and BTB Performance for Gcc. 87
6.13 NLS-Table and BTB Performance for Li. 88
6.14 NLS-Table and BTB Performance for Groff. 89
7.1 A Schematic Representation of the Branch Target Buffer Architecture. 93
7.2 A Schematic Representation of the Precomputed-Branch Architecture. 93
7.3 Alternative Branch Methods 95
7.4 Partitioning a Call Graph without Profiles 99
7.5 Partitioning a Call Graph with Profiles 100
7.6 Branch Execution Penalty for Precomputed-Branch and BTB Architectures 104
7.7 Using the Precomputed-Branch Architecture with a PC-Relative Instruction Set 110

TABLES

TABLE

2.1 Pipeline with no Branch Prediction or Instruction Fetch Prediction. 6
2.2 Pipeline with Branch Prediction but no Instruction Fetch Prediction. 6
2.3 Pipeline with Branch Prediction and Instruction Fetch Prediction. 6
3.1 Description of FORTRAN Applications. 18
3.2 Description of C Applications. 19
3.3 Description of C++ Applications. 20
3.4 Measured Attributes of Traced Programs. 21
3.5 Conditional Branch Quantiles for Traced Programs. 22
4.1 Verification of Zippy Architecture Model 25
4.2 Zippy Performance Assuming Perfect Branch Prediction 27
4.3 Architectures Simulated for System Level Study 29
4.4 Percent Increase in Execution Time for Conditional Branch Architectures 31
4.5 Percent Increase in Execution Time for a Subset of Programs 32
4.6 Average System Level Performance 34
4.7 Average Branch and Instruction Fetch Prediction Performance 35
4.8 Validating the Branch Execution Penalty Metric 37
5.1 Cost, in Cycles, for Different Branches. 44
5.2 Branch Execution Penalty for Static Branch Prediction Architectures with Branch Alignment. 48
5.3 Percent of Fall-Through Branches with Branch Alignment. 49
5.4 Branch Execution Penalty for Pattern History Table Architectures with Branch Alignment. . 50
5.5 Branch Execution Penalty for Branch Target Buffers with Branch Alignment. 51
5.6 Branch Alignment Miss Rates and Relative Instructions Executed for Direct Mapped PHT Architecture. 53
5.7 Branch Alignment Miss Rates and Relative Instructions Executed for Correlated GAg Architecture. 54
5.8 Branch Alignment Miss Rates and Relative Instructions Executed for a 64 entry BTB Architecture. 55
5.9 Branch Alignment Miss Rates and Relative Instructions Executed for a 256 entry BTB Architecture. 56
6.1 NLS Prediction Sources 62
6.2 Instruction Cache Misses Rates for Traced Programs. 67
6.3 Values Used to Calculated RBE Costs for NLS Architectures. 69
6.4 Values Used to Calculate RBE Costs for BTB Architectures. 69
6.5 NLS-Cache Performance for Doduc and Espresso. 73
6.6 NLS-Cache Performance for Gcc and Li. 74
6.7 NLS-Cache Performance for Cfront and Groff. 75
6.8 NLS-Table Performance for Doduc, Espresso and Li 77
6.9 NLS-Table Performance for Gcc, Cfront and Groff 78
6.10 Branch Target Buffer Performance 80
7.1 Number of Static Instructions and Procedures in Traced Programs. 97
7.2 Efficiency of Program Partitioning with 14-bit Branch Displacements 101
7.3 Efficiency of Program Partitioning with 16-bit Branch Displacements 102
7.4 Summary of Performance Information From Trace Driven Simulations 105

7.5 Average Direct Mapped and 4-Way Associative IJB Performance for the Optimal-21 Partitioning 106
7.6 Percent of Mispredicted Branches for IJB with Preorder Partitioning 107
7.7 Percent of Mispredicted Branches for IJB with MaxCut Partitioning 108
7.8 Precomputed-Branch Penalties for Branch Intensive Programs 108

CHAPTER 1

INTRODUCTION

When a branch is encountered in the instruction stream, it can cause a change in control flow. This change in control flow can cause branch penalties that can significantly degrade the performance of today's superscalar pipelined processors. To overcome these penalties, processors typically provide some form of control flow prediction.

Branch Prediction is the process of predicting the branching direction for conditional branches, whether or not they are taken, and predicting the target address for return instructions and indirect branches. The final destinations for conditional branches, indirect function calls and returns are typically not available until late in the pipeline. The processor may elect to fetch and decode instructions on the assumption that the eventual branch target can be accurately predicted. If the processor mispredicts the branch destination, instructions fetched from the incorrect instruction stream must be discarded, leading to several pipeline bubbles. This is called a branch **mispredict penalty**.

Instruction fetch prediction is used each cycle to predict which instructions to fetch from the instruction cache. In most processors, the next instruction is assumed to be fetched and executed. If the decoded instruction causes a change in control flow, the instruction following the branch can not be used, and a new instruction must be fetched after the target address is calculated, introducing a pipeline bubble or unused pipeline step. This is called an instruction **misfetch penalty**, and is caused by waiting to identify the instruction as a branch and to calculate the target address.

Historically, processor design has focused on mechanisms to correctly predict conditional control flow changes because these are simple to implement and results in considerable savings. Only recently have processor designs addressed instruction fetch prediction. As processors issue more instructions concurrently, branch penalties increase, and the instruction misfetch penalty becomes increasingly important. It is more likely that a branch will occur as more instructions are fetched each cycle, decreasing the likelihood that the "fall-through" instruction will be executed. This thesis concentrates on the problem of instruction fetch prediction and eliminating misfetch penalties. This dissertation examines the system level performance of modern instruction fetch and branch prediction architectures, examines a compiler optimization called Branch Alignment to reduce misfetch penalties, and examines two alternative instruction fetch prediction architectures called the Next Cache Line and Set prediction architecture and the Precomputed-Branch architecture. The following paragraphs summarize these four topics.

Usually, branch prediction mechanisms are studied in isolation, ignoring other aspects of the system. In these studies, a variety of metrics are used to assess the performance of branch architectures, but it is rare that the performance of an entire system is reported. There are numerous reasons for this; the complexity of accurately simulating a complex architecture is daunting, the results are then specific to that architecture and it is difficult to translate the performance of that architecture to another, possibly different architecture. However, there are advantages to system-level simulations. They provide a sense of the magnitude and importance of the problem being solved, and indicate where further effort should be placed. Furthermore, the performance metrics, such as execution time and cycles per instruction, are more understandable to the casual reader. Lastly, such studies provide a method to calibrate the performance metrics used in other studies with the impact on system level performance. This dissertation provides a system level study of several branch and instruction fetch prediction architectures, including recently proposed two-level correlated branch history architectures. The system-level simulation is based on the Alpha AXP 21064, a dual-issue statically-scheduled processor.

To examine the instruction fetch problem I first study software techniques to reduce misfetch penalties. I examine algorithms that reorder the structure of a program to improve the accuracy of the branch and instruction

fetch prediction architectures. These code transformations reduce the number of mispredicted branches and the number of misfetched instructions. Essentially, the method is this: restructure the control flow graph so that **fall-through** branches occur more frequently. Then use profile information to direct the transformation, and an architectural cost model to decide if the transformation is warranted. Transformations include rearranging the placement of basic blocks, changing the sense of conditional operations, moving unconditional branches out of the frequently executed path, and occasionally inserting unconditional branches. By making the fall-through the most common execution path the processor executes fewer taken branches, decreasing the need for the taken target address, and the number of misfetched instructions is significantly reduced.

Several effective instruction fetch and branch prediction mechanisms have been proposed including branch target buffers (BTB). A BTB is a cache storing the target address for taken branches, and is used for predicting the next instruction fetch when a branch is encountered. When an instruction is fetched, the instruction's address is given to the BTB; if there is a match in the BTB and the branch is predicted as taken, the next instruction is fetched using the target address specified in the BTB. In this design, exemplified by the PowerPC 604 [54], a hit in the BTB identifies the instruction as a branch and the BTB provides the destination address for only taken branches.

Using a BTB avoids the delay needed to recalculate the branch target address and reduces the misfetch penalty. However, an effective branch target buffer can be large and can possibly increase the cycle time of a processor. An alternative design would be to fetch the instruction following a branch by using an index into the cache instead of a branch target address. I call such an index a **next cache line and set** (NLS) predictor. A NLS predictor is a pointer into the instruction cache indicating the target instruction of a taken branch. Johnson [32] proposed a similar design using cache indices to predict the next instruction fetch. I propose an alternate organization that improves fetch prediction accuracy. In this dissertation, I examine two varieties of the NLS architecture. The NLS-cache is similar to the branch architecture described by Johnson, where each NLS predictor is associated with a cache line. The NLS-table uses NLS predictors stored in a separate direct mapped tag-less memory buffer. I also examine the effects of combining the NLS predictors with modern two-level correlated branch prediction architectures.

Both the BTB and NLS architectures are effective means to eliminate misfetch penalties, though they can require a fair amount of hardware resources. The architecture hardware needs for instruction fetch prediction can be completely eliminated by using software approaches to help solve the hardware problem. I propose that a design used in older computers, such as the PDP-8, be used in modern architectures. This architecture is called the Precomputed-Branch architecture; the branch destination is precomputed for most branch instructions, allowing the branch information to be stored with the instruction. This architecture assumes a flat address space, and eliminates program-counter relative (PC-relative) branches. In this design, the destination address is quickly computed by precomputing the low-order bits of the destination, and concatenating this with the high-order bits of the current address, effectively dividing memory into a number of program segments or "branch spaces". Branches between branch spaces are computed and performed as indirect jumps. This design assumes the program linker or compiler can partition the program into a number of segments and modify the program's structure to select between intra-space and inter-space branches. I also describe how existing architectures using PC-relative branches can be extended to use the Precomputed-Branch architecture by precomputing branch destinations as instructions are fetched into the instruction cache.

1.1 Contributions

This dissertation makes five major contributions:

(1) I provide a detailed architectural system level study of branch and instruction fetch prediction. This research examines current branch and instruction fetch prediction architectures providing the actual execution time improvement for an Alpha 21064 processor. To my knowledge, no previous branch prediction literature has reported this detailed level of results for branch and instruction fetch prediction with respect to other system resources. This study validates the branch execution penalty metric used to compare branch and fetch architecture performance in the remainder of this dissertation.

(2) I examine branch alignment algorithms to eliminate instruction fetch penalties and to improve branch prediction performance. These algorithms take into consideration the architectural cost model and the branch prediction architecture when performing the basic block reordering. Results show that these algorithms can improve a broad range of static and dynamic branch and instruction fetch prediction architectures.

(3) I propose the design of an alternative instruction fetch prediction architecture called Next Cache Line and Set prediction. This architecture uses cache indices instead of target addresses to predict which instructions to fetch from the instruction cache. Results show that the NLS architecture is a competitive alternative to the BTB design.

(4) I examine a hardware/software solution to the instruction fetch problem called the Precomputed-Branch Architecture. This architecture eliminates PC-relative branches, and instead uses precomputed branches to provide the target address. This effectively eliminates all misfetch penalties since the target addresses are precomputed. Simulation results show that the Precomputed-Branch architecture performs better than the BTB architecture, and has significant hardware savings.

(5) I provide branch and instruction fetch prediction analysis for object-oriented C++ programs. To my knowledge this research is the first to examine the branch and instruction fetch prediction issues and performance of object-oriented programs. The results show that current branch and instruction fetch prediction architectures and the architectures proposed in this dissertation accurately predict C++ applications.

1.2 Organization

The rest of this dissertation is organized into seven chapters. Chapter 2 provides the background of this dissertation. It describes the instruction fetch and branch prediction problems, previous research in branch and instruction fetch prediction, previous work in compiler optimizations related to branch alignment, and previous C++ branch prediction studies. Chapter 3 describes the programs simulated in this dissertation, provides basic branching statistics for each of these programs, and describes the simulation tools and metrics used throughout this dissertation. Chapter 4 provides a system level study of an Alpha 21064 architecture varying only the branch prediction architecture. This provides an indication of the importance of instruction fetch prediction for a statically scheduled architecture like the DEC Alpha 21064. It also validates the branch execution penalty (BEP) as a suitable metric for comparing branch and instruction fetch prediction architectures. Chapter 5 describes Branch Alignment and the results of using branch alignment with static and dynamic branch and instruction fetch prediction architectures. Chapter 6 describes the Next Cache Line and Set Prediction architecture and compares the performance of the NLS architecture to the BTB design. Chapter 7 describes the Precomputed-Branch architecture and compares its performance to the BTB design. Chapter 8 presents the conclusion of this dissertation, and Chapter 9 discusses implications and future work.

CHAPTER 2

BACKGROUND

This chapter provides background on branch and instruction fetch prediction architectures and related work. There are a number of mechanisms to ameliorate the effect of uncertain control flow changes, including: N-bit counters, pattern history tables, branch target buffers, delayed branches, prefetching both targets, early branch resolution, branch bypassing and prepare-to-branch mechanisms. These techniques are described in great detail elsewhere by Lee and Smith [38], McFarling and Henessy [42], Lilja [39], and by Perleberg and Smith [48]. This chapter describes the branch and the instruction fetch problems, previous work in branch and instruction fetch prediction, previous work in compiler optimizations related to branch alignment, and previous C++ prediction studies.

2.1 The Branch Problem and The Instruction Fetch Problem

Conventional processor architectures, particularly superscalar designs, are extremely sensitive to control flow changes. A simplified processor pipeline can be divided into the instruction fetch (IF) stage, instruction decode (ID) stage, register file and issue (R/I) stage, first execute stage (EX1), second execute stage (EX2), and a write back stage (WR). This is the pipeline model assumed for the simulation results gathered in Chapters 5, 6, and 7, and we assume that this simplified processor can fetch and execute one instruction per cycle, but this model is also applicable to wide-issue processors. In this processor model, the target address for a PC-relative branch is not known until after the ID stage. The final destinations for conditional branches, indirect procedure calls and jumps, and return instructions is not known until after the EX2 stage has completed. Processors fetch instructions, and discard their results if a branch is incorrectly predicted. Therefore, incorrectly predicted conditional branches, indirect procedure calls and jumps, and return instructions cause a four cycle mispredict penalty. In addition, correctly predicted taken conditional branches, unconditional branches, and direct procedure calls can cause a one cycle misfetch penalty, because the target address is not available until after the decode stage has completed.

The **branch problem** is the problem of figuring out the correct destination for branch instructions in order to start fetching from the correct instruction stream in the instruction cache without causing any pipeline delays. **Branch prediction** is the process of predicting the branching direction for conditional branches, whether they are taken or not-taken, and predicting the target address for return instructions and indirect branches.

The **instruction fetch problem** is the problem of figuring out which instructions to fetch next from the instruction cache. **Instruction fetch prediction** is used to predict which instructions to fetch from the instruction cache when the current instruction fetch contains a branch. If the current instruction fetch does not contain a branch instruction, then the next instruction fetch is easily calculated to be the $PC + fetch\ size$. If the current instruction fetch contains a taken PC-relative branch instruction and the processor does not provide instruction fetch prediction, the $PC + fetch\ size$ is used by default for the next instruction fetch. If the fetched instructions are not used, this introduces a pipeline bubble, or an unused pipeline step called the misfetch penalty. Once the PC-relative target address is calculated in the decode stage, the correct target instruction can then be fetched from the instruction cache. The misfetch penalty is caused by having to wait for the target address to be calculated in the decode stage of our simplified processor.

Assume our simplified processor is a single issue machine, and executes three instructions: an Add, conditional branch (CBr), and a Multiply (Mult) instruction, in this order. Also assume that the conditional branch is taken. Figure 2.1 shows the cycle diagram of the processor when **no** branch or instruction fetch prediction is available. A pipeline stall is represented as a dash (—) in the digram. In this case, the three instructions take 12 cycles to execute. Here the processor must stall for four cycles until the destination of

the conditional branch is resolved, which is only after it has passed through the EX2 stage. At this point the Multiply instruction can be fetched from the instruction cache and execution can continue.

Figure 2.2 shows the same pipeline with the same three instructions when **only** branch prediction is added to the architecture. In this case, we may be able to correctly predict that the conditional branch is taken. Correctly predicting the branch eliminates three of the stall cycles, but not the remaining stall cycle because the target address is not available until after the decode stage. Therefore, the Multiply instruction cannot be correctly fetched from the instruction cache until after the conditional branch has passed through the decode stage.

Figure 2.3 shows the same pipeline as the previous two figures, but with instruction fetch prediction added to the architecture. If a prediction mechanism such as a branch target buffer (BTB) is supplied, then a hit in the BTB indicates that a branch was fetched and the target addresses stored in the BTB allows the misfetch penalty to be avoided. In Figure 2.3, the mispredict penalty is eliminated because branch prediction predicts that the conditional branch is taken, eliminating three cycles as in the previous diagram. In addition, the final stall penalty is removed by using a BTB to predict the correct branch destination to start fetching from in the instruction cache **before** the target address is even calculated. This example shows the benefit of instruction fetch prediction.

All taken PC-relative branches (conditional branches, unconditional branches, and direct procedure calls) can cause misfetch penalties, and these account for the majority of branches executed in a program. Indirect jumps, indirect procedure calls and return instructions are examples of non PC-relative branches. Past branch prediction research has typically dealt only with conditional branch prediction and has usually ignored the penalties due to instruction misfetches, because the biggest performance gain can be seen by eliminating mispredicted conditional branches. The other reason instruction fetch prediction was not an important issue was that past computers were single issue processors, and for these processors it was fairly easy to eliminate most of the misfetch penalties using branch delay slots. This is not the case for future wide-issue superscalar processors. On these future processors, the misfetch penalties can cost several instructions and can span several cycles. Even recent processors designs, the Intel Pentium [17], PowerPC 604 [54], MIPS R8000 TFP processor [31], and the UltraSPARC processor, have deemed instruction fetch prediction an important enough problem to add special purpose hardware for instruction fetch prediction to their processors.

2.2 Branch and Instruction Fetch Prediction

Branch and instruction fetch prediction techniques can be classified as either **static** or **dynamic**. Static branch prediction information does not change during the execution of a program, while dynamic prediction may change, reflecting the time-varying activity of the program. Static methods range from compile-time heuristics [4, 38, 42, 52] to profile-based methods [25, 42, 61]. In general, profile-based prediction techniques out-perform compile-time prediction techniques or techniques that use heuristics based on the direction of the branch target (forward or backward) or instruction opcodes. Among the static techniques, the most common architectures for conditional branch prediction are to predict conditional branches as fall-through (not-taken), taken, backwards taken and forwards not taken (BTFNT), having a "likely" bit encoded in the branch instruction at compile time to predict the direction, and branch delay slots.

While static prediction mechanisms, particularly profile-based methods, accurately predict 70-90% of branches, modern computer architectures increasingly depend on mechanisms that estimate future control flow decisions to increase performance, requiring more accurate branch prediction mechanisms. These architectures use dynamic prediction to achieve an accuracy above 90%. Another advantage of dynamic prediction is that it can also avoid misfetch penalties associated with decoding the branch instruction and calculating the target address. Among the dynamic prediction techniques, the most common architectures are pattern history tables (PHT), branch target buffers (BTB), and architectures that combine branch and fetch prediction information with the instruction cache.

Table 2.1: Pipeline with no Branch Prediction or Instruction Fetch Prediction.

	Cycle											
	1	2	3	4	5	6	7	8	9	10	11	12
IF	Add	CBr	—	—	—	—	Mult					
ID		Add	CBr	—	—	—	—	Mult				
R/I			Add	CBr	—	—	—	—	Mult			
EX1				Add	CBr	—	—	—	—	Mult		
EX2					Add	CBr	—	—	—	—	Mult	
WR						Add	CBr	—	—	—	—	Mult

Table 2.2: Pipeline with Branch Prediction but no Instruction Fetch Prediction.

	Cycle								
	1	2	3	4	5	6	7	8	9
IF	Add	CBr	—	Mult					
ID		Add	CBr	—	Mult				
R/I			Add	CBr	—	Mult			
EX1				Add	CBr	—	Mult		
EX2					Add	CBr	—	Mult	
WR						Add	CBr	—	Mult

Table 2.3: Pipeline with Branch Prediction and Instruction Fetch Prediction.

	Cycle							
	1	2	3	4	5	6	7	8
IF	Add	CBr	Mult					
ID		Add	CBr	Mult				
R/I			Add	CBr	Mult			
EX1				Add	CBr	Mult		
EX2					Add	CBr	Mult	
WR						Add	CBr	Mult

2.2.1 Static Prediction

Some architectures employ static prediction hints, using either profile information, information derived from compile time analysis, or information about the branch direction or branch opcode [4, 42, 52, 69]. David Wall provided an early study on predicting the future behavior of a program using profiles [62]. His results showed that using profiles from a different run of the application achieved results close to that of a perfect profile from the same run. Fisher and Freudenberger confirmed that this observation applied to static branch prediction [25]. They used traces from one execution of a program to predict the outcome of conditional branches for the same and different inputs. They defined **perfect profile** prediction to be the prediction accuracy achieved when the same input was used to trace the program and then used to measure the accuracy of static branch prediction. Their C/Integer results show that, on average, 95% of the perfect profile prediction accuracy is achieved when profiling a program with the best matched previous trace. Only 75% of the perfect profile prediction accuracy was achieved using the worse previous trace. Both of these studies and others have promoted profile based optimizations as a means to achieve increased processing performance.

More recently, studies have been performed using compile time heuristics to estimate profile information [4, 60, 64, 14]. These studies address a number of issues. First, it may be possible to use simple heuristics to estimate profiles, implying that profile-based optimizations can be performed using heuristics. Furthermore, even though many extant compilers perform some profile-based optimizations, most programmers do not use such options, either because the profiling method is not standardized across platforms, they are unaware of the option, they are uncertain of the benefits of profile-based optimization, or they believe that the process of gathering profiles and recompiling their programs is too expensive.

Ball and Larus proposed several heuristics for predicting a program's behavior at compile time [4]. In a later study [14], we found their heuristics are reasonably accurate, resulting in a 25% mispredict rate at compile time without profile information. By comparison, perfect profile prediction had a 8% miss rate for the same collection of programs. Other studies by Wagner **et. al.** [60] and Wu and Larus [64] have focused on using these heuristics and other techniques to fully estimate a program's behavior at compile time.

We examined an alternative technique for predicting program behavior by combining profile information gathered from a number of applications. We collected a "feature vector" describing an individual conditional branch, and then used various machine-learning techniques to determine what combination of features, if any, accurately predicted the branches. We have considered two techniques; the first, described in [14], uses a "neural network" to combine the information from the feature vectors, and the second technique uses "decision trees" to accomplish the same goal. These methods create heuristics to be used at compile time for a specific compiler, language and architecture. Our results show that this technique, called Evidence-based Static Prediction (ESP), results in a 20% mispredict rate, a slight improvement to the Ball and Larus heuristics, which had a miss rate of 25%. In general, profile-based prediction techniques outperform compile-time prediction techniques or techniques that use heuristics based on branch prediction or instruction opcodes.

Static conditional branch prediction can be implemented as a single prediction bit in the instruction to indicate if the branch is to be predicted as taken or not-taken; the bit would be set by the compiler using profiles or heuristics as discussed above. We call this the "Likely" architecture, because the bit predicts the likely direction of the branch. Other implementations allow some action to be taken if a certain opcode is encountered. One popular technique is to predict that backward branches are taken and forward conditional branches are not taken (BTFNT). This technique is implemented by examining the sign bit of the relative displacement for the branch. If the bit is on, indicating a backwards branch, the branch will be predicted as taken, and if the bit is not set the fall-through will be predicted.

In the branch delay slot architecture, each branch instruction is followed by a fixed number of instruction slots that are executed while the branches' destination is resolved. These branch delay slots are filled at compile time in order to remove misfetch and mispredict branch penalties. Recent processors no longer use branch delay slots, since current architectures being developed issue many instructions per cycle and have very deep pipelines. For example, the DEC Alpha 21164 issues four instructions per cycle. If the Alpha used branch delay slots to avoid branch misfetch penalties, four instructions would be needed to fill each branch delay slot.

2.2.2 Dynamic Prediction

Most processors now use some form of dynamic branch and instruction fetch prediction to improve performance. Some branch architectures reduce both misfetch and mispredict penalties while other architectures only reduce mispredict penalties due to conditional branches.

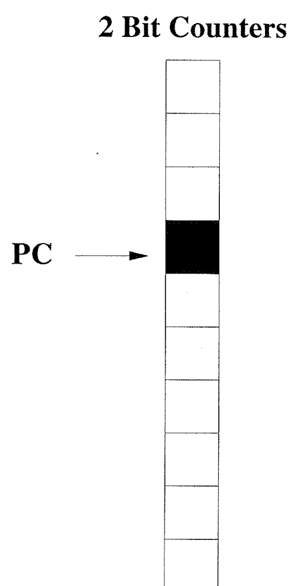
One-bit branch prediction is the simplest form of dynamic conditional branch prediction. This scheme dynamically predicts the direction that the branch was taken the last time it was executed. A 0 stored in the 1-bit predictor indicates that the conditional branch was not-taken the last time it was executed, and a 1 indicates that the branch was taken. The Alpha 21064 used a one-bit branch predictor associated with each instruction in its 8K direct mapped instruction cache to dynamically predict the direction of conditional branches. This bit is initialized with the sign-bit of the PC-relative displacement when the cache line is read in, initializing the dynamic 1-bit predictors to BTFNT prediction. Thus, the BTFNT rule is used the first time the branch is encountered, and a dynamic one-bit predictor is used thereafter.

The pattern history table (PHT) branch architecture is another example of a dynamic architecture used to predict only conditional branches. This architecture uses N-bit counters stored in a table to predict the direction for conditional branches. The most common variants of this design are 1-bit counter tables that indicate the direction of the most recent branch mapping, and 2-bit counter tables which yield much better performance for programs with loops. Two-bit up-down saturating counters have been shown to effectively predict the direction of conditional branches, outperforming 1-bit branch predictors [46, 52].

Figure 2.1 shows the finite state machine for a 2-bit pattern history table. The arrows in the 2-bit state diagram show the state transition when the branch is either taken (T) or not-taken (NT). Once the direction of the branch is known, the counter is updated. For example, if the counter has a value of 0, the conditional branch is predicted as not-taken. If the branches' destination is not-taken, then the counter stays the same with the value of 0. If the branches final destination was taken, then the counter is incremented and its new value is 1. This type of pattern history table is called a direct mapped PHT (PHT-Direct), since the branch address is used to directly index into the pattern history table to find the 2-bit counter used to predict the branch. Since different branch addresses can index into the same table entry, several conditional branches may alias to the same prediction information. For example, in a 4096 entry table, branches at addresses 0, 16384 and 32768 all map to the same entry in the table. When a conditional branch at any of these addresses is executed, the information for entry '0' is used to predict the branch direction, even if that information was recorded for one of the other branches. This aliasing effect can degrade the performance of a PHT, and there have been several studies that have addressed the issue of trying to eliminate these aliasing effects [19, 59, 68]. The advantage of pattern history tables is that they keep track of very little information per conditional branch site and are very effective in practice, achieving a 95% conditional branch prediction accuracy.

Pan **et al.** [47] and Yeh and Patt [65, 67] investigated **branch-correlation** and **two-level** conditional branch prediction mechanisms. Although there are a number of variants, these mechanisms generally combine the history of several recent branches to predict the outcome of a branch. The **degenerate method** (GAg) of Pan **et al.** [47] shown in Figure 2.2 uses a global history register to record the branch correlation. When using a 2^k entry table, the processor maintains a k-bit global history register that records the outcome of the previous k branches (e.g., a taken branch is encoded as a 1, and a not-taken branch as a 0). This register is used as an index into the pattern history table (PHT), much as the program counter is used for a direct-mapped PHT. This provides contextual information and correlation about particular patterns of branches. The problem with this method is that a lot of the table may not be used, depending upon the patterns of the branches executed in the program. McFarling [41] described a variant of the GAg architecture where he uses an exclusive-or of the global history register and the branch address as an index into the PHT. This effectively spreads out the usage of the 2-bit counters across the table.

The Per-Set Pattern History table (PAs), shown in Figure 2.2, is a correlated conditional branch prediction method proposed by Yeh and Patt [65]. It uses a table of history registers instead of a single history register as in the degenerate case. Yeh and Patt found that the best performance for this architecture came from concatenating part of the branch address with the history register and using this as an index into the pattern



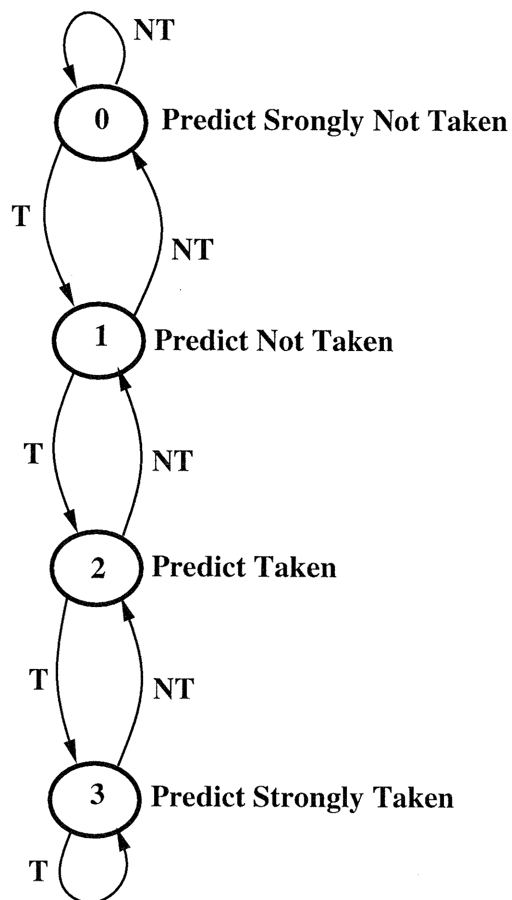
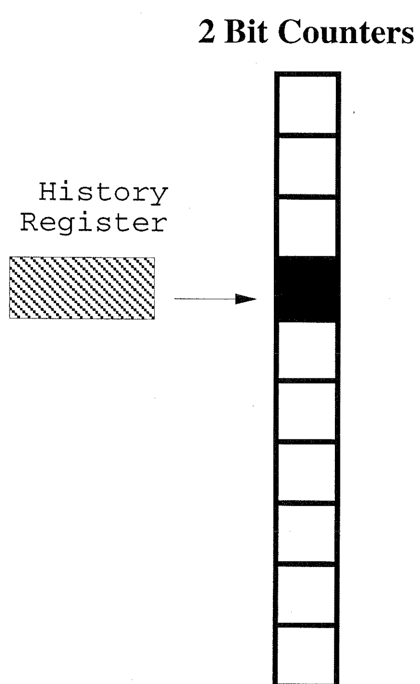


Figure 2.1: Pattern History Table and State Diagram for 2-Bit Counter.



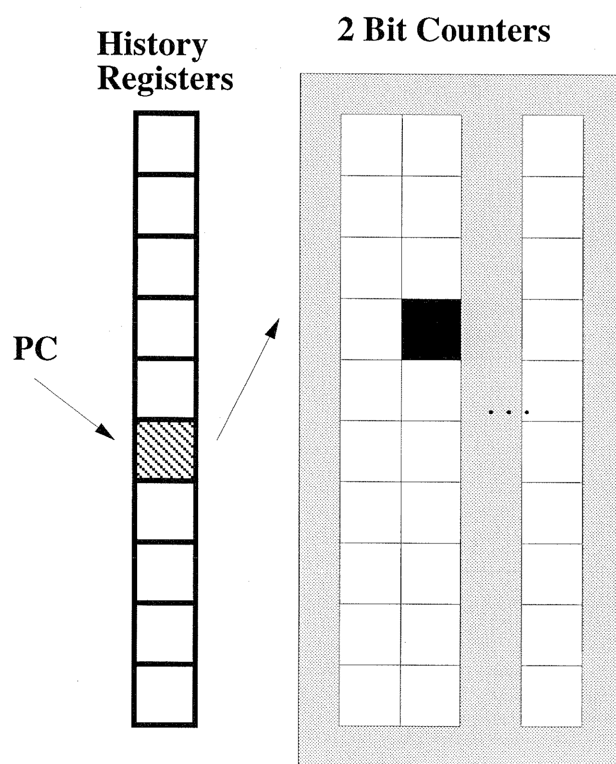


Figure 2.2: Correlated Conditional Branch Prediction.

history table. This effectively partitions the pattern history table into subtables partitioned by the low order bits of the branches address.

Pattern history architectures are very effective at predicting conditional branches, but they provide no mechanisms for instruction fetch prediction. This is because the PHT provides the predicted branching direction but not does not provide the taken target address. Misfetch penalties can be reduced by using a branch target buffer (BTB) [38, 39, 48, 52]. A branch target buffer is a cache that stores the target addresses for the most recently executed branches. For unconditional branches, direct procedure calls, and conditional branches that are predicted as taken, the target address stored in the BTB can immediately be fetched, effectively eliminating the misfetch penalty. The BTB also acts as a branch prediction mechanism for indirect jumps by storing the indirect branches' most recent destination. A BTB can also be used for return instructions, but a return stack [33] is much more accurate. When using a return stack, the BTB does not provide the destination address for return instructions. Instead, the BTB can be used to indicate the instruction is a return instruction so the return stack can be used for the next instruction fetch, avoiding the misfetch penalty.

Typically, a BTB contains from 32 to 512 entries with varying degrees of associativity. A BTB requires considerable storage, because it stores the address of the branch, representing the tag, and the branches' target address. Different kinds of branches use different mechanisms to predict their branch destinations. To be able to select among the different mechanisms, we need to be able to identify the branch type, and some BTB designs store the branch type in the BTB. Figure 2.3 represents the instruction fetch architecture proposed by Yeh and Patt [66]. This is called the "BTB-PAs" architecture, where each BTB entry contains a 6-bit history register. This history register is concatenated with the branch's address in order to form an index into the PHT when predicting conditional branches, as described above for the PAs architecture. In the BTB-PAs architecture, the branch type, along with the conditional branch prediction, is used to indicate whether the "fall-through" address, the "taken" address stored in the BTB, or the return stack address is to be used for the next instruction fetch. We call this a **coupled** branch architecture because the branch prediction information is coupled with the instruction fetch prediction information.

The Intel Pentium is an example of a modern architecture using a coupled BTB design – it has a 256-entry BTB organized as a four-way associative cache. Only branches that are taken are entered into the BTB. If a branch address appears in the BTB and the branch is predicted as taken, the stored address is used to fetch future instructions, otherwise the fall-through address is used. This is called the "BTB-2Bit" architecture because each BTB entry has a two-bit saturating counter used to predict the direction of the conditional branch [38]. The Intel P6 architecture adds to this design by increasing the BTB to a 512 entry 4-way associative design replacing the 2-bit counters with 4-bit history counters. In both the BTB-2Bit and BTB-PAs architectures, the conditional branch prediction information (the two-bit counter and 6-bit history register), is associated or coupled with the BTB's instruction fetch prediction information. Therefore, the dynamic conditional branch prediction information can only be used for branches in the BTB, and branches that miss in the BTB must use less accurate static branch prediction.

A **decoupled** branch architecture separates the conditional branch prediction information from the branch target buffer, so that it can be used to correctly predict a branch even when that branch is not in the BTB. The PowerPC 604 [54] is an example of a decoupled BTB design, where the conditional branch prediction information is not associated with the BTB and is used for all conditional branches, including those not recorded in the BTB. The PowerPC 604 has a 64-entry fully associative BTB that holds the target address of the most recently taken branches, and uses a separate 512 entry PHT to predict the direction of conditional branches. In this architecture, the BTB's main purpose is instruction fetch prediction by providing the taken target address and the branch type. Figure 2.4 shows a decoupled BTB architecture (BTB-GAg) using McFarling's correlated PHT-GAg architecture, described earlier, for predicting conditional branches even on a BTB miss. In related work [7], we showed that a decoupled branch architecture provides similar performance to a coupled design.

The BTB designs described above and the BTB designs used on current processors can be very expensive to implement in terms of chip area costs, and they can have a slow access times since associative BTB configurations are used in order to increase the BTB's hit rate. Part of goals for this dissertation is to

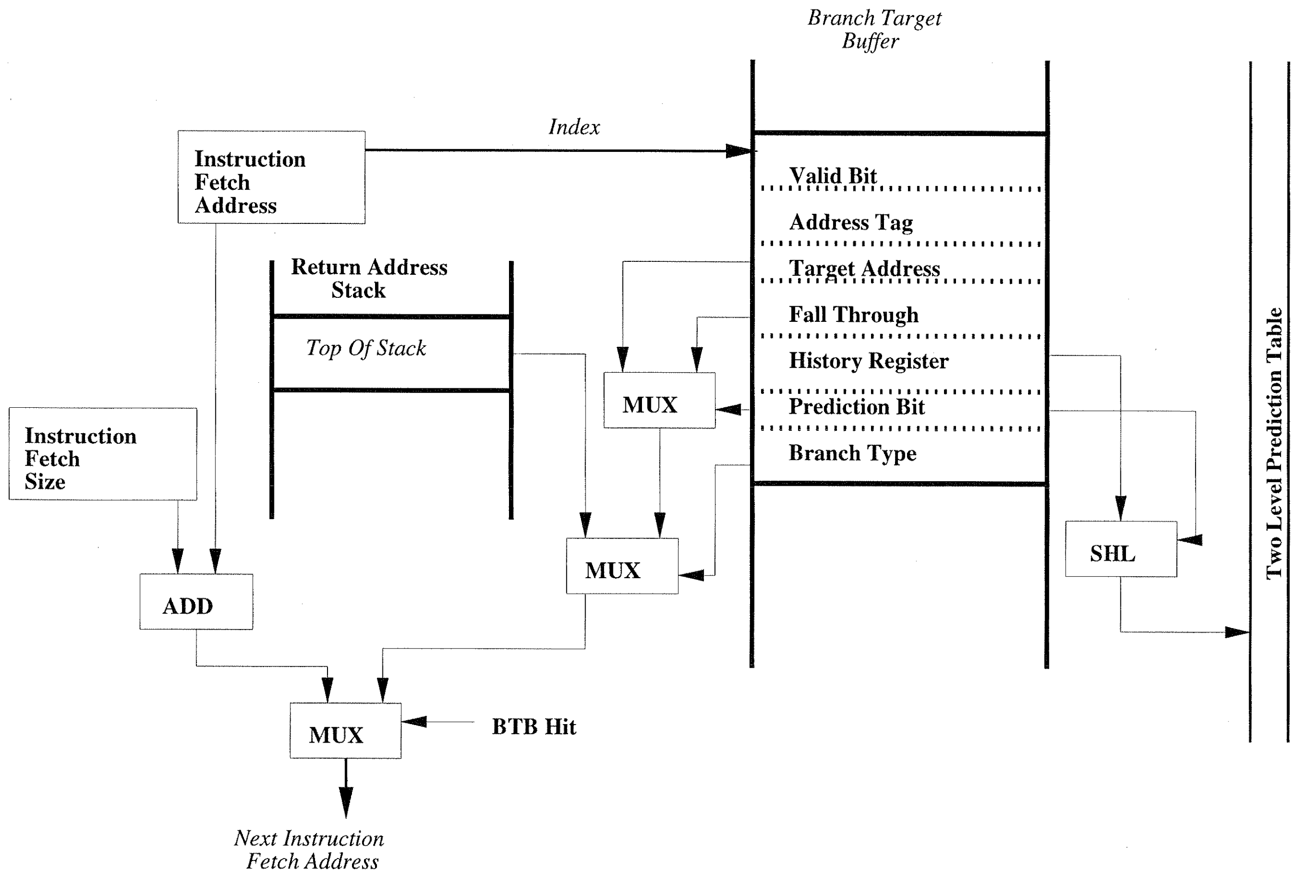


Figure 2.3: A Schematic Representation of the Coupled PAs and Branch Target Buffer Architecture.

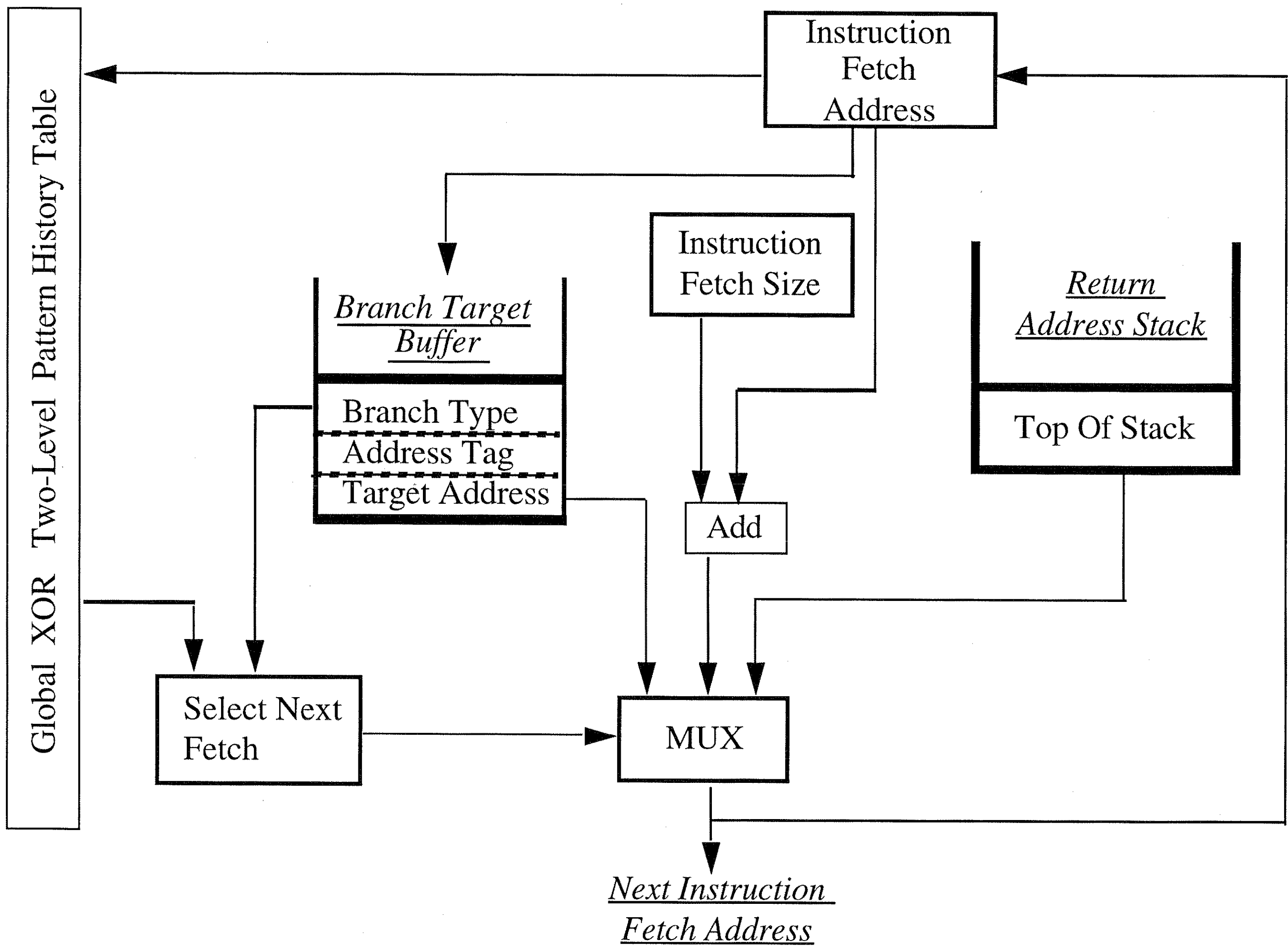


Figure 2.4: A Schematic Representation of a Decoupled GAg and Branch Target Buffer Architecture.

improve upon the BTB design by examining alternative ways of representing the target address for instruction fetch prediction. In Chapter 6, we examine using cache indices instead of full target addresses for instruction fetch prediction. In Chapter 7 we examine using precomputed branches instead of PC-relative branches. These precomputed branch offsets are then used for instruction fetch prediction. Both of these designs require fewer hardware resources then the BTB design, are simple direct mapped designs which have a fast access time, and they achieve better instruction fetch prediction performance than the BTB.

2.3 Profile Code Transformation Optimizations

There has been considerable work on profile-driven program optimization, and this section describes related work on optimizations for instruction caches and branch mechanisms.

2.3.1 Optimization for Memory Hierarchies

Due to the expense of memory on early computers, most early optimizations focused on reducing paging in virtual memory systems. Several researchers explored ways to group related subroutines or basic blocks onto the same virtual memory page [1, 24, 27, 28, 35]. Other researchers extended this work to lower levels of the memory hierarchy, optimizing the performance of instruction caches. McFarling [40] described an algorithm to reduce instruction cache conflicts for a particular class of programs. Hwu and Chang [43] describe a more general and more effective technique using compile time analysis in the IMPACT-I compiler system. Using profile-based transformations, the IMPACT-I compiler inlines subroutines and performs trace analysis. For each subroutine, instructions are packed using the most frequently executed traces, moving infrequently executed traces to the end of the function. Following this, global analysis arranges functions to reduce inter-function cache conflicts. Similar transformations were applied by Pettis and Hansen [49] for programs on the HP PA-RISC.

2.3.2 Optimizations for Control Flow

McFarling and Hennessy [42] described a number of methods to reduce branch misprediction and instruction fetch penalties, including profile-driven static branch prediction, delay slots, delayed slots with squashing, and a form of branch alignment. Their goal for branch alignment was to make the "taken" path the most commonly executed path to improve the performance of the delayed slots with squashing branch prediction architecture. Later, Bray and Flynn [6] studied the same idea, except they examined making the "fall-through" the most common case. They studied the performance gain of their algorithm in terms of conditional branch prediction accuracy for only the branch target buffer architecture.

Yeh **et al.** [66] commented that taken branches could only be reduced from $\approx 62\%$ of the executed conditional branches to $\approx 50\%$ of the executed conditional branches with trace scheduling. An earlier study by Hwu and Chang [43] showed an $\approx 58\%$ fall-through rate after branch alignment. The papers by McFarling and Hennessy, and Bray and Flynn did not report their change in the percentage of taken branches, and they did not describe the basic block order in which their algorithms where applied when laying out the basic blocks in a procedure.

The branch alignment reordering algorithm proposed by Hwu and Chang [43] is a general algorithm which grows the basic block layout list, first forwards then backwards, starting with the basic block edge that has the highest execution count for a given procedure. The work by Pettis and Hansen [49] describes a greedy algorithm for branch alignment which is similar to Hwu and Chang's. The Pettis and Hansen greedy algorithm is more general than the Hwu and Chang algorithm, and performs better in terms of reducing the cost of branches. The Pettis and Hansen algorithm is described in detail in Chapter 5 on branch alignment. One of the goals of this research is to try to improve instruction fetch prediction using compiler optimizations. In Chapter 5 we examine using the Pettis and Hasen algorithm for improving instruction fetch prediction for current static, PHT and BTB architectures. Where the goal is to remove misfetch penalties by making the fall-through path the most frequently executed path.

2.4 Reducing Branch Penalties in C++

This dissertation provides branch and instruction fetch prediction results for many applications, including applications written in the object-oriented programming language C++. These studies are among the first to examine the effects of branch and instruction fetch prediction on object-oriented programming languages.

In a previous study [16], we examined the behavioral differences between C and C++ applications. In this study we found that C++ programs execute many more indirect jumps than C programs since dynamic dispatch calls in C++ are implemented using indirect jumps. This effect is also shown later in Chapter 3, where conditional branches only account for 61% of the branches executed in C++ programs, compared to 80% of the branches executed in C programs and 87% of the branches executed in FORTRAN programs. The difference in the branch execution mix comes from the C++ programs executing many more procedure calls, indirect procedure calls, and return instructions. Another difference we found in our previous study comparing C++ and C programs [16], is that C++ procedures execute 3 times fewer instructions than C procedures, and the average procedure size of C++ methods are 50% smaller the average C procedure size. The small C++ static function size and number of instructions executed, implies that optimizations such as procedure in-lining may be very effective for C++ applications. These optimizations can only be applied to direct procedure calls, and dynamic dispatch calls (indirect jumps) can cause serious problems for procedure in-lining and other compiler optimizations.

This prompted us to perform a related study [9], where we examine eliminating dynamic dispatch calls through compiler optimizations and study the accuracy of existing branch prediction architectures on predicting C++ applications. In this study, we found that many of the dynamic dispatch calls can be eliminated and that C++ programs can be accurately predicted using existing branch prediction architectures. We showed that many of the dynamic dispatch calls can be eliminated using either a link-time compiler optimization called Unique Name elimination or a compiler optimization we call If-conversion. **Unique Name** elimination examines the type signature of a call site and if there is only a single function with that signature, then no other function could be the destination of this call site and the call site can be implemented as a direct procedure call. We found that the Unique Name optimization eliminates on average 32% of the indirect procedure calls, converting these into direct procedure calls. The other optimization, called **If-conversion**, converts a dynamic dispatch call to a series of `if/then/else` or `switch` statements. We can convert an indirect function call, e.g.,`object -> foo();` to a conditional procedure call with the run-time type check:

```
if (typeof(object) == A )
    object -> A::foo();
else
    object -> foo();
```

This transformation is useful for three reasons. First, once this code transformation has been performed, function call `A::foo()` can be in-lined. Secondly, if there is a high likelihood of calling `A::foo()`, this code sequence may be less expensive than performing the indirect procedure call. Lastly, if an architecture providing conditional branch prediction but does not support prediction for indirect function calls, the transformed code can avoid many mispredict penalties. The idea of If-conversion was used to perform in-line cache optimizations in Smalltalk [21], and was used by Hölzle **et.al.** [30] to improve the performance of the SELF language. Normally these optimizations would be performed in an optimizing compiler, possibly eliminating the need to predict indirect jumps. Note, there is a trade off for using the If-conversion optimization, since more conditional branches will be executed, and this may degrade the performance of a PHT conditional branch architecture by creating alias effects. There are many such trade-offs that need to be considered when performing these optimizations.

CHAPTER 3

EXPERIMENTAL METHODOLOGY

In evaluating solutions for instruction fetch prediction, large programs were sought, both in terms of source code size and in terms of number of instructions executed. Where possible, programs that were in widespread use and familiar to a broad audience of users were selected. The SPEC92 programs met this requirement. C++ programs were also included to examine the predictability of object-oriented languages.

3.1 Experimental Tools

This dissertation studies many branch prediction architectures using information from direct-execution simulation. Three tools were used to gather the results for this dissertation: ATOM, OM and Zippy. ATOM is a tool from DEC-WRL [55] used to instrument programs. OM is a link-time optimization tool, also from DEC-WRL [56, 57], that was used to perform the branch alignment optimizations. Zippy is a pipeline-level simulator used to measure program execution time and to attribute portions of that time to different processor subsystems.

3.1.1 ATOM

The trace generation tool ATOM [55] was used to gather static and dynamic statistics of programs and to simulate several different branch prediction architectures. Due to the nature of this tool, it was not necessary to record traces. Instead, the programs were traced and the simulations were ran all at the same time. This allows for the tracing of very long-running applications. The simulators constructed were typically run once to collect information on call and branch targets, and a second time if needed using profile information from the prior run.

3.1.2 OM

The OM [56, 57] linker translates the object code of an entire program into symbolic form, recovering the original basic block and procedure structure of the program. OM also provides a symbolic form of the program that can easily be manipulated to delete, insert and reorder instructions. When OM is invoked, it analyzes and optimizes this symbolic form, and when it is finished it outputs the transformed code, generating a new executable. OM was used to implement different branch alignment algorithms, creating a link-time branch alignment code optimizer.

3.1.3 Zippy

Zippy is a cycle-level simulator for the DEC Alpha architecture that simulates all resources on the Alpha, including the processor bus, the first and second level caches, the main memory, the functional units and all the hardware interlocks. Zippy is a direct-execution simulator; the program being measured is executed, and events generated by the program execution are fed to an event-driven simulator. The version of Zippy used in this study used the ATOM instrumentation system to instrument the applications. A considerable amount of analysis can be performed during the instrumentation phase. For example, interlocks and dependence between instructions can be computed statically. Some aspects of the simulation can be simplified as well. Since virtually-indexed direct-mapped caches are used, instruction references to a cache line can be precomputed and need only occur when a cache line might be fetched. Zippy was used to perform the system level study of different branch and instruction fetch prediction architectures in Chapter 4.

3.2 Program Statistics

We instrumented all the programs from the SPEC92 benchmark suite, the Perfect Club benchmarks, some other C programs, and object-oriented programs written in C++. The description for all the FORTRAN programs are given in Table 3.1, the C programs in Table 3.2, and the C++ programs in Table 3.3. The default input was used for each of the Perfect Club benchmarks. All the programs were compiled on a DEC Alpha 3000-400 running OSF/1 V2.0, using either the DEC FORTRAN, C, or C++ compiler. All programs were compiled with standard optimization (`-O`). The alternative C++ programs were selected because the SPECint92 suite did not typify the behavior seen in C++ programs [9, 16], and we wanted to understand the impact of branch and instruction fetch prediction on C++ programs. For these alternate programs, sizable inputs were used that would hopefully exercise a large part of the program.

Table 3.4 shows the basic statistics for the programs instrumented. The first column lists the number of instructions traced, in millions, and the second column indicates the percentage of those instructions that are branches. The third column shows the percentage of conditional branches that are taken. The next five columns divides the number of branches encountered during tracing into five classes: conditional branches (**CBr**), unconditional branches (**Br**), indirect jumps (**IJ**), procedure calls (**Call**), and procedure returns (**Ret**).

Table 3.5 shows the quantiles for the conditional branches in the traced programs. The columns labeled 'Q-25', 'Q-50', 'Q-75', 'Q-90', 'Q-95', 'Q-99' and 'Q-100' show the number of branch instruction sites that contribute to 25, 50, 75, 90, 95, 99 and 100% of all executed branches in the program. Thus, in `doduc`, three branch instructions constitute 50% of all executed branches. The ninth column shows the total number of static conditional branch sites in each program.

3.3 Metrics

This section gives a brief summary of the metrics used in this dissertation to compare branch architecture performance.

3.3.1 Percent Increase in Execution Time (%IET)

Chapter 4 compares the execution time of each program to a system with an unobtainable "perfect" branch prediction architecture. Perfect branch prediction assumes that branches are never misfetched or mispredicted. Program execution using other branch architectures are specified as a percent increase in execution time (%IET), or slowdown, relative to the perfect branch execution architecture. The slowdown includes the affect of all system components.

3.3.2 Cycles Per Instruction (CPI)

Obviously, there is a considerable variation in the number of instructions issued by different programs, as shown in Table 3.4. Cycles per instruction provides a more application-independent performance metric for a given architecture. In certain cases, the CPI is used to compare branch architecture performance, particularly when determining how well other performance metrics predict program performance.

3.3.3 Branch Execution Penalty (BEP)

Many researchers [6, 48] have used the percent of mispredicted branches to compare the performance of different branch architectures. However, this metric is too simplistic to capture the effects of misfetch penalties. There are two forms of pipeline penalties to be concerned with: misfetching and misprediction. All PC-relative branches can be misfetched, but only conditional branches, indirect function calls and returns can be mispredicted. As described in Chapter 2, the penalty for misfetching is less than the penalty for misprediction. One may be willing to misfetch more branches if it means one can reduce the number of mispredicted branches. Thus, the percentage of misfetched branches (%MfB) **and** the percentage of mispredicted branches (%MpB) are recorded. It is often difficult to understand how these metrics influence processor performance. Yeh & Patt [66] defined a formula to combine these branch penalties called the **branch execution penalty**:

Table 3.1: Description of FORTRAN Applications.

Perfect Club	
Programs	Description
APS	Air pollution, Fluid dynamics.
CSS	Circuit simulation, Engineering design.
LGS	Lattice gauge, Quantum chromodynamics.
LWS	Liquid water simulation, Molecular dynamics.
NAS	Nucleic acid simulation, Molecular dynamics.
OCS	Ocean simulation, 2-D fluid dynamics.
SDS	Structural dynamics, Engineering design.
TFS	Transonic flow, 2-D Fluid dynamics.
TIS	2-electron transform integrals, Molecular dynamics.
WSS	Weather simulation, Fluid dynamics.

SPEC 92	
Programs	Description
doduc	Monte Carlo simulation of the time evolution of a thermohydraulical modelization for a nuclear reactor's component. Input was ref.in.
fpppp	Quantum chemistry benchmark measuring performance of two electron integral derivatives in the Gaussian series of programs. Input was ref.in.
hydro2d	Hydrodynamical Navier Stokes equations are solved to compute galactical jets. Input was ref.in
mdljsp2	Solves the equations of motion for a model of 500 atoms interacting through the idealized Lennard-Jones potential. Input was input.file.
nasa7	A collection of 7 kernels. No input file.
ora	ORA traces rays through an optical system composed of spherical and plane surfaces. Input was ref.in
spice	An analog circuit simulation. Input was ref.in.
su2cor	Quantum physics benchmark where masses of elementary particles are computed in the framework of the Quark-Gluon theory. Input was ref.in.
swm256	Shallow Water Model with 256x256 grid. Input was swm256.in.
tomcatv	A vectorized mesh generation program. Not input.
wave5	A two-dimensional, relativistic, electromagnetic particle-in-cell simulation code used to study various plasma phenomena. No input.

Table 3.2: Description of C Applications.

Programs	Description
alvinn	A program that trains a neural network called ALVINN (Autonomous Land Vehicle In a Neural Network) using backpropagation.
compress	A file compression program, version 4.0, that uses adaptive Lempel-Ziv coding. The test input required compressing a one million byte file.
ear	Simulates the propagation of sound in the human cochlea (inner ear) and computes a picture of sound called a cochleagram
eqntott	A translator from a logic formula to a truth table, version 9. The input was the file int_pri_3.eqn.
espresso	A logic optimization program, version 2.3, that minimizes boolean functions. The input file was an example provided with the release code (cps.in).
gcc	A benchmark version of the GNU C Compiler, version 1.35. The measurements presented shows the execution of the "cc1" phase of the compiler. The input used was a preprocessed 4832-line file (1stmt.i).
li	A Lisp interpreter that is an adaptation of XLISP 1.6 written by David Michael Betz. The input measured was a solution to the N-queens problem where N=8.
sc	A spreadsheet program, version 6.1. The input involved cursor movement, data entries, file handling, and some computation.
bc	GNU arbitrary precision calculator. Input was an example that finds all primes between 2 and 2000.
flex	Fast lexical analyzer generator. Input was flex-scan.l.
gzip	GNU compression utility. Input was the same ref.in used for compress.
indent	Indents and formats program source. Input was a 220 Kbyte C program.
od	Writes the contents of a file to standard output in octal and other formats. Input was the od's own od.rr file.
wdiff	Compares two files, finding which words have been deleted or added. Input, two versions of CU's computer science /etc/hosts files.

Table 3.3: Description of C++ Applications.

Programs	Description
cfront	The AT&T C++ to C conversion program, version 3.0.2. Input was `groff.C`, part of the GNU `troff` implementation. The input was first preprocessed with `cpp`.
db++	A version of the "delta-blue" constraint solution system written in C++. Input was an example program that comes with the Deltablue system.
groff	Groff Version 1.9 — A version of the "ditroff" text formatter. Input was a collection of manual pages.
idl	Sample backend for the Interface Definition Language system distributed by the Object Management Group. Input was a sample IDL specification for an early release of the Fresco graphics library.
lic	Part of the Stanford University Intermediate Format (SUIF) compiler system. It is a **linear inequality calculator**. Input was the largest distributed example.
porky	Part of the Stanford University Intermediate Format (SUIF) compiler system. It performs a variety of compiler optimizations. It was used to perform constant folding, constant propagation, reduction detection and scalarization for a large C program.

Table 3.4: Measured Attributes of Traced Programs.

Program	# Insn Traced (Millions)	% Branches	% CBr Taken	Percentage of Branches During Tracing				
				%CBr	%Br	%IJ	%Call	%Ret
APS	1490	5	51	85	6	0	5	5
CSS	379	9	56	78	10	2	5	5
LGS	956	9	67	77	3	0	10	10
LWS	14183	10	66	80	6	0	7	7
NAS	3604	5	61	64	13	2	11	11
OCS	5187	3	89	99	0	0	0	0
SDS	1109	7	53	99	0	0	0	0
TFS	1694	3	77	92	3	0	2	2
TIS	1722	5	51	100	0	0	0	0
WSS	5422	5	62	87	6	3	2	2
doduc	1150	9	49	81	5	0	7	7
fpppp	4333	3	48	87	8	0	3	3
hydro2d	5683	6	73	96	1	0	1	1
mdljsp2	3344	11	84	95	4	0	0	0
nasa7	6128	3	79	81	6	0	6	6
ora	6036	8	53	70	11	0	10	10
spice	16148	13	72	92	4	0	2	2
su2cor	4777	4	73	76	9	1	7	7
swm256	11037	2	98	100	0	0	0	0
tomcatv	900	3	99	100	0	0	0	0
wave5	3555	6	62	77	6	1	8	8
alvinn	5241	9	98	98	0	0	1	1
compress	93	14	68	89	8	0	2	2
ear	17006	8	90	61	4	0	17	17
eqntott	1811	12	90	93	2	2	1	2
espresso	513	17	62	93	2	0	2	2
gcc	144	16	59	79	6	3	6	6
li	1355	18	47	64	8	2	13	13
sc	1450	21	64	86	3	1	5	5
bc	93	16	42	64	7	1	14	14
flex	15	15	68	85	7	1	4	4
gzip	309	13	61	89	6	0	3	3
indent	32	17	52	85	7	2	3	3
od	210	18	46	72	16	2	5	5
tex	148	10	57	76	10	3	5	6
wdiff	76	17	54	78	16	0	3	3
cfront	17	13	53	76	6	3	8	8
db++	86	18	57	54	2	15	7	22
groff	57	18	49	66	10	3	9	11
idl	21	20	47	50	8	12	9	21
lic	6	17	52	66	9	0	13	13
porky	164	20	60	55	3	3	18	21

Table 3.5: Conditional Branch Quantiles for Traced Programs.

Program	Q-25	Q-50	Q-75	Q-90	Q-95	Q-99	Q-100	Static
	Conditional Branch Quantiles							
APS	18	44	123	283	357	524	1,617	8,926
CSS	8	32	109	211	262	467	2,202	9,670
LGS	3	8	22	42	56	86	1,344	7,306
LWS	2	3	9	18	26	38	1,148	6,927
NAS	2	5	14	34	69	125	1,663	7,614
OCS	1	3	10	46	79	197	1,447	7,084
SDS	1	9	25	43	67	169	1,669	7,585
TFS	6	15	38	122	220	464	1,598	7,270
TIS	2	8	20	31	36	66	863	6,292
WSS	10	41	145	275	344	533	1,756	7,592
dodoc	1	3	40	175	231	296	1,447	7,073
fpppp	5	10	28	51	73	109	744	6,260
hydro2d	7	14	43	74	111	230	1,613	7,088
mdljsp2	3	6	10	14	16	23	1,010	6,789
nasa7	3	8	21	55	94	277	1,083	6,581
ora	2	5	8	11	12	17	641	5,899
spice	1	2	12	38	63	116	1,762	9,089
su2cor	3	8	15	26	34	60	1,569	7,246
swm256	1	2	2	3	3	13	795	6,080
tomcatv	2	3	4	5	7	7	515	5,474
wave5	6	18	40	82	132	276	1,331	8,149
alvinn	1	2	2	2	3	102	430	1,622
ear	1	2	4	6	8	32	530	1,846
compress	2	4	7	12	14	16	230	1,124
eqntott	1	2	2	14	42	72	466	1,536
espresso	14	44	104	163	221	470	1,737	4,568
gcc	55	245	804	1,612	2,309	3,724	7,640	16,294
li	5	16	33	52	80	127	556	2,428
sc	3	14	41	94	153	336	1,471	4,478
bc	14	41	97	160	204	273	753	1,956
flex	4	29	102	190	260	421	1,204	2,969
gzip	2	3	13	29	36	49	342	2,476
indent	8	27	74	159	244	457	1,065	2,272
od	14	30	56	76	84	118	433	1,702
tex	11	39	111	259	416	790	2,365	6,050
wdiff	4	7	11	19	24	29	502	1,618
cfront	33	112	345	946	1,540	3,055	5,783	15,509
db++	2	9	40	96	137	173	421	1,639
idl	4	9	15	38	67	154	1,001	3,839
groff	26	86	182	372	564	1,021	2,511	7,434
lic	42	82	123	146	154	161	2,311	18,897
porky	2	7	43	122	238	557	2,553	8,808

$$\text{BEP} = \frac{\%\text{MfB} \times \text{misfetch penalty} + \%\text{MpB} \times \text{misprediction penalty}}{100},$$

which reflects the average penalty suffered by a branch due to misfetch and misprediction. A BEP of 0.5 means that, on average, each branch takes an extra half cycle to execute; values close to zero are desirable. I have assumed a one cycle misfetch penalty and a four cycle misprediction penalty for the performance studies presented in Chapters 5, 6, and 7 of this dissertation.

CHAPTER 4

SYSTEM LEVEL PERSPECTIVE

Previous branch prediction studies compare different branch prediction architectures using misprediction rates, branch penalties, or an idealized cycles per instruction. With these ideal cycles per instruction and branch miss rate metrics, one cannot really quantify how these branch and instruction fetch prediction architectures will effect a real processor. Therefore, this chapter provides a system level study of several branch and instruction fetch prediction architectures, including recently proposed two-level correlated branch history architectures. The system-level simulations were performed for the Alpha AXP 21064, a dual-issue statically-scheduled processor. The performance of various branch architectures is reported using execution time and cycles-per-instruction. This chapter also shows that the branch execution penalty metric used throughout this dissertation is highly correlated with program performance and is a suitable metric for architectural studies. [1]

4.1 Introduction

Usually, branch architectures are studied in isolation, ignoring other aspects of the system. In those studies, a variety of metrics are used to assess the performance of different branch architectures, but it is rare that the performance of an entire system is reported. There are numerous reasons for this. The complexity of accurately simulating a complex architecture is daunting and the results are specific to that architecture. This makes it is difficult to translate the performance of that architecture to another, possibly different architecture. However, there are advantages to system-level simulations. They provide the reader with a sense of the magnitude and importance of the problem being solved, and indicate where further effort should be placed. Furthermore, the performance metrics, such as execution time and cycles per instruction, are more understandable to the casual reader. Lastly, such studies provide a method to calibrate the performance metrics used in other studies with the impact on system level performance.

This chapter studies several branch and instruction fetch prediction architectures using information from direct-execution simulation. A pipeline-level simulator was used to measure program execution time and attribute portions of that time to different processor subsystems. We first describe the simulator used and compare measurements from that simulator to measurements from an actual system. We then describe the branch architecture's examined in this chapter, followed by their system level performance. The chapter concludes by showing that the BEP is an appropriate metric for comparing branch and instruction fetch prediction architectures.

4.2 Verification of Zippy Simulation Methodology

The system level performance results were gathered using Zippy. Zippy can be configured to use one of a number of processor and system configurations. The system we simulated is a DEC Alpha 3000-500 workstation, with a DECchip AXP 21064 processor clocked at 150Mhz. The system contains an 8KB on-chip direct-mapped instruction cache and an 8KB on-chip direct-mapped store-around data cache. The data cache is pipelined, and has a two-cycle latency and single cycle access time. There is a 512KByte second-level, unified direct-mapped cache with a five cycle latency and access time. Main memory has a 24 cycle latency and 30 cycle access time. The branch misfetch penalty is 1 cycle and the mispredict penalty is 5 cycles. More information about the AXP 21064 can be found in the processor reference manual [22].

Table 4.1 compares the cycle count reported by Zippy to the actual cycle count taken for the same application, recorded by the processor cycle counter on an Alpha 3000-500 processor. We ran each application 1000 times, and the measured values had a 95% confidence interval typically less than 0.06% of the mean.

---

[1] Parts of this chapter were published in the 28th International Symposium on Microarchitecture [13].

Table 4.1: Verification of Zippy Architecture Model

Program	Zippy Estimate	Measured	% Diff	PAL Calls	%PAL Increase
doduc	3744530000	3852783497	-2.89	43	0.0
tomcatv	4211550000	3428500986	22.83	128	0.0
compress	305373000	305036415	0.00	208	3.4
espresso	820013000	798971677	2.63	86	0.2
gcc	383407000	406986172	-6.14	180	1.1
li	3446060000	3840210240	-11.43	32949	0.0
bc	223635000	227787352	-1.85	535	0.6
flex	30794800	34237883	-11.18	35	1.0
gzip	552878000	559296680	-1.16	67	0.7
indent	76213700	80816742	-6.03	133	3.1
od	591999000	545321384	8.55	193	2.9
cfront	61496200	68785345	-11.85	81	1.5
lic	15324900	17928028	-16.98	337	4.9
porky	439329000	472871359	-7.63	241	0.7

Shows the number of cycles estimated by Zippy while simulating the Alpha 3000-500 (Estimated), the number of actual executed cycles measured on an Alpha 3000-500 minus the Pal call cycles (Measured), the percent difference in the Zippy model and measured cycle counts (% Diff), the number of dynamic PAL calls executed in the measured programs (PAL Calls), and the percent increase in number of cycles executed if PAL calls were taken into account.

The measurements in Table 4.1 shows that the average difference between the cycle time reported by the Zippy model and the actual execution time is less than 8% (absolute error) for the programs we measured. There are a number of reasons why the estimated performance of Zippy would differ from the measured performance. First, we used a prototype 3000-500 for our measurements, and the memory subsystem was slightly different for that architecture; we think this explains the large variance of `tomcatv`, since it has a large data cache miss rate. Second, Zippy simulates the processor executing a single program; it does not simulate the underlying operating system and other programs running on the system. Normally, when a process requests a system service, system code displaces parts of the instruction and data cache. Likewise, when a program begins execution, the pages are demand-faulted into the program address space. The layout of pages in memory may cause cache conflicts [5, 36]; these are not detected by Zippy, because it assumes a perfect page coloring scheme. These problems should be most visible in short-running programs or those that make a large number of system calls. Table 4.1 confirms this; short-running programs such as `flex`, `indent`, `cfront` and `lic` have a high variance. On the Alpha, a "PALcall" is used to access system routines and for system-specific functions; the column showing the number of PALcalls shows that `li` executes many system calls, and also has a large variance. Other programs with a large difference in the execution time initiate I/O, which causes cache flushes and more system level disturbances. Even with these aspects of the system not being modeled by Zippy, the measurements in this experiment show that Zippy provides a reasonably accurate performance estimate for the 21064 architecture.

4.3 Processor Performance with Perfect Branch and Fetch Prediction

Table 4.2 shows the average execution time in seconds and the cycles-per-instruction (CPI), assuming perfect branch and instruction fetch prediction, for each program included in this chapter's system level study. The remaining columns give the break down of cycles executed, and attributes those cycles to the instruction issues (Issues), stall cycles due to the inability to issue instructions (Stalls), instruction cache (ICache), data cache (DCache), and the translation look-aside buffer (TLB). For example, in `doduc`, 25.4% of the execution time is from issued instructions. Stalls account for 47.7% of the execution time. Waiting for misses in the instruction and data caches consumed 9.7% and 17.0% of the cycles, and TLB misses accounted for 5.2% of the time. We combined all cache misses from the first and second level cache into the ICache or DCache value.

The "Stalls" entry includes interlocks and data dependence constraints that can be computed statically. For example, the result of a load is not available until two cycles after it is issued. An instruction using the result of that load immediately following the load would have to stall at least that long. The cycles induced by the dependence constraint are attributed to stalls, but any additional cycles induced by cache misses are attributed to those misses. Most hardware interlocks encountered in the 21064 can be statically determined; some interlocks, particularly for floating point computations, may span several basic blocks, and are reported in the "Stalls" column.

4.4 Metrics

We used several metrics to compare branch architecture performance in this chapter. The first two are the **increase in execution time** and **cycles per instruction**. The third metric, **branch execution penalty** (BEP), is often used to select one branch architecture over another. The percent increase in execution time (%IET) compares the execution time of a program for each branch architecture on a 21064 processor to a system with an unobtainable "perfect" branch prediction architecture. This reports the slowdown seen when using a specific branch and instruction fetch prediction architecture relative to perfect branch prediction. Cycles per instruction provides an application-independent performance metric for a given architecture. The CPI is also used to compare branch architecture performance, particularly when determining if the BEP branch performance metric predicts program performance. The BEP reflects the average cycle penalty suffered by branch instructions due to misfetch and mispredict penalties. These three metrics are described in more detail in Chapter 3.

Table 4.2: Zippy Performance Assuming Perfect Branch Prediction

Program	Exec Time (s)	CPI	% of Cycles Issues	Stalls	ICache	DCache	TLB
doduc	24.249	3.195	25.4	47.7	9.7	17.0	0.2
tomcatv	27.654	4.657	16.3	12.6	0.1	70.9	0.2
compress	1.897	3.103	27.0	8.5	0.0	22.3	42.2
eqntott	14.969	1.253	74.1	11.0	2.5	11.4	1.1
espresso	4.666	1.378	62.3	19.4	3.2	13.9	1.2
gcc	2.314	2.439	35.0	13.6	20.8	25.1	5.5
li	20.718	2.317	36.7	17.4	15.5	30.3	0.0
bc	1.338	2.171	38.7	17.0	22.9	19.9	1.4
flex	0.185	1.811	47.9	19.1	11.9	20.6	0.4
gzip	3.363	1.646	54.5	19.2	1.6	24.6	0.0
indent	0.463	2.154	40.6	22.7	18.4	17.7	0.7
od	3.648	2.628	31.7	11.3	23.7	31.5	1.9
wdiff	0.984	1.956	43.5	41.5	0.1	14.9	0.0
cfront	0.392	3.594	23.7	10.3	23.2	29.5	13.3
idl	0.287	2.058	41.1	14.6	10.4	26.2	7.7
lic	0.093	2.361	36.2	19.6	20.0	19.9	4.3
porky	2.592	2.399	35.5	16.6	12.1	27.9	7.9
Avg	6.460	2.419	39.4	18.9	11.5	24.9	5.2

The first column shows the estimated execution time, followed by the cycles per instruction (CPI). The remainder of table indicates what fraction of the processor cycles were attributed to different subsystems in the AXP 21064 system.

4.5 Branch Architectures Simulated

This section briefly describes the branch architectures we simulated for this system level study. Table 4.3 gives a summary of these architectures. Chapter 2 gives a more detailed description of each architecture. We simulated the DECchip 21064 implementation of the Alpha AXP architecture using perfect branch prediction (Perfect), no branch prediction (No-Pred), backwards taken/forwards not taken (BTFNT), the default 1-Bit branch prediction provided on the 21064 (Alpha-Ev4) architecture, pattern history tables (PHT), branch target buffers (BTB), and two types of correlated conditional branch prediction architectures (PAs and GAg).

For Perfect branch prediction we assume that all branch targets are fetched correctly without any penalties, so there are no misfetch or mispredict penalties. This establishes an upper-bound that the branch architecture can have on system performance. For no branch prediction (No-Pred), we assume that the instruction fetch pipeline halts whenever a branch is encountered and the pipeline stalls until the correct branch destination address is known. This is the worst possible branch architecture, and establishes a lower bound on branch performance for the 21064 architecture. For No-Pred, all branches either cause a misfetch penalty or a mispredict penalty depending on the branch type. The only static branch prediction architecture we simulated was the simple backward-taken, forward not-taken (BTFNT) architecture. The BTFNT architecture predicts that backwards branches are looping branches which will be taken, and forward branches are predicted as not taken.

The Alpha 21064 used in the 3000-500 we simulated (Alpha-Ev4) adds one bit to each instruction in the 8K direct mapped instruction cache to dynamically predict the direction of conditional branches. This bit is initialized with the sign-bit of the PC-relative displacement when the cache line is read in, initializing the dynamic 1-bit predictors to a BTFNT prediction. Thus, the BTFNT rule is used the first time the branch is encountered, and a dynamic one-bit predictor is used thereafter.

The pattern history table (PHT) branch architecture is an example of an architecture using two-bit saturating up-down counters. It contains a table of two-bit counters used to predict the direction for conditional branches. In the direct mapped PHT architecture (PHT-Direct), the branch address is used to directly index into the pattern history table to find the two-bit counter to use when predicting the branch. In comparison, the correlated global history architecture uses the history of previous branches to determine which two-bit counter in the table to use. The PHT-GAg method we modeled uses a variant of the correlated branch prediction schemes described by McFarling [41]. This method used the exclusive-or of the global history register and the branch address as the index into the PHT.

The BTB-2Bit architecture models the architecture used in the Pentium, where each BTB entry has a two-bit saturating counter to predict the direction of a conditional branch [38]. The BTB-PAs architecture models the architecture proposed by Yeh and Patt [66], where each BTB entry contains a 6-bit history register. This history register is used as the lower bits of an index into a PHT when predicting conditional branches. The upper bits of the index are the lower bits of the branch address. In both the BTB-2Bit and BTB-PAs architectures, the branch prediction information (the two-bit counter and 6-bit history register), is associated or **coupled** with the BTB entry. Therefore, the dynamic conditional branch prediction information can only be used for branches in the BTB, and branches that miss in the BTB must use less accurate static prediction. The BTB-GAg is an example of a **decoupled** design, where the branch prediction information is not associated with the BTB and is used for all conditional branches, including those not recorded in the BTB. For this architecture, the BTB is used only to identify when a branch has been fetched and to provide target addresses, and a separate PHT-GAg architecture is used to predict the direction of conditional branches.

We simulated a number of configurations of these architectures. In particular, we varied the number of BTB entries, associativity of the BTB, and the size of the PHT. All of the BTB designs we simulated only update the BTB with taken branch addresses. This was shown to be more effective for both the coupled and decoupled architectures because the BTB is not filled with fall-through addresses that are easily calculated [7, 48]. Therefore, on a BTB miss for the coupled design, the branch is predicted as not-taken. The other static prediction alternative is to predict the branch using the BTFNT scheme on a BTB miss. In our studies we have found that statically predicting the fall through direction performs better than BTFNT in this case.

Table 4.3: Architectures Simulated for System Level Study

Perfect	All branches are correctly predicted, so there are no mispredict nor any misfetch penalties.
No-Pred	No Branch Prediction. All branches are either mispredicted or misfetched.
BTFNT	Backwards Taken, Forwards Not Taken. Static branch prediction.
Alpha-Ev4	1-bit dynamic branch prediction for each instruction in the instruction cache.
PHT-Direct	Direct mapped Pattern History Table (PHT)
PHT-GAg	A single global history register is XORed with the program counter (PC) and used to index into the PHT.
BTB-2Bit	A BTB with each entry containing a 2-Bit counter for conditional branch prediction.
BTB-PAs	A BTB where each entry contains a 6-bit history register. This history register is used as the lower bits of an index into a PHT for predicting conditional branches. The upper bits of the index are the lower bits of the branch address.
BTB-GAg	A decoupled BTB that contains no conditional branch prediction information. Instead, conditional branches are predicted with the decoupled PHT-GAg architecture.
Infinite BTB's	An infinite BTB that only suffers from misprediction and cold-starts misfetches. We simulated this model for both the PAs and GAg (XOR) conditional branch prediction schemes.

4.6 Branch Architecture Performance

Throughout this section, the reader should be aware of the limitations of this study, and should not draw inferences beyond the systems model. In particular, the Alpha AXP 21064 is a statically scheduled, dual-issue processor. Dynamically-scheduled processors may be able to mask the effects of the branch architecture. Alternatively, dynamically-scheduled processors may be more sensitive to branch architecture performance, because more instructions would be "in-flight," increasing the mispredict penalty. It is difficult to extrapolate how these results would differ with out-of-order execution, and this topic is an issue of future work.

Table 4.4 shows the increase in execution time over an architecture using Perfect branch prediction for all the branch architectures that do not provide instruction fetch prediction (a branch target buffer). These simulations show that the dynamic branch prediction schemes reduce the branch overhead by approximately 50%, from 11% using BTFNT to $\approx$5% using a PHT architecture. Table 4.4 also shows that the simple direct-mapped PHT has slightly better performance than the more elaborate global history register (GAg) for a small PHT, but that the GAg method is slightly better for larger tables. The GAg method has better performance than the Direct method for large PHTs, since it uses a large number of PHT entries for a single branch, depending on the history of the previously executed branches. This leads to improved performance for a larger PHT, but can lead to more conflicts for a smaller PHT. Table 4.4 demonstrates that the marginal benefit of adding more PHT entries rapidly declines after 512 entries. The 4096-entry table uses eight fold more storage, but it only decreases the execution time by 0.6%. Larger improvements can be had by devoting those resources to more TLB entries or other scarce resources.

For the majority of the programs, further improvement from the branch architecture for the execution time requires that the misfetch penalty be reduced by providing instruction fetch prediction. For example, the `idl` program executes a large number of indirect branches, and additional conditional branch prediction accuracy provides very little performance improvement. Table 4.5 shows the percent increase in execution time for all the branch architectures examined for a subset of the programs simulated. `Doduc` is examined because it represents a branch intensive FORTRAN program. `Gcc` and `cfront` are examined because they have a large number of static branch sites, and `idl` because it has a large number of indirect jumps caused by dynamic dispatch calls in C++. The other SPECint92 programs are provided because they have been examined in detail in earlier branch studies.

Table 4.5 shows two obvious trends: a larger BTB is better than a smaller BTB, and a higher associativity BTB is better. These results aren't surprising, but it is interesting to note how little improvement is gained by adding a larger BTB or PHT, or by increasing the associativity of the BTB. The table shows that changing the architecture from a 64-entry direct mapped BTB-GAg architecture with a 512-entry PHT, to a costly 512-entry 4-way associative BTB-PAs architecture with a 4096-entry PHT, reduces the execution time by only 0.5% for the FORTRAN program `doduc` and up to a 2.7% for the C++ program `idl`.

The rows for the infinite BTB show the added benefit of removing all misfetch penalties except for those due to cold-start BTB misses. The infinite BTB also shows improvement from increased indirect jump prediction, and the prediction accuracy of the infinite BTB-PAs architecture will be further improved because each BTB entry has its own conditional branch history register. In each case, the programs with complex conditional branching behavior (`espresso`, `gcc`) are slightly improved by increasing the size of the PHT – but only by 1% of total execution time. Highly object-oriented C++ programs like `idl` gain the most from a large BTB because they execute a large number of indirect jumps that can be predicted by a BTB. However, even a modest BTB, such as a 64-entry direct-mapped BTB, provides most of the benefits achievable with this program.

Tables 4.6 and 4.7 show performance for all the branch architectures averaged over all programs. The different branch architectures are sorted by percent increase in execution time; architectures with poorer performance appear earlier in the table. Table 4.6 shows the average contribution of the instruction issues (Is), stalls (St), instruction cache (IC), data cache (DC), TLB (TB), mispredict branch (MpB), and misfetch branch (MfB) penalties to the overall execution time, as described earlier for Table 4.2. Table 4.7 shows the actual percentage of mispredicted branches, misfetched branches, and the branch execution penalty, which combines

Table 4.4: Percent Increase in Execution Time for Conditional Branch Architectures

Program	No Pred	BTFNT	Alpha-Ev4	512-Entry PHT Direct	512-Entry PHT GAg	4096-Entry PHT Direct	4096-Entry PHT GAg
doduc	9.68	1.88	1.92	1.02	1.26	0.96	0.94
tomcatv	2.59	1.74	0.51	0.51	0.51	0.51	0.51
compress	18.72	8.98	6.25	5.00	5.04	5.00	4.73
eqntott	44.05	24.15	11.12	9.63	9.29	9.57	9.27
espresso	56.62	19.28	15.98	12.03	10.11	11.93	8.90
gcc	24.84	11.34	9.34	6.79	7.73	6.24	6.23
li	25.82	8.69	9.78	5.82	5.34	5.80	4.59
bc	23.70	7.94	10.28	5.75	6.69	5.32	4.40
flex	34.62	15.30	9.82	7.67	7.81	7.45	6.78
gzip	33.22	12.75	8.51	7.09	7.25	7.06	6.77
indent	32.83	12.29	8.65	6.53	7.89	5.87	6.01
od	22.31	9.15	7.11	3.70	4.55	3.55	3.24
wdiff	34.48	15.52	7.31	5.80	5.85	5.80	5.74
cfront	10.68	3.57	3.51	2.24	3.04	1.87	1.97
idl	35.09	16.44	18.60	10.72	10.85	10.64	10.37
lic	24.23	8.74	9.24	5.41	6.35	5.17	5.07
porky	28.22	11.02	11.86	5.45	5.84	5.40	5.49
Avg	27.16	11.10	8.81	5.95	6.20	5.77	5.35

Percent increase in execution time over perfect branch prediction for an architecture using no branch prediction (No Pred), BTFNT, a direct mapped pattern history table, and a degenerate pattern history table.

Table 4.5: Percent Increase in Execution Time for a Subset of Programs

	BTB		PHT	Percent increase in Execution Time							
	Size	Assoc	Size	doduc	compress	eqntott	espresso	gcc	li	cfront	idl
No Pred				9.68	18.72	44.05	56.62	24.84	25.82	10.68	35.09
BTFNT				1.88	8.98	24.15	19.28	11.34	8.69	3.57	16.44
Alpha-Ev4				1.92	6.25	11.12	15.98	9.34	9.78	3.51	18.60
PHT-Direct			512	1.02	5.00	9.63	12.03	6.79	5.82	2.24	10.72
PHT-GAg			512	1.26	5.04	9.29	10.11	7.73	5.34	3.04	10.85
PHT-Direct			4096	0.96	5.00	9.57	11.93	6.24	5.80	1.87	10.64
PHT-GAg			4096	0.94	4.73	9.27	8.90	6.23	4.59	1.97	10.37
BTB-2Bit	64	1		0.90	2.91	1.80	9.44	6.51	4.61	3.42	5.28
BTB-2Bit	64	4		0.79	2.39	1.02	7.11	6.21	3.96	3.25	2.09
BTB-2Bit	512	1		0.55	2.39	0.93	6.32	4.00	2.94	2.05	1.19
BTB-2Bit	512	4		0.51	2.39	0.89	6.13	3.36	2.59	1.73	0.97
BTB-GAg	64	1	512	0.87	2.52	0.91	4.98	5.67	2.70	2.95	3.25
BTB-GAg	64	1	4096	0.54	2.19	0.88	3.71	4.13	1.99	1.93	2.73
BTB-GAg	64	4	512	0.84	2.40	0.57	4.50	5.57	2.57	3.00	1.82
BTB-GAg	64	4	4096	0.50	2.08	0.54	3.23	4.03	1.89	2.02	1.29
BTB-GAg	512	1	512	0.78	2.40	0.56	4.21	4.96	2.12	2.72	1.23
BTB-GAg	512	1	4096	0.44	2.08	0.53	2.93	3.39	1.45	1.68	0.74
BTB-GAg	512	4	512	0.77	2.40	0.55	4.25	4.80	2.04	2.66	1.13
BTB-GAg	512	4	4096	0.42	2.08	0.52	2.97	3.22	1.36	1.61	0.58
BTB-PAs	64	1	512	0.81	2.62	1.63	6.94	6.19	3.35	3.25	5.23
BTB-PAs	64	1	4096	0.77	2.58	1.62	6.50	5.91	3.15	3.15	5.08
BTB-PAs	64	4	512	0.68	2.10	0.70	4.42	5.87	2.86	3.12	1.97
BTB-PAs	64	4	4096	0.66	2.05	0.70	3.95	5.57	2.58	2.97	1.86
BTB-PAs	512	1	512	0.43	2.10	0.68	3.67	3.64	1.63	1.87	0.96
BTB-PAs	512	1	4096	0.40	2.05	0.67	3.23	3.37	1.51	1.71	0.85
BTB-PAs	512	4	512	0.38	2.10	0.64	3.47	3.01	1.38	1.52	0.73
BTB-PAs	512	4	4096	0.34	2.05	0.63	3.02	2.78	1.27	1.39	0.63
BTB-GAg	Inf		512	0.76	2.40	0.55	4.24	4.69	2.04	2.53	0.99
BTB-GAg	Inf		4096	0.42	2.08	0.52	2.96	3.10	1.36	1.48	0.47
BTB-GAg	Inf		32768	0.33	1.91	0.50	2.42	2.18	1.07	0.96	0.32
BTB-PAs	Inf		512	0.32	2.10	0.64	3.43	2.47	1.38	1.02	0.53
BTB-PAs	Inf		4096	0.30	2.06	0.63	3.00	2.28	1.27	0.89	0.45
BTB-PAs	Inf		32768	0.30	2.05	0.63	2.86	2.12	1.26	0.85	0.44

Percent increase in execution time for a subset of the simulated programs for all the branch architectures we studied. The branch architecture configurations are listed roughly by their order of complexity and cost. All of the configurations using a 512-entry PHT have been darkened in an effort to make the table more readable.

the %MpB and %MfB into one metric. The %MpB contains all branches that can be mispredicted, including indirect jumps, returns and conditional branches.

These tables reaffirm what was shown in Table 4.5; once a reasonable number of BTB entries have been added to a branch architecture, overall performance is best improved by adding resources to other architectural components, including TLB entries and the data cache. The results show that it is very important to have 2-bit conditional branch prediction, such as a PHT, although little performance improvement is seen as the size of the PHT is increased from a 512 entry PHT to a 4096 entry PHT. We found that increasing the PHT size for an infinite BTB beyond 32768 entries provided almost no improvement.

Table 4.7 also shows the improved performance when a BTB is added to the PHT branch architecture. For the 512 or 4096 entry PHT-GAg architecture, adding a direct mapped 64 entry BTB decreases the execution time by 3% and the CPI by 2.4%. Increasing the number of BTB entries beyond 64 results in only a very small performance improvement, and even less improvement is seen by increasing the associativity of the BTB. When increasing the BTB-GAg architecture from 64 entries to 512 entries, the execution time is decreased by 0.65% and the CPI by 0.57%. Increasing the associativity for a direct mapped 64 entry BTB-GAg to 4-way associativity decreases the execution time by 0.22% and CPI by 0.16%. For a 512 entry BTB-GAg architecture, the execution time is decreased by 0.12% and the CPI by 0.12% when the associativity is increased from direct mapped to 4-way. Each decrease in execution time comes with a price, in area, architecture complexity, and access time. Therefore, for a statically scheduled architecture like the Alpha 21064, the most aggressive branch prediction architecture that is justified by our experimentation is a direct mapped 64 entry BTB with a decoupled 512 entry direct mapped PHT.

Table 4.7 lets us directly compare the performance of the decoupled GAg and coupled PAs architectures with equivalent BTB sizes. For the 64 entry BTB designs, the GAg scheme performs better than the PAs architecture because the coupled PAs architecture can only use the conditional branch prediction information on a BTB hit, while the decoupled GAg uses the PHT predictors for all conditional branches. When increasing the size of the BTB to 512 entries, the PAs scheme performance usually surpasses that of the GAg, largely because of the increased BTB hit rate. However, this is not true in all cases; for example, a direct-mapped GAg with 512 BTB entries and 4096 PHT entries has (marginally) better performance than the corresponding PAs method. We feel the simpler GAg design should be used in an actual implementation, because the PAs architecture associates a 6-bit history register with each BTB entry, and this extra storage and complexity is not needed in the GAg architecture.

4.7 Validating the Branch Execution Penalty Metric

Branch architecture research, like all computer architecture research, is a computationally intensive undertaking. Normally, researchers try to identify a number of performance metrics that can be easily calculated, yet indicate the likely impact of a particular architectural feature. For example, the simulations used to accurately simulate the Alpha 21064 and calculate the CPI are approximately 500-1000 times slower than a simulation that only calculates the branch execution penalty (BEP). Numerous studies, including those in the remainder of the dissertation, have used the BEP for this very reason: it yields an intuitive performance metric, and it is significantly easier to calculate than the CPI. Other studies have used the percent of mispredicted branches (%MpB) for the same purpose.

We were curious how accurately the BEP and %MpB predict the CPI for the architectures we examined. The standard statistical test for this problem is to compute the sample correlation coefficient, ρ. Given ρ for two series, ρ^2 represents the probability that the series are linearly related; $\rho^2 = 99\%$ indicates that 99% of the variation in one series is predicted by a change in the second series. The CPI and BEP for all branch architectures averaged over all programs resulted in $\rho^2 = 99.9\%$; i.e., over all the architectures studied, a change in the BEP is a very accurate predictor for the CPI. The corresponding measurement for the %MpB is $\rho^2 = 96.9\%$, indicating that the %MpB is a slightly less accurate predictor of the CPI than the BEP. This is only natural, since the BEP includes the misfetch penalty as well as the mispredict penalty.

The predictive accuracy of the BEP and %MfB is also consistent across a range of branch architectures.

Table 4.6: Average System Level Performance

Arch	BTB Size	A	PHT Size	% IET	CPI	% of Cycles Is	St	IC	DC	TB	MpB	MfB
No Pred				27.16	2.981	30	15	9	20	4	17.87	2.70
BTFNT				11.10	2.647	35	17	11	23	5	4.42	5.33
Alpha-Ev4				8.81	2.605	36	18	11	23	5	1.56	6.39
PHT-GAg			512	6.20	2.549	37	18	11	24	5	0.86	4.91
PHT-Direct			512	5.95	2.542	37	18	11	24	5	0.72	4.82
PHT-Direct			4096	5.77	2.538	37	18	11	24	5	0.64	4.74
PHT-GAg			4096	5.35	2.530	37	18	11	24	5	0.51	4.51
BTB-2Bit	64	1		4.27	2.512	38	18	11	24	5	2.75	1.29
BTB-2Bit	64	4		3.65	2.500	38	18	11	24	5	2.41	1.06
BTB-2Bit	512	1		2.40	2.471	38	19	11	24	5	1.94	0.39
BTB-2Bit	512	4		2.00	2.462	39	19	11	25	5	1.81	0.14
BTB-PAs	64	1	512	3.86	2.504	38	18	11	24	5	2.43	1.25
BTB-PAs	64	1	4096	3.72	2.501	38	18	11	24	5	2.32	1.23
BTB-PAs	64	4	512	3.22	2.492	38	18	11	24	5	2.06	1.02
BTB-GAg	64	1	512	3.13	2.489	38	18	11	24	5	1.21	1.80
BTB-PAs	64	4	4096	3.05	2.488	38	18	11	24	5	1.93	1.00
BTB-GAg	64	4	512	2.91	2.485	38	18	11	24	5	1.37	1.43
BTB-GAg	512	1	512	2.49	2.475	38	19	11	24	5	1.77	0.63
BTB-GAg	512	4	512	2.37	2.472	38	19	11	24	5	2.09	0.21
BTB-GAg	Inf		512	2.33	2.471	39	19	11	24	5	2.25	0.01
BTB-GAg	64	1	4096	2.26	2.469	39	19	11	24	5	0.66	1.54
BTB-GAg	64	4	4096	2.04	2.465	39	19	11	24	5	0.79	1.19
BTB-PAs	512	1	512	1.89	2.461	39	19	11	25	5	1.48	0.36
BTB-PAs	512	1	4096	1.74	2.458	39	19	11	25	5	1.35	0.35
BTB-GAg	512	1	4096	1.61	2.455	39	19	11	25	5	1.02	0.56
BTB-PAs	512	4	512	1.49	2.452	39	19	11	25	5	1.33	0.13
BTB-GAg	512	4	4096	1.49	2.452	39	19	11	25	5	1.25	0.21
BTB-GAg	Inf		4096	1.44	2.451	39	19	11	25	5	1.40	0.01
BTB-PAs	512	4	4096	1.35	2.448	39	19	11	25	5	1.19	0.13
BTB-PAs	Inf		512	1.35	2.448	39	19	11	25	5	1.32	0.01
BTB-PAs	Inf		4096	1.21	2.445	39	19	11	25	5	1.19	0.01
BTB-PAs	Inf		32768	1.17	2.444	39	19	11	25	5	1.14	0.01
BTB-GAg	Inf		32768	1.08	2.443	39	19	11	25	5	1.06	0.01
Perfect				0	2.419	39	19	12	25	5	0	0

Average results for all the branch architectures sorted in terms of Percent Increase in Execution Time. The average cycles executed for each architecture is broken down into Issues, Stalls, I-Cache, D-Cache, TLB, Mispredicted Branches (MpB),and Misfetched Branches (MfB).

Table 4.7: Average Branch and Instruction Fetch Prediction Performance

Arch	BTB Size	A	PHT Size	%IET	CPI	Branch Penalties %MpB	%MfB	BEP
No Pred				27.16	2.981	86.85	13.15	4.47
BTFNT				11.10	2.647	30.95	39.00	1.94
Alpha-Ev4				8.81	2.605	14.12	63.31	1.34
PHT-GAg			512	6.20	2.549	10.51	63.93	1.16
PHT-Direct			512	5.95	2.542	8.88	64.91	1.09
PHT-Direct			4096	5.77	2.538	8.01	65.39	1.05
PHT-GAg			4096	5.35	2.530	6.90	65.74	1.00
BTB-2Bit	64	1		4.27	2.512	14.72	8.03	0.82
BTB-2Bit	64	4		3.65	2.500	12.92	6.91	0.72
BTB-2Bit	512	1		2.40	2.471	8.73	2.20	0.46
BTB-2Bit	512	4		2.00	2.462	7.36	0.77	0.38
BTB-PAs	64	1	512	3.86	2.504	13.51	8.03	0.76
BTB-PAs	64	1	4096	3.72	2.501	13.04	8.03	0.73
BTB-PAs	64	4	512	3.22	2.492	11.69	6.91	0.65
BTB-GAg	64	1	512	3.13	2.489	9.62	15.51	0.64
BTB-PAs	64	4	4096	3.05	2.488	11.13	6.91	0.63
BTB-GAg	64	4	512	2.91	2.485	9.47	12.59	0.60
BTB-GAg	512	1	512	2.49	2.475	9.35	3.94	0.51
BTB-GAg	512	4	512	2.37	2.472	9.32	1.22	0.48
BTB-GAg	Inf		512	2.33	2.471	9.29	0.04	0.46
BTB-GAg	64	1	4096	2.26	2.469	6.02	16.43	0.47
BTB-GAg	64	4	4096	2.04	2.465	5.87	13.44	0.43
BTB-PAs	512	1	512	1.89	2.461	7.17	2.20	0.38
BTB-PAs	512	1	4096	1.74	2.458	6.58	2.20	0.35
BTB-GAg	512	1	4096	1.61	2.455	5.74	4.20	0.33
BTB-PAs	512	4	512	1.49	2.452	5.77	0.77	0.30
BTB-GAg	512	4	4096	1.49	2.452	5.71	1.35	0.30
BTB-GAg	Inf		4096	1.44	2.451	5.68	0.04	0.28
BTB-PAs	512	4	4096	1.35	2.448	5.22	0.77	0.27
BTB-PAs	Inf		512	1.35	2.448	5.09	0.03	0.25
BTB-PAs	Inf		4096	1.21	2.445	4.57	0.03	0.23
BTB-PAs	Inf		32768	1.17	2.444	4.40	0.03	0.22
BTB-GAg	Inf		32768	1.08	2.443	4.13	0.05	0.21
Perfect				0	2.419	0	0	0

Average results for all the branch architectures sorted in terms of Percent Increase in Execution Time. Branch penalties gives the percent of mispredicted branches, percent of misfetched branches and the branch execution penalty.

Table 4.8 lists ρ^2 for each program across all the branch architectures considered and shows that CPI can be accurately predicted by the BEP and the %MpB, although the BEP is a more accurate metric. As expected, the BEP also predicts the %MpB with reasonable accuracy. It is essential to understand that we are asserting that the predictive quality of the BEP for the CPI only applies to the statically-scheduled architecture we model.

It is understandable that the BEP would be correlated with the CPI on the Alpha 21064 architecture. The nominal branch misprediction penalty is five cycles. In some circumstances, these penalties can vary. For example, if a load that follows a conditional branch misses in the data cache, the branch will be resolved before the load issues, masking any misfetch penalty that would have occurred if the load had hit in the cache. Likewise, if only one instruction in an issue-pair can actually be issued, the processor has an additional cycle to determine the next fetch address, effectively masking the misfetch penalty. However, the high correlation indicates that these are infrequent occurrences.

Note that the correlation coefficient does not indicate the degree of importance of the BEP. It simply indicates that the BEP and CPI are linearly correlated. The least-squares solution for all architectures over all programs indicates that $\mathrm{CPI} = 2.35 + 0.073 \times \mathrm{BEP}$. Thus, although the BEP is important in the overall CPI, a difference of 1.0 in the BEP results in a 3.1% increase in the CPI, and most of the branch and instruction fetch prediction architectures examined in this dissertation have a BEP between 0.2 and 1.0.

4.8 Implications of System Level Study

A valid question to ask is how the results presented in this chapter apply to future wide-issue architectures that have wider instruction issues, deeper pipelines, larger branch penalties or out-of-order execution? We feel it is important to understand the scope of these results, and the intent of the evaluation we have conducted. All of the results we have described are relatively specific to the 21064 implementation of the Alpha architecture. Although some general conclusions can be drawn from these results, many branch architecture issues are not addressed.

Within the limitations imposed by the particular architecture examined, this study is important because it provides detailed system performance showing the actual gain in going from no branch prediction (27% increase in execution time) all the way down to Perfect prediction, using various branch and instruction fetch prediction architectures. We are not aware of a previous study that has provided this detailed performance comparison for modern branch prediction architectures. This also demonstrated that the BEP is an accurate predictor for the CPI for the class of architectures we considered. Therefore, the BEP metric can be used when comparing different branch prediction architectures if the real CPI or execution time is not available; however we recommend that this be confirmed for architectures radically different than the 21064 architecture.

These observations come with an important proviso – the importance of a branch architecture is highly dependent on the underlying architecture and the penalties introduced by other subsystems, such as the instruction or data cache. For example, data cache overheads constitute $\approx 24\%$ of the total execution time in the programs we measured. Larger first and second-level caches will reduce the cache overhead. Likewise, a dynamically-scheduled architecture may be able to mask a larger fraction of the cache latency, but is very dependent on the ability to predict branches. Similarly, if profile-directed optimization becomes commonplace, the importance of hardware support for branch architectures will diminish. However, simulating the interaction between the operating system and an application or a mixture of applications may indicate that other branch architectures are more desirable.

4.9 Summary

Microprocessor design is an art that balances many issues, including area and power budgets, design complexity, patent issues, robustness across a number of applications and a particular implementation technology. It is important to understand the contribution of individual architectural features in the context of the full processor design.

This chapter provided a system level study of several branch and instruction fetch prediction architectures including recently proposed two-level correlated branch history schemes. We provide the performance

Table 4.8: Validating the Branch Execution Penalty Metric

	Correlation Coeff. (ρ^2)		
	CPI vs.	CPI vs.	BEP vs.
Program	BEP	%MpB	%MpB
doduc	0.98	0.99	0.94
tomcatv	0.99	0.88	0.89
compress	1.00	0.92	0.92
eqntott	1.00	0.91	0.90
espresso	1.00	0.96	0.94
gcc	1.00	0.94	0.93
li	0.99	0.95	0.92
bc	0.99	0.96	0.93
flex	1.00	0.92	0.92
gzip	1.00	0.93	0.93
indent	1.00	0.95	0.95
od	0.99	0.97	0.94
wdiff	1.00	0.93	0.93
cfront	0.97	0.99	0.94
idl	1.00	0.95	0.95
lic	1.00	0.93	0.91
porky	0.99	0.94	0.90

The square of the correlation-coefficient for each program across all branch architectures. A $\rho^2 = 1$ indicates that changes in the BEP due to a particular branch architecture are predictive of the CPI for that program using that architecture.

comparison in terms of execution time using a full Alpha AXP 21064 architectural simulation model. The results show that, for the 21064 processor examined, a significant performance gain is achieved by adding a small 512 entry PHT to the architecture, and little benefit is gained by increasing the size of the PHT. An additional reduction in execution time, half of that provided by adding the PHT, is seen by adding a small 64-entry direct mapped BTB. A processor using a 64 entry direct mapped BTB with a decoupled 512 entry PHT achieves an execution time within 3% of the execution time for prefect branch prediction. The results show that increasing the BTB size and associativity past the 64-entry direct mapped BTB provides minimal improvements.

These results indicate that continuing to improve the branch behavior for processors and programs similar to the ones we studied is simply "polishing a round ball". This does not mean that reducing the BEP by improving branch and instruction fetch prediction is an unimportant problem; rather, it means that for further reductions in the BEP to become meaningful, other pipeline stalls such as cache and TLB stalls first need to be greatly reduced on this type of architecture. Alternatively, researchers proposing new branch prediction architectures should demonstrate how they will work on a processor with dynamic out-of-order execution, deeper pipelines and wider issue widths. They also should emphasize designs that are faster, less complex, easier to test or easier to implement. This point of striving for a simpler and faster branch architecture design is the focus of the hardware models examined later in this dissertation.

CHAPTER 5

BRANCH ALIGNMENT

This chapter examines a software approach called **Branch Alignment** to improve the performance of branch and instruction fetch prediction architectures by eliminating misfetch penalties and reducing the number of unconditional branches executed. This chapter in effect, describes how to layout basic blocks when compiling a program so that one can get the most from a branch architecture. [1]

5.1 Introduction

Several researchers have proposed algorithms for basic block reordering. The primary emphasis for these algorithms has been on improving instruction cache locality, and the few studies concerned with branch prediction reported small or minimal improvements. As wide-issue architectures become increasingly popular, the importance of reducing branch costs will increase, and branch alignment is one mechanism which can effectively reduce branch costs.

This chapter examines algorithms that reorder the structure of a program to improve the accuracy of the instruction fetch and branch prediction architectures. These code transformations help reduce the number of mispredicted branches and the number of misfetched instructions. Essentially, the method is this: restructure the control flow graph so that **fall-through** branches occur more frequently. Profile information is used to direct the transformation, and an architectural cost model is used to decide if the transformation is warranted. Transformations include rearranging the placement of basic blocks, changing the sense of conditional operations, moving unconditional branches out of the frequently executed path, and occasionally inserting unconditional branches.

5.2 Related Branch Alignment Work

There has been considerable work on profile-driven program optimization. Almost all the basic block reordering optimizations have concentrated on reducing the instruction cache miss rate, as described in Chapter 2, since this has been one of the most important factors in achieving high throughput in past processors. Branch and instruction fetch prediction is a prime candidate for profile-based optimizations, since it is already becoming increasingly important as future static and dynamic architectures execute four to eight instructions at a time. This means that almost every instruction fetch will contain a branch instruction. Thus, insuring the target of the conditional branch is the fall-through path or eliminating unconditional branches from the control flow will improve a processors performance without adding any additional hardware.

McFarling and Hennessy [42] described a number of methods to reduce branch misprediction and instruction fetch penalties. Their goal was to make the "taken" path the most common path. Later, Bray and Flynn [6] studied the same idea, except they examined making the "fall-through" the most common case, but they only studied the performance gain of their algorithm in terms of conditional branch prediction accuracy and for only the branch target buffer architecture.

Yeh **et al.** [66] commented that with trace scheduling, taken branches could only be reduced from $\approx 62\%$ of the executed conditional branches to $\approx 50\%$ of the executed conditional branches. An earlier study by Hwu and Chang [43] showed a $\approx 58\%$ fall-through rate after branch alignment. The papers by McFarling and Hennessy, and Bray and Flynn did not report their change in the percentage of taken branches, and they did not describe the basic block order in which their algorithms where applied when examining all the basic blocks in a procedure.

---

[1] Parts of this chapter were published in the 6th International Conference on Architectural Support for Programming Languages and Operating Systems [8].

The branch alignment reordering algorithm proposed by Hwu and Chang [43] is a general algorithm which grows the basic block layout list, first forwards then backwards, starting with the basic block edge that has the highest execution count for a given procedure. The work by Pettis and Hansen [49] describes a greedy algorithm for branch alignment which is similar to Hwu and Chang's, but their greedy algorithm is more general than the Hwu and Chang algorithm, and performs better in terms of reducing the cost of branches. Therefore, we only describe the Pettis and Hansen algorithm in detail in this chapter.

This chapter describes an algorithm which is an extension of the Pettis and Hansen algorithm, and compares the simulated results to their greedy algorithm. We also improve upon the analysis and the effectiveness of branch alignment over that of McFarling and Hennessy, Bray and Flynn, Hwu and Chang, and Pettis and Hansen. We describe how to efficiently apply branch alignment to various static and dynamic prediction architectures and we measure the effectiveness of branch alignment on these architectures. When performing branch alignment, we do not inline functions, perform global analysis or duplicate code as in previous studies. We perform the analysis using an object code post-processor rather than a compiler. This simplifies the analysis and avoids recompiling the full program for these simple transformations. This also allows us to apply branch alignment to the full program, including portions normally not compiled by the user, such as program libraries, and to process many programs generated by a number of different compilers. With such a post-processor tool, branch alignment would normally be only one of several optimizations applied to the program.

5.3 Branch Prediction Architectures

Not all branch architectures include a mechanism for providing the taken target address to indicate which instruction to fetch from the instruction cache. For these studies the fall-through instruction is assumed to be fetched while a branch is decoded. Therefore, all PC-relative taken branches can incur a misfetch penalty while the branch is decoded and the target address is calculated, and all branch architectures that do not include a BTB **always** incur this misfetch penalty. In evaluating branch alignment, three static prediction architectures and three dynamic prediction architectures were studied. The architectures we examined in the previous Chapter 4 that are also examined in this chapter are the BTFNT, PHT-Direct, PHT-GAg, and the BTB-2Bit branch and instruction fetch prediction architectures. In addition to these four architectures we also examine two additional static branch prediction architectures.

5.3.1 Static Branch Prediction Architectures

The "Fall-Through" model assumes the fall-through execution path is always executed. The "BTFNT" (backward-taken, forward not taken) assumes backward branches are always taken while forward branches are not taken. This branch model is fairly common, and variants of it are implemented on the HP PA-RISC and the Alpha AXP-21064. The "Likely" model assumes that encoded information in the branch instruction indicates whether the branch is likely to be taken or not-taken. This branch model is used by several architectures including the Tera [2]. The "likely/unlikely" flag can be set either using compile-time estimates [4] or profile information [25], as described in Chapter 2. We use profile information since it is accurate and simple to gather with the appropriate tools [55].

Program transformation can help these branch prediction architectures reduce misfetch and misprediction delays. In the Fall-Through architecture, the fall-through path should be executed most frequently, both to reduce misfetch penalties and to improve prediction. In the BTFNT architecture, it is useful to have the fall-through be the most common path, but if that is not possible or cost-effective, the branch target should be placed before the conditional branch so a backwards branch is predicted. Since the branch mispredict penalty is larger than the misfetch penalty, it may be better to correctly predict the backwards branch even if this results in a misfetch. In the Likely model, the compiler can specify the likely branch outcome, therefore the code transformations can only eliminate misfetch penalties by making the fall-through the most frequently executed path. We should expect that there are more opportunities for optimization with the Fall-Through architecture than the BTFNT model because all taken branches will be mispredicted in the Fall-Through method. Likewise, we would expect more optimization opportunities for the BTFNT model than the Likely model since we can

only improve the misfetch rate when transforming programs using the Likely model.

Figure 5.1 shows how code transformations can help each static prediction model. Figure 5.1 shows a portion of the control flow graph from the routine `elim_lowering` in the ESPRESSO benchmark. Nodes are labeled with numbers and the number in parenthesis indicates the number of instructions in that basic block. Edges are labeled by frequency of execution. The edge labeled "16" is executed for 16% of all edge transitions in that subroutine. Unlabeled edges are executed less than 1% of the time. Fall-through edges are darkened while "taken" edges are dotted.

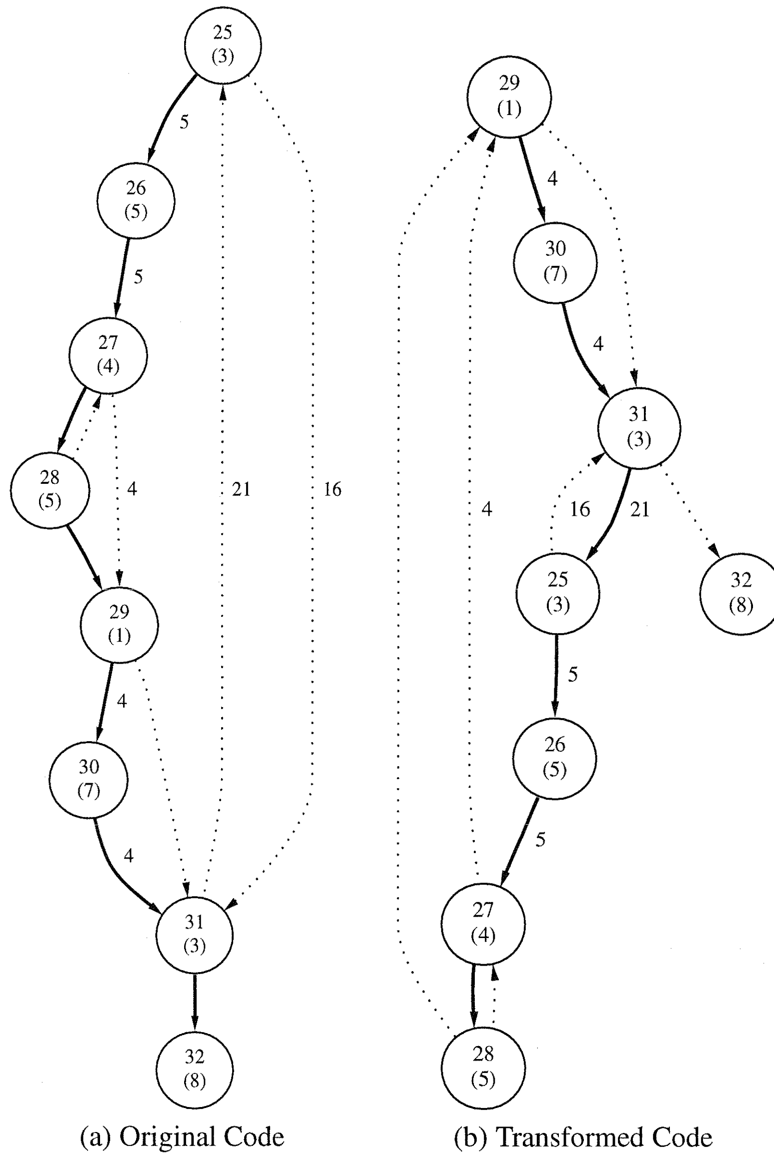
In the original code in Figure 5.1(a), the Likely architecture can correctly predict the most likely targets having misfetch penalties on edges 27 $\rightarrow$ 29, 31 $\rightarrow$ 25, and 25 $\rightarrow$ 31. In comparison, the Fall-Through architecture will mispredict edges 27 $\rightarrow$ 29, 31 $\rightarrow$ 25, and 25 $\rightarrow$ 31 since these are all taken branches. The BTFNT architecture will also mispredict edges 27 $\rightarrow$ 29 and 25 $\rightarrow$ 31, but will correctly predict edge 31 $\rightarrow$ 25 since the target is before the branch instruction resulting in a backwards branch.

The transformed code in Figure 5.1(b) is an efficient layout in terms of branch costs for each of the static prediction architectures. Since node 25 is now the fall-through of node 31, all of the architectures can correctly predict edge 31 $\rightarrow$ 25 without any penalty, and since 31 is laid out before 25 the BTFNT can accurately predict edge 25 $\rightarrow$ 31 with only a misfetch penalty. Also, since node 29 is laid out before 27 the branch 27 $\rightarrow$ 29 can be accurately predicted by the BTFNT architecture with only a misfetch penalty. The transformed program gives an optimal layout for the BTFNT since it will have the same prediction as the Likely architecture. This is also a good layout for the Fall-Through architecture. It still suffers by mispredicting edges 27 $\rightarrow$ 29 and 25 $\rightarrow$ 31, but it can correctly predict the frequently executed edge 31 $\rightarrow$ 25. Notice that in the transformed code there are two taken edges coming out of node 28. Since one of the edges has to be the fall-through, we need to add an unconditional branch to the fall-through, which will in turn jump to the correct destination node. The transformed code in Figure 5.1(b) gives an efficient transformation for each of the static architectures, but in general, a single branch alignment transformation will not always give an optimal alignment for the different architectures.

There are many optimizations such as un-rolling loops that we did not investigate. For example when we traced the `alvinn` program, which is a neural net simulator, we found that 46% of the time was spent in routine `input_hidden` and another 46% was spent in `hidden_input`. Figure 5.2 shows the control flow graph for `input_hidden`. Nearly 100% of the branches in that subroutine, or $\approx$ 46% of all branches in `alvinn`, arise from a single branch from basic block 4. If we unrolled that loop, duplicating this 11-instruction basic block, we could reduce the misfetch penalty for all architectures and improve the branch prediction for the Fall-Through architecture. Normally, loop unrolling is a more complex transformation that also attempts to reduce the total number of executed branches within the unrolled code. We feel that simply duplicating the basic block 4 and then inverting (aligning) the branch condition for the added conditional branches in this example would offer some performance improvement, even if the other optimizations offered by loop unrolling were ignored.

5.3.2 Dynamic Branch Prediction Methods

While static prediction mechanisms, particularly profile-based methods, accurately predict 70-90% of the conditional branches, many current computer architectures use **dynamic prediction**, such as branch target buffers (BTB) and pattern history tables (PHT) to accurately predict 90-95% of the branches. For this study, we simulated two PHTs, a direct mapped PHT and the degenerate two-level correlation PHT (GAg) using a variant that McFarling [41] found to be highly accurate. This method performs an exclusive-or of the branch address with the global history register and uses this as an index into the PHT. Both of the PHTs we simulated contained 4096 2-bit saturating up-down counters, for a total of 1KBytes of storage. We also simulated two BTB configurations. We modeled a 64-entry 2-way associative BTB and a 256-entry 4-way BTB – the latter configuration is used in the Intel Pentium. The BTBs we simulated store only taken branches in the BTB, and predict fall-through on a BTB miss. Each BTB entry contains a 2-bit saturating up-down counter used to predict the destination for conditional branches. The BTB in our simulations hold entries for conditional branches,



(a) Original Code (b) Transformed Code

Figure 5.1. Benefits of code transformation for `elim_lowering` in `espresso`. The darkened edges are fall-through and the dotted edges are taken. Nodes represent basic blocks.

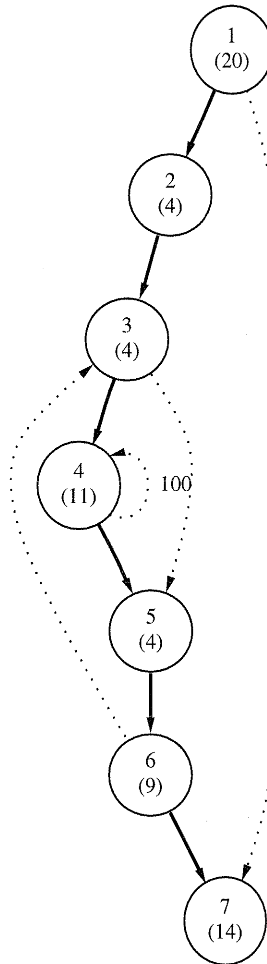


Figure 5.2: Routine `input_hidden` from `alvinn`

unconditional branches, indirect jumps, procedure calls and returns.

5.4 Branch Alignment Algorithms

We implemented the branch alignment algorithm suggested by Pettis and Hansen [49] since their algorithm is more general and performed better for branch alignment than previous algorithms. In our results, branch alignment is performed for each procedure in a program. We are mainly concerned with reducing branch cost, although instruction cache performance may also be improved. We

Procedures were represented by a directed control flow graph containing a set of basic blocks represented by nodes and edges between nodes. The program's execution was traced, recording the number of times each edge is traversed. This is called the edge's **execution weight**.

When transforming a program we look at all nodes that have an out degree of one or two. An unconditional branch is a basic block with a single out-going edge, the 'taken' edge. A conditional basic block has two edges, the 'taken' and the 'fall-through' edges, and a fall through basic block has an out-going 'fall-through' edge. All other edges are given a weight of zero and are not considered when applying branch alignment. Thus, we ignore indirect branches, procedure returns and subroutine calls.

5.4.1 Greedy

Pettis and Hansen [49] proposed two heuristics to align branches. We only describe their bottom-up ('greedy') algorithm, since it has better performance. The Greedy algorithm was directed towards the BTFNT architecture, and did not consider the implications of different branch architectures. In the terminology of [49], a **chain** is a contiguous sequence of basic blocks threaded by 'head' and 'tail' pointers. The first basic block in each chain has a null head pointer and the last basic block in each chain has a null tail pointer.

The algorithm aligns each procedure in turn. The edge $S \rightarrow D$, where S is the source and D the destination, with the largest weight, that has not already been added to the layout, is selected. The algorithm then attempts to position node D as the "fall-through" of node S. If S does not already have a fall-through basic block, and D does not already have a head, then these two basic blocks are combined into a chain. Otherwise, these blocks cannot be linked. If these basic blocks are part of existing chains, the two chains are merged when the basic blocks are linked. This is repeated until all edges have been examined and chains can no longer be merged. Pettis and Hansen implemented their technique for the HP PA-RISC architecture. This architecture uses the BTFNT conditional branch prediction model. After all the edges in a procedure have been examined, a precedence relation is defined between chains to determine an ordering between chains that would achieve the best prediction using the BTFNT model. The chains are then positioned using this precedence relation, inserting unconditional branches when needed.

5.4.2 Adding a Branch Cost Model

The Greedy algorithm does not consider the underlining architecture when constructing chains. We include these underlining architecture costs in our algorithms in order to reduce the cost of branches beyond that of the Greedy algorithm. Our architecture assumes specific costs for different branches, shown in Table 5.1. A "Cost" transformation algorithm would try to minimize the cost of the branches for a procedure using simple heuristics, hoping that each local minimization will result in a global performance improvement.

Table 5.1: Cost, in Cycles, for Different Branches.

Unconditional branch	2	(instruction + misfetch)
Correctly predicted fall-through	1	(instruction)
Correctly predicted taken	2	(instruction + misfetch)
Mispredicted	5	(instruction + mispredict)

As in the Greedy algorithm, the Cost algorithm starts with the edge with the highest weight. When

we pick an edge $S \rightarrow D$, we determine if having D be the fall-through for S will locally benefit the program using our cost model before trying to link S and D. We examine all the predecessors of D to see if it is more cost effective to connect D to another node. The algorithm only considers basic blocks with one and two exit edges. We consider two possible alignments for single-exit nodes; examining the cost of aligning the edge as a fall through (thereby avoiding an unconditional branch) or adding an unconditional branch. For example, if S were a single-exit node, we could either include S and D in the same chain, or insert a jump to D at the end of S, allowing S and D to be in different chains. For conditional branches we examine three possible alignments. Assume S has another edge, $S \rightarrow D_2$. We consider including the $S \rightarrow D$ or the $S \rightarrow D_2$ edge in the current chain or adding a jump to the end of S, making the jump the fall-through of S. This latter transformation may be useful if it is more cost effective to have both D and D_2 as fall-through in other chains.

In certain cases, not aligning either edge of a conditional branch can improve performance on the Fall-Through and BTFNT architectures. For example, consider a loop consisting of a single basic block, such as that shown in Figure 5.2. Using the Fall-Through model, the original loop in node 4 incurs a five cycle penalty (one cycle for the branch instruction and four cycles for the misprediction penalty) using our cost-model. It is cost-effective to invert the sense of the conditional ending the block and follow the block with an inserted unconditional branch instruction. This combination takes only three cycles (the correctly predicted conditional branch, the unconditional branch and a single misfetch penalty). Any loop can be structured this way - we illustrated the point using the single block loop because the Greedy algorithm would not restructure such loops.

5.4.3 Try15

Simulation results showed that the 'Cost' heuristic gave sizable improvements for the Fall-Through architecture, modest improvements for BTFNT and negligible improvements for Likely and dynamic branch architectures. We decided to considered using the cost model to assess the cost of every possible basic block alignment using an exhaustive search and selecting the minimal cost ordering. In practice, this sounds expensive, but in the common case procedures contain between 5 to 15 basic blocks. However, most programs have some procedures containing hundreds of blocks, which makes an exhaustive search impossible for those procedures. For example, the `gcc` program contains a procedure (`yyparse`) containing 712 basic blocks. However, many edges were never executed in those large procedures, and a few basic blocks contribute most of the execution time.

In order to examine an upper bound on how well a 'Cost' heuristic algorithm could perform over a Greedy algorithm we devised a heuristic that balanced time against performance called 'Try15'. For each procedure, we select the 15 most frequently executed edges and attempt all possible alignments for these nodes. We then select the next 15 edges, and so on. This allows us to try all possible combinations for each group of 15 nodes. The possibilities to try for each node is similar to that described for the Cost algorithm. For single-exit nodes (unconditional and fall-through basic blocks) the two possibilities are to make the outgoing edge either a fall-through or taken edge, and for nodes with two-exit edges (conditional branches) we try aligning separately each of the two outgoing edges as the fall-through and then try having neither of the out-going edges as the fall-through. This heuristic takes more time than the Greedy heuristic algorithm, but produced better results and still ran in a few minutes. Considering 10 nodes at a time gave slightly worse results than Try15 for a few programs, taking less than a minute to run and still resulting in better performance than the Greedy algorithm.

To improve the performance of the Try15 algorithm we only examined edges that were executed more than once. This eliminated over half of the edges from consideration in each program. If more profiles were used or combined for a program, one could reduce the execution time of the Try15 algorithm by examining only those conditional branches that account for 99% of the executed branches.

For branch alignment algorithms, aligning loops is difficult, and this is one case where the Try15 heuristics perform better than the Greedy algorithm. Figure 5.3(a) shows a fragment of code with a loop. The Greedy algorithm would not modify this code because it chooses to align edge $A \rightarrow B$ before it chooses edge $B \rightarrow A$ since the cost of $A \rightarrow B$ (201) is greater than the cost of edge $B \rightarrow A$ (160). The Greedy algorithm would then align the edge $B \rightarrow C$ since this edge has the highest cost (41) out of the remaining

alignable edges. Whereas, the Try15 algorithm transforms the code as shown in Figure 5.3(b). Note that in the transformed code the unconditional branch from $D \rightarrow C$ is removed. Using our cost model in Table 5.1 for the Likely and BTFNT architecture, the execution cost for the original code with the edge-weights shown is $201 + 40 * 5 + 160 * 2 + 41 * 5 + 40 * 2 = 1,006$ cycles, while the cost for the Try15 algorithm is $160 + 41 * 5 + 201 * 2 + 40 * 5 = 967$ cycles. The main differences between these two alignments is that the unconditional branch from $D \rightarrow C$ is eliminated from the program, and the loop between A and B is laid in the opposite order. This reduces the branch execution cost by 4%. A larger or smaller percent reduction can be achieved, depending upon the edge weights.

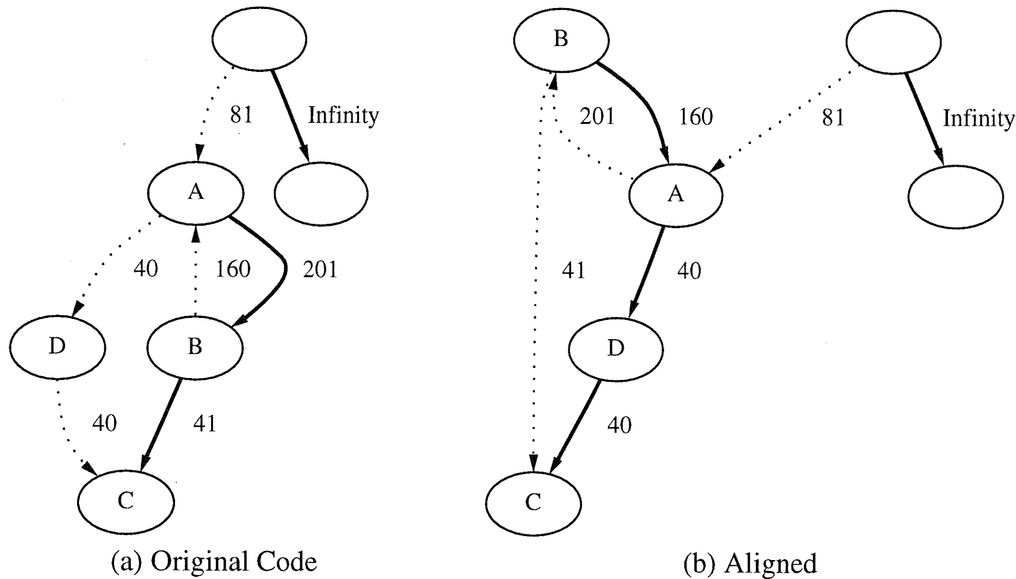


Figure 5.3. Example illustrating where Try15 reduces branch costs. The darkened edges are fall-through and the dotted edges are taken. Nodes represent basic blocks.

Ideally, we want the most likely path through the loop to be in a single chain. The Greedy algorithm does not examine enough of the loop to minimize this cost. This is one of the main reasons why the Try15 heuristic is able to produce better results. The Try15 heuristic can try all the combinations to find the correct place to "break" a loop.

5.5 Methodology

ATOM was used to create several tools to study the behavior of branch alignment on current branch and instruction fetch prediction architectures. The first tool gathered profiles and recorded the control flow of the program. The profiles and the program's control flow were then read into a branch alignment algorithm, which when ran outputted the new control flow graph. This was then read into the branch prediction simulators in order to gather the simulation results. The different branch alignment algorithms were also implemented using the OM [56, 57] system for the Alpha 21064 architecture. The control flow graphs used in the ATOM simulations were the same as those read into OM in order to perform branch alignment on the programs.

5.6 Greedy and Try15 Branch Execution Penalty Results

For all of our results, the branch alignment algorithms only rearrange basic blocks within a procedure. The algorithms do not perform procedure splitting, basic block duplication, or any procedure rearranging. In order to evaluate the performance of the different alignments and architectures, we use a modified version of the branch execution penalty (BEP) described in Chapter 3. When performing branch alignment, the number of branches executed by a program can change because unconditional branches may be added or removed from the execution stream. Since the BEP is based on miss rates, for branch alignment the BEP is calculated by dividing

the branch penalties by the number of branches executed in the **original** program. This gives us a normalized branch execution penalty. This normalized BEP captures the branch performance even for unconditional branches that are added or removed from the programs execution stream. Removing or adding unconditional branches through branch alignment can eliminate or increase misfetch penalties, and this performance is taken into account by reducing or increasing the BEP. What the BEP does not take into account is the increase or decrease in the number of instructions executed by adding or removing unconditional branches. The overall effect of adding or removing unconditional branches is examined later in this section.

For these simulations we assumed a misfetched branch causes a one cycle misfetch penalty and a mispredicted branch causes a four cycle mispredict penalty. For the static branch and PHT architectures, all PC-relative branches cause misfetch penalties, and all indirect jumps are mispredicted. Since the BTB architecture tries to predict all branch types, the misfetch penalty for any PC-relative branch found in the BTB can be eliminated, and the mispredict penalties for indirect jumps can also be eliminated. In all of our static and dynamic architecture simulations we simulated a 32-entry return stack [33] for predicting return instructions.

Table 5.2 shows the relative BEP for each program using the various alignments on the three static branch architectures. Table 5.3 shows the percent of executed conditional branches which are fall-through branches after the alignment has been performed for the various architectures. The percent of fall-through branches does not change for the Greedy algorithm on the varying branch architectures, whereas the fall-through percentage for the Try15 algorithm changes for each architecture since the cost model algorithm is different for each architecture. The fall-through percentages for the PHT and BTB architectures are similar to the Likely architecture for the Try15 algorithm. Table 5.4 shows the relative BEP for the PHT architectures and Table 5.5 shows the relative BEP for the BTB architectures. Arithmetic averages are shown for each group of programs (SPECfp92, SPECint92, and the 'Other' programs). The 'Orig' column for each architecture shows the performance when we instrumented and traced the original program. For each branch architecture, we use the same input to 'align' the program and to measure the improvement from that alignment.

The branch alignment heuristics that use the Try15 architectural cost model usually perform better than the simpler Greedy algorithm – this is particularly notable in the Fall-Through architecture. The Fall-Through architecture is no longer a realistic architecture to consider, but is used in combination with BTBs – the fall-through can be predicted on a BTB miss. This works especially well for branch target buffers that only store taken branches. The improved performance occurs because the Try15 heuristic does not align either of the out-going edges for some conditional branches. Instead, unconditional branches are added to one of the conditional branch edges to take advantage of the Fall-Through prediction cost model. In fact, the Try15 heuristic converts up to 99%, as seen in Table 5.3, of all conditional branches in some programs to be fall-through in the Fall-Through model. Adding an unconditional jump works especially well for single basic block loops which end with a conditional branch, as described earlier for `alvinn` and many of the FORTRAN programs.

The BTFNT architecture sees reasonable improvement from branch alignment. In the BTFNT architecture, it is difficult to create and layout chains. When forming chains, it is not known where the taken branch will be located in the final procedure until the chains are formed and laid out. The destination of a taken branch could be placed before or after the current node, affecting the final branch prediction costs.

The small benefit for the Likely architecture occurs because the Greedy and Try15 algorithms eliminate the misfetch penalty for many branches and they remove unconditional branches from the likely execution path. Eliminating instruction misfetches will be increasingly important as super-scalar architectures become more common – an eight-issue super-scalar architecture could fetch a branch instruction each cycle. These architectures will benefit from having frequent "fall-through" branches.

The cost model used for the static architectures is different than that for the dynamic architectures. When examining the costs of aligning a conditional branch for the static architecture only one of the targets of the conditional branch can be predicted and the other must **always** be mispredicted. In the dynamic architectures this is not the case. In order to compensate for the increased accuracy for predicted conditional branches, our cost model for the PHT architectures assume that conditional branches are mispredicted only 10% of the time. Similarly in the BTB architectures we also assume that conditional branches are mispredicted only 10% of

Table 5.2: Branch Execution Penalty for Static Branch Prediction Architectures with Branch Alignment.

	Fall-Through			BTFNT			Likely		
Program	Orig	Greedy	Try15	Orig	Greedy	Try15	Orig	Greedy	Try15
alvinn	3.86	3.79	0.97	1.03	0.97	0.98	0.99	0.97	0.97
doduc	1.70	1.09	0.45	1.10	0.51	0.51	0.64	0.45	0.45
ear	2.43	2.05	0.82	0.95	0.82	0.82	0.94	0.82	0.82
fpppp	1.76	0.62	0.56	1.68	0.64	0.60	0.89	0.55	0.55
hydro2d	2.84	1.52	0.60	1.56	1.46	0.66	0.88	0.61	0.63
mdljsp2	3.23	0.63	0.54	2.83	0.80	0.54	1.19	0.55	0.54
nasa7	2.72	2.48	0.76	1.01	0.79	0.78	0.88	0.77	0.77
ora	1.69	0.31	0.31	1.55	0.32	0.71	0.72	0.31	0.32
spice	2.69	2.31	1.79	1.22	1.11	1.06	0.98	0.88	0.89
su2cor	2.42	1.65	0.79	1.15	0.91	0.93	1.00	0.81	0.84
swm256	3.93	3.92	1.00	1.01	1.01	1.01	1.01	1.01	1.01
tomcatv	3.97	2.25	0.58	2.30	0.59	0.59	1.02	0.59	0.59
wave5	2.07	1.64	0.65	1.11	0.69	0.67	0.82	0.66	0.67
SPECfp92 Avg	2.72	1.87	0.76	1.43	0.82	0.76	0.92	0.69	0.69
compress	2.51	0.86	0.77	1.89	1.10	0.77	1.12	0.76	0.77
eqntott	3.47	1.75	0.57	2.28	0.61	0.60	1.02	0.58	0.58
espresso	2.36	1.34	0.89	1.64	1.47	1.40	1.14	0.89	0.90
gcc	2.11	0.96	0.71	1.87	0.90	0.88	1.05	0.72	0.72
li	1.51	0.70	0.65	1.48	0.81	0.77	0.86	0.62	0.62
sc	2.42	1.27	0.64	1.73	0.83	0.77	0.96	0.65	0.66
SPECint92 Avg	2.40	1.15	0.71	1.81	0.95	0.86	1.02	0.70	0.71
cfront	1.85	0.92	0.70	1.70	0.79	0.79	0.97	0.70	0.70
db++	1.93	1.24	0.98	1.69	1.05	1.03	1.19	0.99	0.99
idl	1.59	0.79	0.70	1.52	0.72	0.73	0.96	0.71	0.71
tex	2.01	1.02	0.73	1.74	0.91	0.93	1.05	0.75	0.75
groff	1.87	0.67	0.52	1.60	1.03	0.62	0.83	0.52	0.52
Other Avg	1.85	0.93	0.73	1.65	0.90	0.82	1.00	0.73	0.73

Table 5.3: Percent of Fall-Through Branches with Branch Alignment.

Program	% of Fall-Through Conditional Branches				
	Orig	Greedy	Try15		
			Fall-Through	BTFNT	Likely
alvinn	2.23	3.76	99.57	3.71	3.75
doduc	51.32	68.90	95.08	68.77	92.24
ear	9.87	25.87	92.85	25.80	25.80
fpppp	52.26	84.68	87.38	83.39	83.62
hydro2d	26.66	57.68	95.44	53.43	53.45
mdljsp2	16.38	87.08	90.20	77.79	77.79
nasa7	20.70	26.35	96.84	26.27	26.32
ora	46.76	94.67	94.96	90.36	90.50
spice	28.37	38.37	92.31	37.42	37.74
su2cor	26.93	52.27	89.82	38.12	38.12
swm256	1.58	1.78	99.42	1.76	1.76
tomcatv	0.72	43.71	99.38	43.71	43.71
wave5	38.21	51.27	94.09	50.96	51.04
SPECfp92 Avg	24.77	48.95	94.41	46.27	48.14
compress	31.75	81.73	84.14	68.72	68.72
eqntott	9.70	55.20	97.56	54.89	54.90
espresso	38.10	62.66	84.30	60.97	65.83
gcc	40.57	76.63	87.37	74.79	75.35
li	52.70	83.03	85.63	83.03	83.11
sc	33.12	66.37	90.91	65.66	65.72
SPECint92 Avg	34.32	70.94	88.32	68.01	68.94
cfront	46.82	81.05	89.64	80.52	81.20
db++	43.14	73.96	90.23	73.47	74.35
groff	45.86	84.20	94.06	82.16	84.53
idl	53.30	90.37	96.11	89.96	90.00
tex	42.53	73.23	87.43	70.67	71.43
Other Avg	46.33	80.56	91.49	79.36	80.30

Table 5.4: Branch Execution Penalty for Pattern History Table Architectures with Branch Alignment.

	4096 Direct Mapped PHT			4096 Correlation PHT		
Program	Orig	Greedy	Try15	Orig	Greedy	Try15
alvinn	0.99	0.97	0.97	0.98	0.96	0.96
doduc	0.66	0.46	0.47	0.66	0.43	0.46
ear	0.95	0.86	0.87	0.91	0.83	0.82
fpppp	0.73	0.42	0.42	0.70	0.38	0.37
hydro2d	0.85	0.58	0.60	0.83	0.57	0.59
mdljsp2	1.08	0.49	0.49	1.08	0.48	0.50
nasa7	0.87	0.77	0.76	0.87	0.76	0.75
ora	0.73	0.33	0.33	0.68	0.28	0.31
spice	0.98	0.88	0.88	0.87	0.81	0.77
su2cor	1.04	0.84	0.89	1.06	0.85	0.91
swm256	1.01	1.00	1.00	1.01	1.02	1.02
tomcatv	1.01	0.58	0.58	1.01	0.58	0.58
wave5	0.82	0.69	0.67	0.75	0.62	0.59
Fortran Avg	0.90	0.68	0.69	0.88	0.66	0.67
compress	1.11	0.76	0.73	1.05	0.70	0.69
eqntott	1.02	0.57	0.57	0.99	0.55	0.54
espresso	1.00	0.79	0.79	0.81	0.62	0.61
gcc	1.08	0.75	0.75	1.07	0.68	0.68
li	0.87	0.63	0.62	0.70	0.49	0.46
sc	0.82	0.53	0.54	0.79	0.50	0.51
C Avg	0.98	0.67	0.67	0.90	0.59	0.58
cfront	1.02	0.72	0.72	1.06	0.69	0.69
db++	1.13	0.95	0.95	1.02	0.84	0.84
idl	0.95	0.70	0.70	0.93	0.67	0.67
tex	1.04	0.74	0.74	0.95	0.63	0.65
groff	0.82	0.50	0.51	0.77	0.47	0.47
Other Avg	0.99	0.72	0.72	0.95	0.66	0.66

Table 5.5: Branch Execution Penalty for Branch Target Buffers with Branch Alignment.

	64-Entry, 2-way BTB			256-Entry, 4-way BTB		
Program	Orig	Greedy	Try15	Orig	Greedy	Try15
alvinn	0.07	0.02	0.02	0.03	0.02	0.02
doduc	0.39	0.22	0.24	0.24	0.19	0.19
ear	0.22	0.22	0.21	0.20	0.20	0.20
fpppp	0.33	0.26	0.28	0.24	0.24	0.24
hydro2d	0.13	0.19	0.13	0.13	0.19	0.13
mdljsp2	0.27	0.27	0.27	0.27	0.27	0.27
nasa7	0.13	0.10	0.10	0.10	0.09	0.09
ora	0.19	0.17	0.16	0.16	0.16	0.16
spice	0.31	0.29	0.42	0.28	0.29	0.41
su2cor	0.35	0.35	0.35	0.35	0.35	0.35
swm256	0.02	0.02	0.02	0.02	0.02	0.02
tomcatv	0.03	0.02	0.02	0.02	0.02	0.02
wave5	0.34	0.25	0.25	0.22	0.22	0.22
Fortran Avg	0.21	0.18	0.19	0.17	0.17	0.18
compress	0.54	0.47	0.47	0.47	0.47	0.47
eqntott	0.14	0.09	0.08	0.08	0.08	0.08
espresso	0.65	0.53	0.52	0.43	0.47	0.46
gcc	1.14	0.65	0.66	0.77	0.53	0.53
li	0.76	0.43	0.45	0.38	0.35	0.35
sc	0.39	0.20	0.20	0.18	0.17	0.17
C Avg	0.60	0.40	0.40	0.39	0.34	0.34
cfront	1.39	0.70	0.71	0.94	0.55	0.56
db++	0.46	0.24	0.25	0.22	0.17	0.17
idl	0.56	0.15	0.17	0.15	0.11	0.11
tex	0.83	0.52	0.52	0.52	0.45	0.45
groff	0.79	0.40	0.35	0.35	0.29	0.25
Other Avg	0.80	0.40	0.40	0.44	0.32	0.31

the time and in addition, we assume that the BTB architectures have a 10% miss rate. This means that taken PC-relative branches will only cause a misfetch penalty 10% of the time.

As seen in Table 5.4, branch alignment offers reasonable improvement for the PHT architectures. Table 5.5 also shows reasonable improvement for a 64-entry BTB, but only a small improvement is seen for the 256-entry BTB. As with the Likely architecture, the major improvement in performance for the PHT architecture comes from moving unconditional branches from the frequently executed path and reducing the misfetch penalty that occurs for taken conditional branches. The BEP performance for the BTB architecture is already small in comparison to the PHT architecture since it can eliminate misfetch penalties. The small BTB architecture can benefit more from branch alignment than the larger BTB because removing unconditional branches and making more conditional branches execute the fall-through will cause the aligned programs to use less entries in the BTB.

One important observation from these results is that branch alignment reduces the difference in performance between the various branch architectures. The performance of the BTFNT architecture is slightly better than the Likely and PHT architectures, while the BTB architecture has the best overall performance. Before branch alignment was applied to the original program the BTFNT architecture had a 74% higher BEP than the correlated PHT architecture. After Greedy alignment, the BTFNT architecture only has a 34% higher BEP than the PHT architecture. Also, the Greedy aligned BTFNT architecture performs better than the original programs with PHT branch prediction. What is even more interesting is that the 64 entry BTB **with** branch alignment has similar performance to the 256 entry BTB **without** branch alignment.

Note that there is a significant difference between the different program classes. The SPECint92 and 'Other' programs see more benefit from branch alignment than the SPECfp92 programs. A reason for this, as seen in Table 3.5 in Chapter 3, is that for the SPECfp92 programs $\approx$ 6.5% of the instructions executed cause a break in control flow. Whereas in the the SPECint92 and the 'Other' programs $\approx$ 16% of the instructions cause a break in control.

5.7 Breakdown of Greedy Alignment Results

The previous sections results show that the Try15 algorithm has essentially the same performance as the Greedy algorithm for the dynamic branch and instruction fetch prediction architectures. In practice, one would implement the Greedy and not the Try15 algorithm. The Try15 algorithm was examined to give an indication of how well the Greedy algorithm performance could be improved.

Tables 5.6, 5.7, 5.8, and 5.9 show the percent of misfetched branches (MfB), percent of mispredicted branches (MpB), the relative number of instructions executed (RIE), and a combined relative cycles per instruction (CPI) for all the dynamic prediction architectures, for the original program and the Greedy branch alignment algorithm. The RIE is calculated by dividing the number of instructions executed in the Greedy aligned program by the number of instructions executed in the original program. This shows the increase or decrease in the number of instructions executed for each program aligned with the Greedy algorithm. The only difference in the number of instructions executed between the original program and a program aligned with the Greedy algorithm is in the number of unconditional branches executed. The relative CPI shows the impact of adding or removing unconditional branches along with the reduction in branch execution penalties achieved by branch alignment assuming an ideal single issue architecture. The CPI is calculated by adding the branch execution cycle misfetch and mispredict penalties to the number of instructions executed for a given alignment, and dividing this by the number of instructions executed in the original program. This CPI metric is provided as a check to make sure that when unconditional branches are added, the overall program performance is not actually worse than the original program performance.

Table 5.6 for the direct mapped PHT architecture shows that for `idl`, the Greedy alignment executes 1.2% fewer instructions (has an RIE of 0.988) than the original program because of the reduction in unconditional branches executed. In contrast, `espresso` executes 1.4% more instructions than the original program because of the additional unconditional branches executed. Even though the number of instructions executed increases for `espresso`, the overall number of cycles to execute the aligned program, when branch execution penalties

Table 5.6. Branch Alignment Miss Rates and Relative Instructions Executed for Direct Mapped PHT Architecture.

	4096-entry, Direct Mapped PHT						
	Original Alignment			Greedy Alignment			
Program	%MfB	%MpB	CPI	%MfB	%MpB	RIE	CPI
alvinn	97.00	0.48	1.09	95.18	0.45	1.000	1.09
doduc	48.47	4.27	1.06	29.72	4.19	0.996	1.04
ear	74.54	5.16	1.08	65.24	5.16	1.001	1.07
fpppp	50.56	5.66	1.02	19.69	5.60	1.000	1.01
hydro2d	72.01	3.20	1.05	45.34	3.13	1.006	1.04
mdljsp2	80.64	6.84	1.11	21.44	6.84	1.010	1.06
nasa7	76.52	2.63	1.03	66.04	2.62	0.998	1.02
ora	56.78	4.02	1.05	17.09	4.02	0.997	1.02
spice	68.70	7.26	1.12	59.12	7.26	1.000	1.11
su2cor	68.01	8.91	1.05	48.35	8.90	1.000	1.04
swm256	98.27	0.60	1.02	98.02	0.60	1.000	1.02
tomcatv	99.19	0.54	1.03	56.36	0.53	1.000	1.02
wave5	60.08	5.44	1.05	46.63	5.50	0.998	1.04
Fortran Avg	73.13	4.23	1.06	51.40	4.22	1.000	1.04
compress	63.83	11.67	1.15	29.69	11.67	1.016	1.12
eqntott	86.44	3.77	1.12	41.99	3.74	0.998	1.06
espresso	58.25	10.48	1.17	37.29	10.45	1.014	1.15
gcc	53.76	13.47	1.17	23.24	12.99	0.999	1.12
li	47.45	9.77	1.15	23.94	9.77	0.994	1.11
sc	63.99	4.56	1.17	34.89	4.56	1.000	1.11
C Avg	62.29	8.95	1.16	31.84	8.86	1.004	1.11
cfront	49.41	13.10	1.14	24.03	12.09	0.997	1.09
db++	37.84	18.82	1.20	19.64	18.87	0.999	1.17
idl	39.33	13.99	1.19	14.21	13.94	0.988	1.13
tex	56.42	11.83	1.10	27.49	11.60	0.996	1.07
groff	54.36	7.00	1.14	23.67	6.70	1.002	1.09
Other Avg	47.47	12.95	1.15	21.81	12.64	0.996	1.11

Table 5.7: Branch Alignment Miss Rates and Relative Instructions Executed for Correlated GAg Architecture.

	4096-entry, Correlated GAg PHT						
	Original Alignment			Greedy Alignment			
Program	%MfB	%MpB	CPI	%MfB	%MpB	RIE	CPI
alvinn	97.09	0.24	1.09	95.29	0.23	1.000	1.09
doduc	48.51	4.39	1.06	30.35	3.16	0.996	1.03
ear	75.03	3.98	1.07	65.96	4.15	1.001	1.07
fpppp	50.58	4.75	1.02	20.71	4.36	1.000	1.01
hydro2d	72.13	2.79	1.05	45.49	2.94	1.006	1.04
mdljsp2	81.00	6.69	1.11	21.37	6.60	1.010	1.06
nasa7	76.29	2.56	1.03	65.90	2.43	0.998	1.02
ora	57.49	2.61	1.05	17.79	2.51	0.997	1.02
spice	70.36	4.22	1.11	60.74	5.05	1.000	1.10
su2cor	67.68	9.50	1.05	48.07	9.31	1.000	1.04
swm256	98.27	0.59	1.02	97.86	0.93	1.000	1.02
tomcatv	99.20	0.50	1.03	56.38	0.50	1.000	1.02
wave5	60.39	3.57	1.04	46.97	3.67	0.998	1.03
Fortran Avg	73.39	3.57	1.06	51.76	3.53	1.000	1.04
compress	65.36	9.86	1.15	31.54	9.67	1.016	1.11
eqntott	86.88	2.97	1.11	42.68	2.97	0.998	1.06
espresso	59.54	5.25	1.14	39.68	5.47	1.014	1.12
gcc	53.58	13.42	1.17	24.82	10.76	0.999	1.11
li	49.66	5.08	1.12	26.68	5.49	0.994	1.08
sc	64.38	3.59	1.16	35.30	3.64	1.000	1.10
C Avg	63.23	6.70	1.14	33.45	6.33	1.004	1.10
cfront	48.02	14.41	1.14	23.87	11.16	0.997	1.09
db++	39.43	15.71	1.18	21.56	15.65	0.999	1.15
idl	39.42	13.36	1.18	14.73	13.08	0.988	1.12
tex	55.71	9.83	1.09	27.73	8.84	0.996	1.06
groff	55.11	5.49	1.13	25.04	5.49	1.002	1.08
Other Avg	47.54	11.76	1.14	22.58	10.85	0.996	1.10

Table 5.8: Branch Alignment Miss Rates and Relative Instructions Executed for a 64 entry BTB Architecture.

	64 entry, 2-way associative, 2-Bit BTB						
	Original Alignment			Greedy Alignment			
Program	%MfB	%MpB	CPI	%MfB	%MpB	RIE	CPI
alvinn	0.83	1.56	1.01	0.45	0.48	1.000	1.00
doduc	4.04	8.64	1.03	1.65	5.20	0.996	1.02
ear	0.36	5.34	1.02	0.23	5.34	1.001	1.02
fpppp	1.38	7.88	1.01	0.54	6.41	1.000	1.01
hydro2d	0.03	3.34	1.01	6.01	3.30	1.006	1.02
mdljsp2	0.02	6.80	1.03	0.00	6.84	1.010	1.04
nasa7	0.65	3.01	1.00	0.33	2.37	0.998	1.00
ora	1.51	4.32	1.01	0.20	4.12	0.997	1.01
spice	0.66	7.62	1.04	0.78	7.14	1.000	1.04
su2cor	0.13	8.82	1.02	0.07	8.78	1.000	1.02
swm256	0.02	0.54	1.00	0.02	0.55	1.000	1.00
tomcatv	0.09	0.68	1.00	0.04	0.54	1.000	1.00
wave5	4.33	7.35	1.02	2.03	5.84	0.998	1.01
Fortran Avg	1.08	5.07	1.02	0.95	4.38	1.000	1.01
compress	0.01	13.50	1.08	0.00	11.67	1.016	1.08
eqntott	0.87	3.40	1.02	0.04	2.12	0.998	1.01
espresso	1.53	15.79	1.11	8.03	11.28	1.014	1.10
gcc	10.41	25.88	1.18	8.07	14.30	0.999	1.10
li	12.54	15.80	1.13	5.96	9.34	0.994	1.07
sc	4.72	8.67	1.08	2.27	4.46	1.000	1.04
C Avg	5.01	13.84	1.10	4.06	8.86	1.004	1.07
cfront	17.18	30.40	1.19	12.35	14.45	0.997	1.09
db++	7.24	9.64	1.08	3.72	5.07	0.999	1.04
idl	8.92	11.69	1.11	3.49	2.93	0.988	1.02
tex	8.93	18.59	1.08	5.43	11.69	0.996	1.05
groff	11.20	16.88	1.13	10.60	7.28	1.002	1.07
Other Avg	10.70	17.44	1.12	7.12	8.28	0.996	1.05

Table 5.9. Branch Alignment Miss Rates and Relative Instructions Executed for a 256 entry BTB Architecture.

	256 entry, 4-way assoc, 2-Bit BTB						
	Original Alignment			Greedy Alignment			
Program	%MfB	%MpB	CPI	%MfB	%MpB	RIE	CPI
alvinn	0.09	0.68	1.00	0.04	0.48	1.000	1.00
doduc	1.10	5.60	1.02	0.44	4.67	0.996	1.01
ear	0.00	5.11	1.02	0.00	5.11	1.001	1.02
fpppp	0.02	6.09	1.01	0.00	6.04	1.000	1.01
hydro2d	0.02	3.30	1.01	5.99	3.27	1.006	1.02
mdljsp2	0.00	6.84	1.03	0.00	6.84	1.010	1.04
nasa7	0.12	2.46	1.00	0.02	2.29	0.998	1.00
ora	0.00	4.02	1.01	0.00	4.02	0.997	1.01
spice	0.01	7.09	1.04	0.55	7.06	1.000	1.04
su2cor	0.01	8.74	1.02	0.01	8.73	1.000	1.02
swm256	0.00	0.53	1.00	0.00	0.53	1.000	1.00
tomcatv	0.03	0.56	1.00	0.00	0.52	1.000	1.00
wave5	0.02	5.48	1.01	0.01	5.44	0.998	1.01
Fortran Avg	0.11	4.35	1.01	0.54	4.23	1.000	1.01
compress	0.00	11.67	1.06	0.00	11.67	1.016	1.08
eqntott	0.00	2.08	1.01	0.03	2.07	0.998	1.01
espresso	0.36	10.63	1.07	5.33	10.36	1.014	1.09
gcc	4.98	17.96	1.12	2.88	12.60	0.999	1.08
li	2.10	9.10	1.07	0.20	8.72	0.994	1.06
sc	0.40	4.47	1.04	0.19	4.11	1.000	1.03
C Avg	1.31	9.32	1.06	1.44	8.25	1.004	1.06
cfront	9.73	21.19	1.13	5.19	12.41	0.997	1.07
db++	0.69	5.35	1.04	0.62	4.20	0.999	1.03
idl	1.91	3.22	1.03	1.18	2.41	0.988	1.01
tex	2.80	12.33	1.05	2.27	10.77	0.996	1.04
groff	2.80	8.16	1.06	5.76	5.87	1.002	1.05
Other Avg	3.59	10.05	1.06	3.00	7.13	0.996	1.04

are taken into account, decreases from 1.17 CPI down to 1.15 CPI. In this case it is more advantageous to execute a few more unconditional branches if it allows the algorithm to align more conditional branches, greatly reducing the number of misfetched branches.

Tables 5.6 and 5.7 show that for the PHT architectures all the programs that have an RIE greater than one, meaning that the number of instructions executed is increased, all have an aligned relative CPI for the Greedy algorithm smaller or equal to the Original CPI. This shows that increasing the number of instructions executed by adding unconditional branches does not decrease the programs performance for the PHT architectures. For the BTB architectures shown in Tables 5.8 and 5.9 this is not always true. For the 64-entry BTB architecture, two programs, `hydro2d` and `mdljsp2`, actually have a slightly worse CPI for the aligned program using the Greedy algorithm than the original alignment. For the 256-entry BTB architecture there are four programs, `hydro2d`, `mdljsp2`, `compress`, and `espresso`, that have a slightly worse CPI for the Greedy aligned program than the original alignment. This occurs because the BTB architecture can eliminate misfetch penalties by storing PC-relative target address in the BTB, while the PHT architecture can not eliminate any of these misfetch penalties. Therefore, for the BTB architecture it is not always advantageous to make a conditional branch execute the fall-through if it causes an unconditional branch to be added to the execution stream.

5.7.1 Link-Time Performance of Branch Alignment

We implemented both the Greedy and Try15 alignment algorithms. Figure 5.4 indicates the improvement in the total execution time for the SPEC92 C programs on a DEC 3000-600 with an Alpha AXP 21064 processor running OSF/1 version 2.0. For each program, we show the execution time for the original program, as compiled by the native compiler, the transformed program using the Greedy algorithm, and the transformed program using the Try15 algorithm. We scaled the execution time for each program by the time for the original program.

The programs were compiled as previously described, and then OM [57] was used to link the resulting object files and standard libraries. Therefore, the Original program execution times shown in Figure 5.4 use the standard OM link time optimizations. We then modified OM to produce the desired branch alignments and used this to link the programs.

The Alpha AXP 21064 is a dual issue architecture which uses a combination of dynamic and static branch prediction. Each instruction in the on-chip cache has a single bit indicating the previous branch direction for that instruction. When a cache line is flushed, all the bits are initialized with the bit from each instruction where the sign displacement should be located. Thus the performance expected by this architecture is a cross between a direct mapped PHT table and a BTFNT architecture.

Not surprisingly, the floating point programs, `alvinn` and `ear`, do not see any benefit from the branch alignment, which agrees with our simulation results. We believe some benefit could be gained if the single loop basic blocks (shown in Figure 5.2) were duplicated. The `gcc`, `eqntott` and `sc` programs benefit the most from branch alignment. It is difficult to understand the origin of the actual performance improvement from branch alignment, because our tools did not allow us to instrument and measure the transformed programs, and our trace simulations did not completely model the Alpha AXP 21064 architecture.

For the simulations described in the previous sections, two different chain layout algorithms where used for the Greedy and Try15 alignments. One algorithm laid out chains for a procedure starting with the highest executed chain continuing down to the lowest executed chain. The other algorithm laid out chains using the BTFNT model described by Pettis and Hansen [49] for their Greedy algorithm. We implemented both chain layouts in OM and found that the algorithms that laid the chains out from highest executed to lowest executed performed slightly better than the one that laid out chains using the BTFNT model. We believe this performance comes from the fact that laying out the chains from highest to lowest executed satisfies many of the branch priorities for the BTFNT model, and at the same time allowing better cache locality. Therefore the results shown in Figure 5.4 uses the same Greedy alignment used for all of the simulations (except the BTFNT simulation) with the highest to lowest chain ordering.

In OM we also implemented all the BTFNT, PHT and BTB alignments for the Try15 algorithm

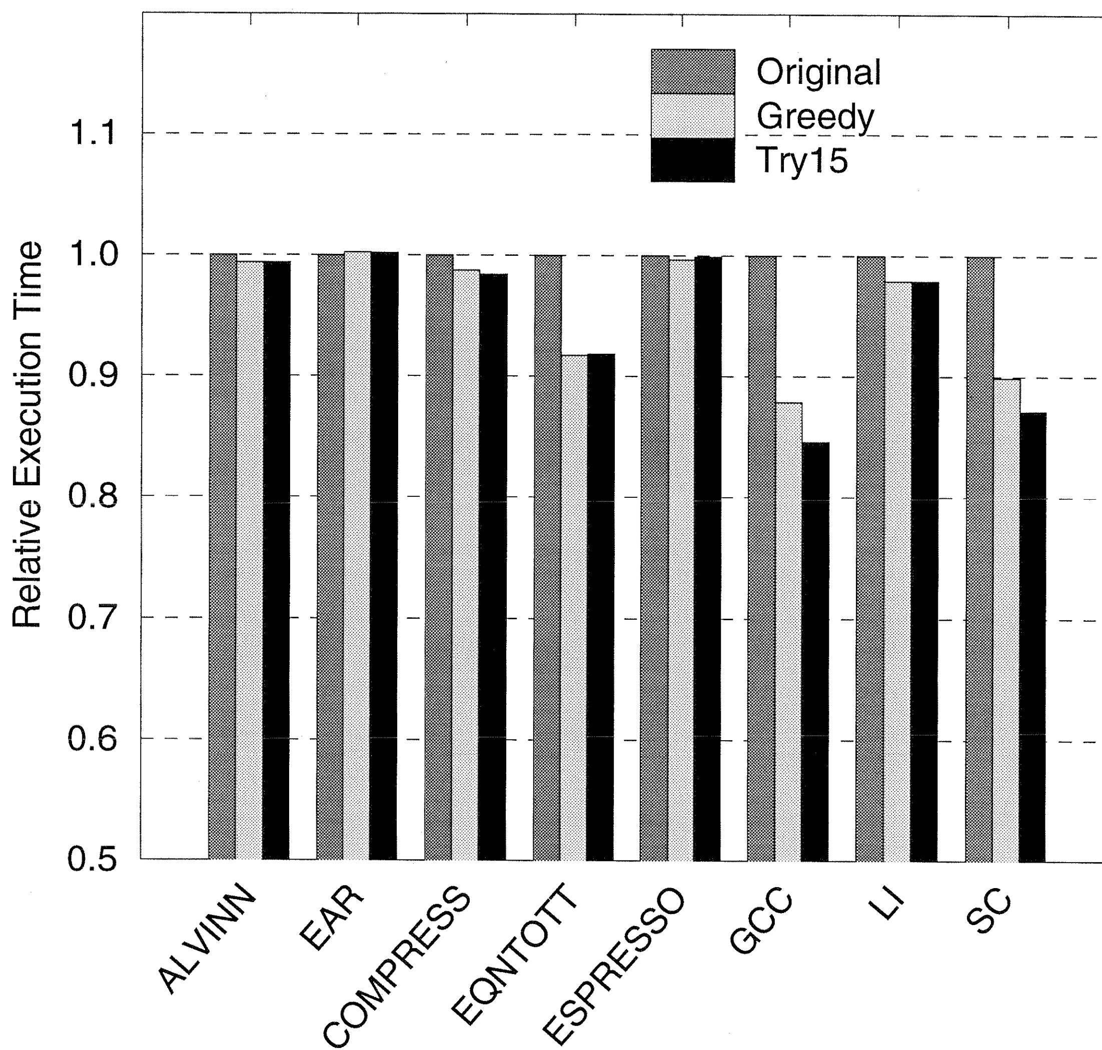


Figure 5.4. Branch Alignment Total Execution Time Improvement on a DEC 3000-600 Alpha AXP for the SPEC92 C Programs.

that were used in the simulations. We found that the Try15 BTB alignment performed the same or slightly better than the Try 15 PHT alignment which was better than the Try15 BTFNT alignment. Recall that when creating a PHT alignment, all taken conditional branches and unconditional branches have a one cycle misfetch penalty in the cost model. In contrast, our BTB cost model assumes a 10% BTB miss rate, which means it assumes the one cycle misfetch penalty only occurs for 10% of the taken branches. In the Alpha AXP 21604 architecture, misfetch penalties can be squashed if the pipeline is currently waiting on other stalls. Therefore, a cost model which would more accurately fit the Alpha AXP 21604 architecture would assume that taken branches are squashed anywhere from 10% to 30% of the time. The Try15 results shown in Figure 5.4 use the same alignment as used for the BTB simulations shown in Table 5.5.

5.8 Summary

This chapter provided BEP simulation results for a number of branch prediction architectures and showed that branch alignment is useful for each architecture. As wide-issue processors become more popular, branch alignment algorithms will have a larger impact on the performance of programs by reducing branch penalties. When these alignment algorithms were implemented, we saw up to 16% improvement in execution time for the dual issue Alpha AXP 21604 architecture. The total reduction in program execution time results from a combination of reduction in the misfetch and misprediction penalties, the instruction cache miss rates, and the number of instructions issued.

We have shown how a simple object code transformation, taking less than a minute to run, even for very large programs, can improve a programs performance. Branch alignment does not benefit all programs, but for C++ and integer programs a reasonable improvement is seen for the various branch prediction architectures. The idea of performing branch alignment as a link-time optimization does not require the recompilation needed by Hwu and Chang [43] or Pettis and Hansen [49]. Branch alignment and other link time optimizations can be applied to all parts of a program, even shared libraries. In related work, we performed a study showing that programs use library code in a very similar fashion between applications [15]. This indicates that it would be beneficial to apply profile-directed optimizations such as Branch Alignment even to shared libraries.

CHAPTER 6

NEXT CACHE LINE SET PREDICTION

As processors issue more instructions concurrently, the likely-hood that a fall-through instruction fetch will be executed decreases. A branch target buffer (BTB) is one mechanism for efficiently predicting the next instruction fetch when a branch is encountered. This chapter proposes the use of an alternative architecture to the BTB called the Next Cache Line and Set (NLS) prediction architecture. A NLS predictor is a pointer into the instruction cache indicating the target instruction of a taken branch. Johnson [32] proposed a similar design using cache indices to predict the next instruction fetch. We propose an alternate organization that improves fetch prediction accuracy. [1]

6.1 Introduction

Chapter 4 showed that branch target buffers can be used to effectively eliminate branch misfetch penalties by storing the target addresses of taken branches. However, branch target buffers can lead to a complex architecture and large BTB's can be costly to implement. In order to examine the cost of the BTB architecture we use the register bit equivalent (RBE) cost model for on-chip memories proposed by Mulder **et al.** [45], where one RBE equals the area cost of a bit storage cell. Figure 6.1 compares the area cost of a direct mapped and 4-way associative 128, 256 and 512 entry BTB to a direct mapped 8K and 16K on-chip instruction cache. The costs assume that static memory cells are used, and that the architecture has a 32-bit address space. The figure shows that the cost for a 512 entry BTB is almost half the cost of an 8K on-chip instruction cache for a 32-bit address space. As the address space increases to 64-bits the BTB costs will double. The fact that a BTB is a costly design is the main motivation for the work in this chapter. One of the goals of this dissertation is to examine alternative architectures that achieve the same or better instruction fetch performance as a BTB for a lower cost, with a simpler design and with a fast access time.

This chapter proposes an alternative instruction fetch and branch prediction architecture called the Next Cache Line and Set prediction (NLS) architecture. We examine two varieties of the NLS architecture. The NLS-cache is similar to the branch architecture described by Johnson [32], where each NLS predictor is associated with a cache line. The NLS-table uses NLS predictors stored in a separate direct mapped tag-less memory buffer. We also examine the effects of combining the NLS predictors with modern two-level correlated branch prediction architectures.

Johnson's previous studies associated each NLS predictor with a cache line and provided only one-bit conditional branch predictors. His cache index predictor stores either the fall-through index or the taken index depending upon the direction of the conditional branch the last time it was executed. The downside to this approach is that the cache index can not take advantage of highly accurate conditional branch prediction information if only one of the target indices, either the fall-through or taken, is available. In contrast, we only store the taken index in the NLS predictor and the fall-through index is calculated during the instruction cache lookup. This effectively provides both the fall-through and taken index and allows the NLS predictor to take full advantage of highly accurate two-level correlated conditional branch prediction architectures. The effect of only storing the taken index in the NLS architecture will also allow the NLS architecture to benefit from compiler optimizations such as Branch Alignment described in the previous chapter. Branch Alignment rearranges basic blocks in order to reduce the number of **taken** branches, and for the NLS architecture this means that fewer NLS predictors would be used, increasing the NLS architecture's hit rate. In contrast, Johnson's predictor provides either the fall-through or the taken index and his predictors would not benifit from such code optimizations. Another difference between the two approaches is that we use the NLS predictors to predict

[1]Parts of this chapter were published in the 22nd Annual International Symposium of Computer Architecture [10].

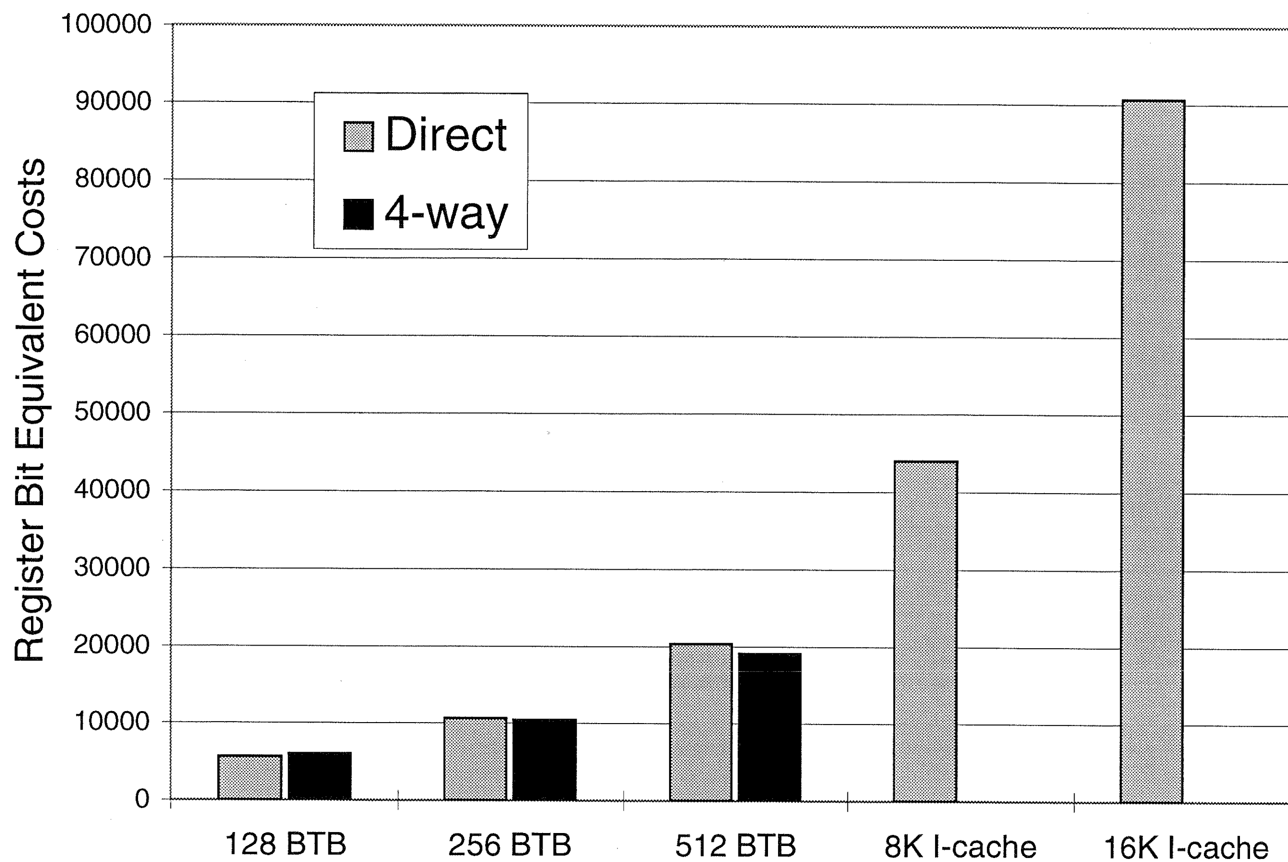


Figure 6.1: Register Bit Equivalent Costs for On-chip BTB and Instruction Cache Architectures.

all PC-relative branches and indirect jump instructions, whereas Johnson's predictors are used to only predict PC-relative branches.

In this chapter, we examine associating the NLS predictors with the instruction cache, as in Johnson's design, and we examine the performance of decoupling the NLS predictors from the cache line and storing them in a separate tag-less memory buffer. Our results show that the decoupled architecture performs better than associating the NLS predictors with the cache line, that the NLS architecture benefits from reduced cache miss rates, and it is particularly effective for programs containing many branches. We also provide an in-depth comparison between the NLS and BTB architectures, showing that the NLS architecture is a competitive alternative to the BTB design.

6.2 The BTB Instruction Fetch Prediction Architecture

Figure 6.2 is a schematic representation of the decoupled BTB and PHT branch prediction and instruction fetch architecture we simulated. In Figure 6.2 the next instruction fetch address is concurrently offered to: the instruction cache, the BTB, and the PHT. The address is also used to compute the fall-through instruction's address. A 32-entry return address stack [33] predicts return instructions, and conditional branches are predicted using the pattern history table organization described by McFarling [41]. This is the degenerate scheme of Pan **et al.** [47] (GAg), where we XOR the global history register with the program counter and use this to index into a 4096 entry (8192 bits) PHT. In this model, we store only taken branches in the BTB, since previous studies have shown this to be more effective [7, 48]. If a branch is not taken while it is in the BTB, we leave the branch (target address) in the BTB until it is removed due to the LRU replacement policy, since we might need the taken target address again in the near future. In this architecture, the BTB's main purpose is to eliminate misfetch penalties by providing the taken target address and the branch type.

6.3 Next Cache Line and Set Prediction Architecture

The NLS architecture is similar to the BTB architecture and is illustrated in Figure 6.3. The difference between these two architectures is that the NLS architecture is a tagless table, providing a pointer into the instruction cache to the next instruction to execute rather than the target address, as in the BTB. Like the BTB, the main purpose of the NLS architecture is to eliminate misfetch penalties by providing a pointer to the cache line and the instruction that is the target of a branch. This allows the next instruction to be correctly fetched from the instruction cache while the branch instruction is decoded and the target address is calculated. The NLS predictor also predicts indirect jumps and provides the branch type.

As shown in Figure 6.3 there are three predicted addresses available for the next instruction fetch. These are the NLS predictor, the fall-through line (previous predicted line + fetch size), and the top of the return stack. Each NLS predictor contains the following fields:

Type Field: Table 6.1 shows the possible prediction sources represented by the NLS type field. The type field is used to determine the proper prediction mechanism, shown in Figure 6.3, to use when fetching the next instruction. Unused NLS entries have "00" stored in the type field indicating the entry is invalid.[2]

Table 6.1: NLS Prediction Sources

		Branch Type	Prediction Source
0	0	Invalid Entry	Fall-Through PC
0	1	Return Instruction	Return Stack
1	0	Conditional Branch	NLS Entry, or Fall-Through PC Depending the PHT Prediction
1	1	Other Types of Branches	Always use NLS Entry

[2]The type field is not needed for the NLS or BTB architectures if the type information can be easily extracted from the fetched instruction before the fetch cycle completes, or from the instruction cache if the information has been pre-decoded.

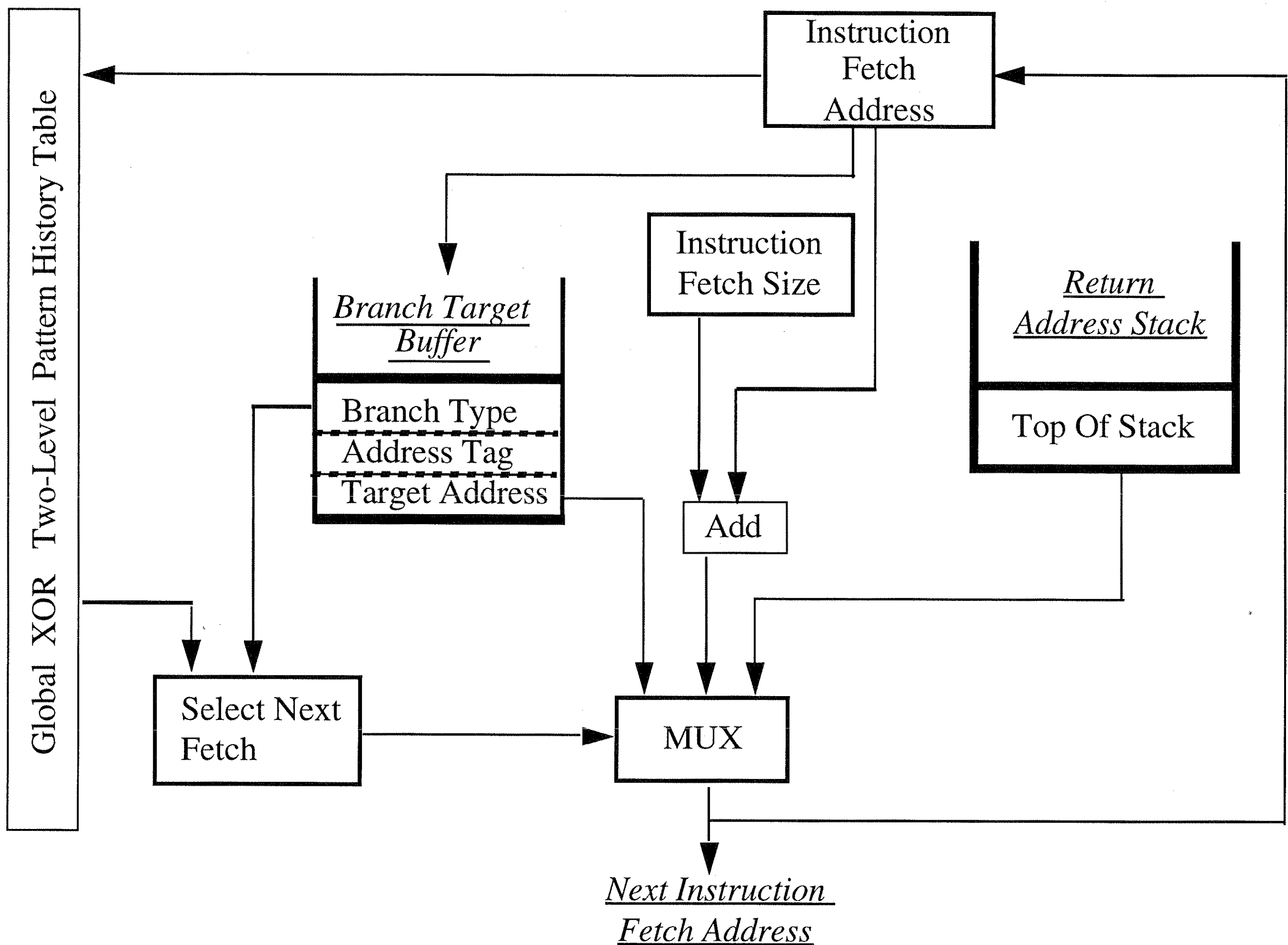


Figure 6.2. A schematic representation of a decoupled BTB branch prediction architecture using two-level correlated branch prediction for conditional branches and a return stack for return instructions.

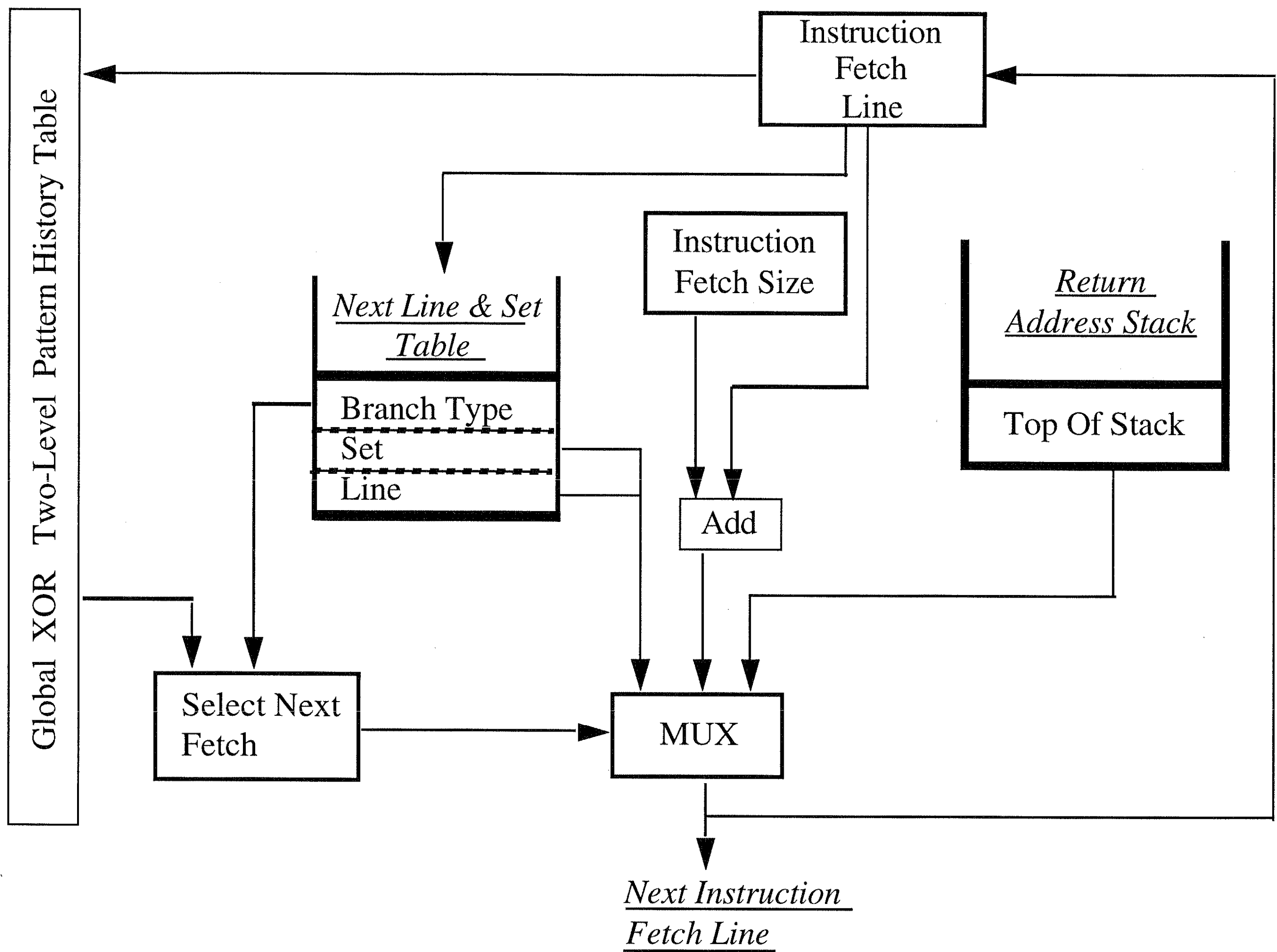


Figure 6.3: A Schematic Representation of the NLS-Table Architecture.

Line Field: This field contains the line number to be fetched from the instruction cache. The high-order bits indicate the line in the instruction cache and the low-order bits are used to indicate the actual instruction in that line.

Set Field: In a multi-associative instruction cache, the destination line may be in any set. The set field is used to indicate where the predicted line is located if a multi-associative cache is used. It is not needed for a direct mapped cache.

The NLS architecture assumes that during the instruction fetch stage of the pipeline, each instruction can easily be identified as a branch or non-branch instruction. The BTB does not have to make this assumption since an instruction is known to be a branch if it hits in the BTB. We assume that if the instruction set encoding does not contain such a distinguishing bit in the instruction, that information can be stored in the instruction cache. Encoding this information in the instruction or the instruction cache improves the fetch accuracy for the NLS architecture, since non-branch instructions fetch the fall-through address while branch instructions use NLS predictors.

If the instruction being fetched from the instruction cache indicates that it is a branch instruction, the NLS predictor is used and the type field is examined to choose among the possible next fetch addresses. Return instructions use the return stack, and unconditional branches and indirect branches use the cache line specified by the NLS entry. If the type field indicates a conditional branch, the architecture uses the prediction given by the PHT, as is done in the BTB architecture. If the branch is predicted as taken, the NLS line and set fields are used to fetch the appropriate cache line and instruction from the instruction cache. If the conditional branch is predicted as not-taken, the precomputed fall-through line address is used on the next instruction fetch.

The NLS entries are updated after instructions are decoded and the branch type and destinations are resolved. The instruction type determines the type field and the branch destination determines the set and line field. Only taken branches update the set and line field, but all branches update the type field. A conditional branch which executes the fall-through should not update the set and line field, since that would erase the pointer to the target instruction. For conditional branches, this allows the branch prediction hardware to use either the NLS predictor for taken conditional branches or to use the precomputed fall-through line, depending on the outcome of the PHT.

6.3.1 NLS-Table Versus NLS-Cache

There are several possible variations on the basic NLS architecture design, and they share many common structures. Figure 6.3 showed one possible design. The intuition behind this architecture is that a branch target address is actually a pointer into the instruction cache. This pointer can be represented by an index pointing to the target instruction of a taken branch.

We considered two possible designs: "NLS-caches" and "NLS-tables". In the NLS-cache, we associate the NLS predictors with each cache line. Thus, the NLS entries share the instruction address tag with the cache line. There may be multiple NLS predictors per cache line and we studied various replacement policies and methods of associating the NLS predictors with specific instructions in a cache line. The second design, the NLS-table, is a simpler and more effective design that uses a tag-less direct-mapped table of NLS predictors. The table is indexed by the branch instruction's address. Both architectures use the NLS entries to predict the next line to fetch for a branch instruction, both architectures use the same conditional branch prediction and return-prediction mechanisms used in the BTB, and both designs replace the BTB with the NLS information.

The NLS-table has three advantages over the NLS-cache design and one disadvantage. These points arise because the NLS predictors are coupled with the cache lines in the NLS-cache design and they are decoupled from the cache in the NLS-table design. For the NLS-cache architecture, we found that associating two NLS predictors with an eight instruction cache line to be the most effective organization. This design restricts the use of the NLS predictors in the NLS-cache, since some cache lines may not have any branches while other cache lines may contain several branches.

In contrast, the NLS-table uses the lower order bits of the branch instructions address to index into a

tagless table. This allows a cache line to use as many NLS predictors as needed. The second advantage comes when an instruction cache line is replaced. In the model we simulated, we assumed that the NLS-cache prediction information associated with a replaced cache line would be discarded while the prediction information for the NLS-table is preserved across cache misses. The final advantage appears when examining different instruction cache sizes. As the instruction cache size doubles, the number of NLS-cache predictors must also double to achieve the same branch prediction performance. Therefore the NLS-cache size increases linearly with an increase in instruction cache size, while the NLS-table size increases only logarithmically. This can greatly increase the cost of the NLS-cache design for large caches.

There is a disadvantage for the NLS-table in making it a tagless table, because prediction information from one branch may be erroneously used for another branch. Our results show that this effect is small for the NLS-table design when compared to the benefits of the three advantages mentioned above.

6.3.2 Using Next Line Addresses with the Instruction Cache

Unlike the BTB architecture, the NLS architecture does not have a full next target address to offer to the instruction cache. It only has the lower order bits of the full target address (the cache line index). This is not a problem for a direct mapped cache, since the tag check against the target address can be performed in the decode stage of the pipeline. When an associative cache is used, the cache needs to be slightly modified in order to properly use the next line address. The following two different approaches may be taken.

The traditional implementation of an associative cache selects the appropriate line from a set by performing a full tag comparison on the tags from the different sets. For all branch instructions, the set field in the NLS predictor is used to predict the instruction cache set instead of performing the tag comparison. When the precomputed fall through line address is used, a full tag comparison is performed. The full fall-through address can be calculated by the time the cache needs to perform the tag comparison using the precomputed fall-through line address, the carry bit from the addition of the fall-through line address calculated in the previous cycle, and the previous instruction's tag.

The second approach to using next line addresses with an associative cache is more elegant and can lead to improved cache performance. In this approach we assume that each cache line has a **set field** associated with it. This set field has the same use as the NLS set field, and it predicts the set where the fall-through line is located for each cache line. For each instruction cache lookup, either the NLS predictor's set field, for a branch instruction, or the previous cache line's set field, for a non-branch instruction, is used to predict the set for the current cache access. Since the set field is used on every cache access, only one cache set is driven at a time during the lookup and the tag comparison can be performed in the decode stage as if the cache where direct mapped. If the set prediction was incorrect and the tag does not match the destination address computed in the decode stage, the other sets in the cache need to be checked in order to find the correct entry or to find if there is a cache miss. This design is suitable for a two-way associative cache. If the first set prediction is incorrect, the remaining set is checked for the instruction. For higher degrees of associativity, other prediction techniques may be applied when the NLS set predictor is incorrect.

This second approach only drives one cache associative set at a time. Therefore the NLS architecture with a fall-through set prediction field in the instruction cache could be used to allow a direct mapped cache to achieve associative cache performance. This idea is similar to the MRU approach by Kessler **et al.** [37], where each pair of cache blocks uses an "MRU bit" to indicate the most recently used block. When searching for an instruction, the set indicated by the MRU bit is probed first. If the instruction is not found, the second set is probed. If the instruction is found in the second set, the MRU bit is inverted, indicating the second set is more recently used than the first set. Other studies have also examined techniques to predict which set information is located in for instruction and data caches in order to achieve associative behavior from a direct mapped cache [12, 18, 53] .

6.3.3 Identifying Instructions as Branch Instructions

The idea of using a NLS predictor could easily be added to the BTB design where one replaces the target address with the NLS predictor. The main difference between this new BTB design and the NLS-table would be that the BTB design still has a tag associated with each BTB entry. We did not examine this design because we felt that there are better mechanisms for achieving what the tag offers. Having a tag in the BTB achieves two objectives. The first is that a match with the tag in the BTB indicates that the current instruction fetch contains a branch. The second feature is that a hit in the BTB indicates that the target address or NLS predictor being used matches the branch being fetched from the instruction cache. If this second feature is compromised, as in the NLS-table design, one branches' prediction information may erroneously be used by another branch, but this is not a problem for the NLS-table since it has many more predictors than the BTB. If this second feature is sacrificed, then the only use for the tag in the BTB design is it indicates that the current instruction fetch contains a branch. We felt there are better and more efficient ways of providing this information. Therefore, we did not examine associating a tag with the NLS-table design.

Information can be added to the instruction cache or stored in a separate table indicating if the current instruction fetch contains a branch or not. These bits can easily be initialized when a cache line is read into the cache for the first time. These branch bits will indicate whether or not the NLS prediction information should be used, since it indicates if the current instruction fetch contains a branch or not. This can easily be accomplished by having a distinguishing bit in the instruction encoding indicating if the current instruction contains a branch. If the instruction encoding is too dense to allow this, then a small amount of predecoding would be necessary to determine this information as the instructions are brought into the instruction cache. The hardware cost for this design is small in comparison to storing a tag with each BTB entry, and the initialization of the branch bits only needs to be performed once when a cache line is place into the instruction cache.

6.4 Simulation Methodology

For each program, we simulated 8KB, 16KB, and 32KB instruction caches with 32 byte cache lines and 4 byte instructions. For each cache size, we simulated direct mapped, 2-way and 4-way associative LRU replacement caches. When simulating the NLS-cache architecture, we used one to four NLS predictors per cache line with varying replacement policies. When simulating the NLS-table architecture, we simulated NLS-table sizes with 512, 1024 and 2048 NLS predictors. For the BTB architecture, we simulated 128-entry and 256-entry BTB organizations with direct mapped, 2-way and 4-way associativity with LRU replacement. We chose these configurations because of their chip area costs. Both the BTB and NLS architectures used a 32-entry return stack [33] to predict procedure returns and a two-level correlated 4096-entry pattern history table for conditional branches. The accuracy of the pattern history table is the same for both the BTB and NLS architectures. This allows us to isolate the instruction fetch prediction performance differences between the BTB and NLS architectures.

Table 6.2: Instruction Cache Misses Rates for Traced Programs.

I-Cache Size	Assoc	Doduc	Espresso	Gcc	Li	Cfront	Groff
8K	Direct	2.91	0.40	4.35	3.27	7.03	4.81
8K	2-way	1.62	0.17	3.76	0.66	6.28	3.92
8K	4-way	1.32	0.10	3.52	0.21	5.84	3.07
16K	Direct	2.15	0.25	2.76	0.49	4.69	2.89
16K	2-way	0.89	0.08	1.97	0.30	3.70	1.72
16K	4-way	0.71	0.02	1.50	0.03	3.21	1.29
32K	Direct	0.48	0.16	1.67	0.06	2.58	1.63
32K	2-way	0.50	0.03	0.91	0.02	1.75	0.82
32K	4-way	0.53	0.00	0.65	0.01	1.44	0.54

Due to the large number of configurations we only simulated the following six programs: `doduc`, `espresso`, `gcc`, `li`, `cfront`, `groff`. This also allows us to examine the performance of each program in more detail. We chose these programs because most of them have a high instruction cache miss rate. Table 6.2 shows the instruction cache miss rates for these programs for an 8K, 16K and 32K direct mapped, two-way and 4-way associative instruction caches. Since the NLS architectures performance is correlated with the instruction cache miss rate (especially the NLS-cache design), these programs would bring to light any performance concerns for the NLS architecture.

We compare the branch architectures using the branch execution penalty performance metric. We record the percentage of misfetched branches (%MfB), and the percentage of mispredicted branches (%MpB). We compute the **branch execution penalty** as before to be:

$$\text{BEP} = \frac{\%\text{MfB} \times \text{misfetch penalty} + \%\text{MpB} \times \text{misprediction penalty}}{100}.$$

The results in this chapter assume a one cycle misfetch penalty and a four cycle mispredict penalty. In all of our BEP graphs, we break the results into two parts. The top part shows the fraction of the BEP caused by the misfetch penalties and the lower part shows the fraction due to the mispredict penalties.

6.5 Calculating Register Bit Equivalent Costs

In order to compare the NLS architecture to the BTB we must first determine the resource costs for each architecture. To evaluate the area implementation costs of the NLS and BTB architectures we used the register bit equivalent (RBE) model for on-chip memories proposed by Mulder **et al.** [45], where one RBE equals the area of a bit storage cell. For the NLS architecture, we used the $area_{static\ memory}$ model since the NLS architecture is a tagless direct mapped memory buffer. Table 6.3 shows the values used for the NLS architecture, where $size_w$ is the size of the structure in words, and $line_b$ is the size of each NLS predictor in bits. The NLS predictor consists of the Line and Set fields along with the 2-bit Type field to predict the branch type. For an 8K instruction cache, the NLS architecture we modeled needs 8 bits to indicate the cache line, 3 bits to indicate the instruction in the cache line to start fetching at, plus 2-bits for the branch Type field. Note that this assumes only one instruction is fetched per cycle. If more instructions are fetched per cycle, fewer bits are needed to represent which instruction to start fetching at for each NLS predictor.

For the BTB architecture, we used the $area_{set\ associative\ cache}$ for calculating the costs for on-chip caches. Table 6.4 shows the values used for the BTB architectures, where $size_b$ is the size of the data-side of the cache in bits, $line_b$ is the size of each BTB-entry in bits, $tags$ is the number of tags used in each configuration, and tsb_b is the size in bits for each tag. We assumed a 32-bit address architecture, and we assumed each instruction was four bytes. Therefore, the number of bits needed for tsb_b is 32 bits, minus 2 bits since each instruction is four bytes, plus 2 bits for the Type field (as in the NLS architecture), and minus $\lg(tags/associativity)$ bits. Also note that as the address size of the processor increases the cost of the BTB can increase significantly.

Figure 6.4 shows the RBE costs for implementing the NLS and BTB architectures using Mulder **et al.** [45] on-chip memory area model. The figure shows the RBE costs for the NLS-cache (with two NLS predictors per cache line) and a 512, 1024, and 2048 entry NLS-table for cache sizes of 8K, 16K, 32K and 64K. It also shows the RBE cost for a 128 entry and 256 entry BTB with associativities of one, two and four.

The RBE cost of the NLS architectures depend on the size of the instruction cache. The NLS-table's RBE cost increases logarithmically as the instruction cache size increases, since the line field for each NLS predictor has to also increase. When the number of lines in the instruction cache are doubled, another bit must be added to each NLS predictor's line field. In the NLS-cache architecture, the number of NLS entries per cache line is constant, and as the cache size increases, the space devoted to NLS entries increases linearly. The RBE cost of the BTB architecture depends on the associativity of the BTB and the size of the address space not on the size of the instruction cache. In performing the BTB calculations, we assumed a 32-bit address space is used. If the address space is increased, the cost of the BTB would also increase.

Table 6.3: Values Used to Calculated RBE Costs for NLS Architectures.

NLS Configuration	$Cache\ Size$	$size_w$	$line_b$
NLS Cache, 512 predictors	8K	208	13
NLS Cache, 1024 predictors	16K	448	14
NLS Cache, 2048 predictors	32K	960	15
NLS Cache, 4096 predictors	64K	2048	16
512 NLS Table	8K	208	13
512 NLS Table	16K	224	14
512 NLS Table	32K	240	15
512 NLS Table	64K	256	16
1024 NLS Table	8K	416	13
1024 NLS Table	16K	448	14
1024 NLS Table	32K	480	15
1024 NLS Table	64K	512	16
2048 NLS Table	8K	832	13
2048 NLS Table	16K	896	14
2048 NLS Table	32K	960	15
2048 NLS Table	64K	1024	16

Table 6.4: Values Used to Calculate RBE Costs for BTB Architectures.

BTB Size	Associativity	$size_b$	$line_b$	$tags$	tsb_b
128 BTB	Direct	4096	32	128	23
128 BTB	2-way	4096	32	128	24
128 BTB	4-way	4096	32	128	25
256 BTB	Direct	8192	32	256	22
256 BTB	2-way	8192	32	256	23
256 BTB	4-way	8192	32	256	24

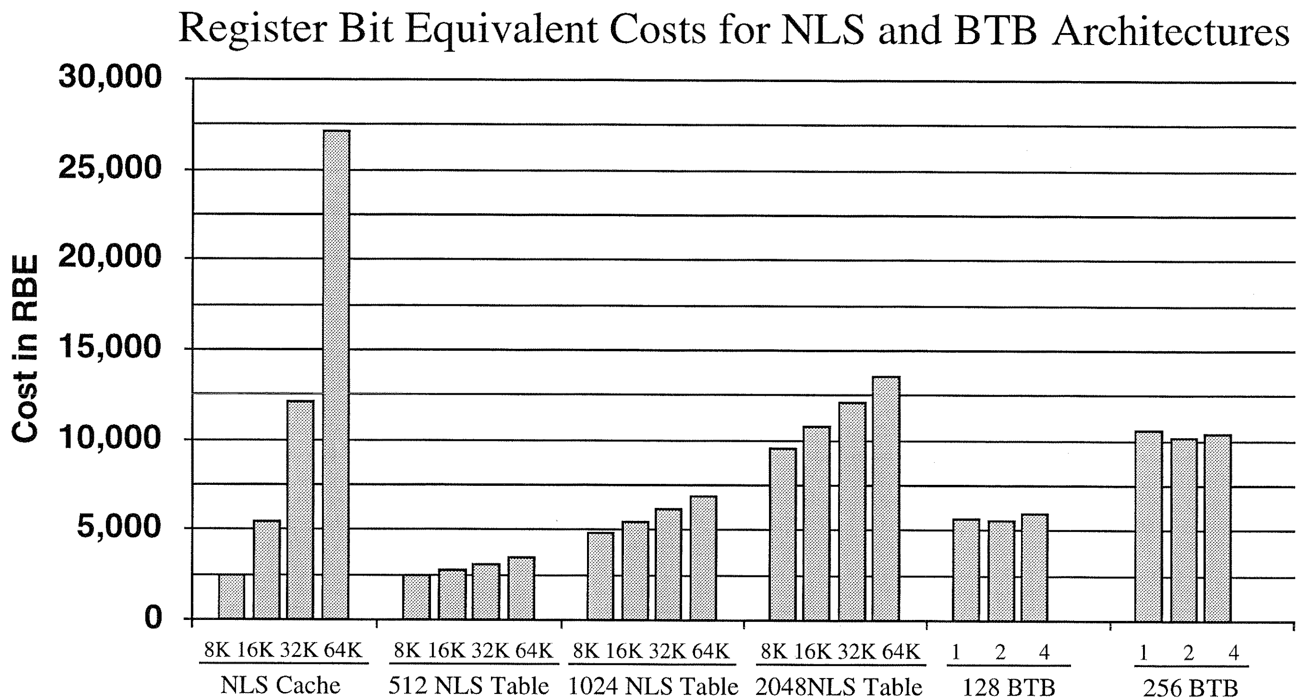


Figure 6.4. Register bit equivalent costs for the NLS-cache and a 512, 1024 and 2048-entry NLS-table for cache sizes of 8K, 16K, 32K and 64K, and for a 128-entry and 256-entry BTB with associativities of one, two and four.

6.6 NLS Architecture Results

This section provides performance results for the NLS-cache and NLS-table architectures, discusses ideas for improving the NLS-cache design, and describes how our designs differ from related work.

6.6.1 Performance of the NLS-Cache Architecture

There are many possible designs and configurations for the NLS-cache architecture. We examined associating from one to four NLS predictors per eight instructions in the instruction cache, and we examined directly (Direct) mapping the predictors onto the instructions and using fully associative mappings with least recently used (LRU) as the replacement policy. Figure 6.5 shows the branch execution results averaged over all the programs in Table 6.5 for the configurations we examined. The figure shows the results for associating 1, 2 and 4 NLS predictors with an eight instruction cache line, directly mapping the predictors onto the cache line. It also shows results for using 2 and 3 NLS predictors with a fully associative mapping onto the cache line with LRU replacement. The results show that two NLS predictors per eight instruction cache line (2-Direct) gives the most cost effective performance. In this configuration, the first NLS predictor is associated with the first four instructions in the cache line and the second NLS predictor is associated with the last four instructions in the cache line. Additional performance is seen by adding more predictors and increasing the associativity, but the improvement in performance is minimal in comparison to the cost of adding the predictors and increasing the associativity.

Tables 6.5, 6.6, and 6.7 show the percent of misfetched branches (MfB), the percent of mispredicted branches (MpB), and the branch execution penalty (BEP) for these NLS-cache configurations and each of the programs we examined.

6.6.2 Performance of the NLS-Table Architecture

Figure 6.6 shows the branch execution penalty results averaged over the programs in Table 6.2. The figure shows results for the 2-Direct NLS-cache, and for 512, 1024 and 2048 entry NLS-tables for varying instruction cache sizes and associativities. Each branch execution penalty (BEP) value is broken into two parts with the upper part representing the fraction of the BEP due to the misfetch penalty, and the lower part the mispredict penalty.

Figure 6.6 shows that the NLS-table consistently outperforms the NLS-cache when architectures with equivalent costs are examined: the NLS-cache and the 512 NLS-table have equivalent costs when using an 8K instruction cache, the NLS-cache and the 1024 NLS-table have equivalent costs when using a 16K instruction cache, and the NLS-cache and the 2048 NLS-table have equivalent costs when using a 32K instruction cache. In terms of the RBE cost, the NLS-cache is practical for only small caches (8K and 16K), but even then, the NLS-table architecture has better performance when comparing architectures with equivalent costs. The difference in performance arises from the NLS-cache discarding useful prediction information for each instruction cache miss and because the predictors in the NLS-cache can only be used for the cache line with which they are associated. In contrast, the NLS-table preserves prediction information across cache misses and a NLS-table predictor's use is not restricted to a given cache line. Figure 6.6 also shows that the difference in performance between the 512 and 1024 NLS-tables is small, and the difference between the 1024 and 2048 NLS-tables is smaller still. In the remainder of this chapter we focus on the NLS-table design and only give results for the 1024 entry NLS-table. Tables 6.8 and 6.9, show the percent of misfetched branches (MfB), the percent of mispredicted branches (MpB), and the branch execution penalty (BEP) for all the NLS-table configurations for each instruction cache size and associativity and for each of the programs examined.

6.6.3 Increasing the Performance of the NLS-Cache Architecture

Other NLS-cache organizations, not examined in this dissertation, can be studied in order to try and match the performance of the NLS-table. The 512 entry NLS-table and the 2-Direct NLS-cache have the same number of predictors for an 8K instruction cache. The difference in performance, as stated above, in these two architectures comes from restricting the usage of the predictors in the NLS-cache design to a fixed number

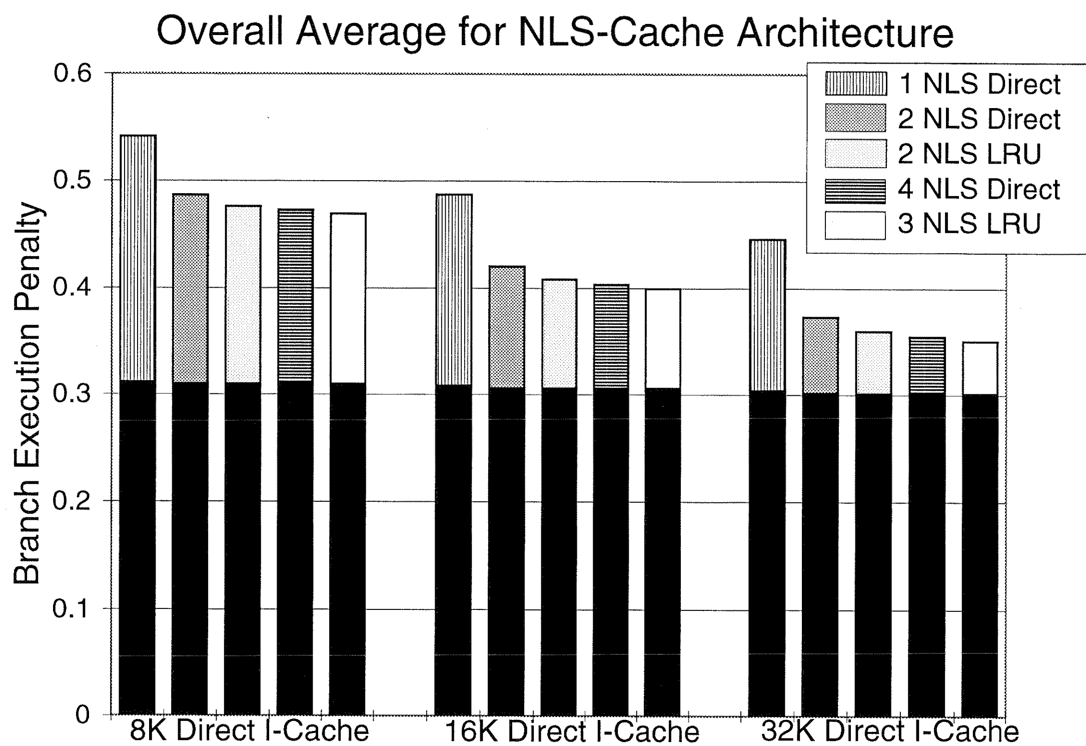


Figure 6.5. Average NLS-Cache Performance. The BEP value is broken into two parts with the upper part representing the fraction of the BEP due to the misfetch penalty, and the lower part the mispredict penalty.

Table 6.5: NLS-Cache Performance for Doduc and Espresso.

I-Cache Size & Assoc.	# NLS Entries	Replace Policy	Doduc			Espresso		
			%MfB	%MpB	BEP	%MfB	%MpB	BEP
8K Direct	1	Direct	12.73	4.39	0.30	15.34	5.11	0.36
	2	Direct	10.87	4.39	0.28	6.11	5.07	0.26
	4	Direct	10.83	4.39	0.28	3.48	5.07	0.24
	2	LRU	10.80	4.39	0.28	4.52	5.07	0.25
	3	LRU	10.80	4.39	0.28	1.99	5.07	0.22
8K 2-way	1	Direct	7.29	4.39	0.25	14.79	5.11	0.35
	2	Direct	5.11	4.39	0.23	5.33	5.07	0.26
	4	Direct	5.06	4.39	0.23	2.49	5.07	0.23
	2	LRU	5.03	4.39	0.23	3.53	5.07	0.24
	3	LRU	5.03	4.39	0.23	1.00	5.07	0.21
8K 4-way	1	Direct	6.18	4.39	0.24	14.55	5.11	0.35
	2	Direct	3.86	4.39	0.21	4.99	5.07	0.25
	4	Direct	3.81	4.39	0.21	2.13	5.07	0.22
	2	LRU	3.78	4.39	0.21	3.17	5.07	0.23
	3	LRU	3.78	4.39	0.21	0.62	5.07	0.21
16K Direct	1	Direct	10.15	4.39	0.28	14.92	5.11	0.35
	2	Direct	7.94	4.39	0.26	5.51	5.07	0.26
	4	Direct	7.90	4.39	0.25	2.79	5.07	0.23
	2	LRU	7.87	4.39	0.25	3.83	5.07	0.24
	3	LRU	7.87	4.39	0.25	1.29	5.07	0.22
16K 2-way	1	Direct	5.24	4.39	0.23	14.50	5.11	0.35
	2	Direct	2.89	4.39	0.20	4.89	5.07	0.25
	4	Direct	2.83	4.39	0.20	2.03	5.07	0.22
	2	LRU	2.81	4.39	0.20	3.08	5.07	0.23
	3	LRU	2.80	4.39	0.20	0.53	5.07	0.21
16K 4-way	1	Direct	4.58	4.39	0.22	14.27	5.11	0.35
	2	Direct	2.24	4.39	0.20	4.58	5.07	0.25
	4	Direct	2.19	4.39	0.20	1.66	5.07	0.22
	2	LRU	2.16	4.39	0.20	2.71	5.07	0.23
	3	LRU	2.16	4.39	0.20	0.15	5.07	0.20
32K Direct	1	Direct	3.93	4.39	0.22	14.64	5.11	0.35
	2	Direct	1.55	4.39	0.19	5.12	5.07	0.25
	4	Direct	1.47	4.39	0.19	2.32	5.07	0.23
	2	LRU	1.43	4.39	0.19	3.37	5.07	0.24
	3	LRU	1.42	4.39	0.19	0.82	5.07	0.21
32K 2-way	1	Direct	4.11	4.39	0.22	14.32	5.11	0.35
	2	Direct	1.75	4.39	0.19	4.66	5.07	0.25
	4	Direct	1.68	4.39	0.19	1.78	5.07	0.22
	2	LRU	1.64	4.39	0.19	2.83	5.07	0.23
	3	LRU	1.64	4.39	0.19	0.27	5.07	0.21
32K 4-way	1	Direct	4.21	4.39	0.22	14.22	5.11	0.35
	2	Direct	1.80	4.39	0.19	4.50	5.07	0.25
	4	Direct	1.74	4.39	0.19	1.57	5.07	0.22
	2	LRU	1.71	4.39	0.19	2.62	5.07	0.23
	3	LRU	1.71	4.39	0.19	0.06	5.07	0.20

Table 6.6: NLS-Cache Performance for Gcc and Li.

I-Cache Size & Assoc.	# NLS Entries	Replace Policy	Gcc %MfB	Gcc %MpB	Gcc BEP	Li %MfB	Li %MpB	Li BEP
8K Direct	1	Direct	25.63	12.63	0.76	23.80	4.14	0.40
	2	Direct	19.97	12.54	0.70	17.90	4.14	0.34
	4	Direct	17.93	12.53	0.68	15.70	4.14	0.32
	2	LRU	18.05	12.53	0.68	16.50	4.14	0.33
	3	LRU	17.44	12.53	0.68	15.70	4.14	0.32
8K 2-way	1	Direct	24.44	12.58	0.75	14.45	4.09	0.31
	2	Direct	18.73	12.48	0.69	6.56	4.04	0.23
	4	Direct	16.60	12.46	0.66	3.64	4.04	0.20
	2	LRU	16.75	12.46	0.67	4.93	4.04	0.21
	3	LRU	16.06	12.46	0.66	3.64	4.04	0.20
8K 4-way	1	Direct	23.58	12.58	0.74	12.45	4.08	0.29
	2	Direct	17.69	12.48	0.68	4.08	4.03	0.20
	4	Direct	15.62	12.46	0.65	1.06	4.03	0.17
	2	LRU	15.76	12.46	0.66	2.38	4.03	0.19
	3	LRU	15.10	12.46	0.65	1.06	4.03	0.17
16K Direct	1	Direct	21.17	12.58	0.71	13.69	4.13	0.30
	2	Direct	14.59	12.48	0.65	5.53	4.08	0.22
	4	Direct	12.14	12.46	0.62	2.52	4.08	0.19
	2	LRU	12.44	12.46	0.62	3.85	4.08	0.20
	3	LRU	11.63	12.46	0.61	2.52	4.08	0.19
16K 2-way	1	Direct	19.26	12.53	0.69	13.21	4.09	0.30
	2	Direct	12.40	12.42	0.62	5.08	4.04	0.21
	4	Direct	9.84	12.40	0.59	2.09	4.04	0.18
	2	LRU	10.16	12.40	0.60	3.42	4.04	0.20
	3	LRU	9.31	12.40	0.59	2.09	4.04	0.18
16K 4-way	1	Direct	17.60	12.52	0.68	11.80	4.07	0.28
	2	Direct	10.41	12.41	0.60	3.29	4.02	0.19
	4	Direct	7.72	12.38	0.57	0.20	4.02	0.16
	2	LRU	8.08	12.38	0.58	1.54	4.02	0.18
	3	LRU	7.16	12.38	0.57	0.20	4.02	0.16
32K Direct	1	Direct	17.69	12.54	0.68	11.85	4.06	0.28
	2	Direct	10.36	12.43	0.60	3.43	4.01	0.19
	4	Direct	7.62	12.41	0.57	0.40	4.01	0.16
	2	LRU	8.03	12.41	0.58	1.73	4.01	0.18
	3	LRU	7.07	12.41	0.57	0.40	4.01	0.16
32K 2-way	1	Direct	15.33	12.50	0.65	11.75	4.06	0.28
	2	Direct	7.67	12.39	0.57	3.23	4.01	0.19
	4	Direct	4.82	12.36	0.54	0.15	4.01	0.16
	2	LRU	5.25	12.36	0.55	1.49	4.01	0.18
	3	LRU	4.26	12.36	0.54	0.15	4.01	0.16
32K 4-way	1	Direct	14.29	12.48	0.64	11.68	4.06	0.28
	2	Direct	6.46	12.37	0.56	3.16	4.01	0.19
	4	Direct	3.52	12.34	0.53	0.07	4.01	0.16
	2	LRU	3.98	12.34	0.53	1.41	4.01	0.17
	3	LRU	2.95	12.34	0.52	0.07	4.01	0.16

Table 6.7: NLS-Cache Performance for Cfront and Groff.

I-Cache Size & Assoc.	# NLS Entries	Replace Policy	Cfront			Groff		
			%MfB	%MpB	BEP	%MfB	%MpB	BEP
8K Direct	1	Direct	32.77	14.01	0.89	27.94	6.38	0.53
	2	Direct	29.52	14.00	0.86	22.07	6.24	0.47
	4	Direct	29.23	14.00	0.85	21.16	6.23	0.46
	2	LRU	29.20	14.00	0.85	21.24	6.23	0.46
	3	LRU	29.08	14.00	0.85	21.11	6.23	0.46
8K 2-way	1	Direct	31.01	13.92	0.87	25.00	6.20	0.50
	2	Direct	27.54	13.91	0.83	18.50	6.06	0.43
	4	Direct	27.26	13.91	0.83	17.52	6.04	0.42
	2	LRU	27.22	13.91	0.83	17.62	6.04	0.42
	3	LRU	27.11	13.91	0.83	17.46	6.04	0.42
8K 4-way	1	Direct	29.80	13.90	0.85	21.70	5.89	0.45
	2	Direct	25.94	13.89	0.81	14.31	5.76	0.37
	4	Direct	25.51	13.89	0.81	13.27	5.75	0.36
	2	LRU	25.47	13.89	0.81	13.41	5.75	0.36
	3	LRU	25.36	13.89	0.81	13.21	5.75	0.36
16K Direct	1	Direct	26.16	13.91	0.82	21.60	6.04	0.46
	2	Direct	21.00	13.90	0.77	14.15	5.89	0.38
	4	Direct	20.56	13.90	0.76	13.05	5.87	0.37
	2	LRU	20.51	13.90	0.76	13.23	5.88	0.37
	3	LRU	20.32	13.90	0.76	13.00	5.87	0.36
16K 2-way	1	Direct	23.64	13.87	0.79	17.68	5.65	0.40
	2	Direct	18.32	13.85	0.74	9.26	5.48	0.31
	4	Direct	17.76	13.85	0.73	8.07	5.46	0.30
	2	LRU	17.73	13.85	0.73	8.28	5.47	0.30
	3	LRU	17.52	13.85	0.73	8.01	5.46	0.30
16K 4-way	1	Direct	22.03	13.86	0.77	15.80	5.53	0.38
	2	Direct	16.14	13.83	0.71	6.81	5.35	0.28
	4	Direct	15.56	13.83	0.71	5.50	5.33	0.27
	2	LRU	15.51	13.83	0.71	5.72	5.34	0.27
	3	LRU	15.31	13.83	0.71	5.43	5.34	0.27
32K Direct	1	Direct	20.09	13.82	0.75	17.16	5.62	0.40
	2	Direct	13.83	13.80	0.69	8.73	5.46	0.31
	4	Direct	13.04	13.80	0.68	7.49	5.44	0.29
	2	LRU	12.99	13.80	0.68	7.68	5.45	0.29
	3	LRU	12.75	13.80	0.68	7.42	5.44	0.29
32K 2-way	1	Direct	17.30	13.78	0.72	14.36	5.48	0.36
	2	Direct	10.16	13.76	0.65	5.31	5.29	0.26
	4	Direct	9.32	13.75	0.64	3.90	5.27	0.25
	2	LRU	9.27	13.76	0.64	4.15	5.28	0.25
	3	LRU	9.01	13.76	0.64	3.82	5.27	0.25
32K 4-way	1	Direct	15.95	13.75	0.71	13.36	5.36	0.35
	2	Direct	8.47	13.72	0.63	3.98	5.16	0.25
	4	Direct	7.56	13.72	0.62	2.51	5.14	0.23
	2	LRU	7.50	13.72	0.62	2.75	5.14	0.23
	3	LRU	7.24	13.72	0.62	2.42	5.14	0.23

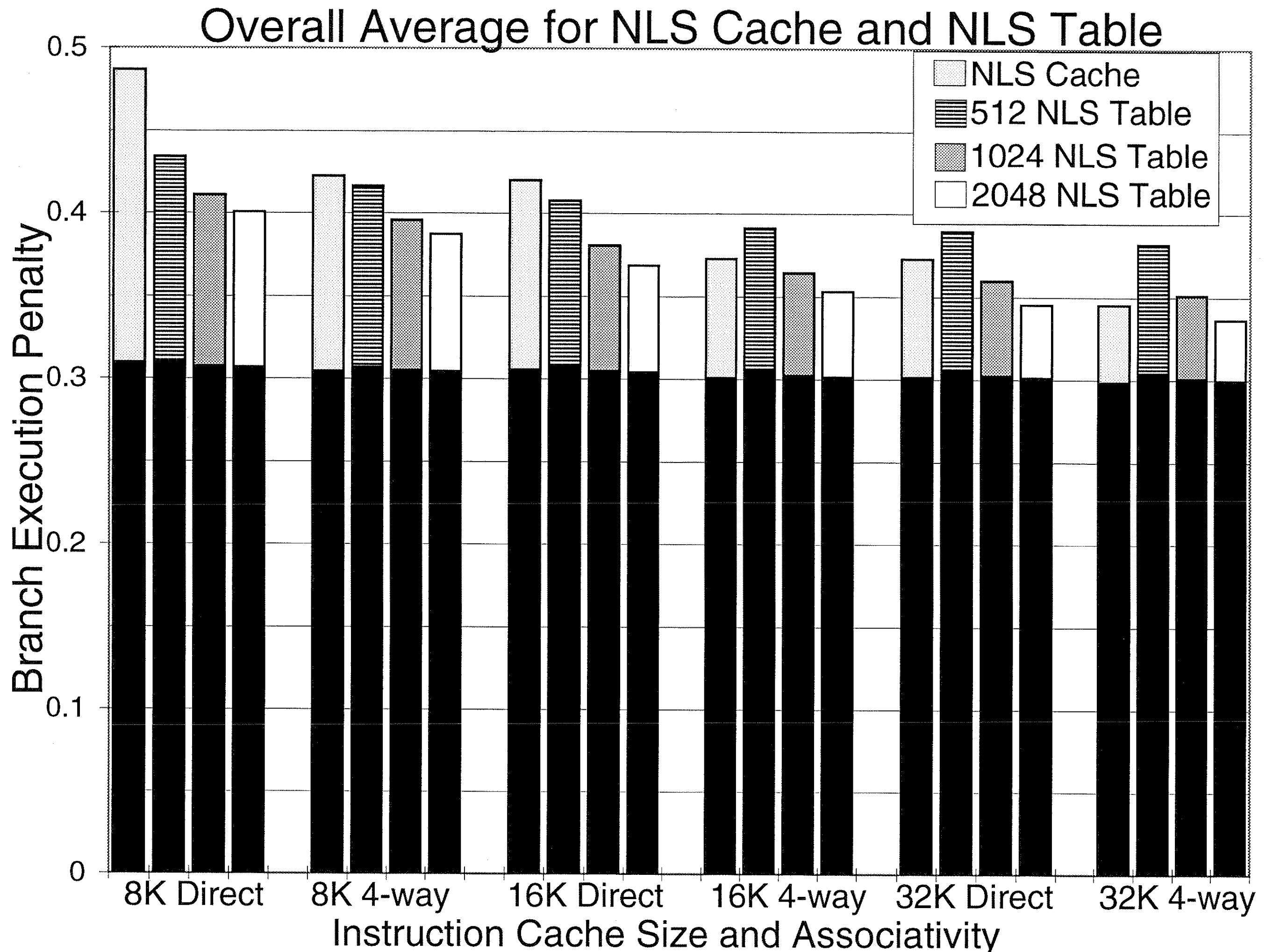


Figure 6.6. Branch execution penalty for the NLS-cache architecture and the 512, 1024 and 2048 entry NLS-table architectures for direct mapped and 4-way associative caches of size 8K, 16K and 32K. The BEP value is broken into two parts with the upper part representing the fraction of the BEP due to the misfetch penalty, and the lower part the mispredict penalty.

Table 6.8: NLS-Table Performance for Doduc, Espresso and Li

Instr Cache	NLS Size	Doduc			Espresso			Li		
		%MfB	%MpB	BEP	%MfB	%MpB	BEP	%MfB	%MpB	BEP
8K Direct	512	6.88	4.39	0.24	1.55	5.07	0.22	8.15	4.10	0.25
	1024	6.02	4.39	0.24	0.90	5.07	0.21	7.35	4.09	0.24
	2048	5.72	4.39	0.23	0.79	5.07	0.21	6.92	4.09	0.23
8K 2-way	512	4.88	4.39	0.22	1.40	5.07	0.22	4.49	4.04	0.21
	1024	4.01	4.39	0.22	0.76	5.07	0.21	3.60	4.04	0.20
	2048	3.96	4.39	0.22	0.68	5.07	0.21	3.19	4.04	0.19
8K 4-way	512	4.17	4.39	0.22	1.23	5.07	0.21	3.42	4.03	0.20
	1024	3.35	4.39	0.21	0.59	5.07	0.21	2.54	4.03	0.19
	2048	3.22	4.39	0.21	0.51	5.07	0.21	2.10	4.03	0.18
16K Direct	512	5.55	4.39	0.23	1.34	5.07	0.22	3.74	4.09	0.20
	1024	4.67	4.39	0.22	0.67	5.07	0.21	2.84	4.08	0.19
	2048	4.36	4.39	0.22	0.55	5.07	0.21	2.39	4.08	0.19
16K 2-way	512	3.59	4.39	0.21	1.20	5.07	0.21	3.79	4.04	0.20
	1024	2.79	4.39	0.20	0.53	5.07	0.21	2.89	4.04	0.19
	2048	2.52	4.39	0.20	0.41	5.07	0.21	2.45	4.04	0.19
16K 4-way	512	3.22	4.39	0.21	1.03	5.07	0.21	3.05	4.02	0.19
	1024	2.39	4.39	0.20	0.34	5.07	0.21	2.14	4.02	0.18
	2048	2.20	4.39	0.20	0.22	5.07	0.20	1.68	4.02	0.18
32K Direct	512	2.72	4.39	0.20	1.22	5.07	0.21	3.05	4.02	0.19
	1024	1.70	4.39	0.19	0.53	5.07	0.21	2.13	4.01	0.18
	2048	1.26	4.39	0.19	0.41	5.07	0.21	1.67	4.01	0.18
32K 2-way	512	2.87	4.39	0.20	1.09	5.07	0.21	3.04	4.02	0.19
	1024	1.95	4.39	0.20	0.41	5.07	0.21	2.13	4.01	0.18
	2048	1.60	4.39	0.19	0.28	5.07	0.21	1.67	4.01	0.18
32K 4-way	512	2.94	4.39	0.21	1.01	5.07	0.21	3.02	4.02	0.19
	1024	2.06	4.39	0.20	0.31	5.07	0.21	2.10	4.01	0.18
	2048	1.76	4.39	0.19	0.18	5.07	0.20	1.64	4.01	0.18

Table 6.9: NLS-Table Performance for Gcc, Cfront and Groff

Instr Cache	NLS Size	Gcc			Cfront			Groff		
		%MfB	%MpB	BEP	%MfB	%MpB	BEP	%MfB	%MpB	BEP
8K Direct	512	15.26	12.53	0.65	25.06	14.05	0.81	17.92	6.26	0.43
	1024	12.69	12.50	0.63	21.11	14.02	0.77	14.15	6.00	0.38
	2048	11.25	12.50	0.61	18.76	14.01	0.75	12.81	5.95	0.37
8K 2-way	512	15.90	12.50	0.66	26.12	13.98	0.82	17.44	6.17	0.42
	1024	13.64	12.47	0.64	22.64	13.97	0.79	13.78	5.92	0.37
	2048	12.55	12.48	0.62	20.82	13.97	0.77	12.74	5.89	0.36
8K 4-way	512	15.51	12.52	0.66	25.66	13.99	0.82	15.79	5.99	0.40
	1024	13.35	12.50	0.63	22.48	13.98	0.78	12.26	5.73	0.35
	2048	12.23	12.50	0.62	20.67	13.98	0.77	11.12	5.68	0.34
16K Direct	512	12.88	12.47	0.63	21.91	13.96	0.78	14.90	6.07	0.39
	1024	9.80	12.43	0.60	17.26	13.91	0.73	10.53	5.79	0.34
	2048	8.12	12.42	0.58	14.37	13.89	0.70	9.02	5.71	0.32
16K 2-way	512	12.48	12.46	0.62	21.40	13.93	0.77	13.43	5.86	0.37
	1024	9.54	12.42	0.59	17.10	13.89	0.73	9.04	5.59	0.31
	2048	7.97	12.41	0.58	14.61	13.88	0.70	7.55	5.51	0.30
16K 4-way	512	11.65	12.46	0.61	20.44	13.92	0.76	12.52	5.86	0.36
	1024	8.49	12.42	0.58	16.04	13.90	0.72	7.99	5.53	0.30
	2048	6.91	12.41	0.57	13.56	13.89	0.69	6.50	5.43	0.28
32K Direct	512	11.50	12.45	0.61	19.28	13.92	0.75	12.91	5.88	0.36
	1024	8.06	12.41	0.58	14.09	13.87	0.70	8.24	5.55	0.30
	2048	6.16	12.40	0.56	10.65	13.85	0.66	6.51	5.41	0.28
32K 2-way	512	10.70	12.44	0.60	18.42	13.89	0.74	12.03	5.81	0.35
	1024	7.18	12.39	0.57	13.10	13.84	0.68	7.19	5.48	0.29
	2048	5.29	12.38	0.55	9.78	13.81	0.65	5.49	5.36	0.27
32K 4-way	512	10.32	12.43	0.60	17.92	13.87	0.73	11.62	5.80	0.35
	1024	6.65	12.38	0.56	12.39	13.81	0.68	6.66	5.44	0.28
	2048	4.66	12.36	0.54	8.95	13.78	0.64	4.90	5.28	0.26

of instructions and because the NLS-cache predictors are flushed when an instruction cache line is flushed. Therefore, one way of improving the performance of the NLS-cache design we modeled is to not flush the predictors when the cache line is flushed. This in itself will not bring the NLS-cache performance equal to the NLS-table performance, because there is only one predictor, in the 2-Direct design, for every four instructions in the cache. When the cache line is swapped out, there is a reasonably high probability that the new cache line will contain a branch which will erase the old NLS predictor's contents. Therefore, when the old cache line is brought back into the cache, the NLS predictor might not contain the prediction information associated with the previous line. If there were more predictors per cache line (e.g., one for every instruction) then the probability of the predictor still retaining useful information across cache misses increases. This leads to the question of how to properly map predictors onto the NLS-cache design in order to achieve similar performance to the NLS-table design with the same number of predictors.

One solution is to have the cache lines share predictors so that each cache line has one predictor associated with every instruction. Then, if a cache conflict exists, the likelihood that the NLS predictor still contains useful information increases. If we assume we have an 8K instruction cache, with eight instructions per cache line (32 bytes each) and 256 lines, we could achieve this goal by having 4 cache lines share 8 NLS predictors. This design would have the same number of NLS predictors for the instruction cache as the 2-Direct NLS-cache design. In this case the first predictor is associated with the first instruction in each cache line, the second predictor with the second instruction in each of the four cache lines and so on. This in effect would allow a branch intensive cache line to use more predictors if it needs, and would also increase the likelihood of predictors retaining their prediction information across cache misses. This could be implemented by spreading the 4 cache lines, that share the 8 NLS predictors, 32 cache lines apart in the 8K instruction cache. This may work better than having the 4 cache lines right next to each other in the instruction cache because of the principle of locality. It would be better to have cache lines that are not used together to share predictors rather than having instructions that are close together share predictors. Notice that each time we modify the NLS-cache design so that cache lines share predictors in this way, we are getting closer and closer to the NLS-table design, and the one disadvantage of this design is that one predictor for a given cache line can be erroneously used to predict a branch for a different cache line.

6.6.4 Related Work

There are several branch prediction strategies related to the NLS design. Our NLS-table architecture was derived from the BTB: each uses a table holding pointers to branch destinations. The primary difference, besides eliminating the tag, is that the BTB encodes the full address, while the NLS encodes only the instruction cache line and set, allowing for larger NLS-tables.

Bray and Flynn [6] described a design similar to the NLS-cache that associated branch target addresses with each cache line. As in our study, they found approximately one entry per four instructions provided the most cost effective design.

Johnson [32], suggested the idea of using cache successor indices as in the NLS-cache architecture for instruction fetch and branch prediction. His architecture associated the cache indices with each cache line as the NLS-cache architecture does. The architecture he studied is slightly different than our NLS-cache design since he only considered using the index for one bit conditional branch prediction. With one bit prediction, the cache index stores either a pointer to the fall-through line or the target line for the next instruction fetch. In order to predict the fall-through line, the cache index is updated even when a non-taken branch is executed. By comparison, we only update the NLS predictor when taken branches are encountered to obtain improved branch prediction accuracy when using a decoupled PHT. In addition, we expanded the design by examining various configurations for associating the predictors with each cache line, including set prediction for associative instruction caches, and we decoupled the predictors from the cache line forming the NLS-table design.

Variations on the NLS-cache design can be found in recent microprocessor architectures. The TFP microprocessor (MIPS R8000) [31] has a 1024 entry NLS-cache architecture similar to the design proposed by Johnson. It has one NLS predictor for every four instructions, and a one-bit branch predictor coupled with each

NLS predictor. The UltraSPARC microprocessor also uses a similar 1024 entry NLS-cache design, associating an NLS predictor with every four instructions. Instead of using one-bit prediction as in the TFP, the UltraSPARC uses a 2-bit dynamic conditional branch predictor for every two instructions in the instruction cache.

The NLS-table design uses an independent table of next line and set predictors. This basic design was recently patented by Steely and Sager [58]. However, they have not published any performance comparisons, and the patented design only addresses direct mapped caches, while our design addresses both direct mapped and associative caches. Furthermore, the patented architecture uses a single "computed goto" register to store the destination of indirect jumps. By comparison, we use the NLS predictor to provide the predicted cache index for **all** branch destinations other than fall-through branches and return instructions. Although we developed our NLS architecture independently, there are other similarities as well as other differences; see [58] for more details.

6.7 Performance of the BTB Architecture

Figure 6.7 shows the average branch execution penalty (BEP) for the programs simulated in this paper. The 1024 entry NLS-table has better performance than the similar costing 128 entry BTB, even when the BTB has a high degree of associativity. The 1024 entry NLS-table and 256 entry BTB exhibit comparable performance even though the 1024 entry NLS-table has roughly half the RBE cost of the 256 entry BTB. Table 6.10 gives the detailed percentage of misfetched branches (MfB), mispredicted branches (MpB), and the BEP for all the BTB configurations simulated.

Table 6.10: Branch Target Buffer Performance

		128 entry BTB			256 entry BTB			512 entry BTB		
Program	A	%MfB	%MpB	BEP	%MfB	%MpB	BEP	%MfB	%MpB	BEP
doduc	1	5.69	4.39	0.23	3.03	4.39	0.21	1.80	4.39	0.19
	2	3.15	4.39	0.21	2.03	4.39	0.20	0.89	4.39	0.18
	4	2.52	4.39	0.20	1.77	4.39	0.19	0.76	4.39	0.18
espresso	1	5.25	5.07	0.26	1.78	5.07	0.22	0.97	5.07	0.21
	2	2.12	5.07	0.22	0.76	5.07	0.21	0.27	5.07	0.21
	4	1.68	5.07	0.22	0.44	5.07	0.21	0.10	5.07	0.20
li	1	11.01	4.00	0.27	4.69	3.95	0.21	2.27	4.01	0.18
	2	8.56	4.06	0.25	1.50	4.02	0.18	0.50	4.01	0.17
	4	7.44	4.06	0.24	0.66	4.01	0.17	0.05	4.01	0.16
gcc	1	18.64	12.46	0.68	12.78	12.41	0.62	8.26	12.38	0.58
	2	17.43	12.46	0.67	10.56	12.40	0.60	5.63	12.35	0.55
	4	16.24	12.45	0.66	9.09	12.39	0.59	3.90	12.35	0.53
cfront	1	28.86	13.94	0.85	21.70	13.87	0.77	13.64	13.79	0.69
	2	27.21	13.90	0.83	17.98	13.85	0.73	10.70	13.78	0.66
	4	25.12	13.93	0.81	15.84	13.82	0.71	8.56	13.73	0.63
groff	1	23.41	6.29	0.49	15.03	5.87	0.39	9.41	5.70	0.32
	2	19.78	5.96	0.44	12.16	5.65	0.35	5.07	5.30	0.26
	4	17.60	5.84	0.41	8.46	5.42	0.30	3.63	5.21	0.24

When comparing the NLS-table to the BTB, one must keep in mind that a direct mapped BTB has a shorter access time than an associative BTB. Figure 6.8 shows the estimated access time, in nanoseconds, for a 128 entry BTB and 256 entry BTB with direct mapped, two, and four way associativity. These estimates were derived using the CACTI timing model of Wilton and Jouppi [63]. Their model derives access times for direct mapped and associative caches such as BTBs, but not for tag-less direct mapped memory buffers. Therefore we do not show the access times for the NLS-table, but we believe it would be similar to that of a direct mapped BTB. This figure shows the access time differences between direct mapped and associative cache structures. The differences arise from the extra time needed to perform the tag comparison. In a direct mapped cache, the tag comparison can be done in parallel while the data output is being driven to the next stage. The figure shows that the 4-way associative BTB access time is 30 to 40% longer than the same sized direct mapped BTB. This

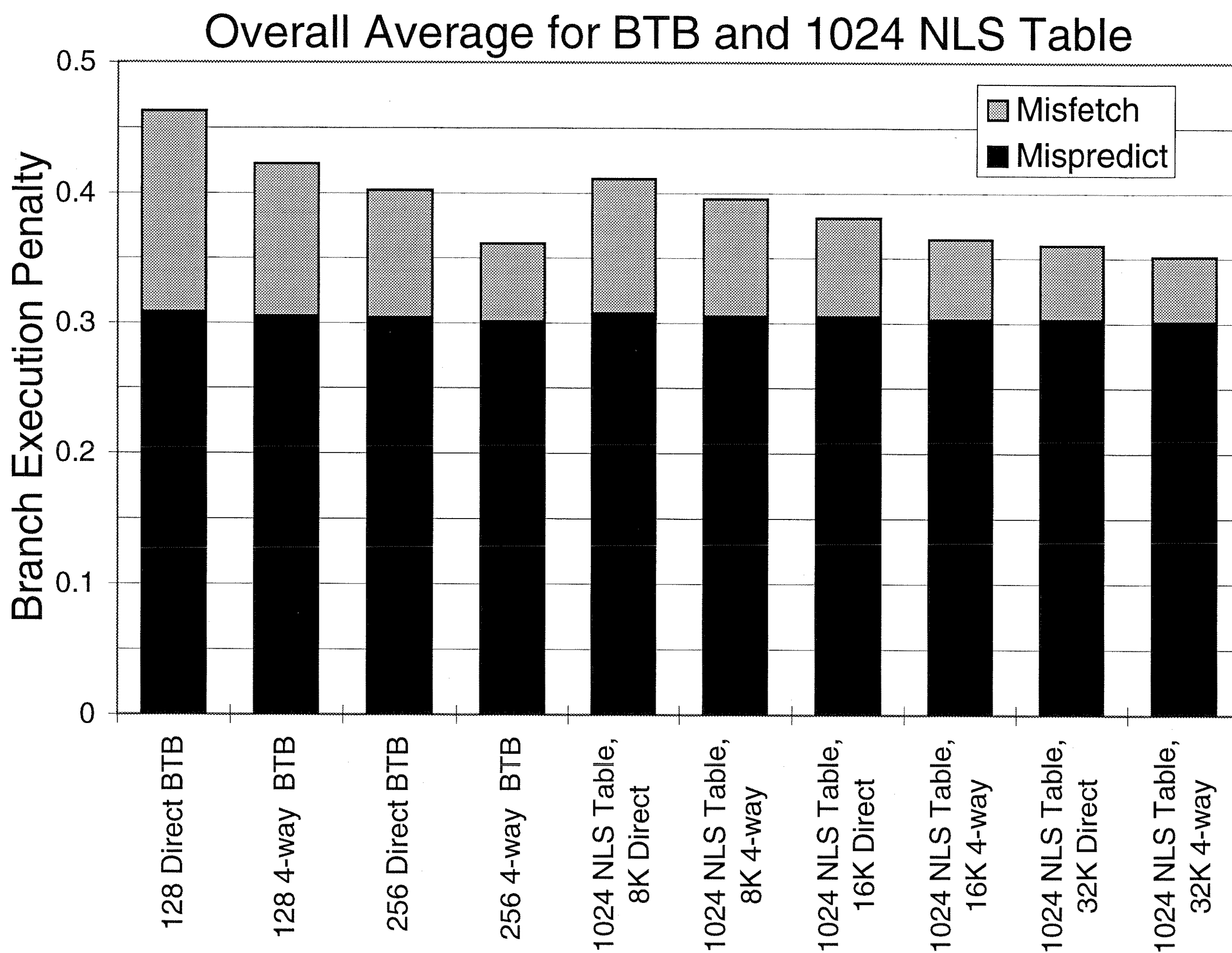


Figure 6.7. Branch execution penalty for the 1024 entry NLS-table architecture for direct mapped and 4-way associativity instruction caches of size 8K, 16K and 32K, and for a 128 entry and 256 entry BTB.

should be considered when comparing the performance of the direct mapped BTB and NLS architecture to the associative BTB architectures since the cycle limitation of the instruction fetch may effect the entire machine. In [31], the designers of the TFP (MIPS R8000) microprocessor stated:

> **We evaluated several well-known branch prediction algorithms for layout size, speed, and prediction accuracy. The most critical factor affecting area was the infrastructure required to support a custom block: power ring and power straps to the ring, and global routing between the branch prediction cache and its control logic. Speed was a problem with tag comparisons for those schemes that are associative. Accordingly we chose a simple direct-mapped, one-bit prediction scheme which can be implemented entirely with a single-ported RAM.**

6.8 Comparison of NLS-Table and BTB Architectures

Figures 6.9, 6.10, 6.11, 6.12, 6.13, and 6.14 compare the performance of the NLS and BTB architectures using the branch execution penalty (BEP) for the programs in Table 6.2. Each graph compares the direct mapped and 4-way set associative 128 and 256 entry BTBs to the 1024 entry NLS-table. It was shown in previous sections that the 1024 entry NLS-table and the 128 entry BTB have similar implementation costs using the RBE model, that the 256 entry BTB implementation cost is twice that of the 1024 entry NLS-table, and that the access time of an associative BTB is 30 to 40% longer than similar sized direct mapped structures.

The differences in the BEP between the BTB and NLS architectures is attributable to differences in the number of misfetched branches. Remember that the BTB and NLS architectures are not used to predict the direction for conditional branches. Conditional branch prediction information is stored in a separate pattern history table (PHT) and the conditional branches are predicted using the PHT. The NLS and BTB architectures are used to eliminate the misfetch penalty associated with the extra cycle taken to determine the branch type and to compute the target address for the next instruction fetch. Once the branch type and target line are predicted, the next fetch line can be chosen from the return stack, precomputed fall-through line, or the predicted target line. Both the NLS and BTB architectures are used to predict the destination for indirect jumps. Table 3.4 shows that indirect jumps constitute 0-5% of the breaks in the programs we instrumented. In these BEP figures, any difference in the mispredict penalty for a given program is attributed to the variation in the mispredict penalty for indirect jumps across the different architectures. The figure shows that the difference in mispredict penalty across the different architectures is only noticeable for `groff`, and even then the difference is insignificant.

These figures show that the BEP for the NLS architecture decreases as the cache size increases or the cache associativity increases. Recall that each NLS predictor indicates the cache line that should be fetched. The information associated with a NLS predictor is only useful if the actual destination of a branch is in the predicted location in the instruction cache. In smaller caches, the NLS predictors will often point to the proper cache line and set, but the desired instruction may not be present or may have been reloaded into a different set. With a NLS predictor, a branch destination that has been displaced from the instruction cache causes a misfetch penalty. When the current instruction is fully decoded, the misfetch is detected and the actual instruction is fetched. In this case, the misfetch penalty is associated with a cache miss. In contrast, the BTB always uses the full target address. This allows the BTB architecture to possibly locate the proper instruction in set associative caches, or to initiate an instruction cache miss a cycle earlier than the NLS architecture. If an associative cache is used, the NLS architecture would have to look in the other set on a misfetched branch, or do a full set lookup. When the cache miss rate is lowered, there is an increased probability that a cache line will still be resident when a NLS predictor is used. The BTB architecture will not benefit from the lower cache miss rate, and there is no change in the BEP for varying cache configurations. Whole-program restructuring [40, 43, 49] is one technique that can be used to reduce the instruction cache miss rate at no additional architectural cost.

Why does the NLS architecture have significantly better BEP performance than the BTB for some programs, such as `gcc`, `cfront` and `groff`, but only slightly better or comparable performance for other programs, such as `doduc` and `espresso`? The program characteristics in Table 3.5 show the programs that benefit most from the NLS architecture have more static branch sites then the programs that show little benefit.

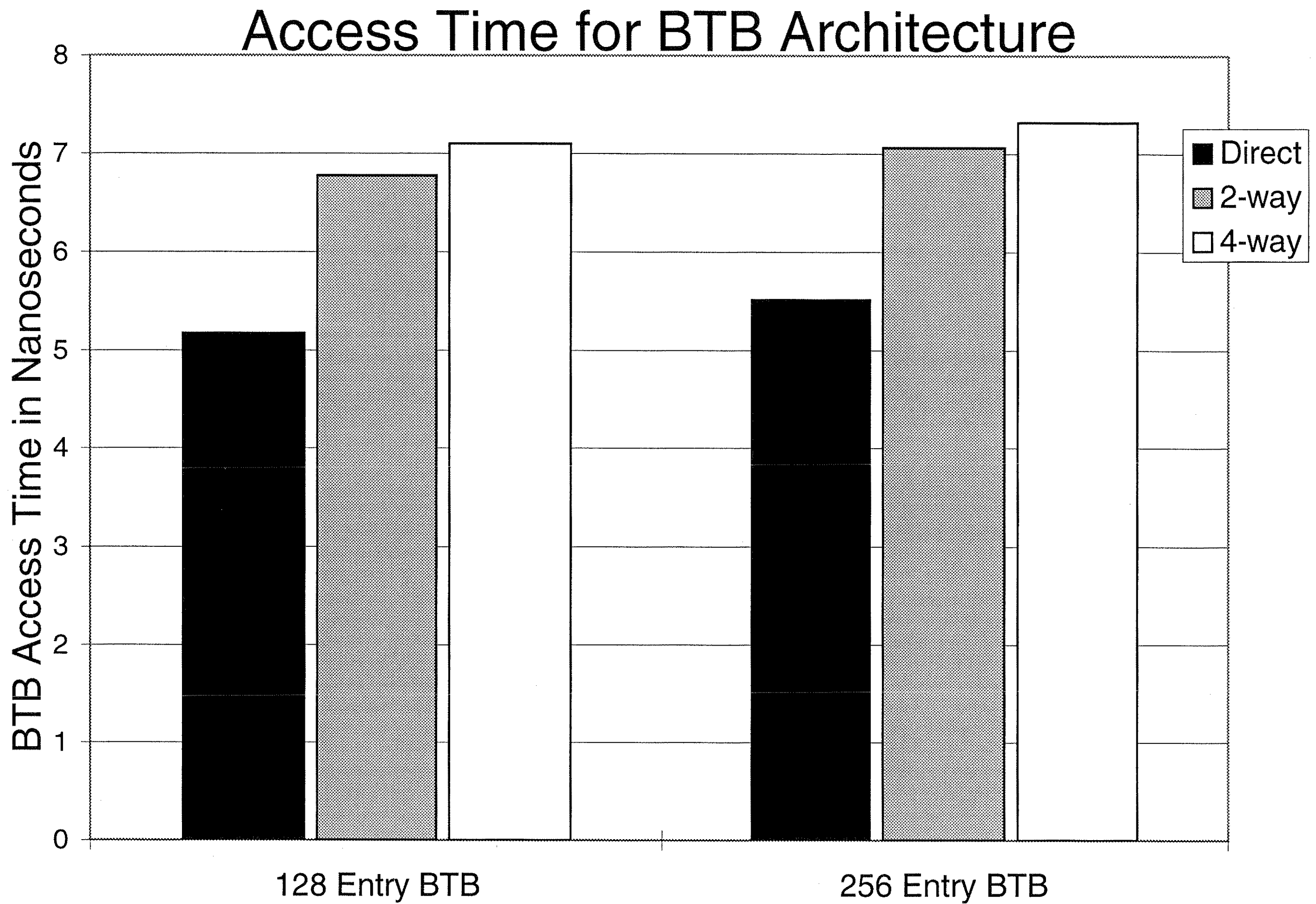


Figure 6.8. Access time for the BTB architecture with varying associativities. The relative values between the BTB access times are more important than the absolute values for a particular processor technology.

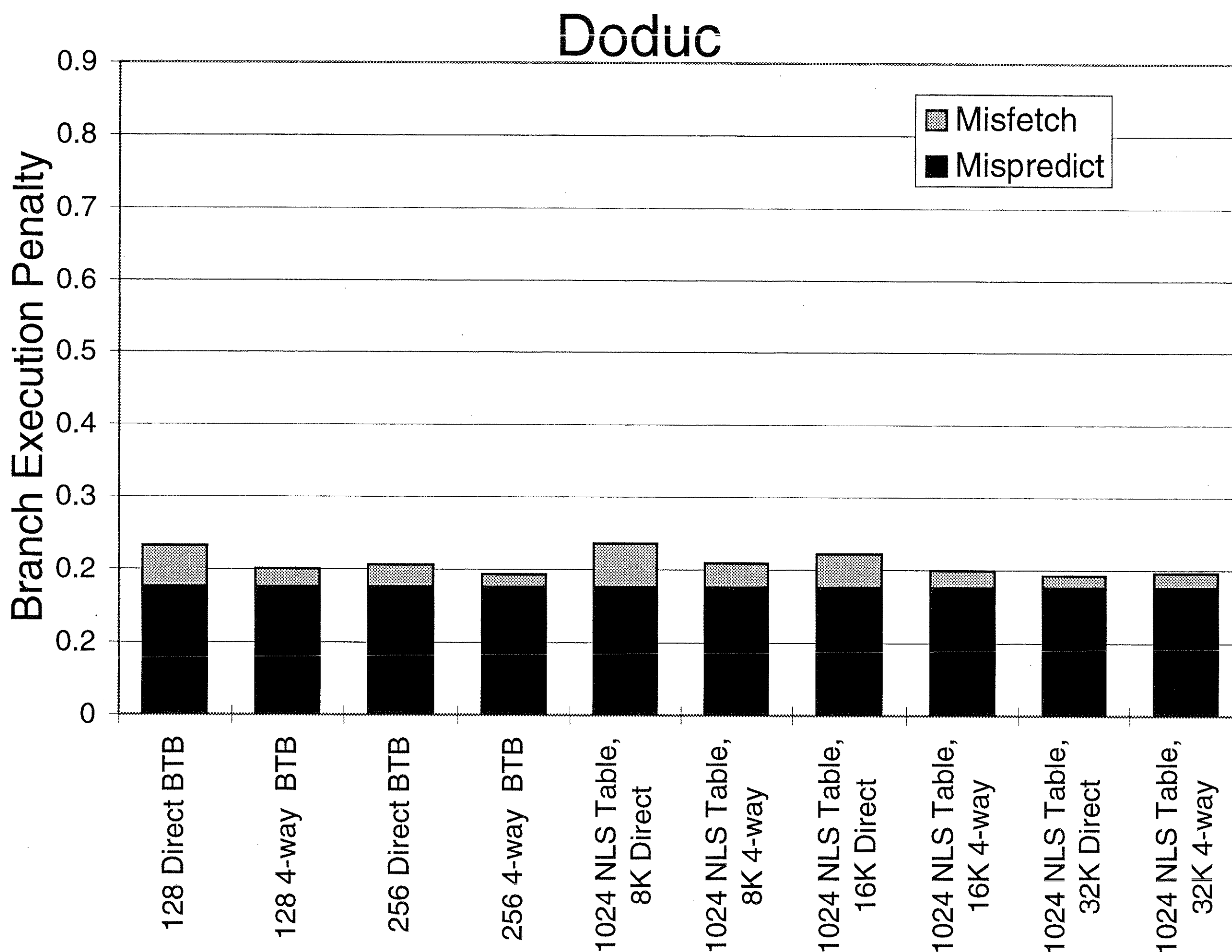


Figure 6.9. Performance comparison between the NLS and BTB architectures using branch execution penalty for Doduc. Each value is broken into two parts. The top represents the fraction of the BEP due to the misfetch penalty and the lower part the fraction due to the mispredict penalty. The NLS results are given for an 8K, 16K and 32K instruction cache with direct mapped and four-way associativity. The BTB results are only shown once, since their results do not change for the different instruction cache configurations. The 1024 entry NLS table and the 128 entry BTB have equivalent implementation costs, and the cost for the 256 entry BTB is approximately twice that of the 1024 NLS-table.

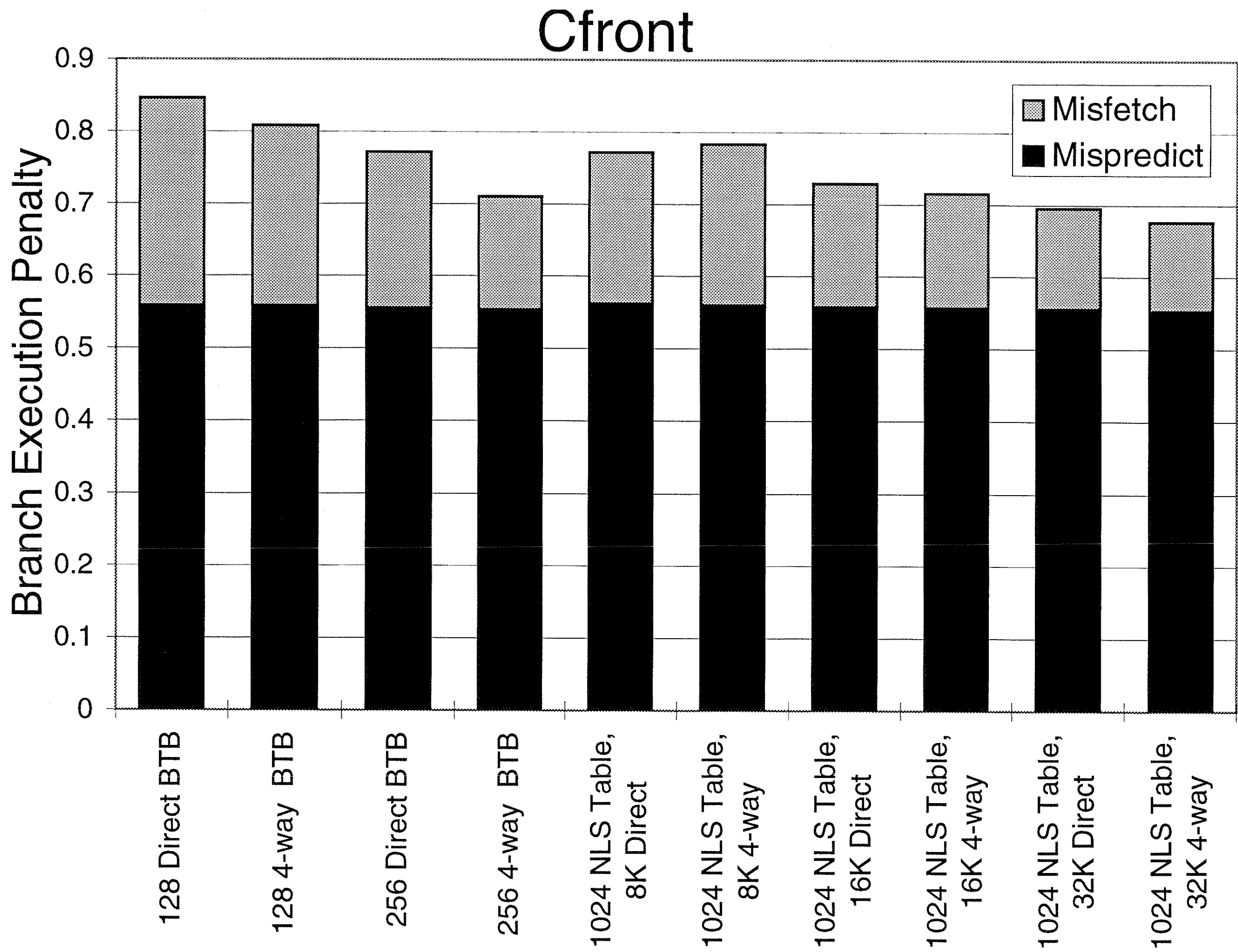


Figure 6.10: NLS-Table and BTB Performance for Cfront.

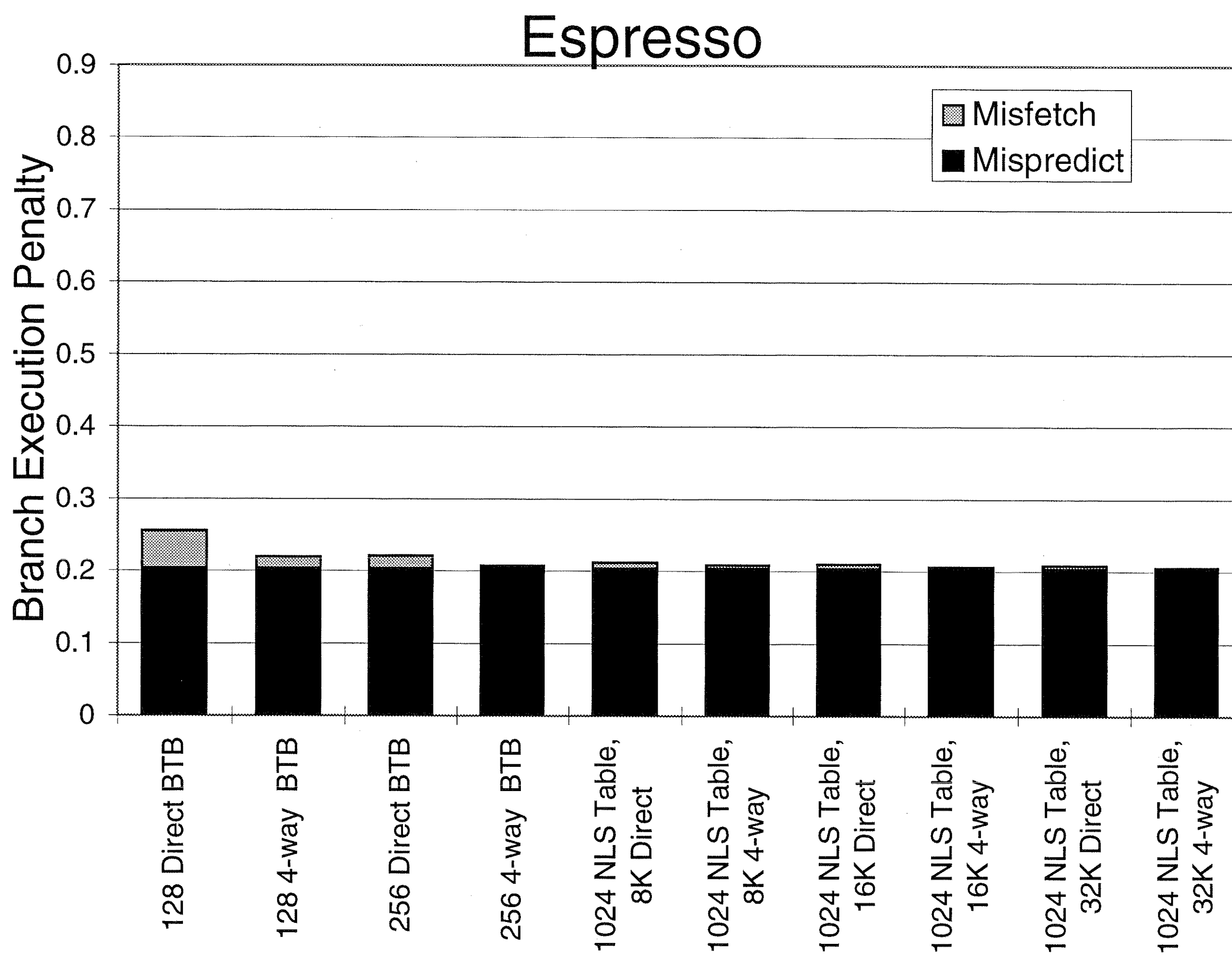


Figure 6.11: NLS-Table and BTB Performance for Espresso.

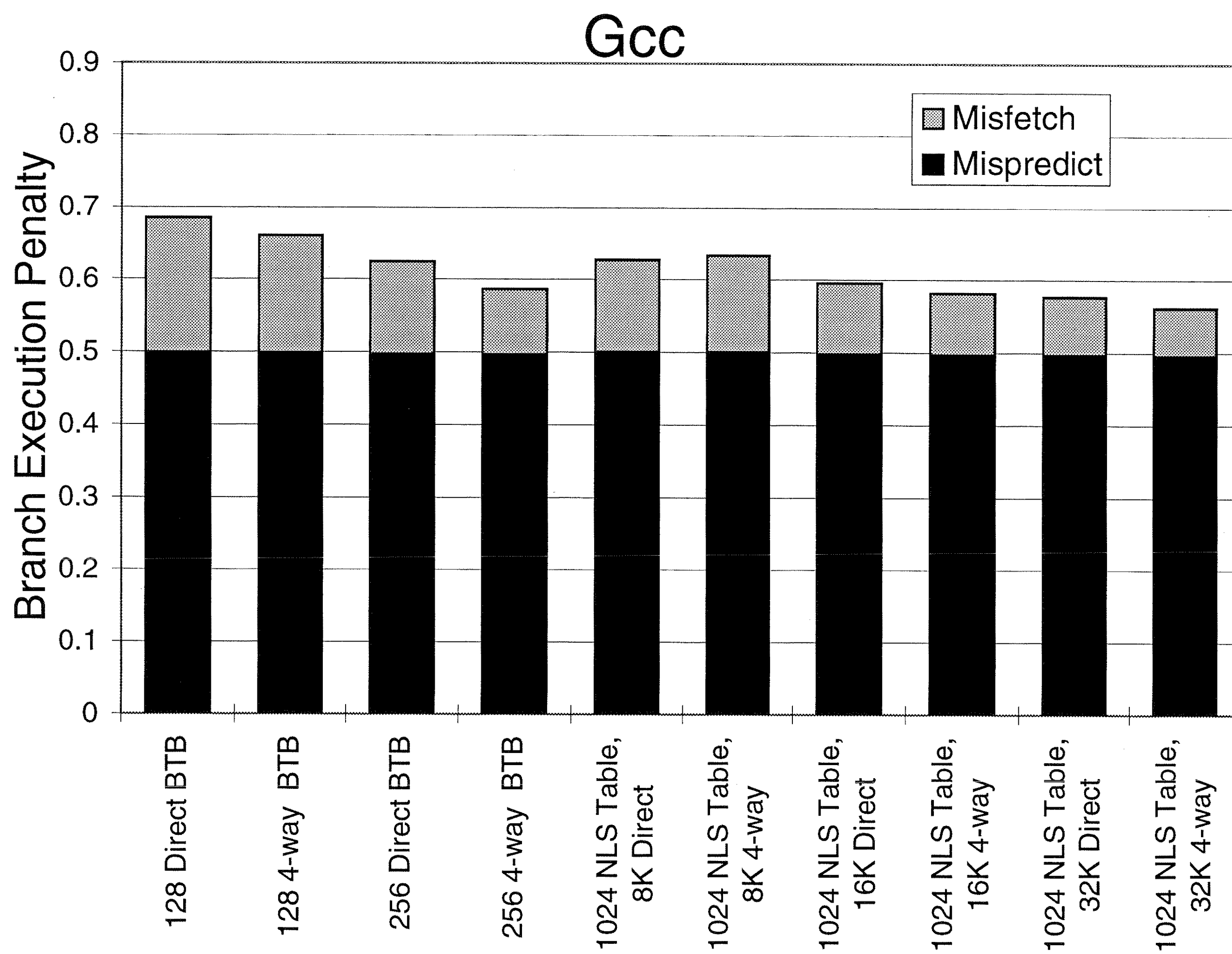


Figure 6.12: NLS-Table and BTB Performance for Gcc.

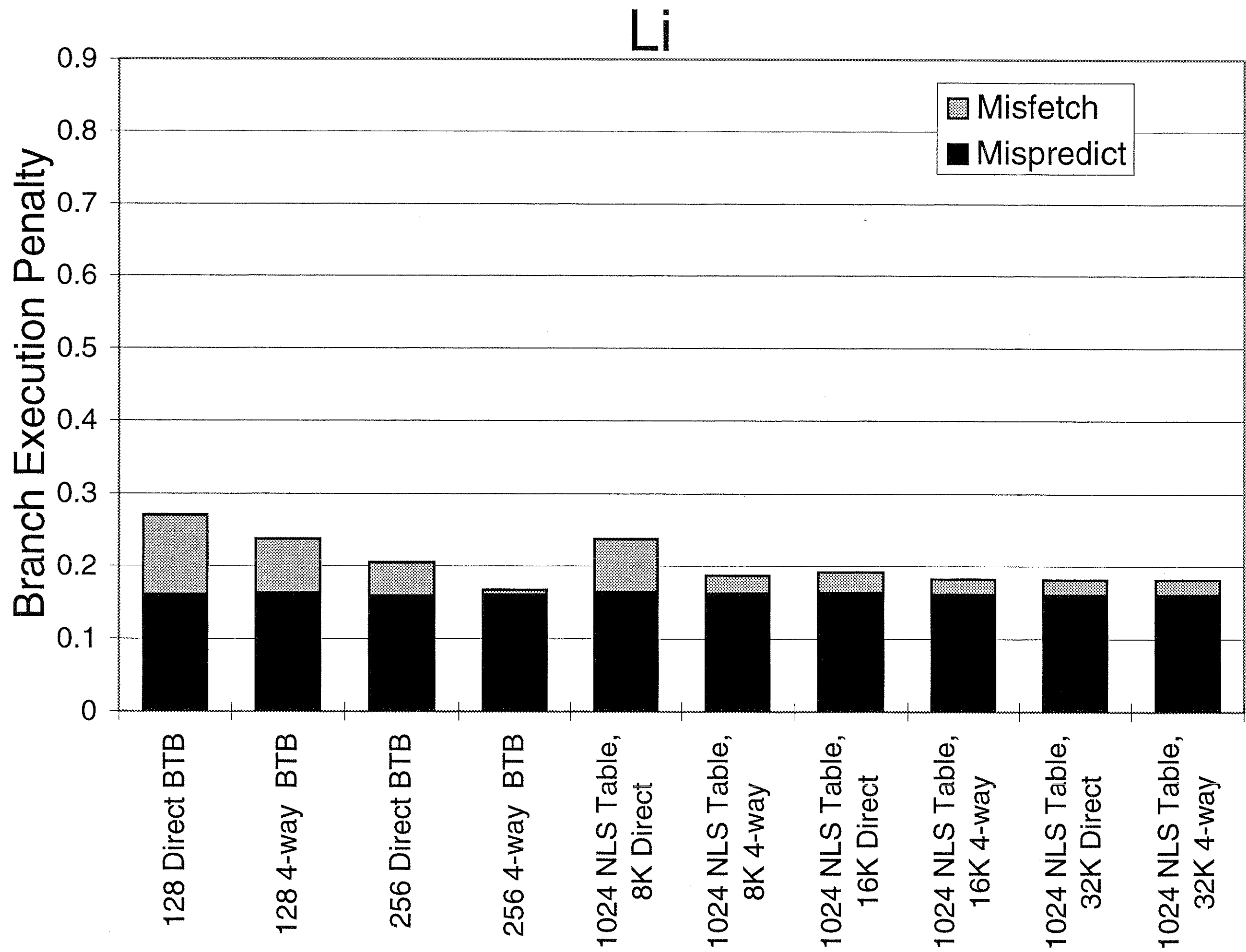


Figure 6.13: NLS-Table and BTB Performance for Li.

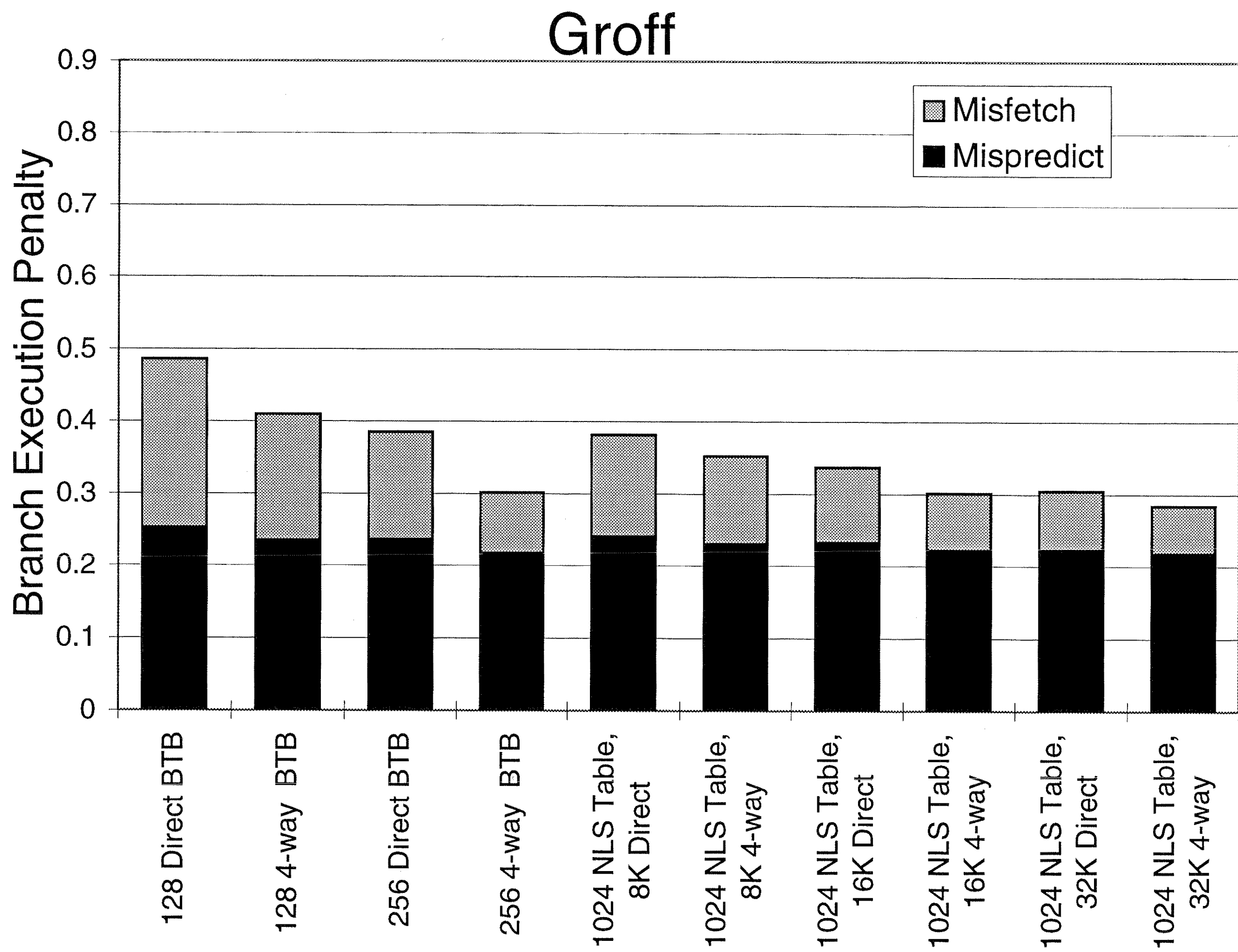


Figure 6.14: NLS-Table and BTB Performance for Groff.

For example, in doduc, three individual branches constitute 50% of the branches encountered during program execution. One need only store those three branches to achieve 50% fetch accuracy. By comparison, gcc, cfront and groff have many more branches encountered during execution. The larger number of branches leads to capacity misses and conflicts in any prediction mechanism using a fixed-size resource. Because each NLS predictor in the NLS architecture is smaller than the comparable BTB entry, the NLS architecture has many more prediction entries using the same resources. Overall, the larger number of less-precise NLS predictors benefits program performance more than the fewer, more precise, BTB entries.

A larger program address space poses several problems for BTB architectures, but is inconsequential for the NLS architecture. As the program address space increases, the tag size used to identify branches in the BTB must either increase, or the information stored in the BTB will become less precise. Likewise, the size of the destination address field must also increase, or the BTB architecture would store only partial target addresses, decreasing the prediction accuracy of the BTB and making the address stored more like a NLS predictor. In our RBE calculations we assumed a 32-bit address space, so the target address stored in the BTB is 30 bits. If the address space was increased, the area needed by the BTB would also increase. By comparison, the NLS-table design does not use a tag nor does it store the full target address, so an increased address space has no effect on the size of the NLS-table. The size of a NLS predictor depends only on the number of lines in the cache and the number of instructions in the line. As the instruction cache size is increased the size of the NLS-table increases logarithmically. In contrast, an increase in cache size has no effect on the size of the BTB.

6.9 Summary

In this chapter, we have presented two alternative NLS architectures, the NLS-cache and NLS-table. Our results show that decoupling the NLS predictors from the instruction cache (NLS-table) performs better than Johnson's [32] approach of associating the NLS predictors with the cache line (NLS-cache), and that the NLS predictors can be combined with highly accurate two-level correlated branch prediction architectures instead of using the NLS predictors for one-bit branch prediction as in Johnson's design. We also found that the NLS-cache is not a scalable design, since the number of NLS predictors increases linearly with the cache size. For the NLS-table design, the results show that there is little benefit from increasing its size past 1024 entries for the programs we examined.

The NLS-table is a tag-less, direct mapped buffer with better instruction fetch prediction than direct-mapped BTBs with similar costs. When comparing the performance of the NLS architecture to associative BTBs, one should keep in mind that the access time for an associative BTB is 30 to 40% longer than the similar sized direct mapped structures. Our results show that the 1024 entry NLS-table performs better than the 128 entry BTB, with similar RBE costs. For a 256 entry BTB, the 1024 NLS-table had comparable performance for approximately half the RBE cost. The NLS-table can offer better performance than the BTB because the cost of a NLS entry is much less than a BTB entry, allowing the NLS-table to contain many more entries than BTB architectures with similar implementation costs. This allows the NLS-table to perform better than the BTB design especially for programs with many branches. For programs with fewer branches, the architectures have comparable performance. The performance of the NLS architecture improves as the instruction cache miss rate is lowered, and its performance can be improved by using whole-program analysis, basic block reordering, and intelligent procedure layout.

CHAPTER 7

PRECOMPUTED BRANCH ARCHITECTURE

A branch target buffer (BTB) is often used to provide target addresses for taken branches and to predict the destination of indirect jumps. The previous chapter described a next cache line and set (NLS) architecture as an alternative design to the BTB, effectively predicting which instructions to fetch from the instruction cache. Using a BTB or NLS architecture avoids the delay needed to calculate the destination address and they effectively reduce the misfetch penalty. However, both of these architectures still require a fair amount of hardware. We propose using a software solution to help solve this problem, thus eliminating the need for the BTB and NLS architectures.

We propose that a design used in older computers, such as the PDP-8, be used in modern architectures. We call this the **Precomputed-Branch** architecture. This architecture precomputes the branch destination for most branch instructions, allowing the branch information to be stored with the instruction. We consider computing branch destinations at link time and as instructions are fetched into the instruction cache; both alternatives offer similar performance with different advantages. A very small branch target buffer is still useful to predict indirect branches, which can not be precomputed. Our results show that the Precomputed-Branch architecture performs better than an architecture using a branch target buffer, and has significant hardware savings. This is particularly true for larger programs with many branches. [1]

7.1 Introduction

Architectures using a BTB or NLS architecture can issue a large number of instructions per cycle because of accurate branch and fetch prediction. However, these architectures can lead to complex designs and can be costly to implement. This chapter shows that we can maintain a low branch execution cost with considerably fewer resources than those needed by architectures using a branch target buffer. Our proposed architecture assumes a flat address space, and eliminates program-counter relative (PC-relative) branches. In this design, the destination address is quickly computed by precomputing the low-order bits of the destination, and concatenating those bits with the high-order bits of the current address, effectively dividing memory into a number of program segments or "branch spaces". Branches between branch spaces are computed and performed as indirect jumps. This design assumes the program linker or compiler can partition the program into a number of segments and modify the program's structure to select between intra-space and inter-space branches [56]. We also describe how existing architectures using PC-relative branches can be extended to benefit from precomputed branches.

The Precomputed-Branch architecture provides area-efficient support for conditional, unconditional and procedural branches, where the branch destination is explicitly specified. Other researchers [33] have described inexpensive mechanisms to predict the destination of procedure returns. The only remaining branches requiring prediction are indirect jumps, where the destination may be specified by values computed during execution. We show that a small branch target buffer, dedicated to predicting only the outcome of indirect jumps, benefits a number of programs, especially object-oriented programs containing a large number of indirect jumps.

7.2 The Design of Two Branch Architectures

We used trace-driven simulation to compare the Precomputed-Branch architecture to a design that uses an efficient branch target buffer. We simulated the decoupled branch target buffer proposed in [7], because this architecture provides similar to better overall branch performance than the coupled models proposed in [66]. In

[1] Parts of this chapter were submitted to the Journal of Computer and Software Engineering [11].

this section, we describe this architecture and follow that with a detailed description of the Precomputed-Branch architecture.

7.2.1 A BTB-based Instruction Fetch Architecture

Figure 7.1 is a schematic representation of a conventional branch prediction and instruction fetch architecture using a branch target buffer. As described earlier, a BTB is used to eliminate misfetch penalties by providing taken target addresses, while a pattern history table (PHT) is used to predict conditional branches. In this design, the global history register is combined with the program counter using an exclusive-or, and the result is used as the index into the PHT [41]. In both the BTB and Precomputed-Branch architecture, we assume the branch type is encoded in the instruction, or pre-decoded and stored in the instruction cache. A 32-entry return address stack is used to predict the outcome of return instructions. The current instruction address is concurrently offered to the instruction cache, providing the actual instruction, and to the BTB. There are three important types of branches: direct or indirect branches, conditional branches and returns. Depending on the branch type and the prefetched branch prediction information from the 2-level PHT, either the BTB target address, the computed fall-through address, or the return stack address is selected as the next instruction fetch. If the branch misses in the BTB, an instruction may be misfetched, but the branch prediction information (PHT and return stack) is used to predict fetch addresses after the decode stage.

When a return instruction is encountered, the branch type indicates that the instruction is a return and the top of the return stack is used to fetch the next instruction. The first time a particular unconditional branch is entered into the BTB, the BTB entry records the computed target address. When the branch is encountered again, the branch type indicates this is an unconditional branch, and selects the target address in the BTB. Conditional branches have similar actions; however, the prediction information from the PHT is used to predict the likely outcome of conditional branches. Depending on the predicted outcome, the stored destination (which is always the 'taken' address) or the fall-through address is used to fetch the next instruction.

7.2.2 The Precomputed-Branch Architecture

The BTB serves two roles. Only taken branches are entered in the BTB, so a BTB hit indicates the instruction is a branch, and the BTB provides the precomputed target address. As mentioned in Chapter 6, we assume that the instruction cache or the instruction itself can identify the branch type. This can either be made explicit in the instruction encoding, or the instruction can be partially decoded when it is brought into the instruction cache; a similar mechanism has been used in several architectures, such as the MIPS R10000 [44]. The only remaining function provided by the BTB is the precomputed destination address for taken branches. The BTB is needed because the destination address specified by a branch instruction can not be fetched from the instruction cache and computed all in a single cycle when using PC-relative destinations.

Figure 7.2 shows our proposed instruction fetch architecture. A program is broken into multiple **branch spaces**. Branches within a single branch space can use a normal branch instruction, while branches between branch spaces must be computed as an indirect branch. The displacement indicates the branch target displacement within the current branch space. The lower-order bits of the branch destination are simply concatenated with the higher-order bits of the current PC address, and no addition is needed. Issues surrounding branch spaces, and the complications that arise, are discussed later.

The concatenating of the Precomputed-Branch eliminates misfetch penalties for conditional branches, unconditional branches and direct procedure calls. This leaves indirect branches as the only branch type without a precomputed branch address. Thus, we use a small BTB to predict indirect branches; otherwise they will always be mispredicted. Since this branch target buffer is only used for indirect jumps, we call this an indirect jump buffer (IJB) to clarify the distinction between the BTB architecture and the Precomputed-Branch architecture using an IJB. Alternatively, we can use profile-based prediction of indirect function calls, which has been shown to be effective for the C++ programming language [9], where such branches occur frequently.

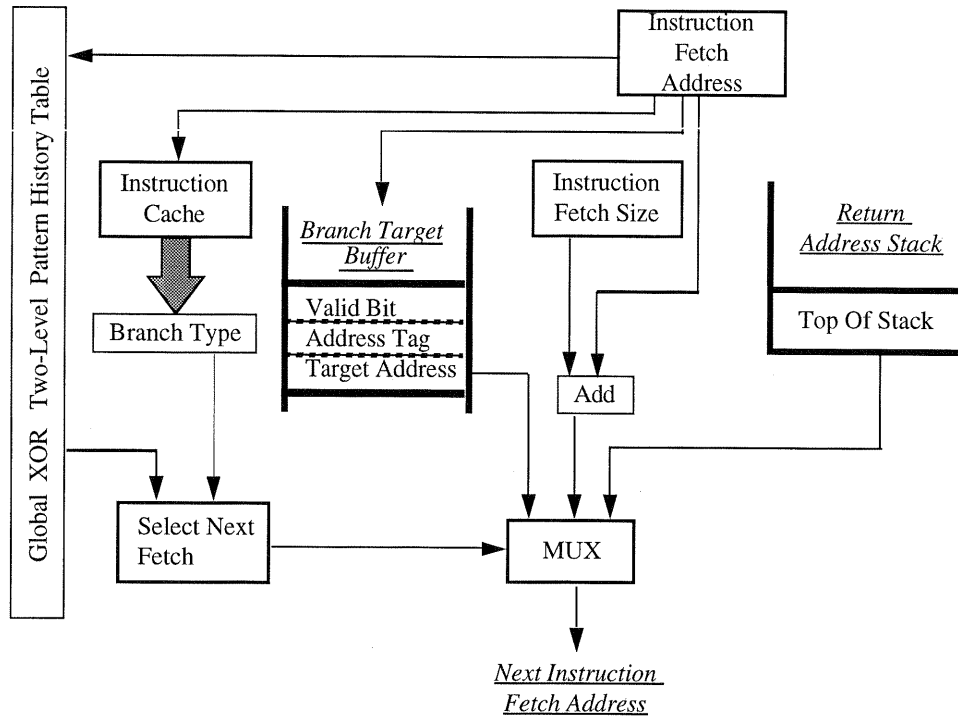


Figure 7.1: A Schematic Representation of the Branch Target Buffer Architecture.

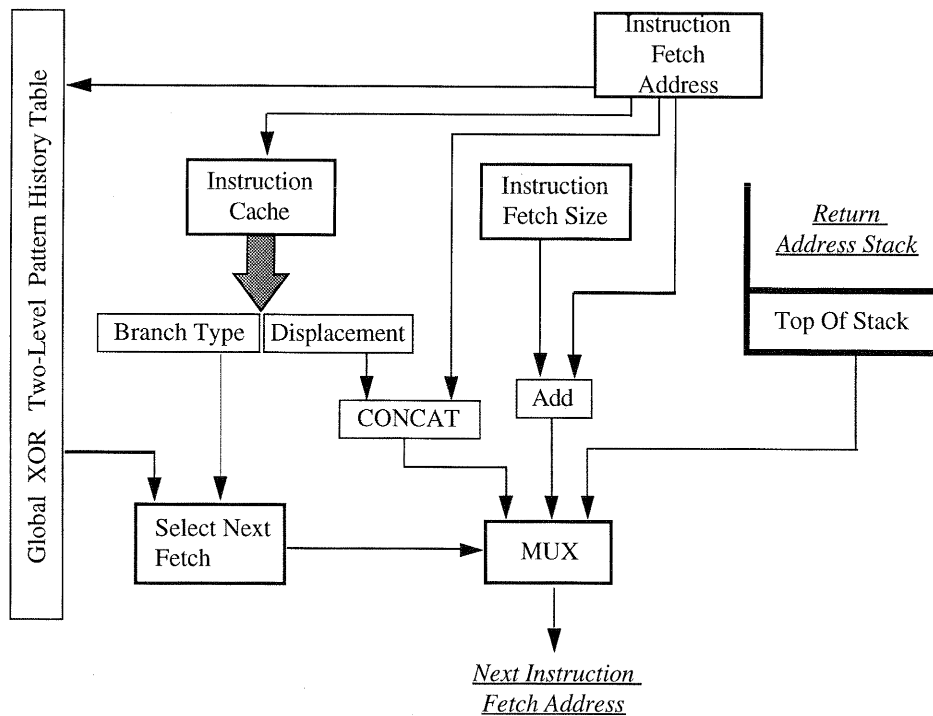


Figure 7.2: A Schematic Representation of the Precomputed-Branch Architecture.

7.2.3 Computing the Branch Target

Traditional branch architectures use a PC-relative displacement; Figure 7.3(a), modeled after the diagrams in [34], schematically illustrates the process. In the encodings, information in lightly outlined boxes is provided or computed at execution time; for example, in Figure 7.3(a), the PC is available during execution. Heavily-outlined boxes show the information provided by the branch instruction – in Figure 7.3(a), the instruction provides $n + 1$ bits. On the right-hand side, the solid boxes show the range of instructions that can be addressed. A displacement stored in the branch instruction is sign-extended to the size of the program counter and added to the program counter. Each branch can directly address instructions at address $PC - 2^{n-1} - 1 \ldots PC + 2^{n-1}$. For simplicity, we assume the program counter is always aligned on instruction boundaries, since we are chiefly concerned with architectures with fixed-width instructions.

Katevenis [34] proposed several branch encodings where the branch displacement field contains the least significant bits of the branch target address. Figure 7.3(b), shows one such encoding. Here, the sign bit for the offset and the carry for the addition of the lower bits are computed by the compiler (or linker) and encoded in the instruction. The lower bits can be immediately used to index the cache. Concurrent with the cache fetch, the higher order bits are computed and matched against the address tags when the cache fetch returns. If the tags are mismatched with the actual PC, an instruction-cache miss occurs and the pipeline is stalled. During the stall, the program counter is corrected. Since the instruction must include both the carry and the sign bit, an n-bit displacement can only address $PC - 2^{n-2} - 1 \ldots PC + 2^{n-2}$.

Figure 7.3(c) diagrams our proposed branch encoding. We use an **explicit displacement** instead of a PC-relative displacement because we need to calculate target addresses in time to use them for the next instruction fetch and, as Katevenis noted, an adder is usually too complex for this purpose. The n-bit displacement is used as the lower order part of the destination address, and is concatenated with the higher order bits of the current PC to form the new fetch address. Each branch can then jump within a **branch space** of 2^n instructions. Every direct branch within a 2^n branch space can only branch within that space. To branch outside that span, an indirect jump must be used.

7.2.4 Other Non-Relative Branch Architectures

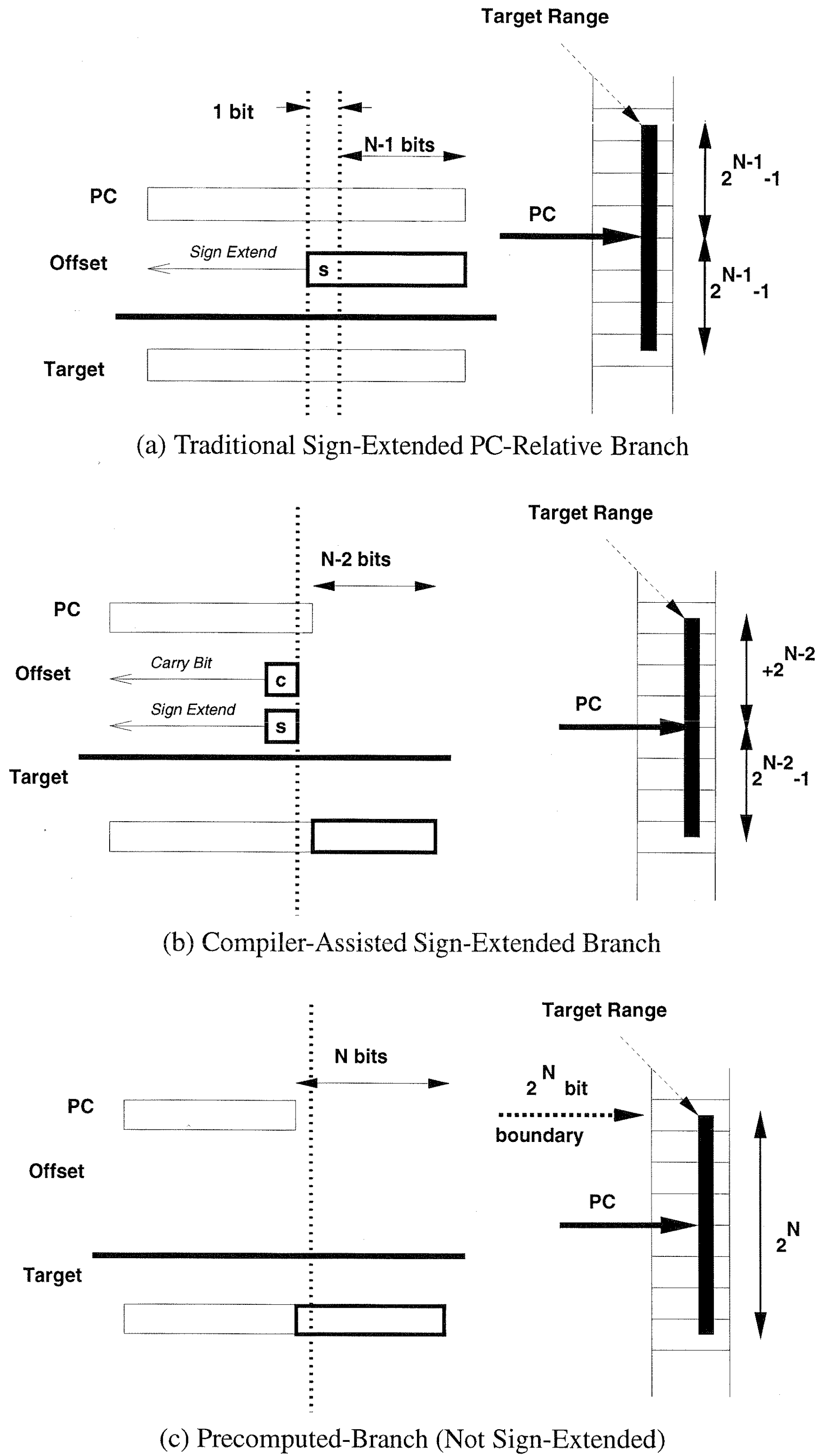
Using non-relative branches is not a new idea, although we are unaware of studies that emphasize the branch layout algorithms, the modern branch prediction architectures, or the large diverse set of applications examined in this chapter.

The memory for the PDP-8 was divided into eight "memory fields," each field was divided into 32 pages of 64 locations (words). A `JMP` or `JMS` instruction could jump within a single memory page (e.g., to one of 64 words), or could specify an indirect reference to a word containing a 12-bit destination. Branches within the same page were precomputed, and all other branches required an indirect reference. In the Crisp processor [23], a branch destination is included in every decoded instruction in the instruction cache, resulting in very large instructions – 192 bits. This technique consumes a considerable amount of space and may limit the processor cycle time. A mechanism similar to that used in the PDP-8 was used in Control Data and IBM processors, where instructions were optimized to execute within an instruction buffer – Lee and Smith [38] provide a good survey.

By comparison, we rely on the program linker to compensate for the limited branching by precomputing branch destinations and reducing the number of complex operations (indirect branches). This applies the "RISC design philosophy" to branch architectures by letting the software (compiler and linker) share the burden of making the hardware efficient and inexpensive. The advantage of the Precomputed-Branch architecture is that it removes all pipeline stalls caused by misfetched branches by providing the precomputed target addresses, so it achieves a lower BEP than the BTB architecture and it requires significantly less area.

7.3 Methodology

Table 7.1 shows the static number of instructions and number of procedures found in the programs we instrumented. The "static" information is a property of the program binary, and is the same for all executions.



(a) Traditional Sign-Extended PC-Relative Branch

(b) Compiler-Assisted Sign-Extended Branch

(c) Precomputed-Branch (Not Sign-Extended)

Figure 7.3: Alternative Branch Methods

These static statistics are of interest when discussing the results for partitioning these programs into branch spaces for the Precomputed-Branch architecture.

We used a modified version of the branch execution penalty to understand the performance improvement of various branch architectures. In the Precomputed-Branch architecture, some branches are changed to indirect branches. We assume this is done by loading the value from memory and performing an indirect jump. A single-cycle penalty for loading the branch destination is included in the BEP for the Precomputed-Branch architecture. Thus, we extend the BEP model to be:

$$\text{BEP} = \frac{\%\text{MfB} \times \text{misfetch penalty} + \%\text{MpB} \times \text{mispredict penalty} + \%\text{EiB} \times \text{indirect br penalty}}{100},$$

where $\%EiB$ is the percentage of branches converted to indirect jumps, expressed as an average cost over all branches. The %EiB is determined when the program is partitioned into branch spaces, as described in the next section. For these simulations we used a single cycle misfetch penalty, a one cycle additional penalty for all branches converted to indirect jumps, and a four cycle mispredict penalty.

7.4 Partitioning Programs Into Branch Spaces

In this section, we show the overhead introduced by the explicit-displacement architecture when the instruction set only permits small displacements. We use static methods to partition the program into multiple branch spaces, and then compare the benefits of partitioning using information from prior execution. In practice, most architectures provide branch instructions intended to be used within a single procedure and different branch instructions used to transfer control to other procedures. Often, the displacements in these instructions are different sizes; for example, the MIPS architecture uses 26-bit displacements for procedure calls, and 16-bit displacements for conditional branches. By comparison, the DEC Alpha uses 21-bit displacements for all branches.

When using PC-relative addresses, procedures can be placed anywhere in virtual memory. By comparison, the explicit displacement architecture places more restrictions on procedure placement. In a Precomputed-Branch architecture, a branch located at address X is located in a particular Z-bit branch space specified by $S(X, Z) = \lfloor \frac{X}{2^Z} \rfloor$. Branches can only reach destinations in the same branch space; thus, to branch from X to Y, we must insure that $S(X, Z) = S(Y, Z)$ in order to use a precomputed branch. If $S(X, Z) \neq S(Y, Z)$ then an indirect jump must be used to branch between spaces.

Assume that conditional branches have a B-bit displacement, while function calls have a C-bit displacement. Branches within a procedure and between procedures must be able to reach their destinations. We reorganize the program trying to insure that all branches and their destination are in the same branch space. If this restriction can not be enforced, the branch is converted into an indirect jump that can span a larger branch space. Typically, several procedures can be organized into a single B-bit branch space.

In this chapter, we partition programs into branch spaces of three different sizes: 14, 16 and 21-bit branch spaces. We choose the 16 and 21-bit branch spaces because they reflect the branch displacement in existing microprocessors, and it is easy to argue that branch spaces of this size are easy to implement. However, all of our sample programs fit within a single 21-bit branch space, and most of the programs fit in two 16-bit branch spaces. Therefore, we also partitioned programs into a 14-bit branch space to give some insight of the overheads that might be encountered by larger programs. We further stipulated that procedures are not spilt across branch spaces; only procedure calls will span branch spaces.

There are many ways to partition programs, and a number of alternatives have been examined in the effort to reduce page faults and and instruction cache conflicts [1, 3, 27]. Ferrari extended this work to include information about working sets [24]. The goals of our study are different than these other studies; we are less interested in efficient use of virtual memory than in reducing the number of indirect jumps that must be introduced.

We used two metrics to compare the program partitioning heuristics. The first is the additional amount of space needed by the program due to wasted space at the end of virtual memory pages. The second metric

Table 7.1: Number of Static Instructions and Procedures in Traced Programs.

	Static	
Program	# Insn's	# Procs
APS	128,694	797
CSS	141,751	818
LGS	90,385	726
LWS	88,661	714
NAS	103,401	740
OCS	90,122	717
SDS	94,615	768
TFS	94,383	715
TIS	74,681	681
WSS	106,227	757
doduc	94,402	708
fpppp	83,999	685
hydro2d	85,808	716
mdljsp2	84,286	733
nasa7	83,867	706
ora	70,604	668
spice	138,312	815
su2cor	93,668	711
swm256	73,412	677
tomcatv	65,625	617
wave5	106,978	762
alvinn	17,811	212
compress	13,144	149
ear	25,079	290
eqntott	19,172	212
espresso	60,674	551
gcc	186,066	1,651
li	33,235	551
sc	59,291	512
cfront	225,064	981
db++	20,784	329
groff	121,191	1,756
idl	79,381	1,459
lic	384,058	5,333
porky	216,678	3,704

is the percent of dynamic branches that cross branch spaces. We felt the number of dynamic branches was more important than the additional space created by the partition, because it has a direct measurable impact on program execution, since any additional memory needs could readily be solved by adding extra memory.

We considered a number of partitioning algorithms. Some methods use profiles, or information about previous executions of the program. Many optimizations require such information, either from previous executions of the program or from estimates using static analysis [14, 60, 64]. We examined depth-first, breadth-first, pre-order, post-order, greedy and max-cut partitioning algorithms. Most of the methods had similar performance, so we present the performance of only three algorithms.

The partitioning algorithms are illustrated in Figures 7.4 and 7.5. In this example, we assume that each branch space can hold three procedures. Branch spaces are indicated by the darkened regions. The "Separate" method, shown in Figure 7.4(a), partitions each procedure into a different branch space, and illustrates the worst-case performance we could encounter. In the "Preorder" partitioning, shown in Figure 7.4(b), nodes are added to a list in a pre-order traversal without using profile information. That list is then partitioned into branch spaces using the size constraints of the architecture. A similar technique is used in the "Depth-First Profile" method shown in Figure 7.5(a); a depth-first search orders the nodes, always first visiting the out-going edge with the highest call frequency. The "MaxCut" partitioning, shown in Figure 7.5(b), uses a conventional max-cut algorithm to partition the graph using the call frequency to guide the partitioning. The costs shown in these figures reflect the number of branches that cross branch spaces for a particular execution of the program. There are a total of 39 procedure calls in the example, and each partitioning algorithm attempts to reduce the number of procedure calls that span branch spaces.

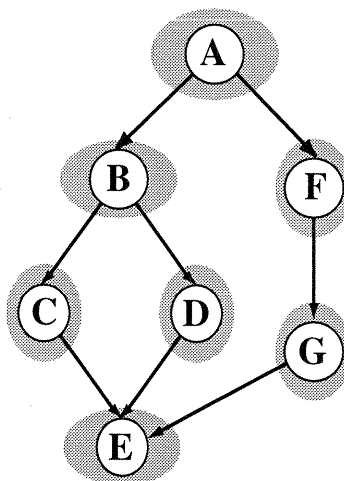
7.4.1 Performance of Program Partitioning Algorithms

All of our programs use a single branch space for 21-bit displacements, illustrating no difference between the performance of the different partitioning algorithms. Table 7.2 shows the performance of the partitioning heuristics into a 14-bit branch space and Table 7.3 shows the performance of the partitioning heuristics into a 16-bit branch space.

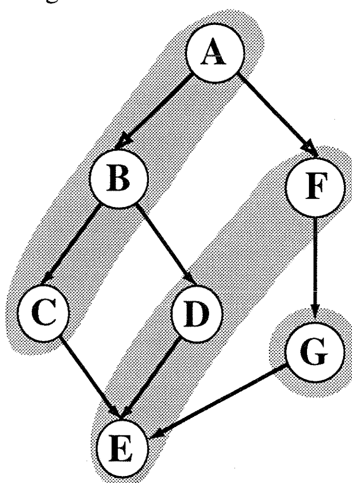
Each table is broken into four major columns, showing the performance of each method. For each method, the first sub-column (Seg) shows the number of branch spaces introduced by the partitioning, and the number of additional program pages needed by the partitioned program over the original program. We assumed an 8KByte page size. The next sub-column (%EiB) shows the increase in indirect function calls introduced by this partitioning. This term is used when computing the branch execution penalty. Table 7.2 shows the average for all the programs partitioned using the "Preorder" method with a 14-bit branch space. For the Preorder algorithm, on average six branch spaces are used, one additional 8KByte memory page is needed and 2.29% of the branches are converted to indirect jumps. The %EiB value is averaged over all branches to simplify the calculation of the BEP, and provides insight to the overhead of partitioning on total branch execution. For the profiled-based partition algorithms, the same program input was used to profile the program and to asses the performance of the partitioning methods; the performance of other executions may differ.

The summary in Table 7.3 shows that partitioning programs into 16-bit branch spaces has little effect on program size or the number of indirect jumps. While the profile information improves the partitioning, it provides only a small improvement. Similar results hold for the 14-bit branch spaces, although the profile approaches are more important for the smaller branch spaces. As mentioned above, all of our sample programs fit into a single 21-bit branch space when using the Precomputed-Branch architecture. Another interesting result shown in Tables 7.2 and 7.3 is that the C++ programs execute up to 7 times the number of static procedures than executed in the C or FORTRAN programs. This causes a large increase in the number of branches converted to indirect jumps for the C++ programs when performing the program partitioning. Many of these procedure calls could be removed by performing inlining and object-oriented compiler optimizations such as If-conversion [9], as described in Chapter 2.

When comparing branch target buffer architectures to the Precomputed-Branch architecture, we will only consider the "Preorder" and "MaxCut" partitioning, and show the results for 14, 16 and 21-bit branch

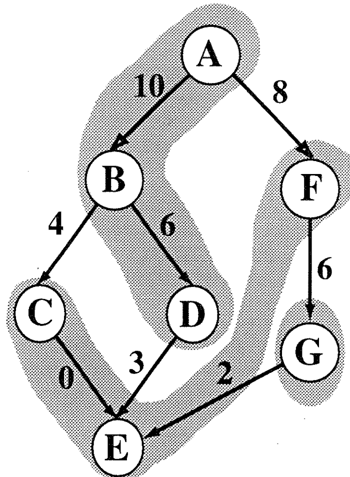


(a) Separate - Each procedure forms a branch space, and no profile information is used. The execution cost is 39, using the same weights as used in Figure 7.5.

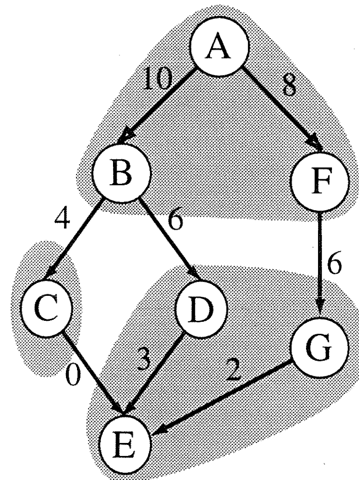


(b) Preorder - The traversal order is {A, B, C, E, D, F, G}, and no profile information is used. The execution cost is 22, using the same weights as used in Figure 7.5.

Figure 7.4. Partitioning a Call Graph without Profiles. In this example we assume that at most three procedures can fit into one branch space. In these graphs the nodes labeled with letters are procedures, the edges are procedure calls, and the shaded areas are the branch spaces the procedures are partitioned into for each algorithm. The cost shown for each partitioning is the number of procedure calls converted to indirect jumps.



(a) Depth-First Profile - The traversal order is {A, B, D, C, E, F, G}. The execution cost is 21, using the weights shown.



(b) MaxCut - Order examined is {A, B, F, D, G, E, C}. The execution cost is 16, using the weights shown.

Figure 7.5. Partitioning a Call Graph with Profiles. In this example we assume that at most three procedures can fit into one branch space. In these graphs the nodes labeled with letters are procedures, the edges are procedure calls, the numbers indicate the number of procedure calls, and the shaded areas are the branch spaces the procedures are partitioned into for each algorithm. The cost shown for each partitioning is the number of procedure calls converted to indirect jumps.

Table 7.2: Efficiency of Program Partitioning with 14-bit Branch Displacements

	Seperate		Preorder		Prof Depth		Max Cut	
Program	Seg	%EiB	Seg	%EiB	Seg	%EiB	Seg	%EiB
APS	797	4.63	9/2	3.38	9/1	0.19	8/0	0.19
CSS	818	5.24	9/2	2.40	9/2	2.47	9/0	1.37
LGS	726	10.14	6/0	6.44	6/1	1.36	6/0	1.68
LWS	714	7.03	6/1	3.14	6/1	0.00	6/0	0.00
NAS	740	10.99	7/1	9.47	7/1	0.43	7/0	0.43
OCS	717	0.44	6/1	0.44	6/0	0.06	6/0	0.00
SDS	768	0.37	6/1	0.02	6/0	0.02	6/0	0.14
TFS	715	2.43	6/0	2.30	6/1	0.28	6/0	0.21
TIS	681	0.00	5/1	0.00	5/1	0.00	5/0	0.00
WSS	757	2.09	7/1	0.97	7/0	0.54	7/0	0.67
doduc	708	6.86	6/1	0.21	6/1	1.06	6/0	1.02
fpppp	685	2.66	6/1	0.22	6/3	0.70	6/0	0.00
hydro2d	716	1.38	6/1	0.00	6/1	0.00	6/0	0.00
mdljsp2	733	0.29	6/2	0.27	6/1	0.00	6/0	0.00
nasa7	706	5.95	6/0	0.22	6/2	1.06	6/0	1.08
ora	668	9.75	5/1	0.40	5/2	0.00	5/0	0.00
spice	815	2.28	9/2	1.73	9/2	0.82	9/0	0.33
su2cor	711	6.92	6/0	4.65	7/2	0.02	6/0	0.02
swm256	677	0.08	5/0	0.08	5/1	0.00	5/0	0.00
tomcatv	617	0.03	5/1	0.01	5/1	0.00	5/0	0.00
wave5	762	7.34	7/1	3.19	7/1	7.12	7/0	1.68
alvinn	212	0.64	2/0	0.00	2/0	0.00	2/0	0.00
compress	149	1.95	1/0	0.00	1/0	0.00	1/0	0.00
ear	290	17.42	2/0	0.05	2/0	0.00	2/0	0.00
eqntott	212	0.69	2/0	0.16	2/0	0.00	2/0	0.00
espresso	551	2.26	4/1	1.22	4/0	0.49	4/0	0.11
gcc	1651	4.90	12/2	1.72	12/2	1.67	12/0	1.92
li	551	12.92	3/0	0.84	3/0	0.84	3/0	0.90
sc	512	4.55	4/1	0.75	4/0	0.92	4/0	0.03
cfront	981	7.69	16/13	4.30	15/7	3.70	14/0	4.02
db++	329	6.75	2/0	0.00	2/0	0.00	2/0	0.00
groff	1756	9.00	8/4	3.91	8/2	3.19	8/0	2.03
idl	1459	9.07	5/1	7.72	5/1	7.66	5/0	7.50
lic	5333	12.52	25/5	6.12	24/2	4.44	24/0	3.27
porky	3704	17.74	14/2	13.78	14/2	4.12	14/0	3.30
Overall Avg	940	5.57	6/1	2.29	6/1	1.23	6/0	0.91

Table 7.3: Efficiency of Program Partitioning with 16-bit Branch Displacements

	Seperate		Preorder		Prof Depth		Max Cut	
Program	Seg	%EiB	Seg	%EiB	Seg	%EiB	Seg	%EiB
APS	797	4.63	2/0	2.73	2/0	0.51	2/0	0.00
CSS	818	5.24	3/0	1.76	3/1	2.83	3/0	0.00
LGS	726	10.14	2/0	0.00	2/0	0.00	2/0	0.00
LWS	714	7.03	2/0	0.00	2/0	0.00	2/0	0.00
NAS	740	10.99	2/0	0.00	2/1	0.00	2/0	0.00
OCS	717	0.44	2/0	0.00	2/0	0.00	2/0	0.00
SDS	768	0.37	2/0	0.00	2/0	0.00	2/0	0.00
TFS	715	2.43	2/0	0.00	2/0	0.00	2/0	0.00
TIS	681	0.00	2/1	0.00	2/1	0.00	2/0	0.00
WSS	757	2.09	2/0	0.44	2/0	0.22	2/0	0.00
doduc	708	6.86	2/0	0.00	2/0	0.00	2/0	0.00
fpppp	685	2.66	2/0	0.00	2/0	0.00	2/0	0.00
hydro2d	716	1.38	2/0	0.00	2/0	0.00	2/0	0.00
mdljsp2	733	0.29	2/0	0.00	2/0	0.00	2/0	0.00
nasa7	706	5.95	2/0	0.00	2/0	0.00	2/0	0.00
ora	668	9.75	2/1	0.00	2/1	0.00	2/0	0.00
spice	815	2.28	3/1	1.69	3/0	0.82	3/0	0.00
su2cor	711	6.92	2/0	0.00	2/0	0.00	2/0	0.00
swm256	677	0.08	2/0	0.00	2/0	0.00	2/0	0.00
tomcatv	617	0.03	2/0	0.00	2/0	0.00	2/0	0.00
wave5	762	7.34	2/0	0.02	2/1	2.93	2/0	0.00
alvinn	212	0.64	1/0	0.00	1/0	0.00	1/0	0.00
compress	149	1.95	1/0	0.00	1/0	0.00	1/0	0.00
ear	290	17.42	1/0	0.00	1/0	0.00	1/0	0.00
eqntott	212	0.69	1/0	0.00	1/0	0.00	1/0	0.00
espresso	551	2.26	1/0	0.00	1/0	0.00	1/0	0.00
gcc	1651	4.90	3/0	1.05	3/0	0.89	3/0	0.97
li	551	12.92	1/0	0.00	1/0	0.00	1/0	0.00
sc	512	4.55	1/0	0.00	1/0	0.00	1/0	0.00
cfront	981	7.69	4/0	2.31	4/0	1.99	4/0	1.06
db++	329	6.75	1/0	0.00	1/0	0.00	1/0	0.00
groff	1756	9.00	2/0	1.40	2/1	0.87	2/0	0.84
idl	1459	9.07	2/0	5.04	2/0	5.04	2/0	5.04
lic	5333	12.52	6/0	2.87	6/0	0.37	6/0	0.29
porky	3704	17.74	4/0	2.44	4/0	3.00	4/0	2.75
Overall Avg	940	5.57	1/0	0.62	1/0	0.56	1/0	0.31

displacements. The "Preorder" partitioning provides the lowest branch cost for those methods we examined since it did not use profile information, while the "MaxCut" method provided the best profile-driven partition.

7.5 Precomputed-Branch Architecture Performance

In this section we compare the performance of the branch target buffer (BTB) architecture to the Precomputed-Branch architecture. We use the branch execution penalty to compare performance, but also report the misfetch and misprediction penalty.

Both architectures used a 4096-entry 2-level pattern history table, using the extension proposed by McFarling [41]. In the BTB architecture, we simulated branch target buffers with 4, 16, 32, 64, 128, 256, 512 and an infinite number of entries. In each case, the BTB was organized as a 4-way associative BTB. The Precomputed-Branch architecture stores the precomputed branch destination as part of the instruction for all branches except indirect jumps and returns, with no additional overhead in the instruction cache. We used a small 4-way associative indirect jump buffer (IJB), containing either 0, 4, 16 or 32 entries, to predict the destination of indirect branches. We also simulated a direct-mapped IJB for the same configurations.

7.5.1 Comparing the BTB and Precomputed-Branch Architectures

Figure 7.6 summarizes the branch execution penalty for the different partitioning methods, branch space sizes and architectures. The "Preorder-14" column uses the "Preorder" partitioning heuristic with 14-bit branch spaces. The "MaxCut-14", "Preorder-16" and "MaxCut-16" uses the obvious partitioning and branch sizes. All these partitions are the same as shown in the previous section. The "Optimal-21" partitioning uses a 21-bit branch space, and with this branch space size each program in our benchmark suite fit into a single branch space. Therefore, for the Optimal-21 partition there is no difference between the "Preorder" and "MaxCut" partitioning. The Precomputed-Branch architecture has no misfetch penalty because the destination for all direct branches is immediately known. The "BTB Misfetch=1" column shows the performance for the decoupled BTB architecture with a single cycle misfetch penalty. A larger misfetch penalty would increase the BEP for the BTB-based architecture, but not the Precomputed-Branch architecture.

Table 7.4 provides more details about the terms used to compute the BEP, showing the average percent of misfetched and mispredicted branches. For the Precomputed-Branch architecture, we also show %EiB which is the cost of converting procedure calls that span branch spaces into indirect branches. We assume that all instructions can either be easily decoded or have been pre-decoded to indicate the branch type in both the BTB and Precomputed-Branch architecture as described in Chapter 6. Recall, that a misfetch branch occurs because we either do not know that an instruction is a branch, or we do not know the destination address for a branch. Therefore, the BTB architecture can cause misfetch penalties when the computed destination has not been entered in the BTB. In comparison, the Precomputed-Branch never misfetches because the destination is stored in the instruction. Table 7.4 shows that the Precomputed-Branch architecture, using a non-profile Preorder partitioning for a 16-bit branch space and only a 16 entry IJB, has the same BEP performance (0.18) as a 256 entry BTB.

Figure 7.6 shows that the Precomputed-Branch architecture with no additional prediction for indirect branches is comparable to the BTB architecture with many more entries. The Precomputed-Branch configurations with 14-bit and 16-bit branch spaces fairs worse than the 21-bit design, because a larger number of branches must span branch spaces (increasing %EiB). The BTB results shown use a 21-bit offset. Therefore, the BTB results are only directly comparable to the Precomputed-Branch architecture results using a 21-bit branch space. These branches are converted into indirect jumps and they may be mispredicted (increasing %MpB). Partitioning using profiles, as in the MaxCut partitioning, helps reduce the penalty for small branch spaces. Figure 7.6 shows that adding a very small indirect jump buffer for indirect branches provides as much benefit as profile-based partitioning, and reduces the BEP for each design. The results show that even without using a small indirect jump buffer that the Precomputed-Branch architecture achieves the same performance as a 64 entry BTB.

Recall that the branch target buffer requires considerable resources, and is organized as a 4-way

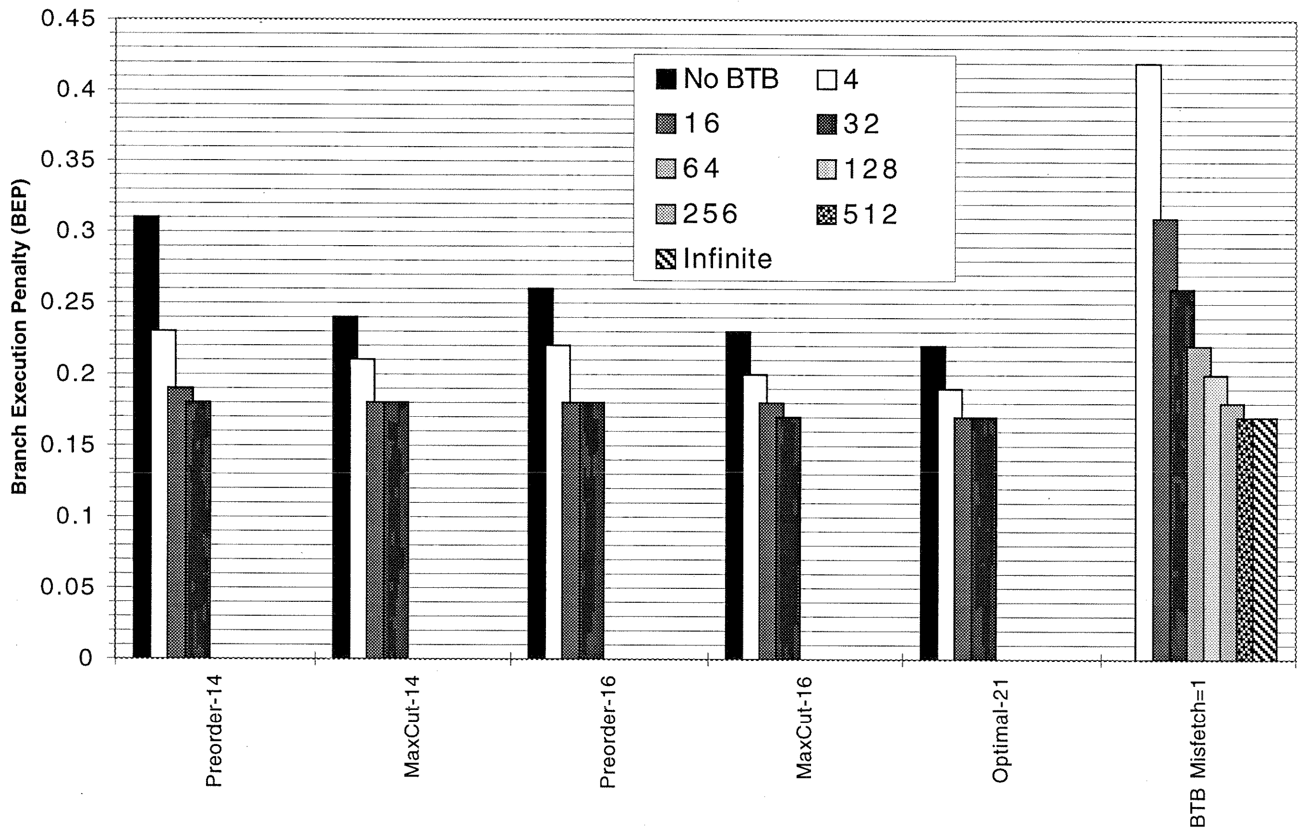


Figure 7.6. The branch execution penalty is computed with a 4-cycle branch misprediction penalty, a 1-cycle misfetch penalty and a 1-cycle penalty for extra indirect branches. Results are shown for a 14-bit, 16-bit, and 21-bit branch space. All the BTB designs are 4-way associative, and for the Precomputed-Branch architecture the BTB is only used to predict indirect jumps.

Table 7.4: Summary of Performance Information From Trace Driven Simulations

Architecture	BTB Size	%MfB	%MpB	%EiB	BEP
BTB	4	22.88	4.85	0.0	0.42
BTB	16	12.64	4.66	0.0	0.31
BTB	32	8.62	4.43	0.0	0.26
BTB	64	4.57	4.37	0.0	0.22
BTB	128	2.57	4.31	0.0	0.20
BTB	256	1.08	4.27	0.0	0.18
BTB	512	0.44	4.25	0.0	0.17
BTB	Infinite	0.02	4.23	0.0	0.17
Preorder-14	0	0.0	7.75	2.29	0.31
Preorder-14	4	0.0	5.69	2.29	0.23
Preorder-14	16	0.0	4.78	2.29	0.19
Preorder-14	32	0.0	4.60	2.29	0.18
MaxCut-14	0	0.0	6.37	0.91	0.24
MaxCut-14	4	0.0	5.37	0.91	0.21
MaxCut-14	16	0.0	4.58	0.91	0.18
MaxCut-14	32	0.0	4.46	0.91	0.18
Preorder-16	0	0.0	6.07	0.62	0.26
Preorder-16	4	0.0	5.13	0.62	0.22
Preorder-16	16	0.0	4.53	0.62	0.18
Preorder-16	32	0.0	4.46	0.62	0.18
MaxCut-16	0	0.0	5.77	0.31	0.23
MaxCut-16	4	0.0	4.97	0.31	0.20
MaxCut-16	16	0.0	4.42	0.31	0.18
MaxCut-16	32	0.0	4.35	0.31	0.17
Optimal-21	0	0.0	5.45	0.0	0.22
Optimal-21	4	0.0	4.69	0.0	0.19
Optimal-21	16	0.0	4.29	0.0	0.17
Optimal-21	32	0.0	4.26	0.0	0.17

The branch execution penalty is computed with a 4-cycle branch misprediction penalty, a 1-cycle misfetch penalty and a 1-cycle penalty for extra indirect branches. The values shown are arithmetic means over all programs.

associative cache, while the Precomputed-Branch architecture uses information recorded in the instructions. Thus, the Precomputed-Branch design takes significantly less area. The access time for a cache (or a BTB) depends both on the size and the associativity of the design [29, 50]. Chapter 6 showed that direct mapped caches can be 40% faster than associative caches, even for very small caches. This occurs because the direct mapped cache contents can be used speculatively, before the full tag comparison finishes. Also, small caches are faster than large caches. Despite the advantages of direct mapped caches, many BTB designs, such as the BTB used in the Intel Pentium and P6 and those proposed by Yeh **et al.** [66] use a large, multi-associative BTBs to reduce the misfetch penalty. Since the instruction fetch cycle often limits processor performance, designs using a large associative BTB may lengthen the cycle time, affecting the performance of the entire processor.

We simulated the Precomputed-Branch architecture with both a direct-mapped and 4-way associative IJB. In Figure 7.4, we only showed the information for the 4-way design to reduce the degrees of freedom in the comparison. Table 7.5 compares the performance of the "Optimal-21" partitioning with a direct-mapped and associative IJB, and shows that there is little difference in performance. Tables 7.6 and 7.7 show the percentage of mispredicted branches for all the programs simulated using no IJB, a 4, 16 and 32 entry IJB, for the Preorder and MaxCut partitions, and for 14-bit, 16-bit, and 21-bit branch spaces. The results show that most of the performance gain achieved by adding an IJB for all the partitions and branch spaces is provided by adding only a 4 entry IJB. The C++ programs benefit the most from the addition of a very small indirect jump buffer. For example, the %MpB for the `db++` program drops from 15.71 to 0.78 with the additional of a four-entry IJB. The only program that significantly benefits form having a 16 entry IJB is the highly object-oriented C++ program `idl`. Both of these programs contain a large number of indirect jumps, as shown by the detailed program information in Table 3.4, and these indirect jumps are very predictable. The predictability of indirect jumps in C++ programs was shown in an earlier study of ours [9], and we have been demonstrated this for many C++ applications.

Table 7.5: Average Direct Mapped and 4-Way Associative IJB Performance for the Optimal-21 Partitioning

		IJB Entries			
		0	4	16	32
Direct Mapped IJB	BEP	0.22	0.19	0.18	0.17
	%MpB	5.5	4.7	4.4	4.3
4-way Associative IJB	BEP	0.22	0.19	0.17	0.17
	%MpB	5.5	4.7	4.3	4.3

In general, the BTB branch architecture has worse performance for large programs in the benchmark suite, although this is not indicated in the mean values we show. In part, this is a reflection of the application mix we used for benchmarking. Table 3.5 indicates that many of the Perfect Club and SPEC applications are dominated by a small number of heavily executed branches, primarily in loops. By comparison, benchmarks such as `cfront` and `groff` have more branches, and these better illustrate the problems of fixed capacity mechanisms such as BTB's. We also feel that these programs better illustrate the performance of branch architectures on actual applications. Programs such as `cfront`, `gcc`, `lic` and `groff` contain a large number of branches, and the Precomputed-Branch architecture performs very well for these programs. Table 7.8, which provides the %MfB and %MpB, shows why. Each program has a high misprediction rate in both architectures, reflecting the unpredictability of the conditional branches in these applications. The BTB architecture has a slightly lower %MpB than the Precomputed-Branch architecture if no IJB entries are used, because the BTB architecture can predict indirect jumps. However, the BTB architecture must also use the BTB to avoid instruction misfetch penalties, and even a 256-entry BTB has a considerable number of misfetches for these large programs.

Table 7.6: Percent of Mispredicted Branches for IJB with Preorder Partitioning

	14 bit Branch Space				16 bit Branch Space				21 bit Branch Space			
Program	No	4	16	32	No	4	16	32	No	4	16	32
APS	6.6	3.9	3.3	3.2	5.9	3.6	3.3	3.1	3.2	3.1	3.1	3.1
CSS	7.6	6.6	6.0	5.5	7.0	6.0	5.3	5.1	5.2	4.5	4.2	4.0
LGS	11.6	5.4	5.2	5.1	5.1	5.1	5.1	5.1	5.1	5.1	5.1	5.1
LWS	8.5	6.3	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4
NAS	13.6	5.0	3.8	3.3	4.2	3.4	3.3	3.3	4.1	3.4	3.3	3.3
OCS	2.9	2.5	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4
SDS	3.3	3.3	3.3	3.3	3.3	3.3	3.2	3.2	3.3	3.3	3.2	3.2
TFS	6.3	5.9	4.0	3.9	4.0	3.9	3.9	3.9	4.0	3.9	3.9	3.9
TIS	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7
WSS	6.5	4.5	3.9	3.8	5.9	4.0	3.8	3.8	5.5	4.0	3.8	3.8
doduc	4.6	4.5	4.5	4.5	4.4	4.4	4.4	4.4	4.4	4.4	4.4	4.4
fpppp	4.9	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6
hydro2d	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8
mdljsp2	7.0	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7
nasa7	2.8	2.4	2.3	2.2	2.6	2.3	2.2	2.2	2.6	2.3	2.2	2.2
ora	3.0	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6
spice	5.9	4.6	4.1	4.1	5.9	4.5	4.1	4.1	4.2	4.1	4.1	4.1
su2cor	14.2	9.3	9.3	9.3	9.5	9.3	9.3	9.3	9.5	9.3	9.3	9.3
swm256	0.7	0.5	0.5	0.5	0.6	0.5	0.5	0.5	0.6	0.5	0.5	0.5
tomcatv	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
wave5	6.8	3.6	3.5	3.5	3.6	3.5	3.5	3.5	3.6	3.5	3.5	3.5
alvinn	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
compress	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9
ear	4.0	3.9	3.9	3.9	4.0	3.9	3.9	3.9	4.0	3.9	3.9	3.9
eqntott	3.1	1.3	1.3	1.3	3.0	1.3	1.3	1.3	3.0	1.3	1.3	1.3
espresso	6.5	5.5	5.3	5.2	5.3	5.1	5.1	5.1	5.3	5.1	5.1	5.1
gcc	15.1	14.0	13.4	13.1	14.5	13.4	13.0	12.7	13.4	12.5	12.3	12.3
li	5.9	4.8	4.3	4.1	5.1	4.0	4.0	4.0	5.1	4.0	4.0	4.0
sc	5.3	4.1	4.0	3.9	4.4	3.8	3.8	3.8	4.4	3.8	3.8	3.8
cfront	18.7	18.1	17.1	16.4	16.7	16.1	15.7	15.3	14.4	13.9	13.7	13.7
db++	15.7	0.8	0.8	0.8	15.7	0.8	0.8	0.8	15.7	0.8	0.8	0.8
groff	11.4	9.6	7.3	6.6	8.9	7.5	6.1	5.8	7.5	6.3	5.5	5.3
idl	21.1	20.9	3.4	2.6	18.4	18.3	3.0	2.2	13.3	13.2	2.2	1.8
lic	13.0	10.4	9.6	9.0	9.8	8.5	8.0	7.3	6.9	6.8	6.7	6.7
porky	19.5	8.5	6.4	5.4	8.2	6.2	5.1	4.4	5.8	4.2	3.5	3.3
Avg	7.7	5.7	4.8	4.6	6.1	5.1	4.5	4.4	5.5	4.7	4.3	4.3

The percentage of mispredicted branches using the Preorder partitioning. The results are shown for a varying number of IJB entries; the IJB entries are only used to predict indirect jumps.

Table 7.7: Percent of Mispredicted Branches for IJB with MaxCut Partitioning

	14 bit Branch Space				16 bit Branch Space				21 bit Branch Space			
Program	No	4	16	32	No	4	16	32	No	4	16	32
APS	3.4	3.2	3.1	3.1	3.2	3.1	3.1	3.1	3.2	3.1	3.1	3.1
CSS	6.6	5.5	4.3	4.1	5.2	4.5	4.2	4.0	5.2	4.5	4.2	4.0
LGS	6.8	5.5	5.1	5.1	5.1	5.1	5.1	5.1	5.1	5.1	5.1	5.1
LWS	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4	5.4
NAS	4.6	3.6	3.3	3.3	4.1	3.4	3.3	3.3	4.1	3.4	3.3	3.3
OCS	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4	2.4
SDS	3.4	3.3	3.3	3.2	3.3	3.3	3.2	3.2	3.3	3.3	3.2	3.2
TFS	4.2	4.1	3.9	3.9	4.0	3.9	3.9	3.9	4.0	3.9	3.9	3.9
TIS	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7	1.7
WSS	6.2	4.1	3.8	3.8	5.5	4.0	3.8	3.8	5.5	4.0	3.8	3.8
doduc	5.4	4.9	4.4	4.4	4.4	4.4	4.4	4.4	4.4	4.4	4.4	4.4
fpppp	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6	4.6
hydro2d	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8	2.8
mdljsp2	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7	6.7
nasa7	3.6	3.1	2.5	2.2	2.6	2.3	2.2	2.2	2.6	2.3	2.2	2.2
ora	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6	2.6
spice	4.5	4.3	4.1	4.1	4.2	4.1	4.1	4.1	4.2	4.1	4.1	4.1
su2cor	9.5	9.3	9.3	9.3	9.5	9.3	9.3	9.3	9.5	9.3	9.3	9.3
swm256	0.6	0.5	0.5	0.5	0.6	0.5	0.5	0.5	0.6	0.5	0.5	0.5
tomcatv	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5	0.5
wave5	5.3	4.0	3.5	3.5	3.6	3.5	3.5	3.5	3.6	3.5	3.5	3.5
alvinn	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2	0.2
compress	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9	9.9
ear	4.0	3.9	3.9	3.9	4.0	3.9	3.9	3.9	4.0	3.9	3.9	3.9
eqntott	3.0	1.3	1.3	1.3	3.0	1.3	1.3	1.3	3.0	1.3	1.3	1.3
espresso	5.4	5.2	5.1	5.1	5.3	5.1	5.1	5.1	5.3	5.1	5.1	5.1
gcc	15.3	14.1	13.4	13.0	14.4	13.2	12.7	12.5	13.4	12.5	12.3	12.3
li	6.0	4.9	4.3	4.1	5.1	4.0	4.0	4.0	5.1	4.0	4.0	4.0
sc	4.5	3.8	3.8	3.8	4.4	3.8	3.8	3.8	4.4	3.8	3.8	3.8
cfront	18.4	17.5	16.4	15.6	15.5	14.9	14.6	14.4	14.4	13.9	13.7	13.7
db++	15.7	0.8	0.8	0.8	15.7	0.8	0.8	0.8	15.7	0.8	0.8	0.8
groff	9.5	8.0	6.6	6.1	8.4	6.9	5.9	5.7	7.5	6.3	5.5	5.3
idl	20.8	20.6	3.1	2.5	18.4	18.3	3.0	2.2	13.3	13.2	2.2	1.8
lic	10.2	8.7	8.3	7.7	7.2	6.8	6.8	6.8	6.9	6.8	6.7	6.7
porky	9.1	7.2	5.6	4.8	8.5	6.6	5.3	4.6	5.8	4.2	3.5	3.3
Avg	6.4	5.4	4.6	4.5	5.8	5.0	4.4	4.4	5.5	4.7	4.3	4.3

The percentage of mispredicted branches using the MaxCut partitioning. The results are shown for a varying number of IJB entries; the IJB entries are only used to predict indirect jumps.

Table 7.8: Precomputed-Branch Penalties for Branch Intensive Programs

	BTB	cfront		gcc		groff		lic	
Config	IJB	%MfB	%MpB	%MfB	%MpB	%MfB	%MpB	%MfB	%MpB
BTB	256	12.52	13.79	6.91	12.39	4.95	5.35	4.80	6.75
BTB	Inf	0.25	13.61	0.03	12.30	0.04	5.13	0.25	6.74
Opt-21	0	0.00	14.39	0.00	13.42	0.00	7.51	0.00	6.90
Opt-21	4	0.00	13.93	0.00	12.48	0.00	6.34	0.00	6.77
Opt-21	16	0.00	13.75	0.00	12.33	0.00	5.45	0.00	6.74
Opt-21	32	0.00	13.66	0.00	12.32	0.00	5.31	0.00	6.74

7.6 Practical Concerns

In the past, segment architectures have been greeted with less than overwhelming enthusiasm, due to limited segment sizes. However, the explicit displacement branch architecture has a single instruction address space with branch instructions that can only access a portion of that address space. The chief complication with an explicit displacement encoding is that PC-relative code relocation, important for shared program libraries, is no longer possible without dynamic relinking. However, consider using an architecture such as the DEC Alpha AXP, which uses a 21-bit branch displacement for word-aligned instructions in a 64-bit instruction address space. The instruction space is broken into $2^{64-21+2}$, or $\approx$ 2 trillion branch spaces of 8MBytes each. Branches within each 8MB branch space use a single explicit displacement. Each segment can be relocated to $\approx$ 2 trillion different locations without modification, and all branches within a given branch space are relative to that branch space. Such large branch spaces would address almost all programs we have encountered.

Another interesting point about using shared libraries is that procedure calls to shared libraries are compiled as indirect jumps on existing systems, because the procedure destinations are not known until execution time [20]. In a related study [15], we found that on average, for 43 C and FORTRAN programs we examined on Digital Unix, 60% of the procedure calls are indirect procedure calls because of shared libraries. Therefore, operating systems using shared libraries may have the need for a small indirect jump buffer, even without the increase in indirect jumps possibly caused by partitioning a program into branch spaces in the Precomputed-Branch architecture. Furthermore, the indirect procedure calls created by the use of shared libraries would decrease the number of procedure calls that need to be converted to indirect jumps when partitioning the program into branch spaces for a Precomputed-Branch architecture.

7.6.1 Non-relative Branches in a Relative World

The Precomputed-Branch architecture performs most branch computation at link or compile time. Traditional relative branches perform the branch computation during instruction issue, and branch target buffers can be used to cache this information. The primary objection to using non-relative branches is that most instruction sets already use relative branches, and the precomputed branch architecture requires a change to the instruction set architecture.

It is also possible to precompute the branch destination when instructions are entered into the instruction cache. Instructions in many architectures are partially decoded when fetched into the cache, simplifying instruction dispatch and scheduling. The destination for a relative branch instruction can also be computed during instruction fetch. This is shown diagrammatically in Figure 7.7. If the carry-bit is set after the PC-relative part of the target address is computed, the branch destination is not in the current branch space. If the branch instruction branches to a destination within the same branch space, a Precomputed-Bit is set to indicate that the destination has already been computed, and the instruction stored in the instruction cache is changed to contain the precomputed address. Then when this branch is executed the precomputed offset is concatenated with the PC as before providing the next instruction fetch. If the destination is not in the same branch space, the Precomputed-Bit is not set indicating that the IJB should be used to predict the branch destination. During the normal instruction issue, the proper destination is computed and used to initialize the IJB for all non-precomputed branches. Then on subsequent accesses to that instruction, the target address stored in the IJB will be used in the next instruction fetch, predicting the branch's destination.

In such an architecture, the partitioning discussed earlier reduces the number of branches that span branch spaces. As in the Precomputed-Branch architecture, this improves the effectiveness of the IJB entries. With partitioning, the cost and performance for this new architecture would be identical to the Precomputed-Branch architecture design, with the addition of an extra cycle of delay when instructions are fetched into the instruction cache. One benefit to this alternate design shown in Figure 7.7, is that the instruction set architecture does not need to be changed since the compiler still creates branches with PC-relative offsets.

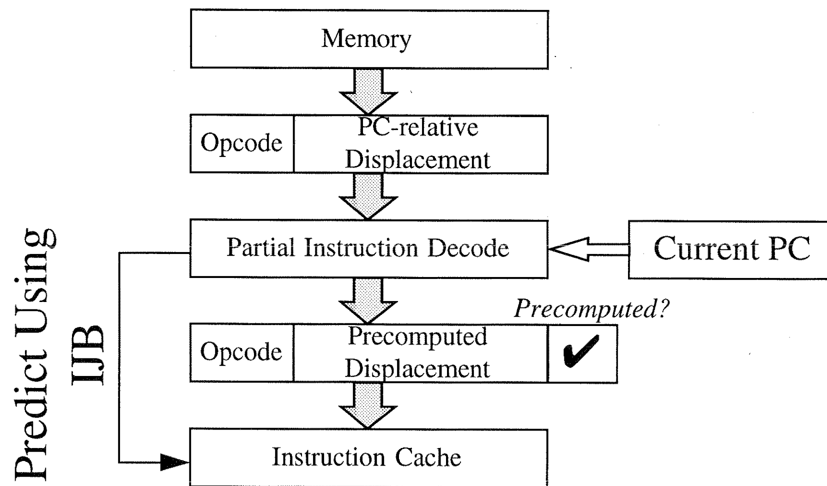


Figure 7.7: Using the Precomputed-Branch Architecture with a PC-Relative Instruction Set

7.7 Design Issues and Comparison of Precomputed-Branch and NLS Architectures

The Precomputed-Branch and NLS architectures are very similar designs, because the precomputed offsets in the Precomputed-Branch architecture provide the same prediction information that the NLS predictors provide. The Precomputed-Branch architecture is similar to a NLS-Cache design where there would be one NLS predictor in the instruction cache for every branch instruction. One advantage of the precomputed offset over using a NLS predictor is that the precomputed offset is always initialized correctly since the value is precomputed by the linker or when the instructions are brought into the instruction cache. Another advantage is that since the precomputed branch is stored as the offset inside the instruction, no additional hardware is needed to provide the predictors, whereas in the NLS architecture extra hardware is needed in order to represent predicted taken cache indices.

One design issue that needs to be addressed is how many ports are needed for these hardware designs. Both the BTB and NLS architectures need to be dual-ported, since potentially in each cycle a predictor must be read and a predictor must be updated. This gives the Precomputed-Branch architecture an advantage over both the BTB and NLS architectures since it only needs a single port for reading prediction information each cycle. The precomputed offsets do not need to be dynamically updated every cycle because their offsets are determined at compile time or when they are brought into the instruction cache.

Another design issues is how to implement both the NLS and Precomputed-Branch architectures for wide-issue architectures. There are two basic options for implementing branch prediction on a wide-issue architecture. The processor could either predict one branch for each instruction issue or predict multiple branches for each instruction issue. When predicting only one branch for each instruction issue, the NLS architecture is a reasonable design, since one NLS predictor would be associated with each instruction fetch predicting the taken index for that issue of instructions. For the Precomputed-Branch architecture, one would need to add a couple bits to each instruction fetch in the instruction cache indicating which instruction's precomputed offset should be used to predict the taken index. Note that this does not fully utilize the Precomputed-Branch architecture for each instruction fetch, since there can be more than one precomputed branch instruction in each instruction fetch. If multiple branches were predicted at the same time then the Precomputed-Branch architecture could take full advantage of all the precomputed offsets stored in the instruction cache. One of the problems with predicting multiple branches each cycle, is that in the decode stage it would be expensive to calculate multiple PC-relative target addresses at a time. This is not an issue with the Precomputed-Branch architecture since all the full target addresses could be easily calculated in the decode stage by concatenating the precomputed offsets with the upper bits of the PC. Therefore, using the Precomputed-Branch architecture and predicting multiple branches at the same time could be a very cost effective design resulting in very accurate branch and instruction fetch prediction.

7.8 Summary

This chapter showed that using a precomputed or non-relative branch displacement is more effective than an architecture that uses a large BTB to cache the destinations for branches. A variety of simple partitioning algorithms were examined that break programs into multiple branch spaces and convert the inter-space branches into indirect jumps. These partitioning algorithms are simple, work well without profile information and work better with real or estimated profile information. Note that even in a PC-relative branch architecture that a similar number of procedure calls would be converted to indirect jumps for those PC-relative displacements that wouldn't be able to fit within the 14-bit or 16-bit offset in the instruction for the branch spaces we examined. We have shown that combining a small indirect jump buffer with the Precomputed-Branch architecture results in a branch architecture that uses few resources and has excellent performance, particularly for programs with a large number of branches. We also described how to use the Precomputed-Branch architecture in existing processors without having to modify the instruction set architecture. One advantage of the proposed Precomputed-Branch design is the precomputed destination does not depend on the size of the address range. By comparison, the size of a branch target buffer would increase as the instruction address range increases, and will pose problems for architects designing 64-bit processors.

This branch encoding has been used in older architectures such as the PDP-8, but is even better suited for modern architectures when a sizable branch displacement field is provided. The results show that the Precomputed-Branch architecture effectively eliminates all misfetch penalties, and has a lower branch execution penalty than the BTB architecture, using very few hardware resources.

CHAPTER 8

CONCLUSIONS

This research was originally motivated by examining the predictability of C++ applications using branch target buffers. This sparked my interest in the instruction fetch problem, since the main hardware mechanism for instruction fetch prediction studied in previous literature is the BTB design, and as shown in Chapter 6 this design can be costly to implement.

Historically, previous literature focused on the BTB as a means for predicting conditional branches, and very few studies examined the BTB's contribution to eliminating misfetch penalties. This is because the penalty for mispredicted conditional branches is larger than the misfetch penalty associated with instruction fetch prediction. Therefore, increasing the accuracy of conditional branch prediction would give the largest gain in program performance. The other reason is that past architectures were single issue processors, and for these processors it was fairly easy to eliminate misfetch penalties using branch delay slots. With the recent development of highly accurate correlated branch prediction combined with the fact that processors are starting to issue many instructions per cycle, making sure a processor also has accurate instruction fetch prediction is increasingly important. This is shown in Chapter 4, where misfetch penalties can degrade a programs performance more than the degradation due to the remaining mispredicted conditional branches. The importance of instruction fetch prediction is also shown in recent processor designs that devote a large amount of hardware resources to branch target buffers in order to eliminate misfetch penalties.

In addressing the issue of instruction fetch prediction, I felt it was important to perform a system level study in order to examine the impact of misfetched branches in terms of overall system performance. Chapter 4 showed that eliminating all misfetch penalties on a dual-issue Alpha 21064 architecture would decrease the average CPI by 6%. This system level study also showed that the BEP is a valid metric for comparing branch and instruction fetch prediction architectures for a statically scheduled processor like the Alpha 21064.

In improving instruction fetch prediction, I first wanted to examine how well I could reduce misfetch penalties on existing branch and instruction fetch prediction architectures using only compiler optimizations. The idea of branch alignment came from a simple observation – the less often a processor executes a taken target address the fewer number of misfetch penalties the processor will have. Chapter 5 showed that reordering basic blocks so that the fall-through path occurs more often greatly reduces the misfetch penalty for current PHT architectures and there is even a noticeable improvement for BTB architectures, especially for small BTBs.

BTB designs that were examined in previous studies typically associated the conditional branch prediction information with the BTB. Once it was realized that decoupling the conditional branch prediction information from the BTB achieves prediction accuracy similar to a coupled design, then the only use for the BTB is for instruction fetch prediction. In Chapter 6, I proposed the idea of using next cache line and set predictors (cache indices) in a NLS-table design, instead of target addresses for instruction fetch prediction. This idea was realized by observing that the target address stored in the BTB is used as an index into the instruction cache, indicating the cache line to start fetching instructions from. Chapter 6 showed that the NLS-table architecture performs better than BTB architectures with similar hardware costs, and that the NLS architecture may be appropriate for high-speed processor designs.

The last solution examined for instruction fetch prediction in this dissertation eliminates the storing of target addresses in the BTB and the use of indices in the NLS architecture. The Precomputed-Branch architecture is an example of a hardware/software co-design solution for instruction fetch prediction. The technique uses an idea from older computers, such as the PDP-8, where a target address is calculated by concatenating the PC with a precomputed branch offset, creating a branch segmented architecture. Chapter 7 showed that the Precomputed-Branch architecture eliminates all misfetch penalties and outperforms the BTB design using very

few hardware resources.

The final contribution of this dissertation is that these results are among the first to examine the effects of branch and instruction fetch prediction on object-oriented programming languages such as C++. The system level study in Chapter 4 showed that branch architectures provided on current processors accurately predict C++ applications. The NLS architecture results in Chapter 6 also showed that the NLS architecture accurately predicts C++ applications. The branch alignment results in Chapter 5 showed that branch alignment provides larger improvements for C++ programs than seen in C and FORTRAN programs, when using a BTB architecture, because C++ programs benefit more from the decrease in BTB usage resulting from branch alignment. The Precomputed-Branch architecture, proposed in Chapter 7, does not provide support for predicting indirect procedure calls in C++. However, the Precomputed-Branch architecture results showed that adding a small 4 to 16 entry direct mapped indirect jump buffer (IJB) can be used to eliminate indirect jump mispredict penalties for highly object-oriented C++ applications such as `db++`, `idl` and `groff`. Alternatively, many of these C++ indirect procedure calls could be eliminated by performing compiler optimizations such as If-conversion and procedure inlining as described in Chapter 2.

This dissertation provided three solutions to the instruction fetch prediction problem ranging from a software solution (Branch Alignment), to a purely hardware solution (NLS architecture), and concluded with a combined hardware/software solution (Precomputed-Branch architecture). All three of these techniques were shown to improve instruction fetch prediction. The hardware solutions were also shown to use fewer hardware resources than the branch target buffer design. These results were reported using the BEP metric assuming a statically scheduled processor. Future research is needed to verify that these results apply to future dynamic out-of-order issue processors, threaded architectures, and multiscalar architectures.

CHAPTER 9

FUTURE WORK

This chapter describes future directions for branch and instruction fetch research, by first describing a few important implications for branch architecture research, then discussing future research for static branch prediction, and concluding with future directions for dynamic branch and instruction fetch prediction research for next generation processors.

9.1 Implications for Branch Prediction Research

An important result missing from branch prediction literature is exactly how important branch prediction is for current and future processors. System level studies are important for future branch architecture research in order to understand the importance of branch prediction on the processors being examined in terms of the overall system performance, and to understand the relevance of proposed improvements for branch prediction. Performing system level studies is also useful in validating branch metrics, as in the validation of the BEP metric in Chapter 4. The system level study performed in Chapter 4, showed that for a statically scheduled architecture, like the Alpha 21064, that improving the conditional branch prediction accuracy is "polishing a round ball". This implies that researchers examining branch architectures for this type of processor should emphasize designs that are faster, less complex, more testable or easier to implement. The performance of future processor designs will depend more upon accurate branch and instruction fetch prediction than the Alpha 21064 architecture studied in this dissertation. Therefore, system level studies are needed in order to understand the importance of branch prediction for these future processor designs.

Many studies that include a form of branch prediction typically only model conditional branch prediction since this is needed for speculative execution and it achieves the largest gain in performance. This dissertation shows that penalties from other branches such as instruction misfetch penalties and mispredicted indirect jumps, also have a large impact on system performance especially for future wide-issue processors. Therefore, it is important that future studies include mispredict and misfetch penalties for all branch types when possible.

Most of the previous studies on branch and instruction fetch prediction only examine a few C and FORTRAN programs from the SPEC benchmark suite. This is useful since these applications are widely available and have been thoroughly studied. The downside to using these applications is that they do not capture the branch behavior in all languages or for larger programs which are more representative of modern applications. The number of programs examined for this dissertation was expanded to include many non SPEC programs and C++ applications. Results presented in this dissertation show that C++ programs have a different branch instruction mix, different conditional branching behavior, and execute many more branch sites than C or FORTRAN programs. Even so, the results indicate that existing branch architectures accurately predict object-oriented C++ applications. Future research is needed to verify these results by examining the effects of branch and instruction fetch prediction on larger applications, and on other languages such as Modula, Ada, and object-oriented languages such as SELF and Cecil.

9.2 Static Prediction

An exciting area of branch prediction research is improving the accuracy of static branch prediction. This is an important problem, since static prediction is used to drive profile-based compiler optimizations. The two basic methods for predicting branches for these optimizations are to either use profiles or program-based heuristics.

Profile prediction has been studied for many years, and several researchers have shown that profiles can accurately predict conditional branches from one run of a program to the next. Even so, there are still several

interesting areas of research dealing with profile prediction. In a recent study, we found that libraries have common behavior between **different** applications [15]. We found that using a profile from one application's use of a library for predicting the library behavior for a different application, achieves conditional branch prediction accuracy close to perfect profile prediction. These results indicate it should be beneficial to use profiles to perform compiler optimizations on libraries before they are shipped, including shared libraries. Another recent profile prediction result that shows promise is correlated static branch prediction. Young and Smith [69] recently proposed a very accurate profile static branch predictor that captures the behavior of correlated branch prediction. The correlated branch prediction can be used at compile time to accurately guide trace scheduling. Their technique accurately predicts conditional branches, but it requires correlated profiles to be gathered for each program in order to perform the optimizations.

The advantage of program-based heuristics over profiles is that a program's performance can be estimated at compile time without the expensive process of gathering profiles. Hank **et al.** [26] showed they were able to perform profile-based compiler optimizations using simple heuristics for predicting conditional branch directions that achieved speedups comparable to the same optimizations performed with profiles. Effective heuristics for predicting branches have been proposed, such as the Ball and Larus heuristics [4], and recent program estimation techniques based on these heuristics have been examined by Wagner **et al.** [60] and Wu and Larus [64]. We have added to this research by examining machine-learning techniques for predicting program behavior resulting in automated compiler heuristics [14].

Accurately estimating program behavior by either using profiles or program-based heuristics as described above, will be an increasingly important area of research as processors rely on compilers to capture increasing levels of instruction parallelism.

9.3 Dynamic Prediction and Future Processor Designs

As the number of instructions issued per cycle increases on superscalar processors, there may be a need to predict multiple branch instructions at the same time. One important future research area is to determine how to expand existing branch and instruction fetch prediction architectures and the architectures proposed in this dissertation to handle predicting multiple branches concurrently.

Another important question for dynamic instruction fetch prediction and branch prediction is how well will these architectures perform for the next generation of processors. Future processor designs may be moving towards either a threaded architecture design, as in the Tera [2] architecture, or a multiscalar architecture design [51]. For both of these architectures, a program can be split into fine grain tasks, and these tasks can be run on one to many processors. It is unclear how important branch prediction is for these processors and the effects of misfetched and mispredicted branches. Branch prediction may be extremely important when running a single application, even more so than on current superscalar architectures. But if many applications are executing on the same processor, the processor may be able to avoid all branch penalties by switching to another task in the same program or another application. In this case, branch prediction penalties may be completely masked without any help from a branch prediction architecture. For instance, in a threaded architecture whenever a branch is encountered, the processor could start executing a different thread, and switch back to the previous thread when the destination of the branch is resolved. This would effectively eliminate all the branch penalties associated with branch prediction without using any branch prediction architecture. On the other hand, if branch prediction is needed, existing branch architectures may perform poorly since the execution streams can change often between different applications or parts of a parallel program causing a degradation in prediction performance due to alias affects in the branch architecture. Future research is needed to examine the importance of branch and instruction fetch prediction for these architectures and to find the branch and instruction fetch prediction designs that are best fitted to these future processors.

BIBLIOGRAPHY

[1] W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformation. **IEEE Transactions on Computers**, C-30(5):341–356, May 1981.

[2] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The tera computer system. In **International Conference on Supercomputing**, pages 1–6, Amsterdam, Netherlands, June 1990. ACM, ACM.

[3] Jean Loup Baer and R. Caughey. Segmentation and optimization of programs from Cyclic Structure Analysis. In **Proceedings of AFIPS**, pages 23–36, 1972.

[4] T. Ball and J. R. Larus. Branch prediction for free. In **1993 SIGPLAN Confernce on Programming Language Design and Implementation**, pages 300–313. ACM, June 1993.

[5] Brian N. Bershad, Dennis Lee, Theodore H. Romer, and J. Bradley Chen. Avoiding conflict misses dynamically in large direct-mapped caches. In **6th International Conference on Architectural Support for Programming Languages and Operating Systems**, pages 158–170, San Jose, California, 1994.

[6] Brian Bray and M. J. Flynn. Strategies for branch target buffers. In **24th International Symposium and Workshop on Microarchitecture**, pages 42–49. ACM, ACM, 1991.

[7] Brad Calder and Dirk Grunwald. Fast & accurate instruction fetch and branch prediction. In **21st Annual International Symposium of Computer Architecture**, pages 2–11. ACM, April 1994.

[8] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In **6th International Conference on Architectural Support for Programming Languages and Operating Systems**, pages 242–251. ACM, 1994.

[9] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In **Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages**, pages 397–408, January 1994.

[10] Brad Calder and Dirk Grunwald. Next cache line and set prediction. In **22nd Annual International Symposium of Computer Architecture**, pages 287–296. ACM, June 1995.

[11] Brad Calder and Dirk Grunwald. The precomputed-branch architecture: Efficient branches with compiler support. Submitted to the Journal of Computer and Software Engineering, January 1995.

[12] Brad Calder, Dirk Grunwald, and Joel Emer. Predictive sequential associative cache. Submitted to the 2nd International Symposium on High-Performance Computer Architecture, June 1995.

[13] Brad Calder, Dirk Grunwald, and Joel Emer. A system level perspective on branch architecture performance. In **28th International Symposium on Microarchitecture**, November 1995.

[14] Brad Calder, Dirk Grunwald, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Corpus-based static branch prediction. In **Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation**, pages 79–92. ACM, June 1995.

[15] Brad Calder, Dirk Grunwald, and Amitabh Srivastava. The predictability of branches in libraries. In **28th International Symposium on Microarchitecture**, November 1995.

[16] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. **Journal of Programming Languages**, 2(4):313–351, 1994. Also available as University of Colorado Technical Report CU-CS-698-94.

[17] Brian Case. Intel reveals pentium implementation details. **Microprocessor Report**, 7(4):9, March 1993.

[18] J. H. Chang, H. Chao, and K. So. Cache design of a sub-micron CMOS System/370. In **14th Annual International Symposium of Computer Architecture**, pages 208–213. IEEE, June 1987.

[19] Po-Yung Chang, Eric Hao, Tse-Yu Yeh, and Yale Patt. Branch classification: a new mechanism for improving branch predictor performance. In **27th International Symposium on Microarchitecture**, pages 22–31. ACM, 1994.

[20] Digital Equipment Corporation. **Assembly Language Programming Manual**. DEC, 1994.

[21] L. Peter Deutsch. Efficient implementation of the smalltalk-80 system. In **Proceedings of the 11th Annual ACM Symposium on Principles of Programming Languages**, pages 297–302. Springer-Verlag, January 1984.

[22] Digital Equipment Corporation, Maynard, Mass. **DECchip 21064 Microprocessor: Hardware Reference Manual**, October 1992.

[23] David R. Ditzel and Hubert R. McLellan. Branch folding in the CRISP microprocessor: Reducing branch delay to zero. In **14th Annual International Symposium of Computer Architecture**, pages 2–9. ACM, ACM, June 1987.

[24] Domenico Ferrari. Improving locality by critical working sets. **Communications of the ACM**, 17(11):614–620, 1974.

[25] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In **Fifth International Conference on Architectural Support for Programming Languages and Operating Systems**, pages 85–95, Boston, Mass., October 1992. ACM.

[26] Richard Hank, Scott Mahlke, Roger Bringmann, John Gyllenhaal, and Wen mei Hwu. Superblock formation using static program analysis. In **26th International Symposium on Microarchitecture**, pages 247–256. IEEE, 1993.

[27] S. J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. **IEEE Transactions on Software Engineering**, 14(11):1640–1644, 1988.

[28] D. Hatfield and J. Gerald. Program restructuring for virtual memory. **IBM Systems Journal**, 10(3):168–192, 1971.

[29] Mark Hill. A case for direct-mapped caches. **IEEE Computer**, 21(12):25–40, December 1988.

[30] Urs Hölzle, Craig Chambers, and David Unger. Optimizing dynamically-typed object-orientred languages with polymorphic inlined caches. In **ECOOP '91 Conference Proceedings.**, pages 21–38. Springer-Verlag, July 1991.

[31] Peter Yan-Tek Hsu. Designing the TFP microprocessor. **IEEE Micro**, 14(2):23–33, April 1994.

[32] Mike Johnson. **Superscalar Microprocessor Design**. Innovative Technology. Prentice-Hall. Inc., Englewood Cliffs, NJ, 1991.

[33] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In **18th Annual International Symposium of Computer Architecture**, pages 34–42. ACM, May 1991.

[34] Manolis G. H. Katevenis. **Reduced Instruction Set Computer Architecture for VLSI**. ACM Doctoral Dissertation Award Series. MIT Press, 1985.

[35] Brian Kernighan. Optimal sequential partitions of graphs. **Journal of the ACM**, 18(1):34–40, 1971.

[36] R. Kessler and M. Hill. Page placement algorithms for large direct-mapped real-index caches. **ACM Transactions on Computer Systems**, 10(4):338–359, November 1992.

[37] R. Kessler, Richard Jooss, Alvin Lebeck, and Mark D. Hill. Inexpensive implementations of set-associativity. In **16th Annual International Symposium of Computer Architecture**, pages 131–139. IEEE, May 1989.

[38] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. **IEEE Computer**, 21(7):6–22, January 1984.

[39] David J. Lilja. Reducing the branch penalty in pipelined processors. **IEEE Computer**, pages 47–55, July 1988.

[40] Scott McFarling. Program optimization for instruction caches. In **Third International Conference on Architectural Support for Programming Languages and Operating Systems**, pages 183–191. ACM, 1988.

[41] Scott McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[42] Scott McFarling and John Hennessy. Reducing the cost of branches. In **13th Annual International Symposium of Computer Architecture**, pages 396–403. ACM, 1986.

[43] Wen mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In **16th Annual International Symposium of Computer Architecture**, pages 242–251. ACM, ACM, 1989.

[44] MIPS Technologies, Incorporated. R10000 microprocessor product overview. Technical report, MIPS Technologies, Incorporated, October 1994.

[45] Johannes M. Mulder, Nhon T. Quach, and Michael J. Flynn. An area model for on-chip memories and its application. **IEEE Journal of Solid-State Circuits**, 26(2):98–105, February 1991.

[46] Ravi Nair. Optimal 2-bit branch predictors. **IEEE Transactions on Computers**, 44(5):698–702, May 1995.

[47] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In **Fifth International Conference on Architectural Support for Programming Languages and Operating Systems**, pages 76–84, Boston, Mass., October 1992. ACM.

[48] Chris Perleberg and Alan Jay Smith. Branch target buffer design and optimization. **IEEE Transactions on Computers**, 42(4):396–412, April 1993.

[49] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In **Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation**, pages 16–27. ACM, ACM, June 1990.

[50] Steven Przybylski, Mark Horowitz, and John Hennesy. Characteristics of performance-optimal multi-level cache hierarchy. In **16th Annual International Symposium of Computer Architecture**, pages 114–121. IEEE, 1989.

[51] Gurindar Shoi, Scott Breach, and T.N. Vijaykumar. Multiscalar processors. In **22nd Annual International Symposium of Computer Architecture**, pages 414–425. ACM, June 1995.

[52] J. E. Smith. A study of branch prediction strategies. In **8th Annual International Symposium of Computer Architecture**, pages 135–148. ACM, 1981.

[53] Kimming So and Rudolph N. Rechtschaffen. Cache operations by MRU change. **IEEE Transactions on Computers**, 37(6):700–709, June 1988.

[54] S. Peter Song, Marvin Denman, and Joe Chang. The PowerPC 604 RISC microprocessor. **IEEE Micro**, 14(5):8–17, October 1994.

[55] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In **1994 Programming Language Design and Implementation**, pages 196–205. ACM, June 1994.

[56] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimizations at link-time. **Journal of Programming Languages**, pages 1–18, March 1992. (Also available as DEC-WRL TR-92-6).

[57] Amitabh Srivastava and David W. Wall. Link-time optimizations of address calculation on a 64-bit architecture. In **Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation**, pages 49–60. ACM, 1994.

[58] Simon C. Steely and David J. Sager. Next line prediction apparatus for a pipelined computer system. US. Patent #5,283,873, Feb. 1994.

[59] A. R. Talcott, M. Nemirovsky, and R. C. Wood. The influence of branch prediction table interference on branch prediction scheme performance. In **3rd International Conference on Parallel Architectures and Compilation Techniques**, pages 88–98, June 1995.

[60] Tim A. Wagner, Vance Maverick, Susan Graham, and Michael Harrison. Accurate static estimators for program optimization. In **Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation**, pages 85–96, Orlando, Florida, June 1994. ACM.

[61] D. W. Wall. Limits of instruction-level parallelism. In **Fourth International Conference on Architectural Support for Programming Languages and Operating Systems**, pages 176–188, Boston, Mass., 1991.

[62] David W. Wall. Predicting program behavior using real or estimated profiles. In **Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation**, pages 59–70, Toronto, Ontario, Canada, June 1991.

[63] Steven J. E. Wilton and Norman P. Jouppi. An enhanced access and cycle time model for on-chip caches. Report 93/5, DEC Western Research Lab, 1993.

[64] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In **27th International Symposium on Microarchitecture**, pages 1–11, San Jose, Ca, November 1994. IEEE.

[65] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch predictions. In **19th Annual International Symposium of Computer Architecture**, pages 124–134, Gold Coast, Australia, May 1992. ACM.

[66] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In **25th International Symposium on Microarchitecture**, pages 129–139, Portland, Or, December 1992. ACM.

[67] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In **20th Annual International Symposium of Computer Architecture**, pages 257–266, San Diego, CA, May 1993. ACM.

[68] Cliff Young, Nicolas Gloy, and Michael D. Smith. A comparative analysis of schemes for correlated branch prediction. In **22nd Annual International Symposium of Computer Architecture**, pages 276–286. ACM, June 1995.

[69] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In **6th International Conference on Architectural Support for Programming Languages and Operating Systems**, pages 232–241, October 1994.