UNIVERSITY OF CALIFORNIA
RIVERSIDE

Hybrid Branch Prediction Using Limited Dual Path Execution

A Thesis submitted in partial satisfaction
of the requirements for the degree of

Master of Science
in
Computer Science

by
Kelsey Lynn Lick

December, 1996

Thesis Committee:

Professor Gary Tyson, Chairperson

Professor Frank Vahid, Co-Chairperson

Professor Y. C. Hong

Copyright by
Kelsey Lynn Lick
1996

The Thesis of Kelsey Lynn Lick is approved:

Committee Co-Chairperson

Committee Chairperson

University of California, Riverside

ACKNOWLEDGMENTS

I would like to give special thanks to my advisor, Gary Tyson, for his guidance in my work. I am grateful for his gracious attitude in giving time and helpful comments. His words of encouragement always came at just the right times. To my committee members, Frank Vahid and Y. C. Hong, I am thankful for their time and interest in my work.

I am appreciative of all my friends, fellow graduate students, and professors who have enhanced my UCR experience in fellowship and knowledge. Special thanks to Linus and Brian who helped motivate me, made the cramming times endurable, and turned the lab into a fun and friendly atmosphere.

I would like to express my deep appreciation to my husband, Chris, for his patience and self-sacrificing love. His support and encouragement were a constant, and his humor made the tough times fun.

Most importantly, I thank my Father God in heaven who deserves all the credit and glory, for "apart from Him, I can do nothing". John 15:5

ABSTRACT OF THE THESIS

Hybrid Branch Prediction Using Limited Dual Path Execution

by

Kelsey Lynn Lick

Master of Science, Graduate Program in Computer Science

University of California, Riverside, December, 1996

Professor Gary Tyson, Chairperson

Professor Frank Vahid, Co-Chairperson

Branches continue to have a large effect on high performance processors as the issue widths and pipeline depths continue to increase. A steady stream of instructions is needed to keep the processor resources full. Branches disrupt this flow by creating stalls as the path of the branch is resolved. Thus, a highly accurate branch predictor is necessary in order to benefit from these processor enhancements. Hybrid predictors have been introduced to exploit the strengths of each single-scheme component predictor, using a selector to choose which prediction to use for a particular branch based on which component has the best accuracy thus far. Another scheme to reduce branch penalty is dual path execution which executes down both paths of a branch. This thesis proposes a new hybrid predictor which combines a current single-scheme predictor with dual path execution. A predicting set is created to limit the occurrence of dual path execution. Predictors are typically created to target types of branch instructions and sets of correlated branches in order to increase branch prediction accuracy. The predicting set for

this new hybrid predictor targets branch histories, or patterns. Patterns are included in the set that are believed to predict well, leaving the dual path execution resources for those patterns that are thought to have low prediction accuracies. This method provides a great increase in accuracy in comparison to other branch prediction methods, achieving up to an average of 98%. This is a 33% decrease in the misprediction rate of the best known predictors.

Contents

1 Introduction **1**

1.1 Contributions 8

1.2 Organization 9

2 Managing Branch Penalty **10**

2.1 Static Schemes 10

2.2 Dynamic Schemes 12

2.2.1 One-bit Prediction 12

2.2.2 Two-bit Counter 13

2.2.3 Two-Level Adaptive 16

2.2.4 Gshare 22

2.2.5 Hybrid Predictors 23

2.3 Dual Path Execution 25

2.4 New Branch Prediction Approaches 26

2.5 Summary 28

3 Assigning Confidence to Branch Prediction **31**

3.1 Methodology 31

3.1.1 Experimental System 32

3.1.2 Implemented Branch Predictors 32
3.1.3 Description of Benchmarks 33
3.1.4 Branch Characteristics 37
3.2 The Hybrid Approach 39
3.2.1 DPE/PAg 41
3.2.2 DPE/2-bit 42
3.3 Accuracy and Coverage Relationship 43
3.4 Analysis of Patterns and the Predicting Set 44
3.5 Fanout Constraints 49
3.6 Summary 51

4 Evaluating Hybrid DPE Performance **53**
4.1 The Fixed Predicting Set 54
4.2 Profile-based Predicting Sets 59

5 Conclusions **67**
5.1 Future Work 70

Bibliography **73**

List of Figures

1.1 Simple Pipeline without Branch Prediction 4

1.2 Superscalar Pipeline without Branch Prediction 5

1.3 Simple Pipeline using Branch Prediction 7

2.1 State Diagram for the One-Bit Predictor 13

2.2 State Diagram for the Two-Bit Predictor 14

2.3 One-Bit versus Two-Bit Predictor 15

2.4 Structure of the Two-Level Adaptive Branch Predictor 18

2.5 Structures for the Global Two-Level Adaptive Branch Predictors . . 20

2.6 Structures for the Per-set Two-Level Adaptive Branch Predictors . 20

2.7 Structures for the Per-address Two-Level Adaptive Branch Predictors 21

2.8 Example of Branch Correlation 22

2.9 Structure of Gshare Branch Predictor 23

2.10 Global History and Gshare Comparison 24

2.11 Hybrid Branch Predictor Selector 25

2.12 Comparison of Dual Path Execution and Disjoint Eager Execution . 27

3.1 Branch Instructions and Total Branch Executions Relationship . . . 38

3.2 Branch Instructions and Total Misprediction Relationship 40

3.3 Relationship between Accuracy and Coverage 45

3.4 Pattern Sets based on the Number of TAKEN Paths 48

4.1 Misprediction Comparison of PAg and DPE/PAg 55
4.2 Misprediction Comparison of 2-bit and DPE/2-bit 57
4.3 Comparison of PAg, DPE/PAg, and DPE/2-bit 58
4.4 Analysis of Fanouts Allowed 60
4.5 Misprediction Rates for DPE/PAg(12) using Profile-based Predicting Sets 64
4.6 Analysis of Fanouts Allowed using Profile-based Predicting Sets . . 66

List of Tables

2.1 Variations of Two-Level Adaptive Branch Prediction. 19

2.2 Summary of Branch Prediction Schemes 30

3.1 Benchmark Descriptions 34

3.2 Program Statistics 35

3.3 Branch Misprediction Rates for Various Predictors 36

3.4 Top 20 Referenced Patterns for each Benchmark 46

3.5 Accuracy and Coverage Assuming Unlimited DPE Resources 50

4.1 Number of Fanouts and the Percentage of the Total Executions . . 61

5.1 Misprediction Rate of the Patterns that performed DPE 68

5.2 Summary of DPE/PAg 69

5.3 Summary of the Best Branch Prediction Schemes 70

Chapter 1

Introduction

Recent trends in the development of high performance processors have seen an increase in pipeline depths as well as an increase in the number of instructions issued each cycle. While these trends provide additional capabilities to improve execution performance, each increase tends to exacerbate the negative impact caused by changes in control flow. Deeper pipelines have the potential for higher throughput, where throughput is a measure of the number of instructions a processor can finish executing per unit time. Deeper pipelines have shorter clock cycles since each stage requires less work, which allows for instructions to be completed closer together in time. Superscalar processors have been designed to have multiple instruction pipelines allowing multiple instructions to be issued per cycle. However, there must be a steady supply of instructions in order to maintain efficient use of the greater number of pipeline stages and multiple pipelines.

To keep a pipeline full, processors must locate enough instructions which are independent (those that do not depend on the results of each other). Programs that contain these independent instructions are said to have instruction level paral-

lelism (ILP). Similarly, architectures that allow multiple independent instructions to be executed at the same time are said to exploit ILP. Thus, normal sequential execution can be replaced by executing instructions in an order that exploits instruction level parallelism. Previous studies [8, 12] have shown, however, that there are limits to the amount of parallelism that can be found within programs, often limited by the basic block size. Scheduling instructions to be run in parallel can at times be difficult, especially at points in the program in which there is the possibility of a change in control flow. In order to understand the effects of control flow, we must examine the pipeline structure. An example pipeline may contain four pipeline stages: Instruction fetch (IF), Instruction decode (ID), Execute (EX), and Write-back (WB). Changes in control flow create the need for a decision to be made as to which instruction to fetch next so that the fetching stage can load the next instruction to begin processing. The decision process due to a conditional branch may delay the fetching of the next instruction until the branch condition is resolved. This may hinder the steady stream of instructions.

Conditional branches cause this delay because the next fetch address is unknown. Normally, the instruction to be executed next is the instruction that follows sequentially. However, a conditional branch has two possible instructions that could follow. When a conditional branch is processed, the next instruction could be fetched from the next consecutive instruction address, called the *fall through* instruction, or fetched from the target address, the *target* instruction. The branch problem arises due to the need for a conditional branch to wait until the condition is resolved and the next address calculated (in the Execute stage) before the next instruction can be fetched. This causes a delay in the processor and degrades performance.

Due to the necessity to stop processing and wait until the direction of the branch has been discovered, stalls are introduced in the pipeline. The number of stalls necessary is determined by the number of pipeline stages between the fetching stage and the stage in which a branch is resolved. Figure 1.1 illustrates the stalls a branch produces in a sample four stage pipeline. Both the high-level code and its equivalent assembly-level code are shown. The conditional branch is fetched during cycle 3. Two stalls must then be inserted to wait until after the branch has been executed and the branch direction is known. Thus the MULT instruction cannot be fetched until cycle 6, after the branch has completed the EX stage. The penalty (in terms of instruction slots wasted because of the branch) can be written as:

$$BranchPenalty = \#of\,stalls * \#of\,branches\,executed \tag{1.1}$$

Figure 1.2 shows the increased branch penalty in a four issue superscalar processor. All pipelines must halt processing until the branch is resolved, creating four times the number of stalls. Performance also degrades as the pipeline grows deeper since the number of stages before the branch is resolved increases. This increase of stages between the fetch and execute directly leads to more cycles being wasted while the direction of the branch is decided. The trend towards processors with deeper pipelines and wider issue width can be seen by examining some real machines. The 80486 [2] uses a pipeline with 5 stages and an issue width of 1 and the Pentium has 5 stages with an increased issue width of 2. The more recent PentiumPro [2] has 14 stages in its pipeline and has increased its issue width to 3.

a)

```
d = x + y;
if (d != 0)
      z = a * b + d;
```

b)

```
            ADD     x, y, d
            CMP     d, 0
            BE      skip
            MULT    a, b, r1
            ADD     r1, d, z
skip:
```

c)

CYCLE

	1	2	3	4	5	6	7	8	9	10
IF	ADD	CMP	BE	—	—	MULT	ADD			
ID		ADD	CMP	BE	—	—	MULT	ADD		
EX			ADD	CMP	BE	—	—	MULT	ADD	
WB				ADD	CMP	BE	—	—	MULT	ADD

Figure 1.1: Simple Pipeline without Branch Prediction

	CYCLE						
	1	2	3	4	5	6	7
IF	ADD	—	—	MULT			
	CMP	—	—	ADD			
	BE	—	—				
	—	—	—				
ID		ADD	—	—	MULT		
		CMP	—	—	ADD		
		BE	—	—			
		—	—	—			
EX			ADD	—	—	MULT	
			CMP	—	—	ADD	
			BE	—	—		
			—	—	—		
WB				ADD	—	—	MULT
				CMP	—	—	ADD
				BE	—	—	
				—	—	—	

Figure 1.2: Superscalar Pipeline without Branch Prediction

An early approach to reducing branch penalty involved the use of delay slots. Delay slots are the stall slots that come after a branch that go unused since a processor must wait to find out the path of the branch. Delay slots will occur regardless of the branch outcome. Instructions that come before the branch can be placed after the branch provided the branch does not dependent on the results of these instructions. This scheme was useful for early RISC machines with a single short pipeline since there are not many slots to fill. However, as the number of slots increases it becomes more difficult to find enough independent instructions from before the branch to fill these slots. New superscalar designs, such as the Pentium Pro, may require finding 41 instructions to fill the delay slots, making the task impossible since basic blocks average 5-6 instructions in length. The ability to locate instructions for the delay slots will only decrease as the issue width and pipeline depth of future generation processors continues to increase.

Another approach to reduce the branch penalty is to allow the stall slots to be filled with instructions from the most likely execution path. Branch prediction fetches the next instruction and speculatively executes it before the branch is resolved. Branch prediction is the technique of guessing the direction that a conditional branch will take. The direction of a branch is defined to be NOT-TAKEN if the fall through instruction should be executed next and TAKEN if the target instruction follows. The advantage comes when the prediction is correct causing the stalls to be filled with useful instructions. Figure 1.3 shows the effects of a correctly predicted path. The instructions that are speculatively executed are in italics. The stalls that usually must be inserted are no longer necessary because the slots have been filled with the instructions that would have followed anyway. If the path is incorrectly predicted however, any effects of the instructions that

CYCLE

	1	2	3	4	5	6	7	8
IF	ADD	CMP	BE	*MULT*	*ADD*			
ID		ADD	CMP	BE	*MULT*	*ADD*		
EX			ADD	CMP	BE	*MULT*	*ADD*	
WB				ADD	CMP	BE	*MULT*	*ADD*

Figure 1.3: Simple Pipeline using Branch Prediction

were processed on the mispredicted path must be undone and then the proper instructions must be brought into the pipeline.

Using branch prediction, the penalty incurred by a branch gets redefined as:

$$BranchPenalty = C * Tb + misprediction\ penalty * Mb \qquad (1.2)$$

where C is the cost of executing a branch and Tb and Mb are the total number of branches executed and the number of mispredicted branches respectively. The misprediction penalty is the time it takes to undo any necessary instructions and the number of cycles that processed useless instructions. The number of mispredicted branches is greatly dependent on the accuracy of the method used to predict the branch path. Thus, a predictor with extremely high accuracy is needed to take advantage of the potential of high performance processors.

A third method aimed at reducing the costs associated with branch instructions, is referred to as Dual Path Execution. Dual Path Execution executes the instructions down both the taken and not taken paths. Once the branch is resolved, all results on the incorrect path must be discarded. This method often is not realizable however, since one branch may lead to another branch before the first is resolved leading to a cascade of speculative execution paths (each competing for available processor resources). The advantage comes if branches do not occur close together producing a constant availability of resources and eliminating branch penalty.

1.1 Contributions

1. In this thesis, I examine techniques to assign branch prediction confidence to branch patterns of current branch predictors.

2. I study the idea of partial coverage predictors and their effects on branch penalty. Partial coverage predictors are predictors that do not give a prediction for every branch.

3. I will propose a new hybrid model combining branch prediction and dual path execution, which targets the dual path execution to only those patterns which are poorly predicted. This should reduce the cascade effect while retaining most, if not all, of the benefits of both branch prediction and dual path execution.

4. I provide analysis to show the validity of the performance increase in executing down both paths of a branch.

5. I will examine the effects of branch resolution on the hybrid dual path execution scheme.

1.2 Organization

The rest of this thesis is organized into five chapters. Chapter 2 gives a background on various branch prediction methods and their accuracies that have been proposed and implemented over the years. Chapter 3 describes the experimental methodology, the benchmarks used in the simulations, and some introductory statistics of these programs. The design of the proposed hybrid branch predictor is also given. Chapter 4 gives the performance results of the proposed predictor and substantiates its usefulness. Chapter 5 discusses the conclusions and future work.

Chapter 2

Managing Branch Penalty

As pipelines get deeper and the issue width of processors increases, the adverse effects of branches on performance grows. Branch prediction can reduce the penalty caused by branches by fetching instructions from the predicted path before the branch is resolved. Many branch prediction strategies have been explored over the years. These methods fall into two categories, static prediction and dynamic prediction. The next two sections will examine the common static and dynamic schemes. This is followed by describing the alternate strategy for reducing branch penalty called dual path execution.

2.1 Static Schemes

Static branch prediction techniques reduce branch penalty by speculatively executing instructions from the path most likely to be taken once the branch is resolved. Static branch prediction uses information that can be obtained at compile-time to determine the most frequent direction the branch takes. Thus, these schemes make the same prediction every time a particular branch is encountered.

A simple static branch predictor scheme is to predict that the branch will always be taken. This is called the *Branch Always* method. This method usually increases performance since it has been found that the majority of branches are taken. The results, however, vary greatly between different programs based on the programmer awareness and the compiler. An average of 60% accuracy [16] is obtained using this technique. Programs with many loops may even obtain accuracies as high as 90% or more.

An improved static scheme is to use the direction to the target address as a means of predicting the branch [9]. If the branch is a backward branch, the target address has a lower address than the branch address, the branch is predicted to be taken. Forward branches are predicted to not be taken. This scheme also targets loops in order to achieve its higher accuracy. Loops are a common occurrence in programs and have a strong tendency to be terminated by backward conditional branches that are disproportionately taken. Thus, this approach works well for loop-intensive programs. It achieves an average of 70% accuracy across a variety of benchmark applications [16].

A branch can also be predicted based on the opcode of the branch [1]. A study showed that "branch if negative", "branch if equal", and "branch if greater than or equal" are usually taken. This is due to these types of branches usually being used to terminate loops. This scheme tends to outperform the previously described methods by averaging 75% accuracy [16].

Information gathered from profiling has been used to decrease the effects of branches on performance. The profile of a program gives the number of times that a basic block and control flow direction is executed [13]. The technique of profiling [4, 5, 6] uses data gathered from previous runs to determine the path

tendencies of branches. Most branches have a strong bias towards being either mostly TAKEN or mostly NOT-TAKEN. These biases can be found by running the program and keeping track of their patterns. This information is then used to predict the path of a branch and initiate instruction execution on this path when running the program in the future. Profiling helps increase performance, however it can not adapt to changes in the path direction of branches during the execution of a program. Another drawback to profiling is the time and effort it takes to run the program prior. This effort will prove to be worthwhile, however, if the program is one that is to be run frequently. The average accuracy of this scheme was found to be 85% [6].

2.2 Dynamic Schemes

An increase in prediction accuracy can be gained by allowing the predictor to adapt to changes during the program. Dynamic prediction methods use information gathered at run-time to determine the prediction path. Thus, the prediction can vary during the run of the program, adapting to branch patterns and changes in the program.

2.2.1 One-bit Prediction

A simple version of dynamic prediction is to make the prediction based on the direction the branch took the last time it was executed [9]. This can be done on a per-branch basis or based on the direction of the last execution of any branch. A scheme is referred to as a *global* scheme if the path of one branch affects the future prediction of all other branches; if the prediction of a branch depends only on the

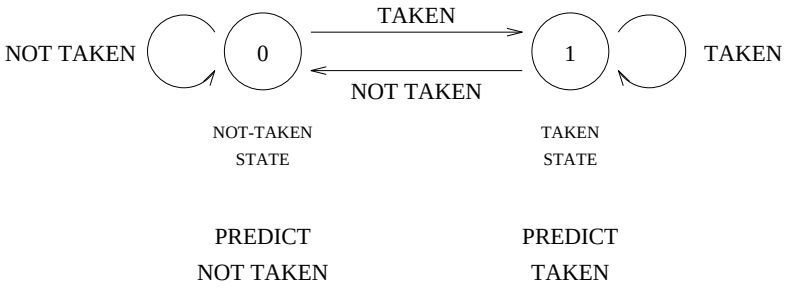

Figure 2.1: State Diagram for the One-Bit Predictor

previous history of that branch it is referred to as *local*. This scheme works well for programs with branches that have high tendencies for a particular path. Incorrect predictions are made every time the branch changes path directions. These changes in direction, however, do not occur frequently since most branches have a bias towards one direction. Thus, the one-bit tends to be comparable to the best of the static schemes since they both benefit from the same conditions, the highly biased branches. The state diagram for this predictor is shown in Figure 2.1. If the branch is in state 0, the *not-taken* state, a NOT-TAKEN prediction will be made. State 1, the *taken* state, works similarly. The actual path of the branch determines the state the counter is set to next. This scheme helps increase performance for programs that are loop-intensive since loops tend to go in one direction for several iterations. This simple dynamic scheme obtains an accuracy averaging 85% [16].

2.2.2 Two-bit Counter

Further improvement in prediction accuracy can be made by increasing the amount of history that is kept for each branch. Two-bit saturating up-down counters are employed to maintain information about previous branch activity. Figure 2.2

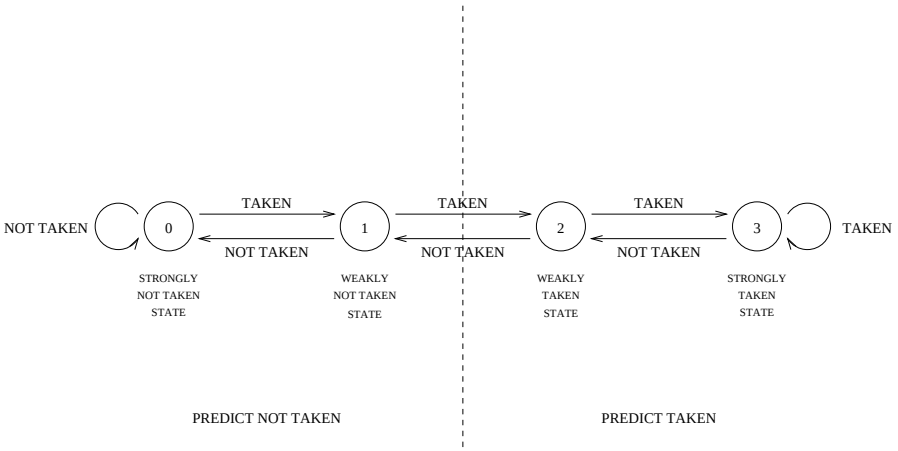

Figure 2.2: State Diagram for the Two-Bit Predictor

shows the state transitions for this approach. A prediction is made by looking at the current state. If the state is 0 or 1 then a NOT-TAKEN prediction is made. These states are referred to as the *strongly not taken* state and the *weakly not taken* state respectively. States 2 and 3 make a TAKEN prediction and are called the *weakly taken state* and the *strongly taken state* respectively. After the branch has been resolved, if the branch was taken the state is increased by one, saturating at state 3. If the branch was not taken the state is decremented without going below state 0. The use of both strongly and weakly taken/not-taken states prevents an anomalous change in direction from affecting the prediction path. Keeping track of more history provides a way to differentiate between more patterns allowing for a higher accuracy.

Figure 2.3 shows an example of how the added history can benefit the prediction accuracy. The predictions, states, and correctness are followed for the branch pattern *111011* using both the 1-bit and 2-bit predictors. The first column shows the history of the pattern encountered thus far. The second column gives the

BRANCH PATTERN = 111011

		ONE-BIT			TWO-BIT		
HISTORY	PATH	CURRENT STATE	PRED.	CORRECT	CURRENT STATE	PRED.	CORRECT
	T	TAKEN (STATE 1)	T	√	WEAKLY TAKEN (STATE 2)	T	√
1	T	TAKEN (STATE 1)	T	√	STRONGLY TAKEN (STATE 3)	T	√
11	T	TAKEN (STATE 1)	T	√	STRONGLY TAKEN (STATE 3)	T	√
111	NT	TAKEN (STATE 1)	T		STRONGLY TAKEN (STATE 3)	T	
1110	T	NOT-TAKEN (STATE 0)	NT		WEAKLY TAKEN (STATE 2)	T	√
11101	T	TAKEN (STATE 1)	T	√	STRONGLY TAKEN (STATE 3)	T	√

Figure 2.3: One-Bit versus Two-Bit Predictor

path the branch takes at that point in its execution. The next columns show the current state for the 1-bit predictor, the prediction this scheme would make given the current state, and whether the prediction was correct. The same information is also shown for the 2-bit predictor. Since branches are taken more frequently than not taken, both predictors are originally set to a *taken* state. The example begins with the path being TAKEN. With the current state set to the *taken* state, the 1-bit scheme would predict TAKEN which is correct as seen by the check mark. The 2-bit predictor beginning in the *weakly taken* state would also correctly predict TAKEN. The second row shows that the history now contains *1* and the path at that point is TAKEN. The state for the 1-bit predictor remains in the *taken* state and thus it again predicts TAKEN and is correct. The 2-bit predictor was moved into the *highly taken* state. The prediction for this state is TAKEN

which is correct. Row 3 follows similarly. In row 4, the history is *111* and the branch will be NOT-TAKEN at that point. The 1-bit predictor is still in the *taken* state and thus it predicts TAKEN since the branch was taken the last time it was executed. This however is incorrect. The 2-bit predictor predicts TAKEN and is also incorrect. Row 5 shows the results after the history *1110* has been seen and the branch is being TAKEN. The 1-bit predictor was moved into the *not-taken* state due to the last path being not taken. This state predicts NOT-TAKEN assuming it will take the same path as last time. This is incorrect again. The 2-bit predictor continues to predict TAKEN even though the state has been changed to *weakly taken*. This allows the 2-bit scheme to make a correct prediction this time. Thus, one can see that the overall pattern *111011* is overwhelmingly taken. Being constrained to predict based on only the last path causes the 1-bit predictor to incorrectly predict twice. The 2-bit predictor assumes that the first change in direction is an anomalous change and continues to predict the same direction. The history shown is one that is often found in loops. One direction is seen many times in a row, then a jump out of the loop occurs which predicts the opposite (*1110*). Later, the branch may be executed again and the same pattern may be seen. Thus the accuracy of the 2-bit predictor can achieve higher accuracies for loop-intensive programs as well as those in which branches have strong tendencies towards one direction. It averages an accuracy of around 92% [16].

2.2.3 Two-Level Adaptive

Yeh and Patt proposed a two-level adaptive scheme [14, 15] to take advantage of history-based branch correlation. The first level keeps track of the direction of the last k branches. This information is kept in a register called the Branch History

Register (BHR). The pattern seen by the history of the first level is used as an index into the second level. The second level maintains the branch direction of the last j occurrences of a particular pattern (based on the first level). Therefore the second level maintains the branch direction of the last j occurrences of the specific history pattern of the last k branches. A Pattern History Table (PHT) is used to hold this behavior. The prediction is most commonly made by a two-bit saturating up-down counter based on the pattern history bits, much like the two-bit counter described previously. A counter is associated with each entry in the PHT. Figure 2.4 shows the organization of the two-level adaptive scheme. In the example, the last k branches encountered had a history of 11...01. This history is then used to index into the pattern history table which keeps the most recent paths of when this pattern was encountered. The two-bit counter uses this pattern history to make predictions. After the branch is resolved the BHR is shifted left, placing the actual path of the branch in the most recent history position. Also, the state of the two-bit counter for the branch history pattern is updated.

The two-level adaptive approach has nine variations that are based on changing the associativity of both levels [14, 16]. The possibilities for the first level are global (G) which means that k is the last of all branches encountered and that all the branches map to the same BHR. The next two options use a table of branch history registers called the Branch History Register Table (BHRT). The first of these is per-address (P) which implies that every branch has its own BHR which contains the last k times that particular branch was executed. This variation, however, may sometimes not be realizable since one can never know how many branches are in every program. The third choice is per-set (S) which maps a set of branches to the

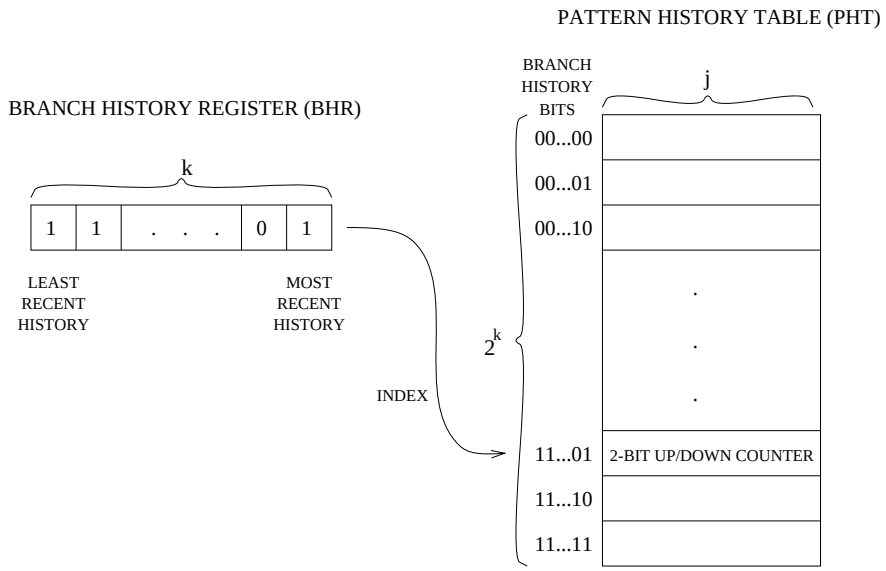

Figure 2.4: Structure of the Two-Level Adaptive Branch Predictor

same BHR and thus the BHR holds the history of the last k branches encountered of that set.

These three options are also available for the second level. Global (g) would mean that all BHRs would use the same pattern history table. Per-address (p) would supply a different PHT for each BHR, and per-set (s) would give one table for each set of BHRs. Table 2.1 summarizes these variations. The number of bits used in the BHR are expressed in parenthesis after the variation type, as in GAg(12), which implies that one global BHR was used, keeping the history of the last 12 paths encountered in the program. Set associative variations are denoted as PAs(6,16) which keeps 6 bits of history for each of the 16 branch sets. The structure of each of these variations is shown in Figures 2.5, 2.6, and 2.7.

Global history schemes allow for the correlation between different branches to be used in the prediction. Thus, this set of schemes works well for branches in

Variation	Description
GAg	Global Adaptive Branch Prediction using one global pattern history table.
GAs	Global Adaptive Branch Prediction using per-set pattern history tables.
GAp	Global Adaptive Branch Prediction using per-address pattern history tables.
SAg	Per-set Adaptive Branch Prediction using one global pattern history table.
SAs	Per-set Adaptive Branch Prediction using per-set pattern history tables.
SAp	Per-set Adaptive Branch Prediction using per-address pattern history tables.
PAg	Per-address Adaptive Branch Prediction using one global pattern history table.
PAs	Per-address Adaptive Branch Prediction using per-set pattern history tables.
PAp	Per-address Adaptive Branch Prediction using per-address pattern history tables.

Table 2.1: Variations of Two-Level Adaptive Branch Prediction.

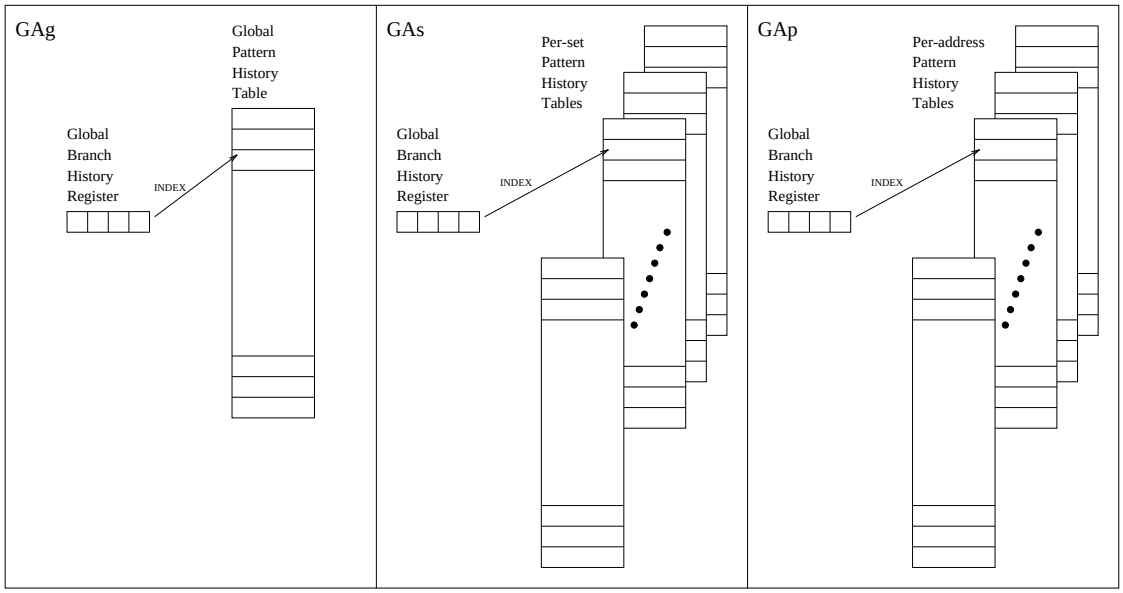

Figure 2.5: Structures for the Global Two-Level Adaptive Branch Predictors

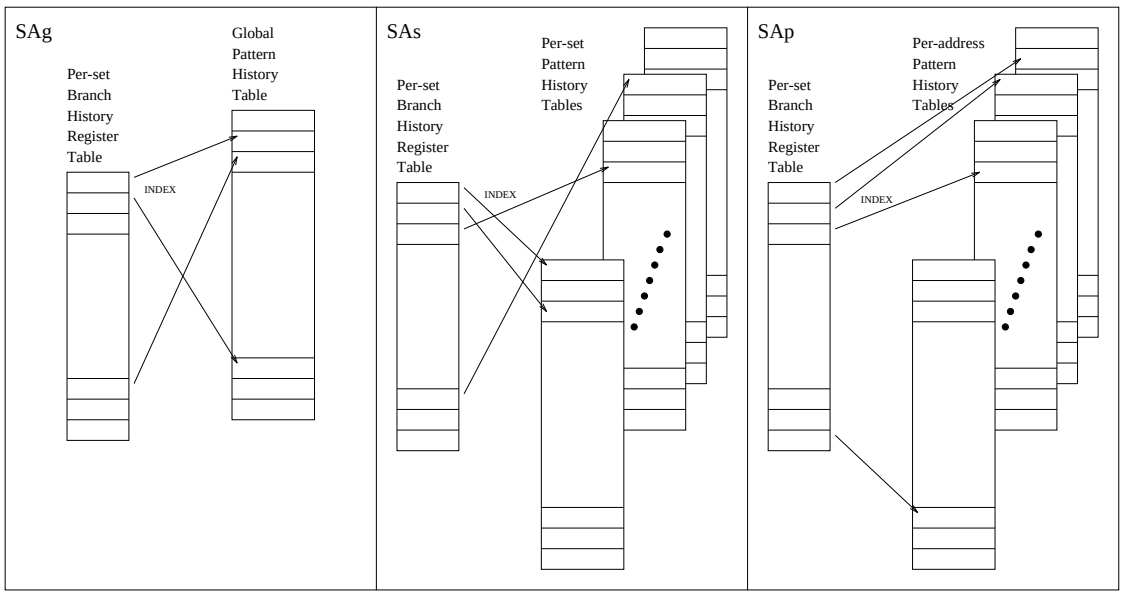

Figure 2.6: Structures for the Per-set Two-Level Adaptive Branch Predictors

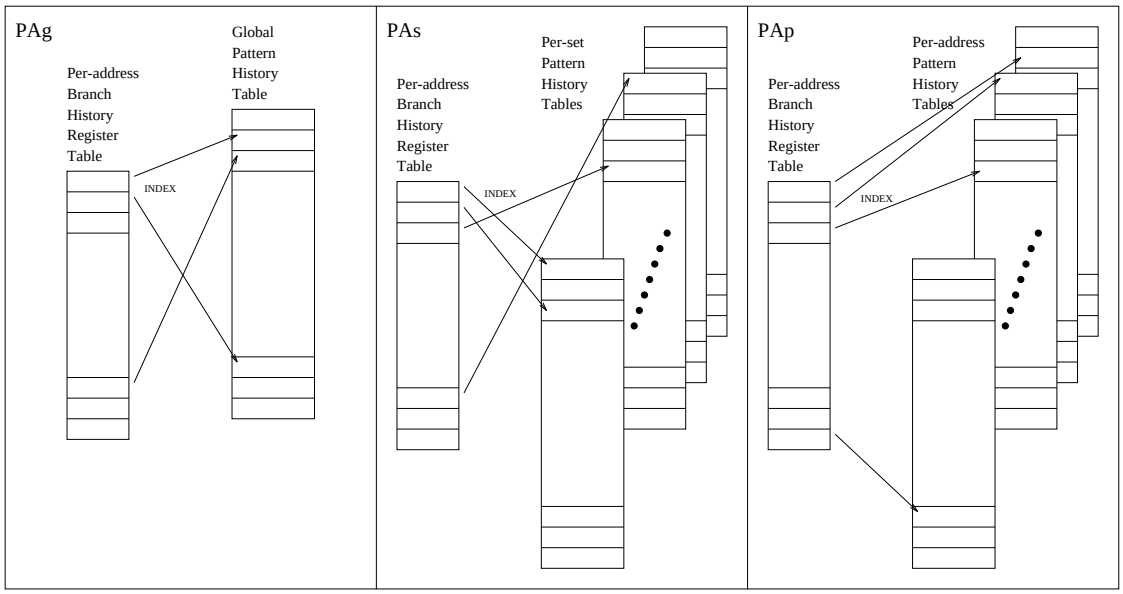

Figure 2.7: Structures for the Per-address Two-Level Adaptive Branch Predictors

which the path of one branch depends on the result of a previous branch. Figure 2.8 gives some example code which illustrates how the results of a particular branch can depend on the results of prior branches. In this example, if branch 1 is not taken then a is set to 1. If branch 2 is not taken then b is set to 1. This implies that if both branch 1 and 2 were not taken, branch 3 would have to be taken since a and b would be equal. Global history schemes tend to increase in performance as the number of history bits increases. They are also affected by the number of pattern history tables used. Since using a global branch history combines the history of all the branches, increasing the number of history bits or the number of pattern tables tends to lessen the interference between patterns. Though, global history schemes achieve high accuracy, they require a long history and a large amount of pattern history tables to maintain this accuracy, thus their cost can be great. Per-address schemes work well for branches that have very distinct, predictable patterns and

```
B1:     if (x == 0)
            a = 1;
B2:     if (y == 0)
            b = 1;
B3:     if (a != b)
            x++;
```

Figure 2.8: Example of Branch Correlation

programs which do not have much correlation between branches. The per-address history designs are not as affected by the increase in history bits and the number of pattern tables and therefore cost less to be effective. Per-set schemes allow for some correlation while remaining sensitive to the patterns of particular branches. Per-set schemes also require high cost in order to alleviate interference although their accuracy is not as affected to the number of pattern history tables as the global schemes. PAs was found to be the best design for the cost. It can achieve an average of 96% accuracy.

2.2.4 Gshare

Gshare is a variant on the two-level scheme which McFarling [7] introduced as a way to use both the branch address and the global history to gain more accuracy than pure global schemes. Global history information does not separate the identity of branches which means that the prediction is made based on the current pattern regardless of the branch being executed at that time. Gshare attempts to distinguish between more combinations of patterns and branches by exclusive ORing the branch address and the global history as illustrated in Figure 2.9. If

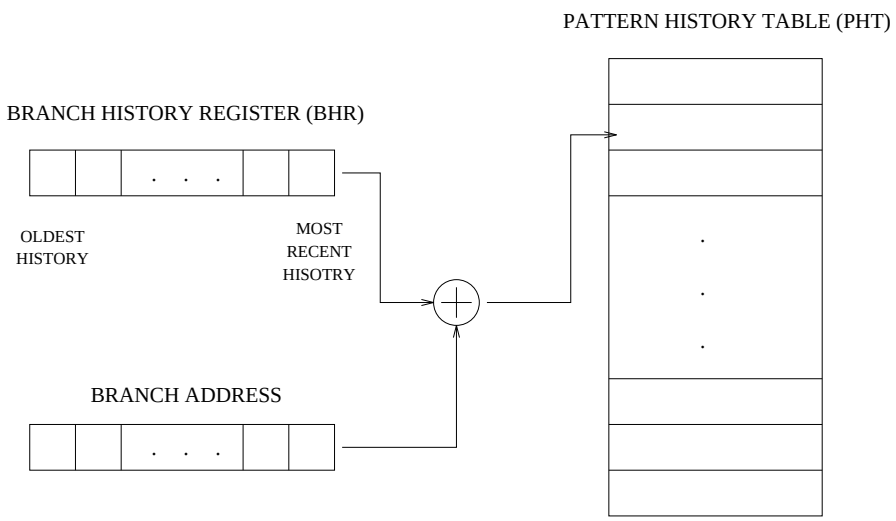

Figure 2.9: Structure of Gshare Branch Predictor

the branch address is *11110001* and the global pattern currently is *10101010*, the index into the PHT would be *01011011*. Figure 2.10 shows how this method can differentiate between more branches and their histories. Shown is two branches with two history patterns each. Global history methods can only find three distinct patterns creating a conflict between of branches when the global pattern is *00000000*. Gshare can differentiate between all four cases. The advantage of gshare over the regular 2-level adaptive global schemes is that the history of branches can be differentiated some without the usual high cost.

2.2.5 Hybrid Predictors

Although the two-level adaptive approach gives average accuracies of 96%, there is still a great need for better predictors. Hybrid branch predictors were introduced as a means to use several predictors at once in order to take advantage of the best predictor at a particular time in the program. McFarling proposed the first hybrid

BRANCH ADDRESS	GLOBAL HISTORY	GSHARE
00000000	00000001	00000001
00000000	**00000000**	11111111
11111111	**00000000**	00000000
11111111	10000000	01111111

Figure 2.10: Global History and Gshare Comparison

branch predictor [7] which combines various predictor methods into a single scheme. A selector is used to determine which predictor component to use for each branch. Hybrid predictors are useful in combining the strengths of its single predictors and thus achieving even higher accuracy [3]. McFarling's selector keeps track of which of the two predictors being used has the highest accuracy for that branch thus far, and then uses that prediction. After the branch is resolved the selector is incremented or decremented based on which predictor was correct. If they both performed the same, no change would be made to the selector's counter. Figure 2.11 illustrates the selector that McFarling used. The first two columns show the possible update options. A *0* indicates that the predictor made an incorrect prediction, and a *1* implies that a correct prediction was made. The third column shows what happens to the selector counter after the branch is resolved. For example, if predictor 1 was incorrect and predictor 2 made a correct prediction, the selector counter would be decremented. A positive selector counter means that predictor 1 has been correct

P1 CORRECT	P2 CORRECT	P1 CORRECT - P2 CORRECT
0	0	0
0	1	-1
1	0	1
1	1	0

Figure 2.11: Hybrid Branch Predictor Selector

more often than predictor 2. A negative counter means that predictor 2 has been correct more often. A hybrid predictor consisting of gshare and PAs was found [3] to achieve a prediction accuracy of around 97%.

2.3 Dual Path Execution

To reduce the branch penalty to zero, dual path execution can be implemented. Dual Path Execution (DPE) executes the code on both paths of a branch. Due to one branch leading to another, a cascading effect can occur causing the resources that would be necessary for this method to be extremely high and even unrealizable in particular implementations or programs. In order to limit the cost necessary for dual path execution Uht [11, 10] proposed Disjoint Eager Execution (DEE) which limits the number of concurrent executing paths by using a selector to decide which paths to execute. The overall probability of each path is calculated; this is referred to as *cp*, the cumulative probability. The probabilities are figured by taking the product of the local path probability and the local probabilities of each of the path's

predecessor paths. A local probability is defined to be the probability of the path being taken with respect to its predecessor path. Figure 2.12 illustrates the design of both the dual path execution model and the disjoint eager execution scheme. Examples of cumulative probabilities are shown next to the path for the disjoint eager execution model. The circled numbers represent the order that resources are assigned to paths. The paths with the highest overall *cp* are chosen to be executed. In the example, if there were 6 resources available all of the darkened paths would be undergoing execution concurrently. If only 5 resources were available, the path marked with the 6 would need to wait for a free resource. In the DPE model, the resources become filled with instructions that will have to be undone. Since only one path at each node is actually taken, over half of the effects of the resources will be negated. The positive aspect of this scheme is that there is no branch penalty since no prediction is being done. In comparison, DEE has the potential for increasing performance by concurrently executing farther down a path than DPE. If the paths that had the highest *cp*s were the paths that were actually taken, then the execution would proceed 2 branches farther down the path with fewer instructions needing to be undone. Disjoint eager execution was shown [11] to perform better than other predictors and dual path execution for the same cost.

2.4 New Branch Prediction Approaches

This thesis will combine the Dual Path Execution model with a current branch predictor to form a hybrid predictor that can use the strengths of each. Dual path execution will be limited by assigning confidences to patterns. A high confidence implies that the pattern is predicted well and thus the prediction should be used.

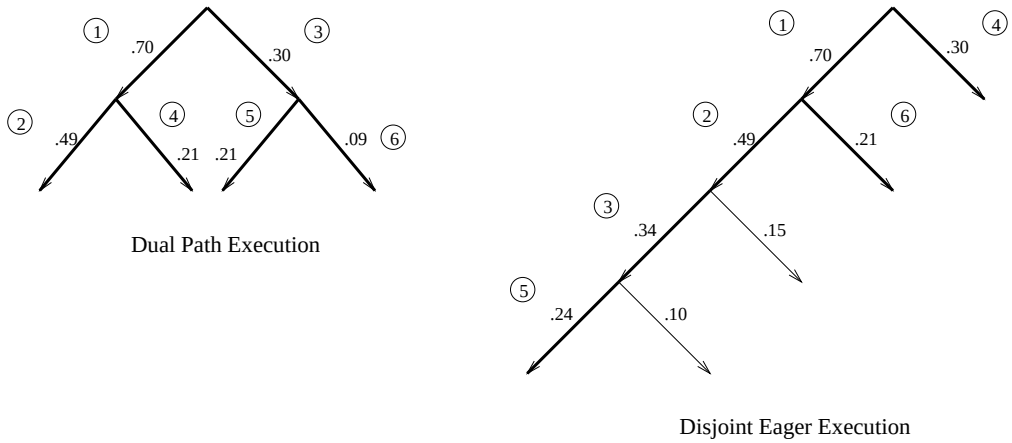

Figure 2.12: Comparison of Dual Path Execution and Disjoint Eager Execution

A low confidence corresponds to a pattern which has a relatively low average prediction rate. Rather than using the prediction, execution will continue down both paths of the branch. The number of consecutive branches which can implement dual path execution will be varied to analyze the effects of the number of available resources. Hybrid predictors using Dual Path Execution combined with PAg and combined with a two-bit counter scheme will be examined. These schemes have the distinct advantage of using a proven predictor for the patterns that predict well. Many patterns do not predict well, however, and rather than using a predictor that has been shown to have a low accuracy in these cases, there is an assurance to begin execution in the correct direction by performing DPE at this point. The second combination, DPE/two-bit, will also use the average prediction accuracy of patterns to limit DPE. Since the patterns that will be predicted have high prediction accuracies, using a more simple predictor can save on cost.

2.5 Summary

Table 2.2 shows a summary of the static and dynamic predictors described. Their average prediction accuracies are shown as well some of the requirements of the predictor. n is the number of branch instructions. Static predictors tend to not need as much storage space yet do not give as high of accuracies. Dynamic predictors use more space, however often the amount of space required is impossible since there is no bound on the number of branches a program may contain. Thus there may be a set of branches that hash to the same history bit which can create predictions based on erroneous information. By allowing the predictor to find patterns and change its prediction as the program runs, dynamic predictors can average much higher accuracies. Because of this, dynamic predictors are used more often than static predictors in high performance processors despite their higher cost. When choosing which dynamic predictor to use, cost has been a major factor. However, as more and more transistors are able to fit on a chip, the amount of hardware that is allocated to a branch predictor increases. Thus, the cost factor is becoming less important. Another trend that has had a large influence in choosing a predictor is the progression towards deeper and wider pipelines. In order to be able to see the performance increase these processors are capable of, a constant stream of instructions is needed. Although branch predictors are reaching 97% accuracy, branches are still often one of the major limiting factors in these high performance processors. Thus, designers are willing to spend even more hardware dedicated to branch prediction. The Dual Path Execution and Disjoint Eager Execution schemes target these ideas. Many more resources are necessary for these models to be implemented, yet the argument for DEE is that the cost is offset by

the performance increase. Overall, the predictors shown give a variety of schemes to predict branches. The techniques to be described in this thesis use ideas from many of these different predictors. The DPE hybrid model will show that current implementations can be combined to achieve even higher accuracy. The second model, the limiting two-bit predictor, will take the opposite approach, letting cost be a factor.

Predictor scheme	Type	Reference	Average Accuracy	Cost
Branch Always	static	[16]	60%	0
BTFNT	static	[9, 16]	70%	0
Opcode based	static	[1, 16]	75%	0
Profiling	static	[4, 5, 6, 13]	85%	0
One-bit	dynamic	[9, 16]	85%	n
Two-bit	dynamic	[16]	92%	2n
GAs(11,32)	dynamic	[16]	97%	128K
PAs(6,16)	dynamic	[16]	96%	8K
gshare(12)	dynamic	[7]	93%	8K
gshare(18)/PAs(15,4)	dynamic	[3, 7]	97%	128K

Table 2.2: Summary of Branch Prediction Schemes

Chapter 3

Assigning Confidence to Branch Prediction

Hybrid branch predictors have been introduced to exploit the strengths of its single scheme components. Selectors are used to determine which component is more confident in its prediction. In the hybrid predictor proposed in this thesis, confidences will be assigned to the branch pattern and a selecting set will be used to determine whether to predict or perform dual path execution. This section first describes the system used to gather and analyze the data. Then the background of the design of this new predictor will be discussed and the details of the predictor will be given.

3.1 Methodology

The following subsections describe the experimental system used to gather the data for the results of this thesis. Descriptions of the benchmarks used and some program statistics for each will also be given.

3.1.1 Experimental System

The experiments for this thesis were conducted using UltraSPARC model 170E systems. The data was gathered using a tracing tool developed at SUN Microsystems Laboratories called Shade. Shade provides exhaustive dynamic information at the instruction-level by dynamically translating the code of the application into host machine code while incorporating analysis code to generate the traces. This new code is then directly executed to emulate and trace the application code. To enhance performance, code is cached, decreasing the overhead caused by translation. An additional benefit of Shade is that the analyzer can be restrained to only collect specified information at particular times further improving analysis time.

3.1.2 Implemented Branch Predictors

Four different types of branch predictors are implemented in order to evaluate how the accuracy of different programs are affected by differences in a variety of current prediction schemes. The data gathered will also serve as a base to evaluate the performance of the proposed predictors. The first and simplest predictor implements a per-address scheme in which every static branch is given its own two-bit saturating up/down counter. The second scheme simulates gshare(12), with twelve bits used to keep track of the first level of history. This requires a pattern history table of size 4096 indexed by the lower 12 bits[1] of the branch address XORed with the pattern. Gshare has been found to achieve high accuracies without the high cost needed for many other predictors. The last two predictors studied are varia-

[1]The two least significant bits of the branch target address in the SPARC architecture are zero because the branch targets must be aligned on a 32-bit boundary. These two bits are shifted out before XORing.

tions of the two-level adaptive scheme. Both PAg and PAs are implemented. PAg is one of the schemes used in the proposed hybrid predictor and thus comparison is vital. PAs was chosen since it is considered to be the best two-level adaptive predictor for its cost. PAg uses a per-address branch history register table. The depth of the history bits is varied between 8, 12, and 16 bits which then indexes into a global PHT. PAs(6,16) keeps 6 bits of history for each branch address. Each BHR hashes to one of 16 PHTs. There was assumed to be no contention among the BHRs with any of the predictors.

3.1.3 Description of Benchmarks

The benchmarks analyzed were those from the SPEC95 suite. Six of the integer benchmarks and three of the floating point benchmarks are used. Although *compress* is included in the Spec95 suite, Berkeley compress version 5.9 was used due to problems with running the SPEC95 version. A description of each program is given in Table 3.1. Table 3.2 shows many basic statistics of each of the programs used. The table lists the input files used, the total number of instructions executed, the number of conditional branch instructions, the total number of branches executed, and the percent of branches that are TAKEN. Most programs contain many conditional branch instructions that are never executed. These instructions are not included in the count in the fourth column. Only the branches that are executed at least once will be examined, since these are the branches that affect the accuracy of a branch predictor. The average percent of branch executions that are taken is around 60% corresponding to the average branch prediction accuracy of the static *Branch Always* predictor.

Table 3.3 shows the misprediction rate for each program when using the predic-

Program	Description
go	An artificial intelligence application which plays the game of *go* against itself.
compress	Performs file compression using adaptive Lempel-Ziv coding.
li	A LISP interpreter based on XLISP 1.6.
jpeg	A video compression algorithm.
perl	An interpreted language optimized for scanning arbitrary text files, extracting information from those files, and printing reports based on that information.
gcc	Based on the GNU C compiler version 2.5.3.
swim	Shallow Water Model using a 1024x1024 grid.
apsi	Solves for the mesoscale and synoptic variations of potential temperature, wind components, mesoscale vertical velocity pressure and distribution of pollutants.
fpppp	A Quantum chemistry application that measures performance of two electron integral derivations in the Gaussian series of programs.

Table 3.1: Benchmark Descriptions

Program	Input File	Total # of Instruction Executions (in millions)	Branch Instructions (executed at least once)	Total Branch Executions (in millions)	% Taken
go	9stone21.in	34423	7263	4296	59.24
compress	in	81	1364	12	71.12
li	*	57924	1880	9830	50.07
ijpeg	penguin.ppm	42619	2407	3534	67.73
perl	primes.pl	15450	2902	2284	54.41
gcc	1insn-recog.i	599	14782	113	57.50
gcc	1cccp.i	1344	21438	253	59.89
gcc	emit-rtl.i	173	18771	31	55.01
gcc	regclass.i	127	18660	23	56.70
gcc	explow.i	241	14969	45	56.80
swim	swim.in	273176	876	725	99.20
apsi		394597	1806	17112	64.63
fpppp	natoms.in	396365	980	5297	66.12
gcc average		497	17724	93	57.18
int average		25183	5100	3345	60.01
fp average		354713	1221	7994	76.65
total average		135026	3807	4799	65.56

Table 3.2: Program Statistics

Program	2-bit % misp	PAg(8) % misp	PAg(12) % misp	PAg(16) % misp	PAs(6,16) % misp	gshare(12) % misp
go	20.444	20.037	19.150	15.284	20.938	27.251
compress	11.815	10.955	10.550	10.154	10.601	10.006
li	10.738	5.301	3.880	2.560	5.207	4.896
ijpeg	9.051	6.239	5.905	5.571	6.294	5.855
perl	4.255	0.037	0.025	0.020	0.042	2.131
gcc	9.925	8.184	6.399	5.153	8.780	11.638
swim	0.266	0.197	0.197	0.197	0.197	0.213
fpppp	15.869	7.093	4.319	3.410	7.222	5.600
int average	11.04	8.46	7.65	6.46	8.64	10.30
fp average	8.06	3.64	2.26	1.80	3.71	2.91
total average	10.30	7.25	6.30	5.29	7.41	8.45

Table 3.3: Branch Misprediction Rates for Various Predictors

tors: 2-bit, PAg(8), PAg(12), PAg(16), PAs(6,16), and gshare(12). The mispredictions given for *gcc* are the average for the five separate inputs. *swim* achieves very high accuracies using all of the predictors. The percentage of the branches that are TAKEN for this program is extremely high, reaching over 99%. This allows *swim* to maintain a high accuracy regardless of the predictor. Since most of the branches in *swim* are TAKEN, this benchmark will not benefit from maintaining more history information or attempting to correlate between branches. For other benchmarks, it can be seen that as the number of history bits increases in the PAg scheme, the accuracy increases. Increasing the depth of history that is maintained allows more patterns to be differentiated, allowing a predictor to find the best prediction per pattern.

3.1.4 Branch Characteristics

There are many interesting characteristics that can be seen among the benchmarks that often depend upon the nature of the program. The integer benchmarks typically use more branch instructions than the floating point benchmarks, with *gcc* having a very large working set of active branch instructions.[2] *gcc* averages around 17000 branch instructions while the floating point benchmarks only typically execute about 1200 different branch instructions. It can also be seen that the floating point programs tend to execute a large number of branches throughout the run of a program. Combining this with the fact that they use fewer branch instructions, it means that a small number of instructions are executed a large number of times. Figure 3.1 illustrates the relationship between the number of branch instructions and the percentage of total branch executions. The data shown in the graphs is gathered from the predictor PAg(12). It is clear that the majority of branch executions are from a small set of branch instructions. For example, in *perl*, only 80 branch instructions, less than 3% of the total branch instructions, are responsible for over 80% of all the executed branches. This phenomena indicates that it might be beneficial if a branch predictor could target the few branches that are responsible for the majority of the executions.

The examination of the distribution of mispredictions among branches shows that even fewer branches are responsible for the majority of mispredictions. Figure 3.2 shows the correlation between the number of branch instructions and the percent of total mispredictions. For *perl*, less than 2% of all branch instructions, 55 branches, create over 90% of the total mispredictions. This again emphasizes

[2]The number of branch instructions in this thesis is defined to be the number of conditional branches instructions that are executed at least once during the run of a program.

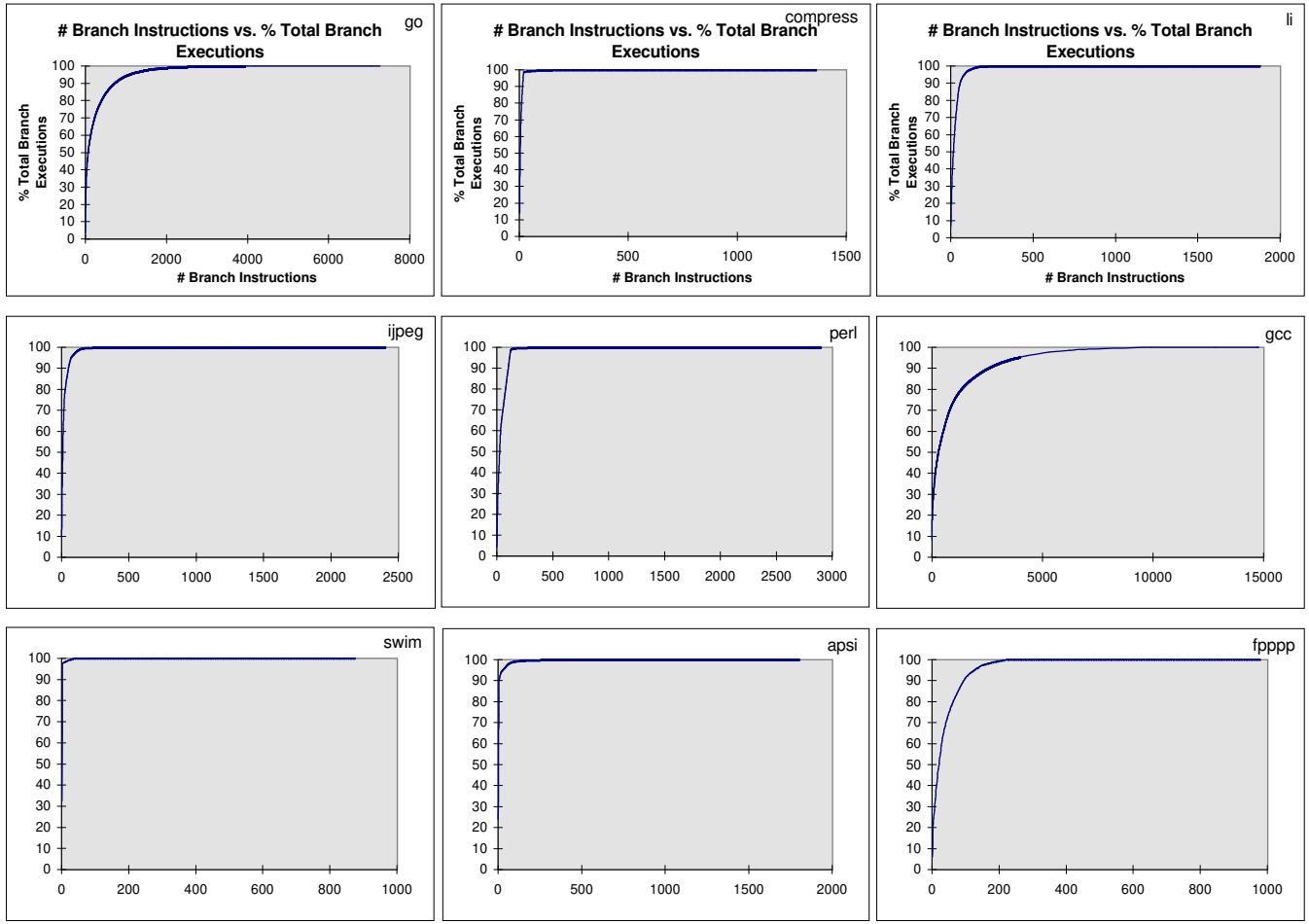

Figure 3.1: Branch Instructions and Total Branch Executions Relationship

the importance of identifying the branches that are generating the largest disturbances in the prediction accuracy. Thus, it is extremely valuable to the accuracy of a predictor, if the branches which are referenced often or have high misprediction rates are discovered and targeted.

Static branch predictors have made efforts to target certain types of branches; it was seen that this effort was still not obtaining high accuracies. Thus, this thesis focuses on targeting patterns. The history of a branch creates a pattern. The two-level adaptive approach attempts to find the best prediction for each pattern. Increasing the depth of history has allowed for more patterns to be differentiated; however, many of these new patterns may not be referenced often. The characteristics of branches examined previously can also be seen when looking at patterns. In *perl*, using the predictor PAg(12), 4 patterns generate over 90% of the total references. It has been shown that branches are highly biased to a particular pattern or direction and since branches may map to more than one pattern throughout the execution of a program, targeting patterns can encompass more branch references and branch mispredictions.

3.2 The Hybrid Approach

In an effort to achieve higher accuracies, new branch predictors are introduced that target a variety of specific characteristics. Focusing on the tendency for branches to more often be TAKEN than not, led to the predictor *Branch Always*. Other early predictors targeted particular branches, based on opcode or target instruction direction, to decide when to predict TAKEN. Dynamic predictors have begun to target history characteristics of both branches and patterns in an effort to

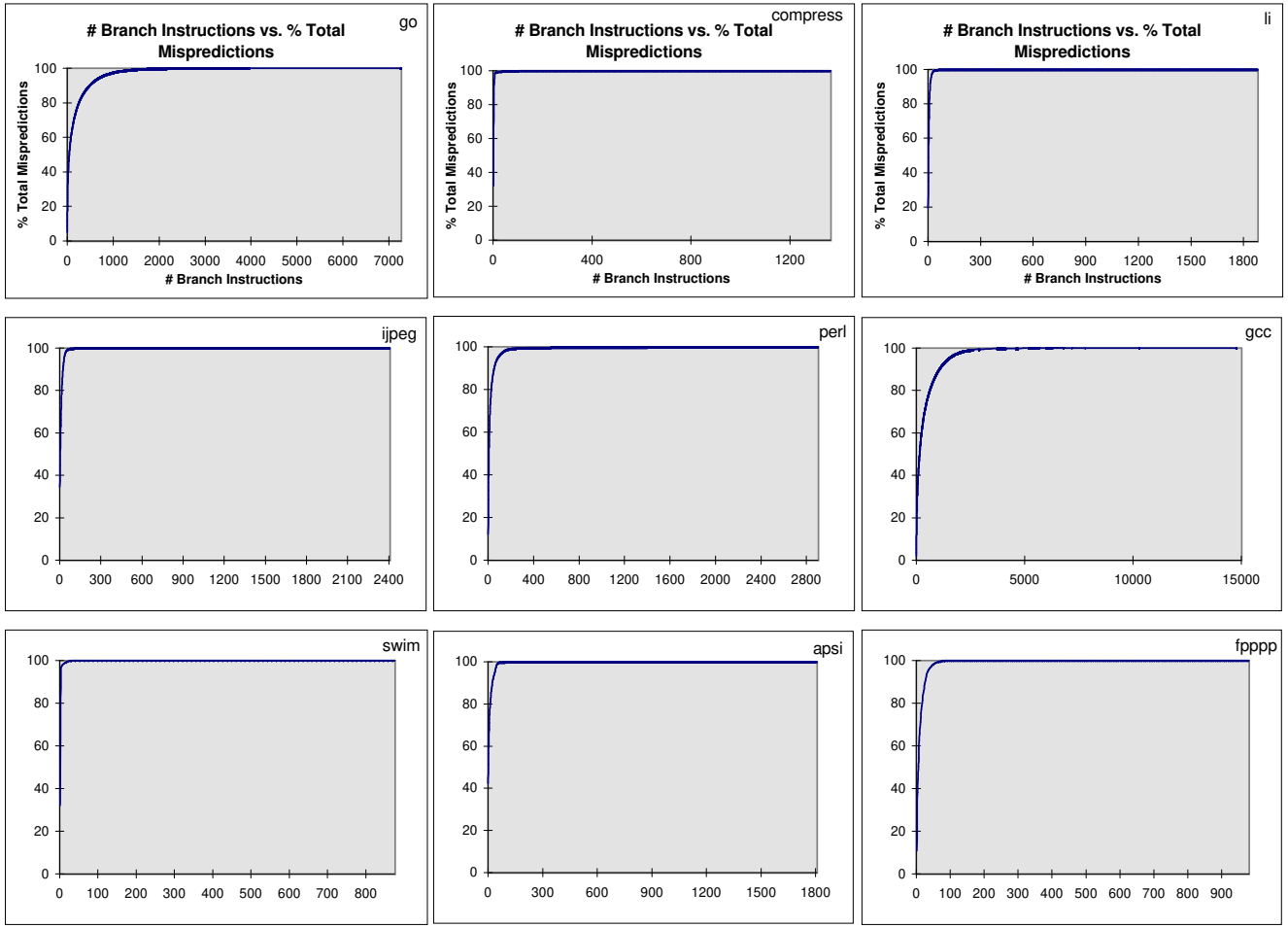

Figure 3.2: Branch Instructions and Total Misprediction Relationship

find the best prediction for a particular instance. This thesis introduces a hybrid predictor which targets specific patterns. However, rather than using history as a means of identifying which patterns to target, the number of references and the misprediction rate will be used.

As described in Section 2.3, Dual Path Execution is a technique which eliminates branch penalty. This technique is quite beneficial in that the correct path always begins execution immediately after the branch instruction. The drawback is an increase in resources necessary to fetch instructions from both the target and fall through paths. If branches occur too close together, then two instruction paths may be insufficient requiring a stall or prediction. Disjoint Eager Execution provides the benefit of using a set of criteria to limit how often DPE, or a fanout, will occur. This thesis attempts to find some optimal set of patterns, based on reference count and misprediction rate, to use as a limiting factor for DPE. This is accomplished by separating the patterns into a predicting set and a DPE set. The predicting set consists of those patterns which are anticipated to have good prediction accuracy, while the DPE set contains the remaining patterns.

The prediction scheme being proposed combines DPE and a current prediction model into a hybrid scheme. This thesis looks at two different combinations, DPE/PAg and DPE/2-bit.

3.2.1 DPE/PAg

The DPE/PAg scheme uses the same prediction structure as PAg. When a branch occurs, the associated BHR is examined to determine whether the branch history, or associated pattern, contained in the BHR is in the predicting set. If the pattern is found to be in the predicting set, a prediction is made and execution continues

down the predicted path. If the pattern is not in the predicting set, implying that the pattern is one that usually does not predict very well, then the ability to fanout is evaluated. If there is currently only a single instruction path (we are not in a fanout state), then execution begins down both paths of the branch. If a prior fanout has occurred within the branch resolution time, however, then a prediction must be made. A prediction is made by examining the state of the PHT, as in normal PAg operation. The benefits of this scheme come from reducing the occurrence of DPE. Those patterns that are predicted well continue to be predicted rather than using DPE resources, and the patterns that are believed to mispredict often are allowed to execute down both paths eliminating the prediction (and thus a misprediction).

3.2.2 DPE/2-bit

The same approach is used to create a second DPE hybrid predictor replacing the PAg scheme with a simple two-bit predictor. The difference between this two-bit component and its single scheme predictor is that there is knowledge of a depth of history that is maintained for each branch, and this knowledge is used to better target processor resources. When a branch occurs, the pattern is evaluated to determine whether to predict or fanout, as in the DPE/PAg scheme. Since the two-bit table cannot not differentiate between as many patterns, a set history depth is used.

3.3 Accuracy and Coverage Relationship

Both accuracy and coverage play a major role in deciding which patterns to include in the predicting set. Coverage is the percent of total references encompassed by the patterns in the predicting set. Given that there is not an infinite set of resources to always allow fanouts without delay, it would be optimal to only predict the patterns that achieve an average accuracy above a certain threshold. This would allow a fanout to occur for patterns that have been shown to average a low accuracy which detracts from the overall accuracy. However, the higher the accuracy threshold is set, the more patterns fall below that accuracy and compete for fanout resources. If two branches compete for dual path execution resources, then the first branch to be started will be allocated the resources regardless of the prediction accuracies of their respective patterns. Thus, a less accurate pattern may be forced to be predicted because the resources are already filled with a more accurately predicted pattern. This can lead to a reduction in overall accuracy when the predicting set does not contain enough patterns, implying that the coverage is too small. A compromise needs to be made between choosing patterns with high accuracies and those that create a large coverage. Including patterns with a high percentage of the total references would reduce how often fanouts would be attempted. Choosing a predicting set that targets either accuracy or coverage, while being optimal in the local point of view, may not improve overall performance. Figure 3.3 illustrates the relationship between accuracy and coverage for each benchmark. As the number of patterns in the predicting set increases, the accuracy drops and the coverage increases. These curves were obtained by sorting the patterns according to accuracy and then the accuracy and coverage were deter-

mined as each pattern was added into the predicting set. These curves assume that those patterns that are not in the predicting set will have the resources to fanout. Since the coverage curve increases quickly, the predicting set can remain small, with the hope of not compromising the accuracy greatly. If the correct patterns could be identified, *go* could achieve an overall accuracy of over 95% with 60% coverage. This is a dramatic increase from its normal overall accuracy of just over 80% (translating to over a 75% drop in misprediction rate). Thus, the importance of choosing the optimal patterns can be seen.

3.4 Analysis of Patterns and the Predicting Set

There are a few different approaches that can be used to obtain the predicting set. Patterns can be analyzed in all the benchmarks and a set can be chosen that should perform well for all programs. Another approach would be to chose a separate set for each benchmark. While this method should attain higher accuracies for each program, a generic predicting set is more versatile. The following describes the techniques used to obtain one predicting set for all benchmarks. Profile-based predicting sets will be used and compared later in this thesis. Another aim when creating the set is to be able to create a simple circuit to identify whether a pattern is part of the predicting set.

When choosing the most beneficial patterns to include in the predicting set for both the accuracy and coverage concerns, it is helpful to look for trends among the patterns across the benchmarks. For each benchmark, two lists of patterns were crucial to determining the predicting set; patterns sorted in order of references and patterns sorted in order of accuracy. The list of the top referenced patterns

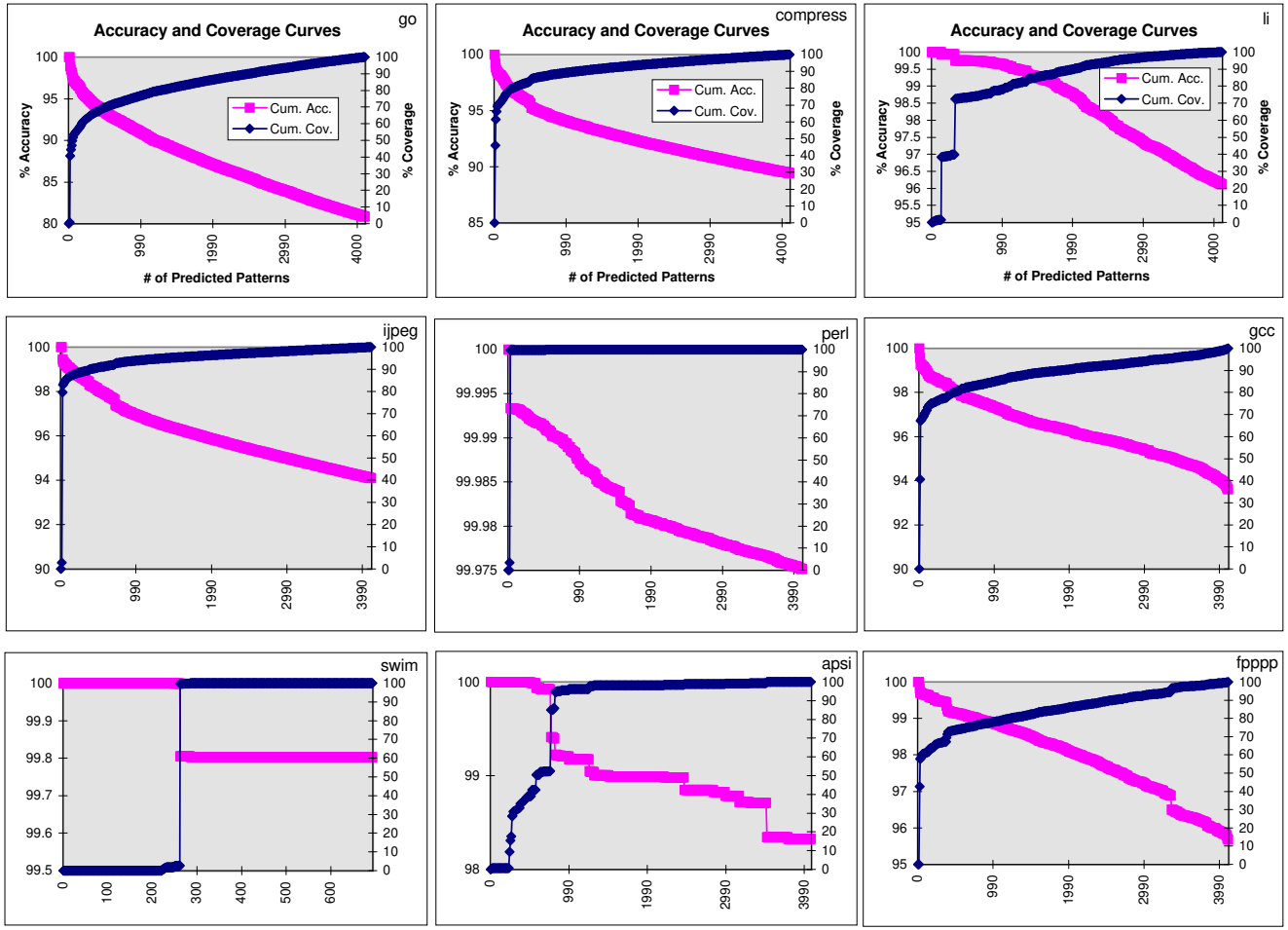

Figure 3.3: Relationship between Accuracy and Coverage

go	ijpeg	perl	gcc	swim	apsi	fpppp
111111111111	111111111111	111111111111	111111111111	111111111111	111111111111	111111111111
000000000000	000000000000	000000000000	000000000000	000000000000	000000000000	000000000000
110111011101	101010101010	101101101101	010101010101	011111111111	101010101010	101110111011
101110111011	010101010101	110110110110	101010101010	111111111110	010101010101	110111011101
111011101110	111101111111	011011011011	011111111111	111111111101	010101010010	111011101110
011101110111	111110111111	101001100010	111111111110	101111111111	010010101010	011101110111
011111111111	111111011111	010100110001	111110111111	111111111011	111110111111	101101101101
111111111110	111111101111	001010011000	111111011111	110111111111	111111011111	110110110110
111111011111	011111111111	100110001010	111101111111	111011111111	101010010010	011011011011
111110111111	111111111110	000101001100	111111101111	111111110111	010101001001	010101010101
111111111101	011111110111	110001010011	111011111111	111111011111	010100100101	101010101010
101111111111	111011111110	100010100110	111111110111	111111101111	101001001010	110110111011
111111101111	101111111011	010011000101	101111111111	111101111111	100101010100	011011101110
111101111111	110111111101	011000101001	110111111111	111110111111	001010101001	110111011011
110111111111	101111111111	001100010100	111111111011	101010101010	101010100100	101101110111
111111111011	110111111111	101010101010	111111111101	010101010101	010010010101	011101110110
111111110111	111111111101	010101010101	000000000001	101110111011	100100101010	111011101101
111011111111	111011111111	111110111111	100000000000	011101110111	001001010101	111011011101
010101010101	111111110111	111111011111	000000100000	110111011101	111101111110	011101101110
101010101010	111111111011	111101111111	000001000000	111011101110	011111101111	101110110111

Table 3.4: Top 20 Referenced Patterns for each Benchmark

revealed a noticeable trend. Table 3.4 lists the top 20 referenced patterns for each benchmark. In every benchmark the top two referenced patterns were those that contained either all ones or all zeros, meaning that the history is exclusively TAKEN or exclusively NOT-TAKEN. These patterns will be referred to as the always-taken pattern and the always-not-taken pattern respectively. This trend implies that these patterns are beneficial to the coverage component for choosing the predicting set. Fortunately, looking at the list of patterns that have the best accuracies, the always-taken and always-not-taken patterns are found in the top 1% of all patterns for most of the benchmarks and in the top 5% for all benchmarks. Thus, the always-taken and always-not-taken patterns are obvious candidates for the predicting set; they have good prediction accuracy as well as good coverage. When analyzing these two lists, the necessity of using both accuracy and coverage together is clear; many of the most accurate patterns obtain such a high accuracy due to very low references.

Another interesting characteristic found in the sorted patterns is that patterns with only a single NOT-TAKEN (zero state) branch often have a high reference count and achieve fairly high accuracies. This set of patterns is referred to as the

almost-always-taken patterns. This phenomena led to the classification of patterns into sets based on the number of TAKEN paths in the history. These sets were analyzed through graphs that showed how they compared in terms of references and accuracies. Figure 3.4 shows these graphs for three benchmarks, *go*, *ijpeg*, and *fpppp*. Each point gives the accuracy or reference for the set of patterns with X number of TAKEN branches. For example, the pattern *0001000100000000* would be included in the set of patterns with two TAKEN branches. The overall accuracy is also shown on the graphs for comparison. The graphs shown use data obtained by the predictor PAg(16). These graphs reaffirm the inclusion of the always-taken pattern and the always-not-taken pattern in the predicting set since these patterns consistently achieve higher accuracies than the overall accuracy. Both *go* and *ijpeg* show that the almost-always-taken patterns also give a higher accuracy than the overall accuracy. Thus the patterns with only one NOT-TAKEN path in their history are included in the predicting set.

Some of the benchmarks show that the almost-always-not-taken pattern set (those containing only one TAKEN branch in a stream of NOT-TAKEN branches) also achieves high accuracy. While this is not true for all the benchmarks, these patterns were chosen to be included in the predicting set due to the simplicity of the circuit that could identify these patterns. The references to these patterns will also help limit the number of fanouts attempted. A few benchmarks, such as *fpppp*, which do not have good accuracy for the almost-always-not-taken patterns, do not reference this pattern set often enough for it to have a negative effect on the accuracy of the new hybrid predictor.

Peaks can be seen in many of the programs for the pattern sets that were TAKEN during half the history and NOT-TAKEN for half. After further ex-

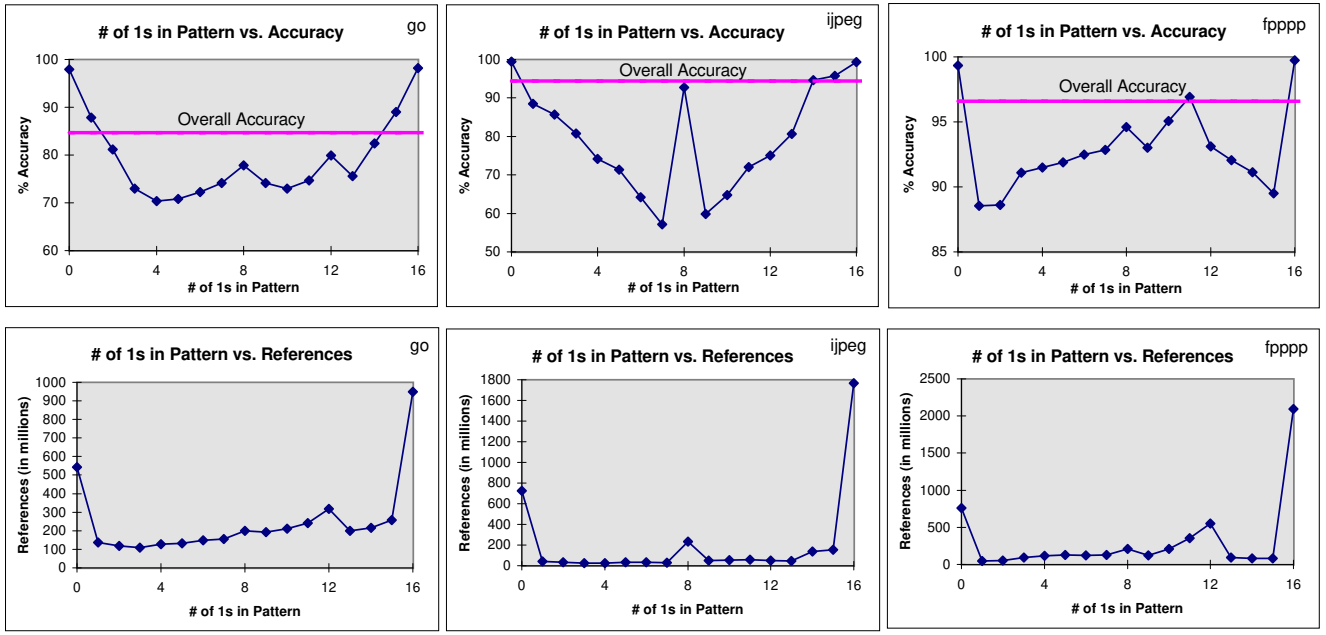

Figure 3.4: Pattern Sets based on the Number of TAKEN Paths

amination, it was found that the patterns that alternate between TAKEN and NOT-TAKEN paths were responsible for the majority of references in this pattern set. These patterns will be referred to as the alternating patterns. Since these patterns can be predicted well[3], they too are included in the predicting set. Thus, the predicting set used for this thesis will contain the always-taken pattern, the always-not-taken pattern, the almost-always-taken and almost-always-not-taken patterns, and the alternating patterns.

The accuracy and coverage that occur due to the chosen predicting set is shown in Table 3.5. The table shows the percentages for each parameter varying the history depth between 8, 12, 16 bits. The actual PAg(x) accuracy, without using any fanouts, is shown for comparison. As more history is maintained, the coverage decreases, attempting to fanout more often. The accuracy is seen to increase with the history. These graphs assume that there is always a DPE resource available when needed, making the only patterns to be predicted those in the predicting set. Given ample resources, it is clear that there would be a very beneficial increase in accuracy due to this new hybrid predictor.

3.5 Fanout Constraints

The frequency of fanouts, or dual path execution, is dependent on the amount of resources available and the time it takes for a branch to be resolved. As shown in Equation 1.2, the cost of executing a branch, measured in clock cycles, is the branch resolution time. In order to accurately simulate the correct cycle count, one would need a full pipeline model and the results would be implementation dependent.

[3]The overall accuracy of the half TAKEN/half NOT-TAKEN is brought down by those patterns which are not alternating.

	DPE/PAg(8)		PAg(8)	DPE/PAg(12)		PAg(12)	DPE/PAg(16)		PAg(16)
Program	% Cov	% Acc	% Acc	% Cov	% Acc	% Acc	% Cov	% Acc	% Acc
go	59.091	94.734	79.963	49.688	97.290	80.850	44.559	98.247	84.716
compress	78.330	96.252	89.045	71.011	97.620	89.450	65.781	98.602	89.846
li	75.185	98.690	94.699	71.716	99.449	96.120	70.535	99.623	97.440
ijpeg	88.277	99.099	93.761	84.394	98.804	94.095	81.563	99.200	94.429
perl	90.637	99.983	99.963	90.606	99.988	99.975	90.581	99.991	99.980
gcc	81.881	96.987	91.816	76.420	98.285	93.601	73.156	98.823	94.847
swim	99.947	99.803	99.803	99.938	99.803	99.803	99.929	99.803	99.803
fpppp	63.131	98.191	92.907	59.145	99.233	95.681	57.497	99.534	96.590

Table 3.5: Accuracy and Coverage Assuming Unlimited DPE Resources

In order to create a branch resolution time that is independent of architecture, branch resolution could be based on instruction count. This would require that all instructions be trapped and traced however, greatly increasing the time it would take to run each program. An approximation to instruction count can be done with a branch count. One branch is roughly equivalent to five instructions. This method is much faster since branches are being traced anyway and it still gives a good first level approximation. Future work in this area will look at the actual instruction count.

When determining whether to allow a pattern that is not in the predicting set to fanout, the branch resolution time is compared to the number of branches that have occurred since the last fanout. If fanouts occurs too close together, then a prediction is forced. A resolution equal to *2* corresponds to the requirement that at least two branches must be predicted between fanouts. Throughout this thesis, the effects of the branch resolution time is analyzed as it varies from one to six. When looking at the resolution in units of instructions there would actually be more allowance in the number of instructions. Rather than implying that either 5 or 10 instructions must be executed (a resolution of 1 or 2), smaller increments could actually be chosen and the basic block size would not be a factor in the resolution time of a branch.

3.6 Summary

Examining branch distributions shows that programs typically execute a few branches a large number of times. A small number of branches also tend to be responsible for the majority of mispredictions. These two trends imply that much could be

gained in a branch predictor if these branches could be targeted. This thesis proposes that patterns should be targeted rather than branches due to branch biases and a larger reference coverage which can be obtained by patterns. The implementation proposed combines dual path execution with a current single scheme branch predictor. Two different variations are studied: DPE/PAg and DPE/2-bit. A predicting set is used to limit the occurrence of dual path execution to those patterns that are believed to have low prediction accuracies. A great deal of study was done to determine a good predicting set. While initially it may seem to be a good choice to use an accuracy threshold to choose patterns, it is shown that the relationship between accuracy and coverage is important when determining the predicting set.

Chapter 4

Evaluating Hybrid DPE Performance

This thesis proposes a new hybrid branch predictor combining dual path execution and a current single scheme predictor. Two different variations were studied by combining DPE with PAg and with the 2-bit counter method. The idea behind this new hybrid predictor is to eliminate branch penalty by executing down both paths of a branch. Due to resource limits however, DPE can not be performed on every branch. Thus, a predicting set is created to limit how often DPE is performed. The predicting set includes patterns that have high accuracy and cover a large number of references. This allows only those patterns that are believed to be poor predictors to fanout. The predicting set that was used in this thesis contained the always-taken, always-not-taken, almost-always-taken, almost-always-not-taken, and alternating patterns. This set will be referred to as the fixed predicting set since it does not change with the benchmark being used. The following sections give the results of both the DPE/PAg and DPE/2-bit predictors.

The first section uses the fixed predicting set. The second section uses an alternate method for partitioning patterns, between the DPE and predicting set, based on profiling information.

4.1 The Fixed Predicting Set

The misprediction rates for PAg and DPE/PAg are shown in Figure 4.1[1]. The depth of history maintained is varied between 8, 12, and 16. The branch resolution is varied from 1 up to 6. These variations are referred to as H(R), where H is the number of history bits and R is the branch resolution. Thus, 12(5) would imply that the DPE/PAg predictor uses 12 bits to maintain the branch history and 5 branches are required to occur between fanouts. As the branch resolution decreases, the misprediction also decreases. A shorter branch resolution allows more fanouts to occur and thus more patterns with poor prediction accuracies do not have to predict. Looking at *li* with a history depth of 12 and a branch resolution of 1, there is a reduction of over 78% in the misprediction rate. Increasing the resolution to 6 while keeping the same number of history bits gives a reduction of the misprediction rate by 33%. Thus, even using a resolution of 6, approximately 30 instructions between fanouts, a significant increase in performance is seen. This allows the use of DPE even with minimal resources to be beneficial.

Figure 4.2 shows the misprediction rates of DPE/2-bit and its single scheme component 2-bit. The reduction in misprediction rates is even greater when comparing these two schemes. *ijpeg* shows a 73% reduction in misprediction rate when changing from a 2-bit counter method to the DPE/2-bit with a history of 8 and a

[1]The results for *apsi* are not reported due to errors in the DPE runs.

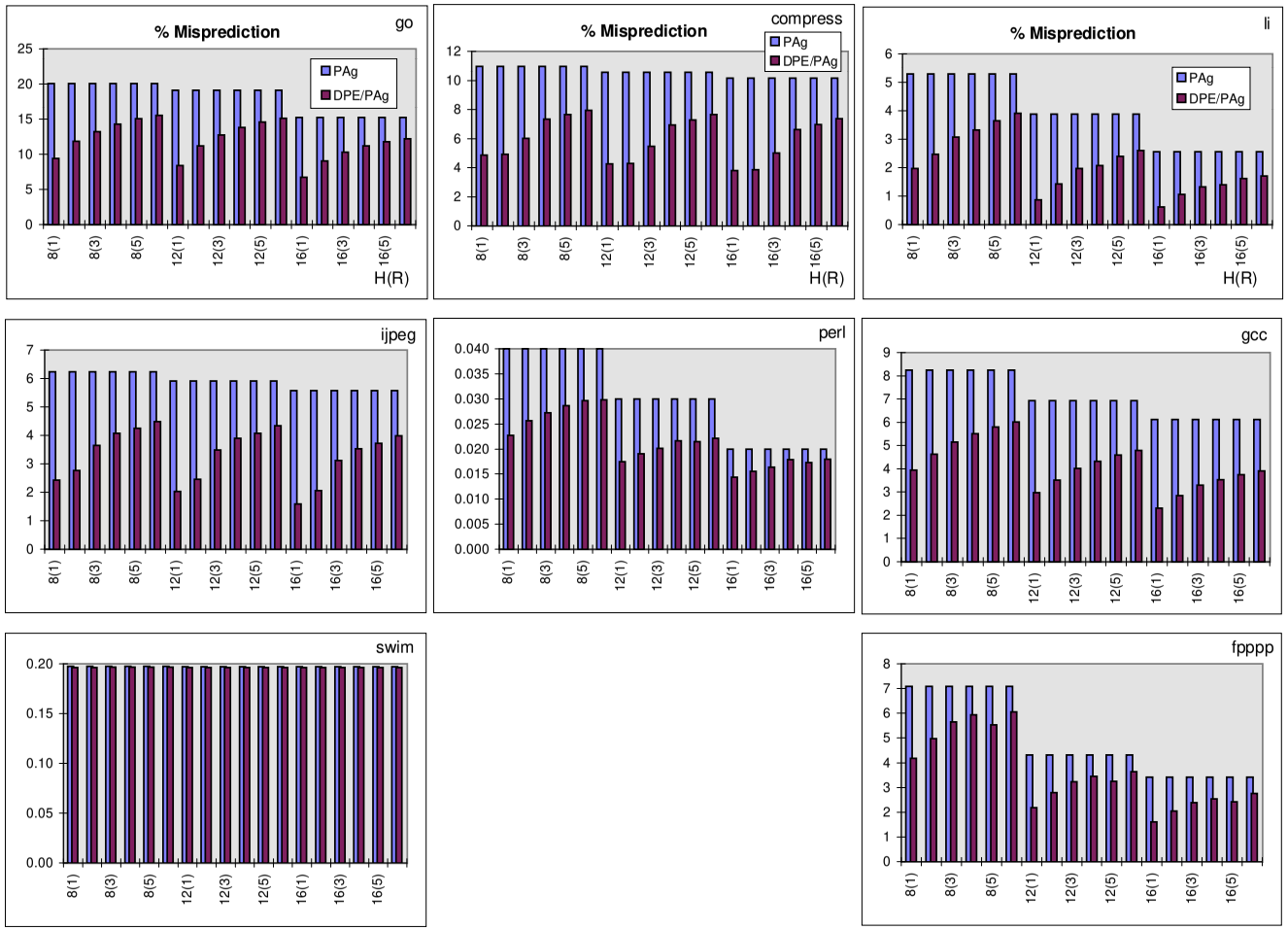

Figure 4.1: Misprediction Comparison of PAg and DPE/PAg

resolution of 1. Increasing the resolution to 6 still gives an impressive reduction of 50% in the misprediction.

These results become more interesting when comparing the single scheme predictor PAg to both the hybrid variations. These comparisons are shown in Figure 4.3. Cost would probably not be willingly given to DPE unless one was willing to use the PAg single scheme over the 2-bit single scheme. One can see that the two DPE hybrid predictors often perform similarly and the DPE/2-bit often outperforms the PAg. When looking at these, one may notice that it may be more beneficial to cut cost in the predictor in order to use that cost for dual path execution resources. If enough resources could be allocated to give a branch resolution of 1 or even 2, then one could benefit on most benchmarks by using the DPE/2-bit with a history of 8 over the single scheme predictor PAg, even with a history of 16. Simple circuits can be made to maintain whether the pattern is in the fixed predicting set without keeping all the bits of the history depth. Using the fixed predicting set, *compress* would give a reduction in misprediction of over 50%. This is a dramatic increase in accuracy and can tremendously increase performance in high performance processors. Thus the cost may be better served if put towards DPE.

When analyzing the results of the hybrid DPE predictor, the results of the predicting set should not be ignored. The performance of the hybrid predictor is dependent on the accuracy and coverage of the predicting set. If the accuracy is kept too high, the coverage will be reduced and there will often be contention for the DPE resources between patterns; forcing a pattern with a low prediction to predict because a pattern with a higher accuracy has fanned out previously taking the resource. Thus the coverage of the predicting set is also very important.

Figure 4.2: Misprediction Comparison of 2-bit and DPE/2-bit

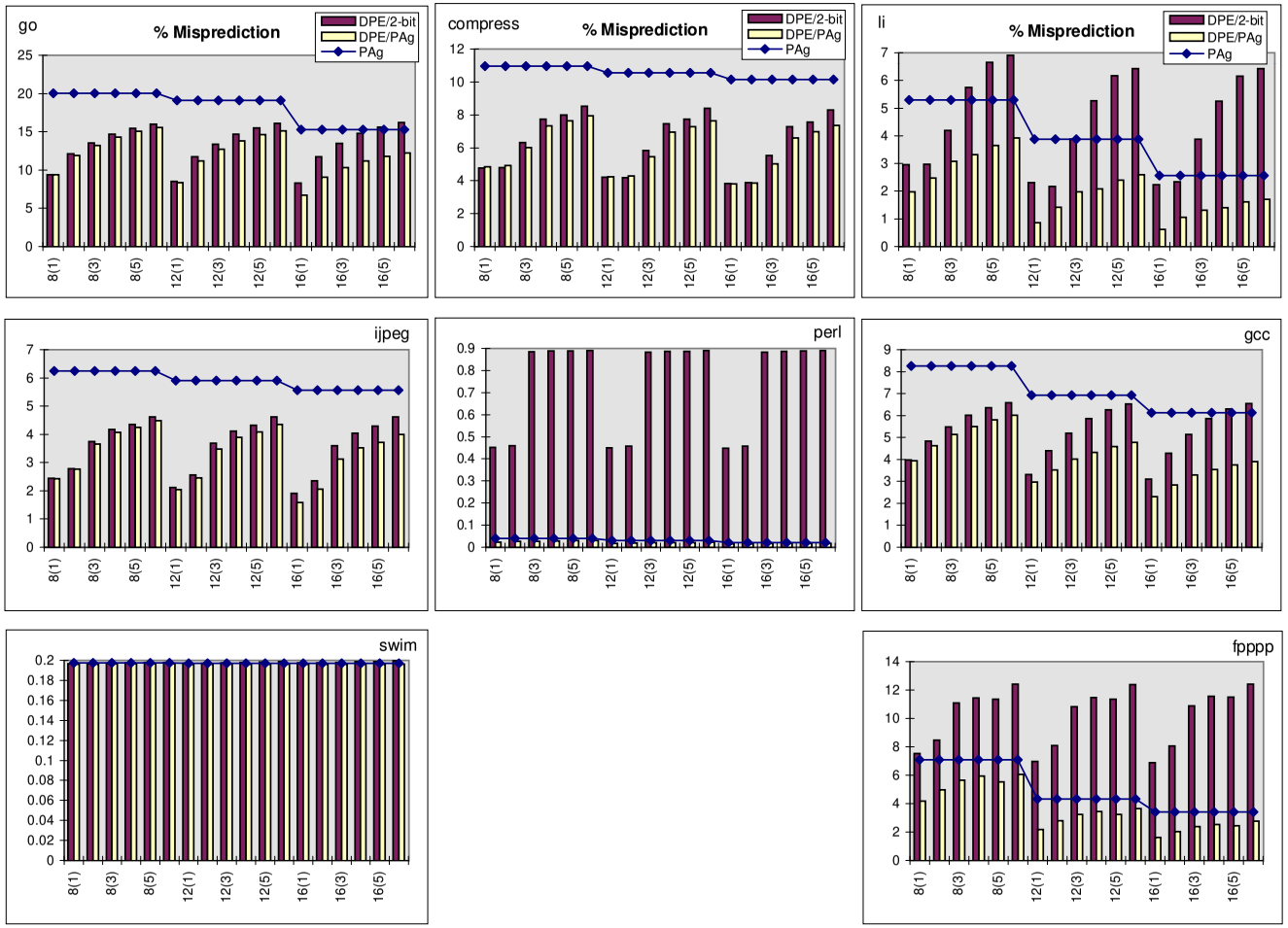

Figure 4.3: Comparison of PAg, DPE/PAg, and DPE/2-bit

Figure 4.4 shows the percent of patterns that were actually able to fanout. This percentage is in terms of patterns that were contending for the DPE resources, those not in the predicting set. Looking at a history depth of 12 and a branch resolution of 1, the fixed predicting set was able to perform between 60% and 94% of all fanouts, with an average of 77%. Increasing the branch resolution to 6, between 23% and 83% of all fanouts were given DPE resources, with an average of 41%. Thus, how much cost is to be devoted to dual path execution and the branch resolution have a large affect on the coverage of the predicting set. Table 4.1 shows the number of fanouts executed and the percentage of total executions that avoided prediction for history depths of 8 and 12. An average of 16% of the total branches had patterns that were allowed to fanout when the history depth is 12 and the resolution is 1. Thus, the fixed predicting set does a good job of limiting the occurrence of dual path execution.

4.2 Profile-based Predicting Sets

A second approach to creating the predicting set was used for comparison. A predicting set was created especially for each benchmark. Profiling was used in order to decide which patterns should be included in the predicting set. These predicting sets will be referred to as the profile-based predicting sets. The patterns were sorted in order of decreasing accuracy and kept in a file along with their corresponding references. When the program is run, the desired coverage is specified for that run. At the start of a run, the profiled pattern file is read in. Starting from the most accurate pattern, patterns are included into the predicting set until the desired coverage is met. A bit corresponding to the pattern is set if the pattern is

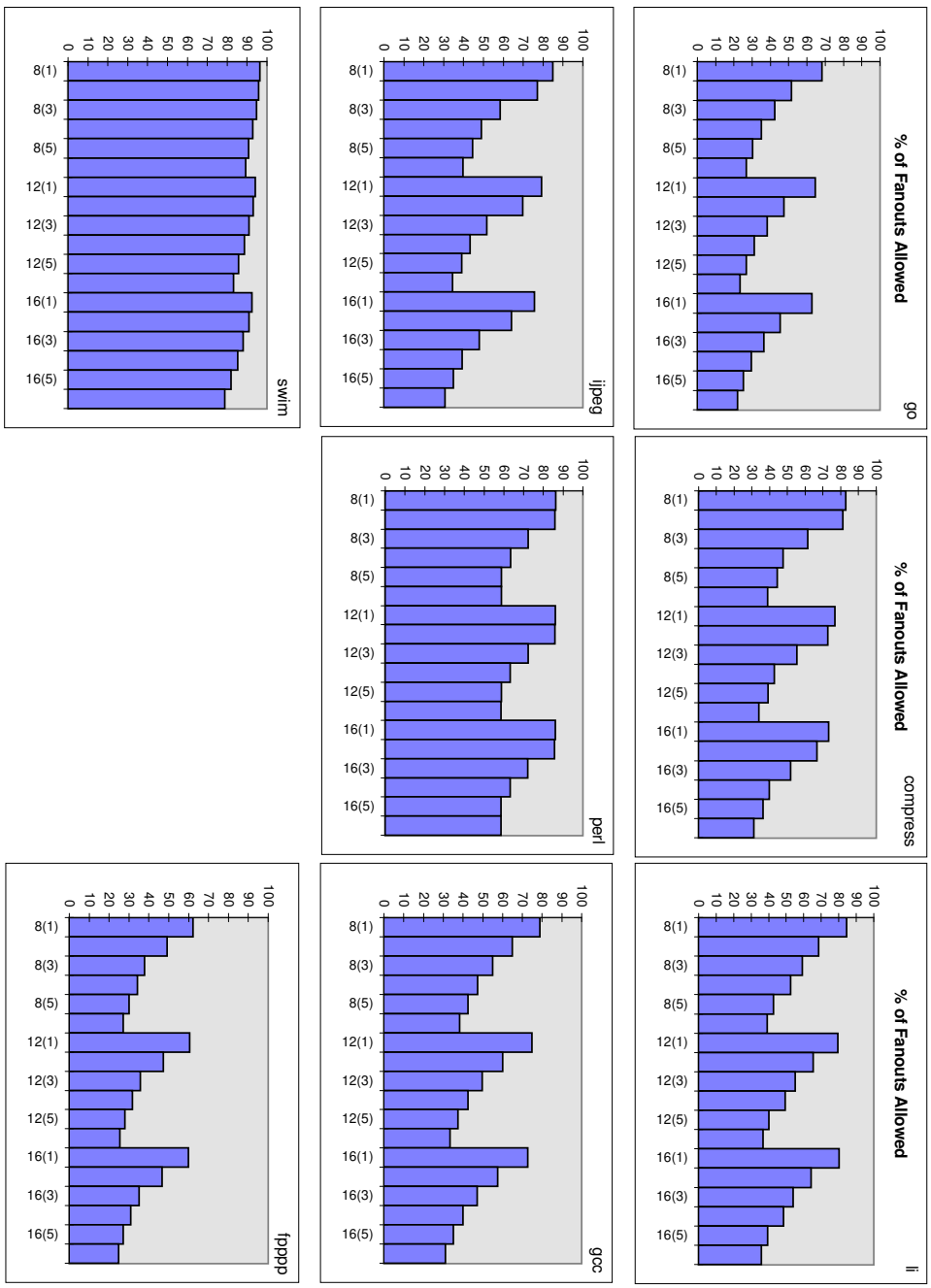

Figure 4.4: Analysis of Fanouts Allowed

	DPE/PAg(8)											
	Resolution = 1		Resolution = 2		Resolution = 3		Resolution = 4		Resolution = 5		Resolution = 6	
Program	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot
go	1199	27.9	906	21.1	745	17.3	616	14.3	531	12.3	469	10.9
compress	2.30	17.9	2.26	17.6	1.71	13.3	1.32	10.3	1.24	9.6	1.08	8.4
li	2060	20.9	1671	17.0	1443	14.7	1281	13.0	1043	10.6	953	9.7
ijpeg	351	9.9	319	9.0	241	6.8	202	5.7	184	5.2	164	4.6
perl	184	8.1	184	8.0	155	6.8	135	5.9	126	5.5	126	5.5
gcc	16.2	14.2	13.3	11.7	11.2	9.9	9.7	8.6	8.7	7.7	7.9	7.0
swim	0.37	0.05	0.37	0.05	0.37	0.05	0.36	0.05	0.35	0.05	0.34	0.05
fpppp	1214	22.9	961	18.1	738	13.9	670	12.6	588	11.1	529	10.0

	DPE/PAg(12)											
	Resolution = 1		Resolution = 2		Resolution = 3		Resolution = 4		Resolution = 5		Resolution = 6	
Program	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot	# Fans	% Tot
go	1393	32.4	1021	23.8	826	19.2	674	15.7	576	13.4	506	11.8
compress	2.86	22.2	2.70	21.0	2.06	16.0	1.59	12.3	1.46	11.4	1.26	9.8
li	2212	22.5	1820	18.5	1532	15.6	1377	14.0	1114	11.3	1018	10.36
ijpeg	437	12.4	385	10.9	285	8.1	238	6.7	215	6.1	189	5.3
perl	184	8.1	183	8.0	154	6.7	135	5.9	126	5.5	126	5.5
gcc	20.0	17.6	16.1	14.2	13.3	11.7	11.3	9.9	10.0	8.8	8.9	7.8
swim	0.42	0.05	0.42	0.05	0.41	0.05	0.40	0.05	0.39	0.05	0.37	0.05
fpppp	1308	24.7	1023	19.3	774	14.6	687	12.9	602	11.4	549	10.4

Table 4.1: Number of Fanouts and the Percentage of the Total Executions

added to the predicting set. Then execution of the program continues, and when a branch occurs, the bit associated with the branch pattern is checked. If the pattern is in the predicting set, then the pattern is predicted; otherwise dual path execution is attempted as before. The advantage to using this scheme for creating the predicting set is that each benchmark only includes those patterns that it has previously found to achieve the highest accuracy. Patterns that may perform well in many other benchmarks, but not on this one, will not be included. Another advantage is that the coverage can be adjusted easily to restrict fanning out to only those patterns that really require it due to their poor prediction accuracy. The coverage from the fixed predicting set given before varies among benchmarks. A drawback to this scheme is that the program must be run prior to application execution in order to determine the predicting set. The results of using this scheme are shown and compared with the fixed predicting set scheme.

Six different benchmarks were run with profile-based predicting sets. Figure 4.5 shows the misprediction rates for the DPE/PAg(12) predictor using this predicting set. *swim* is not shown due to the results remaining essentially the same among all runs. Both coverage and branch resolution are varied. The variables are referred to in the graphs as C/R, where C is the coverage specified at the beginning of the run and R is the branch resolution in terms of branches. Runs are shown for a branch resolution equal to 2 and 3. For *gcc*, the resolution is shown for 1, 2, 3, and 4. Asterisks (*) show the results for the previous study with the fixed predicting set. These graphs emphasize the compromising relationship between coverage and accuracy. Fixing the branch resolution, the misprediction rate decreases and then increases as the coverage is varied. If the coverage is too low, then patterns with low prediction accuracies may be forced to predict since there is a lot of contention

for the DPE resources. If the coverage is too high, then the DPE resources are not being used enough and patterns with low accuracies are being included in the predicting set and thus are being forced to predict. Each benchmark has its own ideal coverage for each length of branch resolution. For example, *ijpeg* performs best when the branch resolution is 2, when the coverage is 85, while *go* performs best with a coverage of 50. *gcc* performs best at a coverage of 70 when the branch resolution is 1, 2 and 3, yet benefits from a coverage of 85 when the resolution increases to 4. Thus, the architectural limits of DPE resources should be considered when choosing a coverage.

Comparing the misprediction rate for the ideal coverage and that of the coverage obtained by using the fixed predicting set, one can see that the rates do not vary by much. Often the two coverages are even fairly close together. For a resolution of 2, *compress* has an ideal coverage of 70 and achieves a misprediction rate of 4.07. The fixed predicting set has a coverage of 71 and achieves a misprediction of 4.29. These correspond to decreases in misprediction rate of the DPE/PAg scheme of 78.7% and 77.6% respectively. The closeness in the coverages and the misprediction rates imply that the fixed predicting set does a good job of identifying the patterns that are good predictors and that those patterns really do not vary much with the benchmark. Thus, profiling may not be necessary.

Figure 4.6 shows the percentage of fanouts that were allowed. The percentage decreases as the resolution increases since more patterns are forced to predict during the time that a prior fanout is occurring. It is interesting to note that the percentages of the ideal coverages are close given the same resolution. For example, with a resolution of 2, the ideal coverages range from 50% to 69% in the percentage of fanouts allowed, with an average of 58%. A resolution of 3, ranges

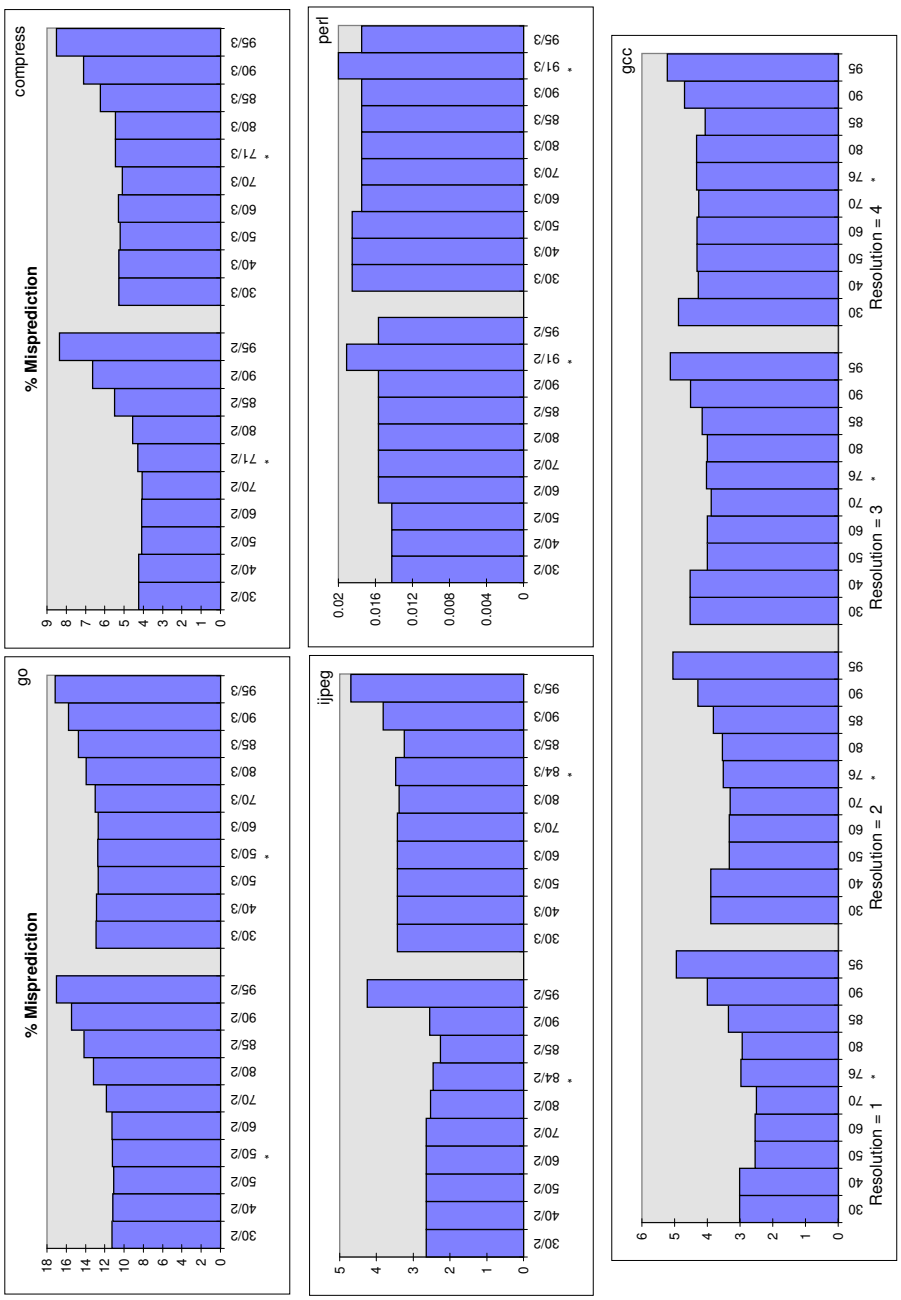

Figure 4.5: Misprediction Rates for DPE/PAg(12) using Profile-based Predicting Sets

from 40% to 60%, with an average of 47%. Thus, although it may seem beneficial to choose a predicting set that allows a high percentage of fanouts to occur, it is actually ideal to force the patterns that were not included in the predicting set to predict around half the time.

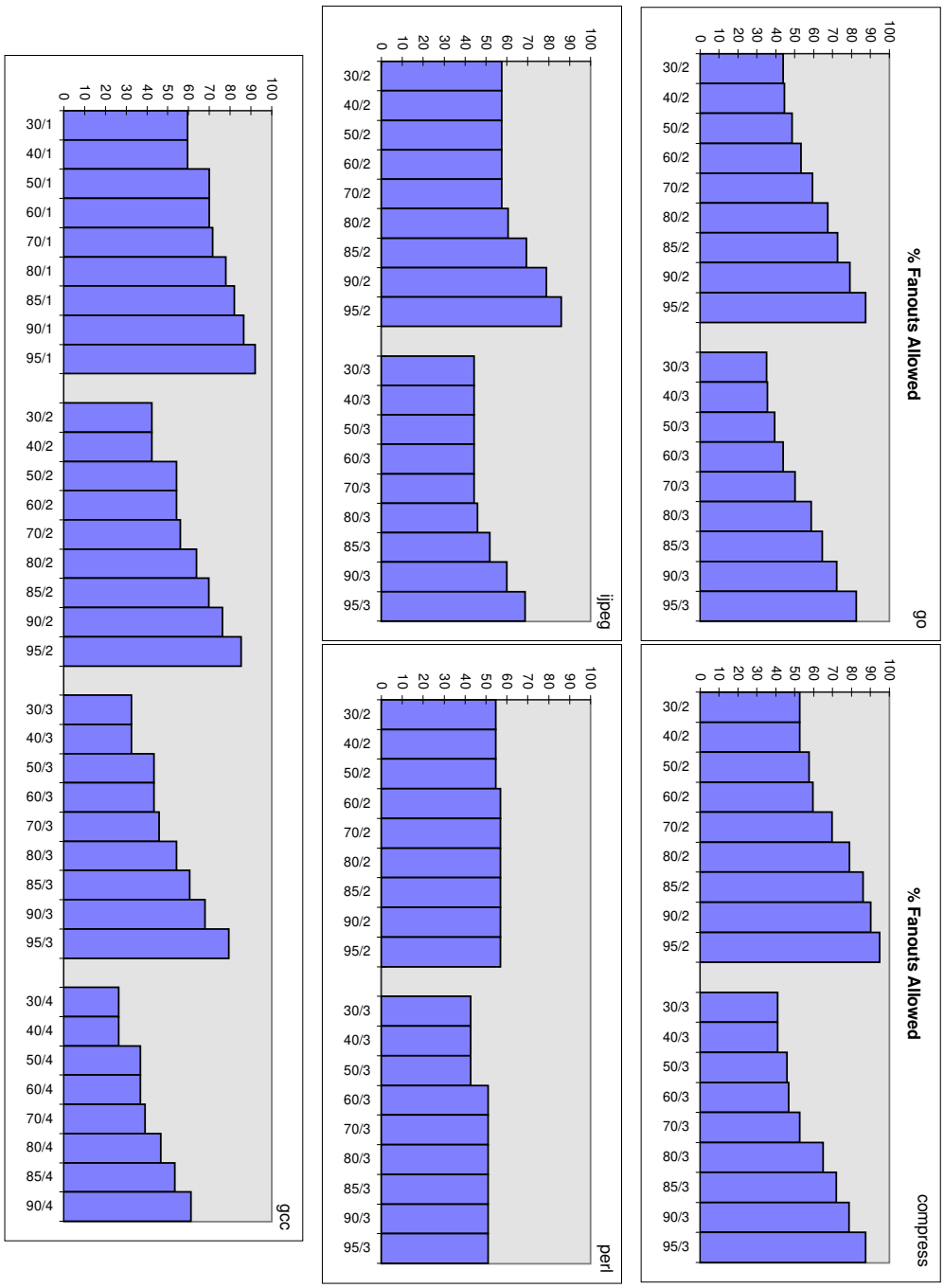

Figure 4.6: Analysis of Fanouts Allowed using Profile-based Predicting Sets

Chapter 5

Conclusions

Specific branches have long been the focus in branch predictors. Programs have a tendency to execute a few branches a large number of times. This implies that it might be beneficial if a branch predictor could target those branches that are responsible for the majority of the branch executions. In an effort to cover more branches, this thesis targets patterns since branches are biased to a particular path or pattern. The accuracy and reference count of each pattern is used to assign a confidence to that pattern. The confidence of the pattern determines whether that pattern should be predicted or should attempt to avoid a prediction by executing down both paths.

Using the pattern confidences, a partial coverage predictor can be implemented. In the past, branches have either always been predicted or are never predicted. The idea of a partial coverage predictor takes a predictor that is known to typically predict well and limits the time it predicts to those high accuracy cases. Predictors used in the proposed implementation, evaluate the confidence assigned to a pattern to decide whether to make a prediction. If the pattern has been known to predict

Patterns that Fanned Out	
Method	Misprediction
DPE/PAg(8)	35%
DPE/PAg(12)	28%
DPE/PAg(16)	23%

Table 5.1: Misprediction Rate of the Patterns that performed DPE

well, then the predictor is allowed to predict, otherwise a prediction is not made. This thesis presents a hybrid branch predictor scheme that uses dual path execution and a predicting set based on patterns to effectively reduce the branch penalty in high performance processors. While dual path execution is beneficial in eliminating branch penalty, it is unrealizable due to the large, and unknown limit, in cost that is needed. Branches occur too frequently and can cause a cascading effect in the number of resources necessary. Combining DPE with another current prediction scheme can limit the times that a fanout would occur reducing the cost. When limiting DPE, it is beneficial to predict when the prediction is believed to be a good prediction and to allow the DPE resources to be used instead of making a poor prediction. A predicting set is created to select when to predict and when to execute down both paths, creating a partial coverage predictor. The predicting sets given in this thesis do a good job in allocating the DPE resources and allow the strengths of both the predictor and dual path execution to remain, while also reducing the negative costs of DPE. Table 5.1 shows the misprediction rate that would have occurred for those branches that were fanned out. When these branches are not predicted in favor of dual path execution, those mispredictions are eliminated.

Comparing the DPE hybrid predictor model to the single scheme approach PAg,

Method	% Misprediction	% Decrease
DPE/PAg(8)/1	3.37	53.6
DPE/PAg(8)/2	3.97	45.3
DPE/PAg(8)/3	4.62	36.4
DPE/PAg(8)/4	5.08	30.0
DPE/PAg(8)/5	5.26	27.5
DPE/PAg(8)/6	5.52	24.0
DPE/PAg(12)/1	2.61	58.6
DPE/PAg(12)/2	3.24	48.6
DPE/PAg(12)/3	3.89	38.3
DPE/PAg(12)/4	4.34	31.1
DPE/PAg(12)/5	4.55	27.8
DPE/PAg(12)/6	4.79	24.0
DPE/PAg(16)/1	2.11	60.1
DPE/PAg(16)/2	2.64	50.1
DPE/PAg(16)/3	3.20	39.5
DPE/PAg(16)/4	3.62	31.6
DPE/PAg(16)/5	3.81	28.0
DPE/PAg(16)/6	4.02	24.0

Table 5.2: Summary of DPE/PAg

dramatic decreases in misprediction rate were seen. The PAg(16) gave an average misprediction of 5.29%. The DPE/PAg(16)/1 gives an average misprediction of 2.11% and the DPE/2-bit(16)/1 achieves a misprediction rate of 3.08%. These correspond to decreases of 60% and 36% respectively. Table 5.2 shows the overall results for the DPE/PAg model and their misprediction decreases compared to PAg. These results imply that dual path execution which often is thought to be a resource consuming method may be a worthy approach if restricted with a predicting set.

The DPE/PAg(16)/1 predictor averages a misprediction rate of 2%. The best predictor proposed in literature is said to have a misprediction rate of 3%. Thus

Predictor Scheme	Accuracy	Cost
gshare(12)	93	8K
PAs(6,16)	96	8K
GAs(11,32)	97	128K
gshare(18)/PAs(15,4)	97	128K
DPE/2-bit(16)/1	97	*
DPE/PAg(16)/1	98	*

Table 5.3: Summary of the Best Branch Prediction Schemes

there is a decrease of 33%. Table 5.3 summarizes the average accuracies for some of the best predictors. The cost of DPE was not studied in this thesis. However, performance is continuing to be more and more limited by branch prediction, creating a willingness to devote more cost to the branch predictor. In addition, as more transistors are able to fit on a chip, cost does not become such a large factor.

5.1 Future Work

The proposed hybrid branch predictor scheme can greatly help increase the performance of processors. While it is clear that this scheme provides an increase in prediction accuracy over current schemes, assumptions that were made throughout the experiments that should be addressed in order to further validate the results of this predictor approach. Future work in this area will eliminate some of these assumptions.

- A assumption was made when dealing with the branch resolution time. This variable should actually have units in terms of clock cycles, yet this would then become architecturally dependent. A good approximation would be to use instructions as the units for branch resolution rather than branches as

was done. In the future, resolutions will be in terms of instructions.

- Conducting a full-pipeline simulation would allow for this approach to be studied in great detail. Branch resolution could be done in terms of clock cycles. Also, it would be quite beneficial to study resource allocation using dual path execution and how the utilization of resources is affected by this approach and as the branch resolution is varied.

- Another area that should be further investigated is to further study the optimal profile-based and mutual predicting sets. To find an optimal profile-predicting set the patterns can be added to the predicting set one-by-one in order of best accuracy. Once the optimal profile-based predicting set is found for each benchmark, further research can be done to find better fixed predicting sets. Other approaches to creating a predicting set would also be beneficial.

- Other hybrid dual path execution variations should be studied. The current predictors used in this study were PAg and two-bit, however, other current predictors might prove useful if combined with DPE.

- The cost of implementing of dual path execution was not looked into for this study. The cost of varying the branch resolution time should also be studied. Knowing these costs would allow a better comparison between predictors to find the best predictor for a given cost.

- In this study, execution was limited to two instruction streams at a time. The effects of allowing more instruction streams to execute simultaneously would be an interesting analysis. This essentially allows for instructions that

are an unresolved branch path to be executed, creating more instructions to fill the pipeline, however also creating the possibility for many more results to be found to be useless later.

- A larger set of benchmarks always proves useful to validate the results and should be run. The IBS benchmarks would be useful to run since these programs have become more commonly run among branch prediction studies and could prove useful in comparing accuracies. Branches invoked through system calls should also added into the study.

Bibliography

[1] T. Ball and J. R. Larus. Branch prediction for free. Technical Report TN-1137, CS Department, Univeristy of Wisconsin-Madison, Feb 1993.

[2] UC Berkeley CPU Info Center. Cpu info and system performance summary. Web address: http://infopad.eecs.berkeley.edu/CIC/summary/local/.

[3] Po-Yung Chang, Eric Hao, and Yale N. Patt. Alternative implementations of hybrid branch predictors. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, pages 252–257, Nov 1995.

[4] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Oct 1992.

[5] W-M Hwu, T. M. Conte, and P. P. Chang. Comparing software and hardware schemes for reducing the cost of branches. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 224–233, May 1989.

[6] S. McFarling and J. Hennessy. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pages 396–403, 1986.

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

[8] B. Ramakrishna Rau and Joseph A. Fisher. Instruction-level parallel processing: History, overview, and perspective. *The Journal of Supercomputing*, Oct 1992.

[9] J. E. Smith. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 135–148, May 1981.

[10] Augustus K. Uht. Reducing the ill effects of branches. *To appear in: IEEE COMPUTER*, 1996.

[11] Augustus K. Uht and Vijay Sindagi. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, pages 313–325, Nov 1995.

[12] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, 1991.

[13] Yougend Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pages 1–11, Nov 1994.

[14] T-Y Yeh and Y. N. Patt. Alternative implementations of two-level adaptive branch prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, May 1992.

[15] T-Y Yeh and Y. N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 257–266, May 1993.

[16] Tse-Yu Yeh. *Two-Level Adaptive Branch Prediction and Instruction Fetch Mechanisms for High Performance Superscalar Processors.* PhD thesis, University of Michigan, 1993.