ENHANCING THE INSTRUCTION FETCHING MECHANISM USING DATA COMPRESSION

by

I-Cheng Chen

A dissertation submitted in partial fulfillment of
the requirement for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1997

Doctoral Committee:

Professor Trevor Mudge, Chair
Professor Richard B. Brown
Professor Peter M. Chen
Professor Edward S. Davidson
Professor Ronald J. Lomax

© I-Cheng Chen 1997
All Rights Reserved

This dissertation is dedicated to my parents,

Dr. Chih-Hsien Chen and Mrs. Ching-Mei Chen.

ACKNOWLEDGEMENTS

First and foremost, I would like to thank my advisor, Professor Trevor Mudge, for giving me his precious advice and guidance. He, by setting an example himself, also showed the importance of working smart and being well-rounded.

I also want to thank my dissertation committee. I would like to thank Professor Richard Brown who leads the PUMA project with dedication and provides us with an excellent research environment. Professor Edward Davidson brought me into the field of computer architecture with his solid and intellectually stimulating lectures. I would also like to thank Professor Ronald Lomax and Professor Peter Chen for their helpful comments on this dissertation.

I am deeply indebted to my parents for their continuous support throughout my life. Without their caring encouragement, I would never have been able to complete my Ph.D. program. And I would also like to thank my brother I-Hong Chen for his assistance.

I would like to acknowledge Professor Sean Coffey and Professor Peter Bird for giving me new perspectives and advice on different research directions.

I also wish to thank Chih-Chieh Lee, who has been a classmate of mine since high school, for his great partnership. Through numerous night-long discussions, he has helped me create and implement various research ideas. I really cherish the opportunity to have him as my academic as well as social partner.

I gratefully thank my group members, Brian Davis, Charles Lefurgy and Bruce Jacob for their patient assistance on my English. They are the best tutors one can ever find. I would also like to thank other group members, Jim Dundas, Matt Postiff, Kristian Flautner, Mike Riepe, David Van Campenhout, Victor Kravets, Todd Basso, Spencer

Gold, Tim Strong, Sean Stetson, Phiroze Parakh, Claude Gauthier, Keith Kraver, Mini Nanua, and Mike Kelly, for creating a pleasant and productive working environment. In addition, I like to thank my colleagues, Jude Rivers and Wee Teck Ng, for their assistance.

Finally, I would like to thank Hong-Yi Wei for her company and kind support.

TABLE OF CONTENTS

DEDICATION ii

ACKNOWLEDGEMENTS iii

LIST OF FIGURES viii

LIST OF TABLES x

CHAPTER 1
INTRODUCTION 1

1.1 The need for fast instruction fetching 1
1.2 Data compression as a solution 2
1.3 Organization of this dissertation 3

CHAPTER 2
ANALYSIS OF BRANCH PREDICTION VIA DATA COMPRESSION . . . 6

2.1 Introduction 6
2.2 System model and overview of branch prediction 7
2.2.1 A general conceptual system model for branch prediction 8
2.2.1.1 Source 8
2.2.1.2 Information processor 9
2.2.1.2.1 Selector 9
2.2.1.2.2 Dispatcher 10
2.2.1.3 Predictor 10
2.2.2 Overview of current branch prediction schemes 10
2.3 Data compression and prediction 11
2.3.1 Prediction by Partial Matching 12
2.3.1.1 Markov predictors 13
2.3.1.2 Combining Markov predictors to perform PPM 13
2.4 Two-level branch prediction as an approximation of PPM 15
2.4.1 Description of two-level predictor 15
2.4.2 Two-level branch predictors as Markov predictors 16
2.4.3 Approximation to optimal predictors 18

2.5 Impact of optimal predictors and further improvement 19
2.5.1 Implication of optimal predictors 19
2.5.2 Assumptions for optimal predictors 19
2.5.3 Illustration of modest improvements using PPM techniques 20
2.6 Cost-effectiveness of PPM 26
2.7 Discussion of further improvement 30
2.7.1 Improvement of the predictor 30
2.7.1.1 Implementation of full-fledged optimal predictors 31
2.7.1.2 Design of other optimal predictors 32
2.7.1.3 Design of efficient non-optimal predictors 32
2.7.1.4 Improvement of the information processor 32
2.7.2 Improvement of the source 33
2.8 Conclusions and further work 33

CHAPTER 3
FURTHER EXAMINATION WITH OPTIMAL ALGORITHM AND EXACT ANALYSIS 35

3.1 Description of Lempel-Ziv predictor 35
3.2 Implementation details and consideration 39
3.3 Simulation results 40
3.4 Verification with an exactly analyzable program—Quicksort 41
3.4.1 Description of Quicksort 41
3.5 Predictability of branches in Quicksort 43
3.6 Simulation results 45
3.7 Summary 47

CHAPTER 4
DESIGN OPTIMIZATION FOR HIGH-SPEED PER-ADDRESS TWO-LEVEL BRANCH PREDICTORS 48

4.1 Introduction 48
4.2 Per-address two-level branch predictors 50
4.2.1 Tagless implementation 52
4.3 Performance analysis for tagless predictors 53
4.3.1 Miss handling policies 53
4.3.2 Simulation methodology 54
4.3.3 Simulation results 55
4.3.4 Analysis using transitional-state and steady-state error 59
4.4 Cost-benefit analysis for tagless predictors 61
4.4.1 Cost/performance analysis 62
4.4.2 Design principles 68

4.5 Conclusion 69

CHAPTER 5
IMPROVING INSTRUCTION FETCH BANDWIDTH AND I-CACHE PERFORMANCE USING DATA COMPRESSION 71

5.1 Introduction 71

5.2 Intrinsic compressibility (redundancy) in programs 72
  5.2.1 Patterns 72
  5.2.2 Instruction fetch bottleneck 74

5.3 Description of the compression technique 74

5.4 Simulation results 76
  5.4.1 I-cache fetch behavior 78
  5.4.2 I-cache miss behavior 80
  5.4.3 Pattern table sizes 81

5.5 Discussion of implementation issues 86

5.6 Related research 87

5.7 Conclusion and future research 88

CHAPTER 6
PREFETCHING USING BRANCH PREDICTION INFORMATION 90

6.1 Introduction 90

6.2 Description of prefetching schemes 91
  6.2.1 Related prefetching schemes 91
  6.2.2 Branch prediction-based prefetching 92

6.3 Simulation environment 95
  6.3.1 Simulation of speculative execution 95
  6.3.2 Description of benchmarks 96
  6.3.3 Hardware assumption 97
  6.3.4 Bus arbitration policy 98

6.4 Simulation results and analysis 99

6.5 Discussion of implementation issues 112

CHAPTER 7
CONCLUSIONS 116

BIBLIOGRAPHY 120

LIST OF FIGURES

Figure 2.1: A conceptual system model for branch prediction 8
Figure 2.2: A two-step model for data compression 12
Figure 2.3: Example of a Markov predictor of order 2 14
Figure 2.4: Prediction flowchart of a PPM predictor of order m 15
Figure 2.5: Popular variations of two-level predictors 16
Figure 2.6: A two-level branch predictor vs. a Markov predictor 17
Figure 2.7: Misprediction rate for direct-mapped BTB with 1024 entries 24
Figure 2.8: Improved accuracy of PPM predictor with a direct-mapped BTB wit 128 entries 25
Figure 2.9: Comparison of hardware requirement of a PPM and a PAg of the same history length 27
Figure 2.10: Branch predictors as a subset of predictors used in data compression . . . 31
Figure 3.1: Example of a Lempel-Ziv predictor 37
Figure 3.2: A Quicksort program and its two comparison branches 42
Figure 3.3: Comparison of prediction accuracy for Quicksort 46
Figure 4.1: Schematic for a per-address two-level branch predictor 50
Figure 4.2: Tagged per-address two-level branch predictor 51
Figure 4.3: Tagless per-address two-level branch predictor 51
Figure 4.4: An example comparing the flush and no-flush policies 54
Figure 4.5: Misprediction rate of PAg with 8-bit history length for IBS 55
Figure 4.6: Misprediction rate of PAg with 8-bit history length for SPEC CINT95 . 56
Figure 4.7: Misprediction rate of PAg with 14-bit history length for IBS 57
Figure 4.8: Misprediction rate of PAg with 14-bit history length for SPEC CINT95 57
Figure 4.9: Illustration for the three parameters of per-address scheme cost function 61
Figure 4.10: Equal-cost contours for 128 branch history entries 62
Figure 4.11: Equal-cost contours for 8k branch history entries 63
Figure 4.12: Misprediction rate vs. budget for SPEC CINT95 64

Figure 4.13: Misprediction rate vs.budget for IBS . . . 65
Figure 4.14: The optimal configuration for each budget in SPEC CINT95 . . . 66
Figure 4.15: The optimal configuration for each budget in IBS . . . 67
Figure 5.1: Organization of the compression scheme . . . 75
Figure 5.2: Bytes needed and bus cycles used for SPEC CINT95 benchmarks . . . 78
Figure 5.3: Bytes needed and bus cycles used for SPEC CFP95 benchmarks . . . 79
Figure 5.4: Average miss ratio for SPEC CINT95 benchmarks . . . 81
Figure 5.5: miss ratios for each individual benchmark . . . 82
Figure 5.6: Comparison of the compression effects with 32 patterns and 128 patterns . . . 85
Figure 5.7: A five-stage pipeline using predecoded information to reduce the delay of decompression lookup . . . 86
Figure 6.1: Conceptual organization of BP-based prefetching scheme . . . 93
Figure 6.2: Flowchart of BP-based prefetching . . . 93
Figure 6.3: A gshare branch predictor . . . 98
Figure 6.4: Performance measure: stall overhead for different schemes . . . 100
Figure 6.5: Total prefetches generated in each scheme and the classification of these prefetches . . . 102
Figure 6.6: Further classification of useful prefetches . . . 105
Figure 6.7: Bus traffic for different schemes . . . 107
Figure 6.8: Percentage of utilization for the bus to level-2 cache for different schemes . . . 110
Figure 6.9: A cache with BP-based prefetching achieves lower execution time than a plain cache of 4 times the size . . . 112
Figure 6.10: A possible implementation of branch prediction-based prefetching . . . 115

LIST OF TABLES

Table 2.1: Summary of current popular prediction schemes 11
Table 2.2: Description of Instruction Benchmark Suite (IBS) workloads 21
Table 2.3: Input data set used for SPEC CINT 95 benchmarks 22
Table 2.4: Static and dynamic conditional branch counts in the IBS and SPEC CINT95 programs 23
Table 3.1: Prediction accuracy of GAg style of two-level predictor and Lempel-Ziv on SPECInt92 programs 40
Table 4.1: Detail misprediction rates for tagged and tagless predictors 58
Table 5.1: Instruction counts 76
Table 5.2: Pattern table sizes in bytes 85
Table 6.1: Statistics of the benchmarks used 97

CHAPTER 1
INTRODUCTION

1.1 The need for fast instruction fetching

The speed of current microprocessors continues to increase due to advanced technology and aggressive designs. This high speed mainly results from increasingly higher clock rate, more functional units, wider instruction issuing and deeper speculative execution.

However, the increasing speed of microprocessors also stresses the need for ever faster instruction fetching rate. To sustain the full speed of microprocessors, the instruction fetching rate must increase proportionally to supply the instructions required. Fast instruction fetching rate heavily depends on the solution to these critical issues: accurate branch prediction, high instruction cache hit rate, and high bandwidth for fast transfer rate.

To enable aggressive speculative execution and issuing beyond one basic block per cycle, accurate branch prediction is necessary. This is because conditional branch instructions depend on values produced by previous instructions, but, in current deep-pipelined designs, those previous instructions generally take several cycles to complete. Therefore, to execute at least one branch every cycle, we need accurate branch prediction to predict the direction of branches and to speculatively execute beyond one basic block.

Since memory speeds can not keep up with microprocessor speeds, multiple levels of cache structures are designed to reduce this gap. Consequently, high hit rates in caches become critical for fast and steady instruction supply. In addition, high-bandwidth buses are needed for fetching instructions into the cache. This may be a problem because

microprocessors usually have limited pin counts, which limit the widths of buses.

Furthermore, to increase cache hit rates and to hide the transfer latency among various levels of cache structures, efficient prefetching is needed. Prefetching can reduce misses in the cache by anticipating the instructions needed in the future and fetching these instructions into the cache before they are requested. By fetching in advance, prefetching also hides the transfer latency by overlapping it with regular microprocessor execution.

1.2 Data compression as a solution

Data compression is a mature and well established field that has been studied for decades. However, there is little attempt to borrow techniques from data compression to solve critical issues in microprocessor design. In this dissertation, I will show opportunities to apply data compression to improve various important aspects for fast instruction fetching: branch prediction, instruction fetch bandwidth, and instruction cache performance.

I will first show that data compression is closely related to branch prediction and can be applied to improve prediction accuracy. Then, using techniques in data compression, I will establish a theoretical basis for current branch predictors as well as point out directions for future improvements. Data compression can even provide us optimal predictors, yet the design of branch predictors still remains a cost-effective optimization problem due to implementation and budget constraints. In addition, I will demonstrate that, by compressing instruction streams, data compression offers an excellent way to improve the cache hit rate and instruction fetch bandwidth. When carefully designed, compression algorithms can be implemented efficiently in hardware to be incorporated into microprocessors. Then using these techniques as basic building blocks, I design an effective prefetching algorithm using branch prediction information to further improve the instruction fetching rate.

1.3 Organization of this dissertation

This dissertation is based on work described in [Chen96a, Chen96b, Mudge96, Chen97a, Chen97b, Chen97c] and is organized into five chapters. Chapter 2 first demonstrates the correspondence between branch prediction and data compression. Then we establish a theoretical basis for current branch predictors by showing that "two-level" or correlation based predictors are, in fact, simplifications of an optimal predictor in data compression, Prediction by Partial Matching (PPM). If the information provided to the predictor remains the same, it is unlikely that significant improvements can be expected (asymptotically) from two-level predictors, since PPM is optimal. However, there is a rich set of predictors available from data compression, several of which can still yield some improvement in cases where resources are limited. To illustrate this, we conduct trace-driven simulation running the Instruction Benchmark Suite and the SPEC CINT95 benchmarks. The results show that PPM can outperform a two-level predictor for modest sized branch target buffers.

After showing that PPM can be successfully applied to branch prediction, Chapter 3 further examines the performance of another optimal algorithm from data compression: Lempel-Ziv algorithm (found in Unix *compress*). Moreover, we will show that, for some programs, the theoretical limit of predictability can be derived using exact analysis. We have chosen Quicksort algorithm to illustrate this point and then use this limit to calibrate the performance of various branch prediction schemes.

Although optimal predictors can be derived using techniques from data compression, the actual design of branch predictors still remains a cost-effective optimization problem due to implementation and budget constraints. In particular, optimal designs vary with target technology and hardware budget. To identify optimal designs, we need to consider all parameters in a branch predictor and evaluate their interaction.

Chapter 4 shows how the design style of optimal predictors changes due to high

clock rate, and how a comprehensive analysis can be done to determine the best design configuration. We choose per-address two-level branch predictors for illustration, because they have been shown to be among the best predictors and have been implemented in current microprocessors.

Chapter 5 introduces a simple and efficient data compression technique to improve instruction cache hit rates and to increase instruction fetch bandwidth. After code generation, a program is executed and profiled to find frequently used instruction sequences. By mapping these instruction sequences into single byte opcodes, we can effectively compress multiple-instruction, multi-byte operations onto a single byte. When these compressed opcodes are detected in program execution, they are dynamically expanded within the CPU into the original instruction sequences. By restricting the instruction sequence within a basic block (excluding branches), branch instructions and their targets are unaffected by this technique allowing compression to be decoupled from compilation.

Although the compiler was not optimized to exploit our instruction compression technique, we effectively reduced both the I-cache byte fetch requirements and the I-cache miss rates for the SPEC95 benchmarks. The average bytes needed from level-1 cache were reduced by 50% for integer benchmarks, and 70% for floating point benchmarks. The average bus cycles needed to fetch instructions from level-1 cache were reduced by 35% for integer benchmarks, and 65% for floating point benchmarks. In addition, a compression enhanced cache has a lower miss rate than a plain cache twice the size for integer benchmarks.

Using branch prediction as a basic building block, Chapter 6 presents a novel instruction prefetching algorithm to further increase instruction fetching rate. Instruction prefetching can effectively reduce instruction cache misses and hide transfer latency, thus improving the performance. We propose a prefetching scheme, which employs a branch predictor to run ahead of the execution unit and to prefetch potentially useful instructions.

Branch prediction-based (BP-based) prefetching has a separate small fetching unit, allowing it to compute and predict targets autonomously. Our simulations show that a 4-issue machine with BP-based prefetching achieves higher performance than a plain cache 4 times the size. In addition, BP-based prefetching outperforms other hardware instruction fetching schemes, such as next-*n* line prefetching and wrong-path prefetching, by a factor of 17-44% in stall overhead.

Finally, Chapter 7 presents summaries of previous chapters and some concluding remarks.

CHAPTER 2

ANALYSIS OF BRANCH PREDICTION VIA DATA COMPRESSION

2.1 Introduction

As the design trends of modern superscalar microprocessors move toward wider instruction issue and deeper super-pipelines, effective branch prediction becomes essential to exploring the full performance of microprocessors. A good branch prediction scheme can increase the performance of a microprocessor by eliminating instruction fetch stalls in the pipelines. As a result, numerous branch prediction schemes have been proposed and implemented on new microprocessors [MReport95b, MReport95c, MReport96].

Many researchers focus on designing new branch prediction schemes based solely on comparing simulation results. However, very few studies address the theoretical basis behind these prediction schemes. Knowing the theoretical basis helps us to assess how good a prediction scheme is, as well as how much more we can improve the existing predictors.

To establish a theoretical basis, we first introduce a conceptual system model to characterize components in a branch prediction process (this work has also been published in [Chen96a]). Using this model, we notice that many of the best prediction schemes [Pan92, Yeh92b, Yeh93, McFarling93, Chang94, Nair95b] use predictors similar to a "two-level" adaptive branch predictor [Yeh91]. Then, we demonstrate that these "two-level like" predictors are, in fact, simplified implementations of an optimal predictor in data compression, Prediction by Partial Matching (PPM) [Cleary84, Moffat90]. This establishes a theoretical basis for current two-level predictors that can draw on the

relatively mature field of data compression.

In particular, the potential benefit of applying data compression techniques to branch prediction is readily apparent in the similarity of predictors used in both methods. In practice, the predictors used in branch prediction are only a very small subset of the predictors developed in data compression. To illustrate the potential improvement using data compression techniques, we have conducted trace-driven simulations. The results show that PPM outperforms an equivalent two-level predictor on the Instruction Benchmark Suite (IBS) [Uhlig95a] and the SPEC CINT95 [SPEC95] benchmarks. The improvement is not great, because two-level predictors are near optimal. However, in the case of modest size systems, the improvement is more significant.

This chapter is organized into seven sections. In Section 2.2 we introduce a conceptual system model to describe the process and components of branch prediction. We then use this model to summarize current popular branch prediction schemes. In Section 2.3, we show that data compression is relevant to branch prediction because it also requires prediction.

In Section 2.4, we develop a theoretical basis for two-level predictors by demonstrating that they are simplified versions of an optimal predictor, PPM. Section 2.5 considers the implications of the availability of optimal predictors, and shows that, in some cases, branch prediction can still benefit from data compression. We verify this with trace-driven simulation again running the IBS and the SPEC CINT95 benchmarks. Section 2.7 discusses the potential benefits from data compression and further improvement in each component of our conceptual prediction model. Finally, we present conclusions and further work in Section 2.8.

2.2 System model and overview of branch prediction

To explain branch prediction schemes, a conceptual view of a branch prediction scheme is introduced. This conceptual view allows us to compare various branch

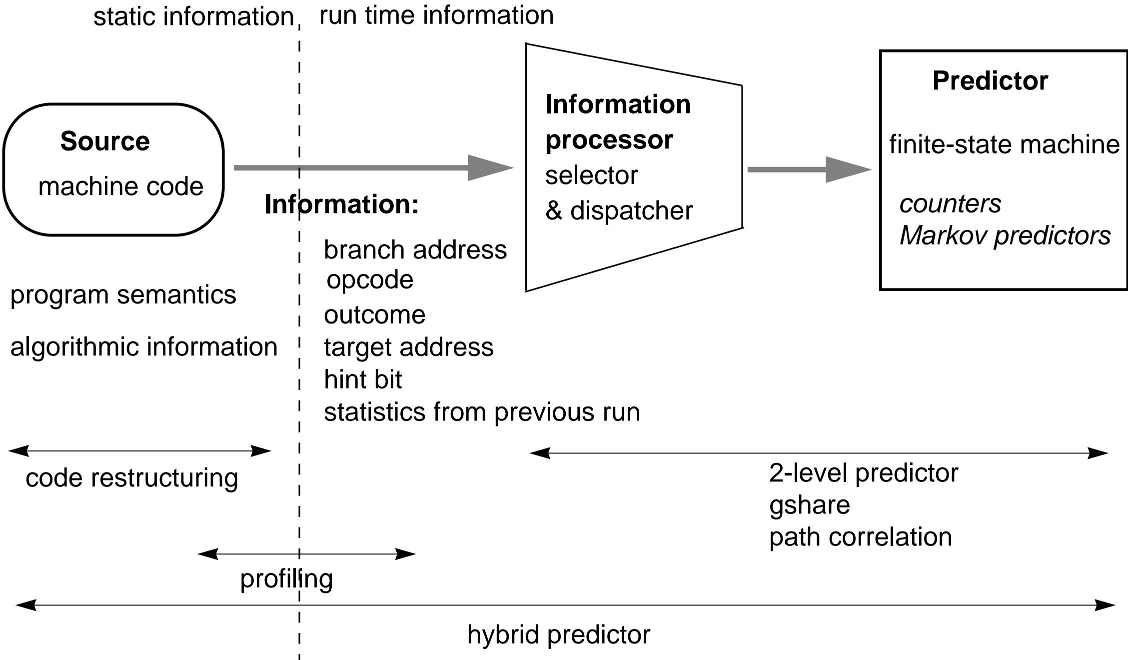


Figure 2.1: A conceptual system model for branch prediction

prediction schemes. It also enables us to focus and improve each component by clearly defining its function. This conceptual model elaborates on the one in [Young95]. Our model extends it to accommodate most popular branch prediction schemes.

2.2.1 A general conceptual system model for branch prediction

The general conceptual model we introduce for branch prediction consists of three major components: a source, an information processor, and a predictor, as illustrated in Figure 2.1. Although some components are often combined in a hardware implementation, this three-part model is useful in explaining the principles behind different prediction schemes.

2.2.1.1 Source

The source is simply the machine code of the programs we are running. The source contains program semantics and algorithmic information. To aid branch prediction, this

information can be explored and extracted during the compile-time. It can be stored and passed on to be used during execution. A hint bit in branch instructions is one means of passing this information. In addition, the source can be modified to produce more predictable branches using statistics from previous test-runs. This is how code restructuring and code profiling work.

2.2.1.2 Information processor

In a hardware implementation, the information processor is often combined with predictors and, hence, overlooked. However, the information processor plays a key role in the prediction process and thus deserves a close study. Conceptually, it can be subdivided into two components: selector and dispatcher.

2.2.1.2.1 Selector

The selector selects which run-time information should be used for branch prediction and encode it. This information can include branch address, operation code, branch outcome, target address, hint bits, or statistics from test-runs. Prediction accuracy depends heavily on the mix of run-time information that is employed. Indeed, significant improvement in future branch prediction schemes are likely to come from providing new prediction information.

Once the information is determined, the selector decides how to represent the information. For example, suppose branch outcomes and branch addresses are selected as information upon which predictions are to be made, then the selector can combine the outcomes with addresses into one single stream or keep outcomes as individual streams classified by branch addresses. A good encoding can result in a concise and efficient representation that helps prediction.

2.2.1.2.2 Dispatcher

The dispatcher determines how the information is mapped (fed) to the various predictors, since multiple information streams and predictors may exist in a prediction scheme. The mapping can be one-to-one, many-to-one, one-to-many, dedicated, or multiplexed (time-shared). Different mappings often have great influence on the final prediction accuracy.

2.2.1.3 Predictor

A predictor is simply a finite-state machine that takes input and produces a prediction. It does not need to know the meaning of the input. Common examples are a constant or static predictor, a 1-bit counter, a 2-bit up-down saturating counter [Smith81], and a Markov predictor. A Markov predictor forms the basis of recent two-level prediction schemes and is discussed in detail in Section 2.3 For the moment, a Markov predictor is simply a finite-state machine that generates predictions based on a finite number of previous inputs.

2.2.2 Overview of current branch prediction schemes

Using the conceptual model just introduced, we can summarize current popular branch prediction schemes in Table 2.1. This table describes the basic components used in each prediction scheme. It lists whether source modification or profiling are used for each prediction scheme. For the information processor, it describes both the information used by the selector and the way dispatcher maps information. Finally, the predictor used in each prediction scheme is also listed. From this table, we notice that many of the best prediction schemes [Pan92, Yeh92b, Yeh93, McFarling93, Chang94, Nair95b] use Markov predictors. We will explain this further in Section 2.4.

Prediction scheme	Source modification or profiling	Information processor		predictor
		selector	dispatcher	
forward not-taken, backward taken (FNTBT)	no	(address - target)	many-to-one	constant
2-bit counter	no	outcome, classified by address	one-to-one, mapped with address	2-bit counters
path correlation	no	target, in execution order	one-to-many, mapped with address	several Markov predictors
gshare	no	address, outcome, XOR together	one-to-one	a Markov predictor
GAg	no	outcome, in execution order	one-to-one	a Markov predictor
GAs	no	outcome, in execution order	one-to-many, mapped with address	several Markov predictors
PAg	no	outcome, classified by address	many-to-one, multiplexed	a Markov predictor
PAs	no	outcome, classified by address	many-to-many, mapped with address	several Markov predictors
PSg	no	outcome, classified by address	many-to-one, multiplexed	constant
branch correlation	yes	statistics from previous runs, hint bit	one-to-one, mapped with address	constant
hybrid predictor	yes	combinations of above	combinations of above	combinations of above

Table 2.1: Summary of current popular prediction schemes

2.3 Data compression and prediction

Like branch prediction, data compression relies on prediction. In data compression, the goal is to represent the original data with fewer bits. The basic principle of data compression is to use fewer bits to represent frequent symbols, while using more

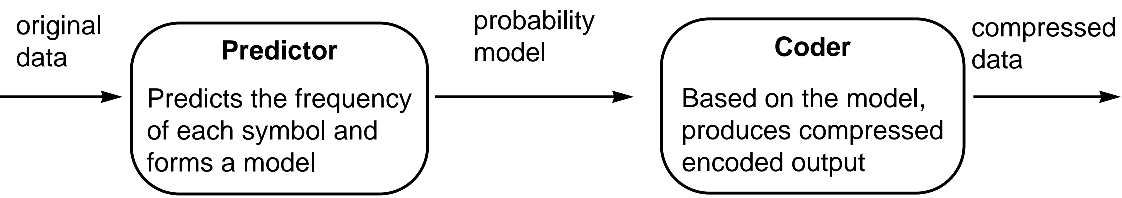


Figure 2.2: A two-step model for data compression

---

bits to represent infrequent symbols. Thus, the net effect is to reduce the overall number of bits needed to represent the original data. In order to perform this compression effectively, a compression algorithm has to predict future data accurately to build a good probabilistic model for the next symbol [Bell90]. Then, as shown in Figure 2.2, the algorithm encodes the next symbol with a coder tuned to the probability distribution. Current coders can encode data so effectively that the number of bits used is very close to optimal and, consequently, the design of good compression relies on an accurate predictor. The problem of designing efficient and general universal compressors/predictors has been extensively examined. In our experiments we draw on these techniques, adapting them to the new context of branch prediction.

2.3.1 Prediction by Partial Matching

Prediction by partial matching (PPM) is a universal compression/prediction algorithm that has been theoretically proven optimal and has been applied in data compression and prefetching [Cleary84, Krishnan94, Kroeger96, Moffat90, Vitter91]. Indeed, it usually outperforms the Lempel-Ziv algorithm (found in Unix *compress*) due to implementation considerations and a faster convergence rate [Curewitz93, Bell90, Witten94]. As described above, the PPM algorithm for text compression consists of a predictor to estimate probabilities for characters and an arithmetic coder. We only make use of the predictor. We encode the outcomes of a branch, taken or not taken, as 1 or 0

respectively. Then the PPM predictor is used to predict the value of the next bit given the prior sequence of bits that have already been observed.

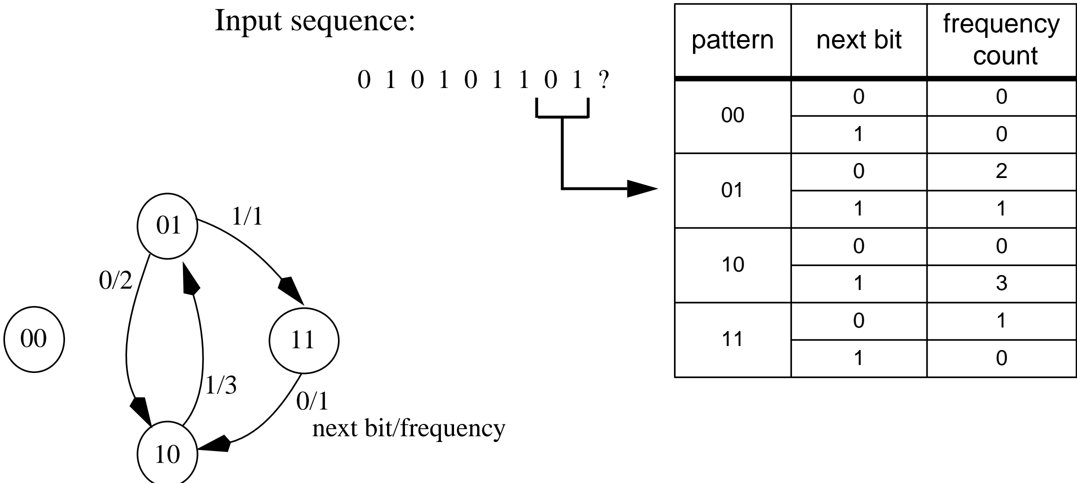
2.3.1.1 Markov predictors

The basis of the PPM algorithm of order m is a set of $(m + 1)$ Markov predictors. A Markov predictor of order j predicts the next bit based upon the j immediately preceding bits—it is a simple Markov chain [Ross85]. The states are the 2^j possible patterns of j bits. The transition probabilities are proportional to the observed frequencies of a 1 or a 0 that occur given that the predictor is in a particular state (has seen the bit pattern associated with that state). The predictor builds the transition frequency by recording the number of times a 1 or a 0 occurs in the $(j + 1)$-th bit that follows the j-bit pattern. The chain is built at the same time that it is used for prediction and thus parts of the chain are often incomplete. To predict a branch outcome the predictor simply uses the j immediately preceding bits (outcomes of branches) to index a state and predicts the next bit to correspond to the most frequent transition out of that state.

Figure 2.3 illustrates how a Markov predictor works. Let the input sequence seen so far be 010101101, and the order of Markov predictor be 2. The next bit is predicted based on the two immediately preceding bits, that is, 01. The pattern 01 occurs 3 times previously in the input sequence. The frequency counts of the bit following 01 are: 0 follows 01 twice, and 1 follows 01 once. Therefore, the predictor predicts the next bit to be 0 with a probability of 2/3. The (incomplete) 4-state Markov chain is shown at the left of the figure. Note that a 0-th order Markov predictor simply predicts the next bit based on its relative frequency in the input sequence.

2.3.1.2 Combining Markov predictors to perform PPM

We noted earlier that the basis of a PPM algorithm of order m is a set of $(m + 1)$ Markov predictors. The algorithm is illustrated in Figure 2.4. PPM uses the m



pattern	next bit	frequency count
00	0	0
	1	0
01	0	2
	1	1
10	0	0
	1	3
11	0	1
	1	0

Figure 2.3: Example of a Markov predictor of order 2

The Markov chain at left corresponds to the information collected from the input sequence in the table at right. Note that the chain is incomplete, because of 0 frequency count transitions.

---

immediately preceding bits to search a pattern in the highest order Markov model, in this case *m*. If the search succeeds, which means the pattern appears in the input sequence seen so far (the pattern has a non-zero frequency count), PPM predicts the next bit using this *m*th-order Markov predictor as described in the previous subsection. However, if the pattern is not found, PPM uses the ($m - 1$) immediately preceding bits to search the next lower order ($m - 1$)-th order Markov predictor. Whenever a search misses, PPM reduces the pattern by one bit and uses it to search in the next lower order Markov predictor. This process continues until a match is found and the corresponding prediction can be made.

There are a number of variations on how the frequency information in the individual Markov predictors can be updated as the PPM process proceeds. In our experiments we use *update exclusion*. This means that we only update the frequency counters for the predictor that makes the prediction and the predictors with higher order. Lower order predictors are not updated.

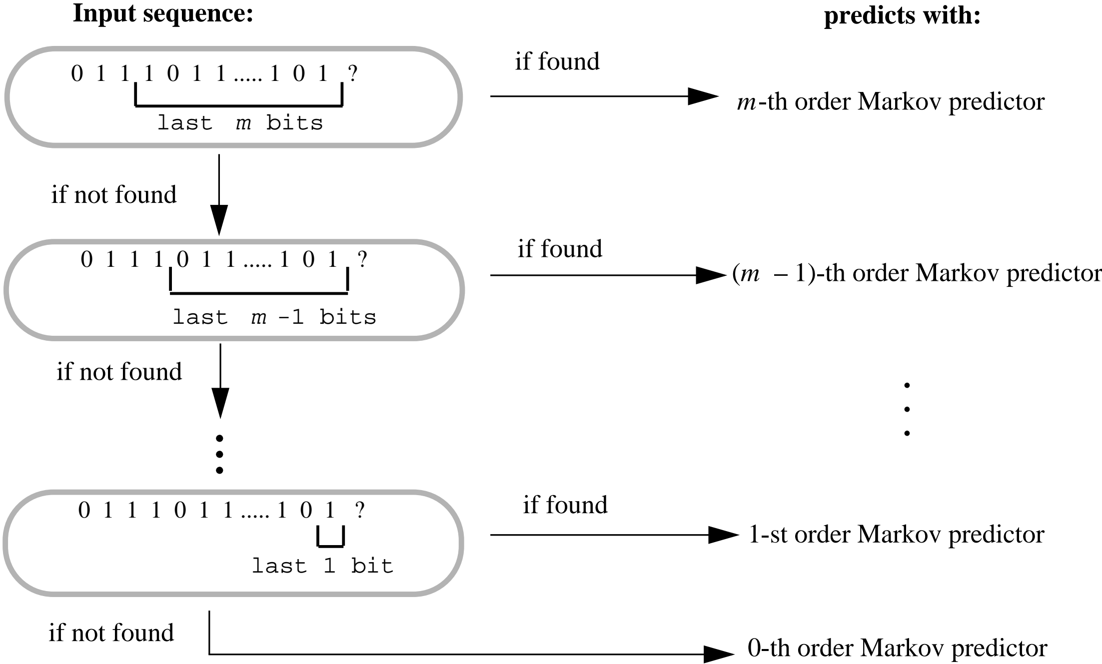


Figure 2.4: Prediction flowchart of a PPM predictor of order *m*

2.4 Two-level branch prediction as an approximation of PPM

In this section, we show that recently proposed two-level or correlation based predictors are approximations of, PPM, an optimal prediction algorithm.

2.4.1 Description of two-level predictor

Among the various branch prediction schemes, two-level or correlation based predictors are among the best. In addition, these predictors all share very similar hardware components. As Figure 2.5 shows, they have one or more shift-registers (branch history registers) to store history information in the first level and have one or more tables of 2-bit counters (pattern history tables) in their second level [Yeh91]. The contents of the first level shift-registers are typically used to select a 2-bit counter in one of the second-level tables. Predictions are made based on the value of the 2-bit counter selected.

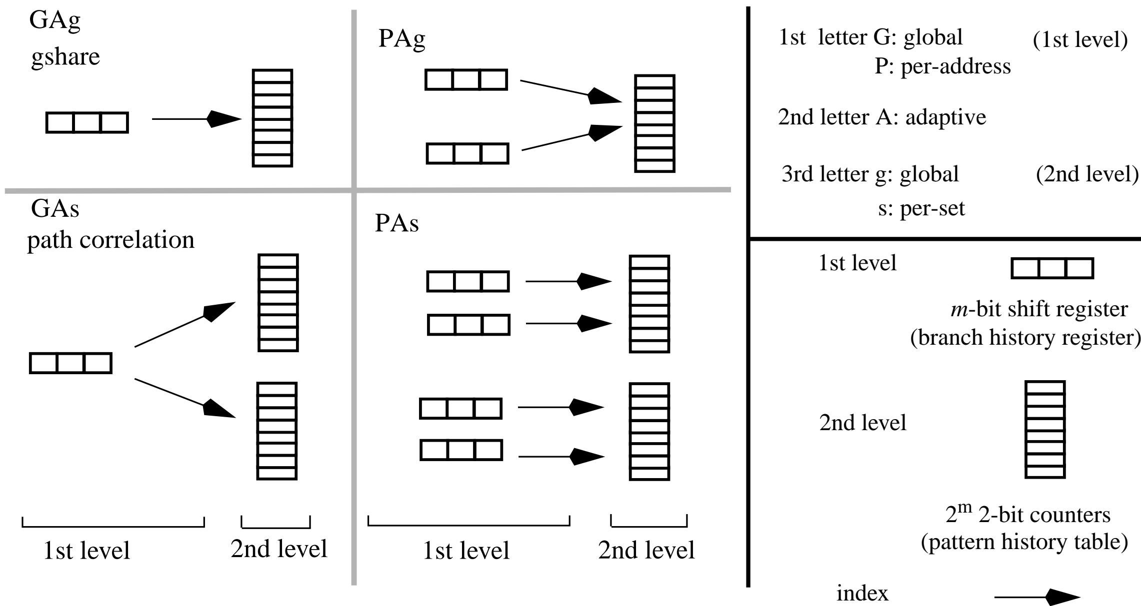


Figure 2.5: Popular variations of two-level predictors

2.4.2 Two-level branch predictors as Markov predictors

From the above discussion on two-level adaptive branch predictors and the one on Markov predictors in Section 2.3.1.1, it can be seen that there are strong similarities. Though different schemes of two-level branch predictors exist, they differ only in what information is used for history and what subsets of branch outcomes are used to index and update the counters. As a result, there exists a corresponding Markov predictor for each scheme.

Figure 2.6 shows the similarity between a two-level predictor and a Markov predictor. Both predictors behave exactly the same in the first level. They both use the last *m* bits of branch outcome to search the corresponding data structure. Note that an *m*-bit shift register serves two functions: first, it limits the information used for prediction to *m* previous outcomes and, second, it uniquely defines a finite-state machine in which each state has exactly two predefined next states. In the second level, the Markov predictor uses a frequency counter for each outcome, while the two-level predictor uses a saturating up-

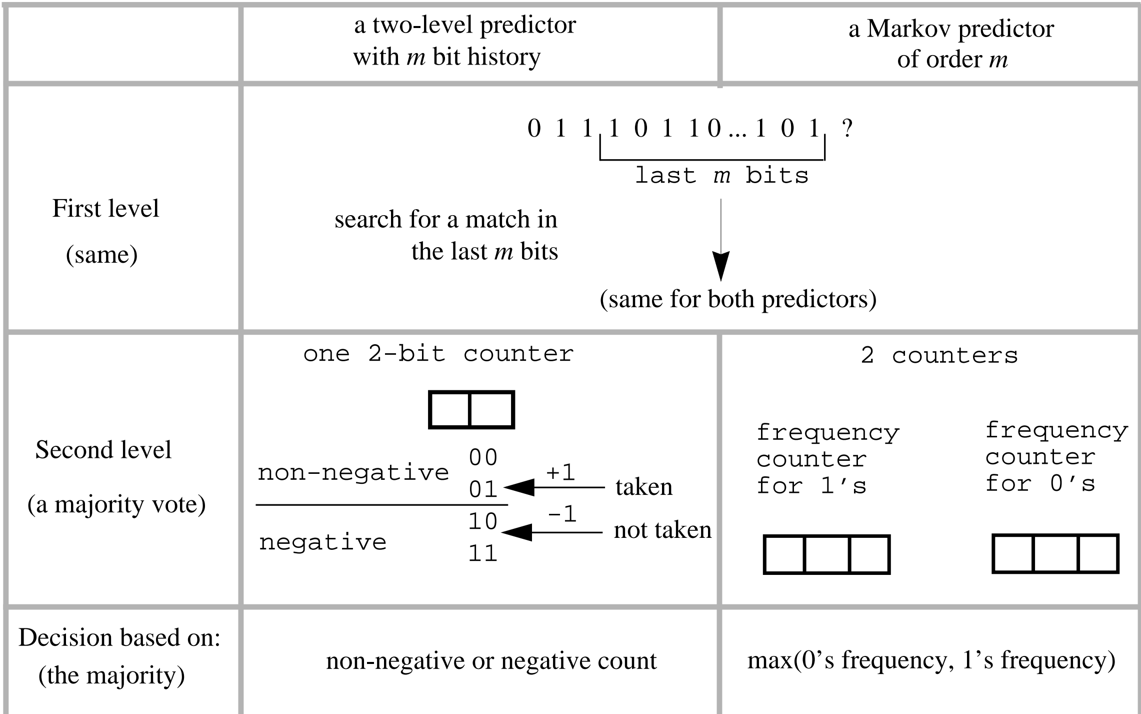


Figure 2.6: A two-level branch predictor vs. a Markov predictor

---

down 2-bit counter [Smith81]. Whenever a branch is taken/not taken, the 2-bit counter increments/decrements. The decision for a two-level predictor depends on whether the value of the counter falls in the non-negative half or the negative half. Similarly, a Markov predictor simply predicts the next branch to be the most frequent outcome based on two frequency counters. Both predictors are utilizing a majority vote via different implementations. The saturating counter is an approximation to this that can be realized in hardware efficiently.

An interesting illustration is to see how a two-level predictor, the per-address branch history register with global pattern history table (PAg), corresponds to a Markov predictor. This per-address scheme uses one table of 2-bit counters and multiple shift registers where each register records only outcomes of a particular branch. Although multiple shift registers exist, all shift registers operate the same and correspond to the same transition rule for a finite-state machine (state diagram). In addition, all shift

registers share the same global table of 2-bit counters and, hence, share the same value (counter) in each state. Therefore, this per-address scheme uses one Markov predictor that is time-shared and updated among various branches.

2.4.3 Approximation to optimal predictors

Given the arguments in the previous sections, we now show that a two-level predictor is an approximation of an optimal predictor. As mentioned in Section 2.3.1, PPM is a theoretically proven optimal predictor consisting of a set of Markov predictors. Though performance is inferior at the beginning, a single Markov predictor can approach the performance of PPM in the long run (asymptotically) [Bell90]. Furthermore, we have shown that a two-level predictor is a simplified Markov predictor. Therefore, we can see that a two-level predictor is an approximation of an optimal predictor, PPM.

Under hardware implementation constraints, a two-level predictor is a reasonable simplification of PPM. The complete PPM predictor can be viewed as a set of two-level predictors, having not one size of predictor (m) but a set that spans m down to 0 (a simple two-bit counter—equivalent to a per-address predictor with zero history length). These extra small predictors help to reduce "cold starts," i.e., lack of information at the training period. Although two-level predictors do not include small predictors, they still can perform well since cold starts are far less severe in branch prediction than in text compression. To see how cold starts differ in the two fields, we consider the number of all possible combinations of m outcomes. In branch prediction, there are 2^m possible combinations since a branch has only two possible outcomes (taken or not taken). In text compression, on the other hand, there are roughly 128^m possible combinations where 128 is the number of printable ASCII symbols. Compared to the large number of branches executed in typical programs, these 2^m cold starts are negligibly small and hardly decrease the overall prediction accuracy. Another simplification made by two-level predictors is the use of a 2-bit counter instead of an n-bit counter. This is a cost-effective choice, since two

bits is the minimal number needed so that the direction of the predictor is not changed by the single exit in a loop statement [Lee84, Smith81].

As an aside, note that it is not coincidental that a 2-bit saturating up-down counter is the best among 4-state predictors [Nair95a]. This is because, with four states, one 2-bit saturating up-down counter is the best way to mimic the majority vote used in the Markov predictor. In the original Markov predictor, this voting prediction is done with two frequency counters (one for each outcome).

2.5 Impact of optimal predictors and further improvement

2.5.1 Implication of optimal predictors

Having shown that a two-level predictor is an approximation of an optimal predictor, we have established a theoretical basis for this type of branch predictor. Rather than just comparing simulation results, which does not tell us how well these predictors perform in general, we can now have a reasonable degree of confidence in the performance of two-level predictors. It is unlikely that, by improving the predictor component alone, we can generate significant improvement in prediction accuracy excepting in special cases discussed below. This is because two-level predictors already perform close to optimal under the constraints imposed by the information they are given. Of course, the inclusion of more information (e.g., knowledge about the program executing) can always be used to improve the prediction accuracy.

2.5.2 Assumptions for optimal predictors

It is important to clarify just what we mean by “optimal.” The optimality of PPM assumes that the source of data (branch outcomes in our case) is a stationary and ergodic random process; in other words, outcomes are time invariant and future outcomes can be predicted with samples drawn from present ones. This type of source modeling has been

successfully applied to a broad range of applications including English text [Bell90]. In our work, we have assumed this source model applies to branch outcomes. It seems reasonable to assume that the statistics of branches is not influenced by the point in execution time when one starts to observe them, because we consider our branch outcomes to be drawn from a sequence created by the ordered execution of all well-formed programs. With these assumptions, a source model that is ergodic and stationary is a reasonable choice. However, there are some concerns that limit the extent to which the optimality claims which follow from the model apply. The most obvious is that optimality, in the sense used here, only implies that a scheme will perform as well as any other scheme in the long run. However, nothing is guaranteed about the rate of convergence, or the time before a scheme reaches its best performance. Therefore, though an optimal scheme will perform well eventually, in practice, it may take a long time to reach that state. In addition, the implementation of an optimal scheme may need a large memory and other resources.

From the above statement we cannot conclude whether an optimal predictor will perform well if a program ends quickly, or if the time between context switches is short. Fortunately, a typical time frame between context switches is relatively long and may contain anywhere from about 10 thousand to 100 thousand branches. Furthermore, empirical evidence described in the next section and reported in [Bell90, Witten94] suggests PPM has a fast convergence rate.

2.5.3 Illustration of modest improvements using PPM techniques

In this section, we illustrate that techniques from data compression can still, in some cases, yield modest improvements to branch prediction. To assess and confirm the potential improvement, we conduct trace-driven simulations. As input for the simulation, we use the Instruction Benchmark Suite (IBS) benchmarks [Uhlig95a] and the SPEC CINT95 benchmark suite [SPEC95] for our simulation.

IBS benchmarks	
Benchmarks	Description
groff	GNU C++ implementation of the Unix ‘nroff’ text formatting program. Version 1.09.
gs	‘Ghostscript’ version 2.4.1. Displays single page of text and graphics.
mpeg_play	‘mpeg_play’ version 2.0. Displays 85 frames.
nroff	Unix text formatting program whipped with Ultrix 3.1.
real_gcc	GNU C compiler version 2.6.
sdet	Multiprocess benchmark from the SPEC SDM suite.
verilog	Verilog-XL version 1.6b. Simulates a microprocessor.
video_play	Modified ‘mpeg_play.’ Displays 610 frames.

Table 2.2: Description of Instruction Benchmark Suite (IBS) workloads

---

The IBS benchmarks are a set of applications designed to reflect realistic workloads. A brief description of benchmarks in IBS is listed in Table 2.2. The traces of these benchmarks are generated through hardware monitoring of a MIPS R2000-based workstation. We use the traces collected under the operating system Ultrix 3.1, which include both kernel-level and user-level instructions.

For the SPEC CINT95 benchmark suite, we used ATOM [Eustace95], a code instrumentation interface from Digital Equipment Corporation, to collect our traces. The benchmarks are first instrumented with ATOM, then executed on a DEC 21064-based workstation running the OSF/1 3.0 operating system to generate traces. These traces contain only user-level instructions. The input data set used for generating traces is summarized in Table 2.3.

The statistics of traces from the IBS and the SPEC CINT95 are summarized in Table 2.4. All traces are identical in format and are used as input to our simulator.

By using a set of small predictors, PPM can predict relatively well in situations where little history information is available, such as conflict misses or cold starts. In particular, this occurs in per-address prediction schemes where a finite branch-target

SPEC CINT 95 benchmarks	
Benchmarks	input data set
compress	reduced version of *bigtest.in* (reference data), reduced to 30,000 elements (instead of 14,000,000)
gcc	*jump.i* (one of the reference data sets)
go	*2stone9.in* (training data), reduced the game_level to 19 (instead of 50)
ijpeg	*specmun.ppm* (test data)
li	*train.lsp* (training data)
perl	reduced version of *scrabbl.in* (reference data), reduced to the first 5 items (instead of 7)
vortex	reduced version of training data, reduce iteration counts and data to the first 10 items (instead of 250)

Table 2.3: Input data set used for SPEC CINT 95 benchmarks

Input to the SPEC CINT95 benchmarks was a reduced input data set; each benchmark was run to completion.

---

buffer is used to record individual branch history, since the history of a particular branch may be overwritten by that of other branches due to contention in the finite buffer (conflict misses). For these cases, PPM can be used to alleviate the problem.

To illustrate potential improvement, we compare the PAg two-level predictor scheme and PPM. PAg means that the inputs are divided into per-address branch outcome streams, and then they are fed into one global predictor. Compared to global schemes, the advantage of PAg, and other per-address schemes, is that aliasing that may arise by mixing streams from different branch histories is reduced. However, conflict misses in the branch target buffer (BTB) become a more significant problem because only finite records of distinct branches can be maintained. This is due to limited buffer size; consequently, the history will be replaced and lost from time to time. The conflict miss problem gets worse as the number of distinct branches increases, since more individual branch outcome streams must be recorded.

Figure 2.7 shows the misprediction results for a direct-mapped BTB with 1024

	Benchmarks	static conditional branches	dynamic conditional branches
SPEC CINT95	compress	95	10,216,264
	gcc	15,647	24,048,361
	go	4,742	18,168,554
	ijpeg	902	40,854,598
	li	345	24,977,690
	perl	1,576	31,309,305
	vortex	5,963	24,979,201
IBS	groff	6,333	11,901,481
	gs	12,852	16,307,247
	mpeg_play	5,598	9,566,290
	nroff	5,249	22,574,884
	real_gcc	17,361	14,309,867
	sdet	5,310	5,514,439
	verilog	4,636	6,212,381
	video_play	4,606	5,759,231

Table 2.4: Static and dynamic conditional branch counts in the IBS and SPEC CINT95 programs

---

entries. The vertical axis indicates branch misprediction rate, and the horizontal axis indicates the size of predictors in term of the storage bits needed. The top graph is the average of the IBS benchmarks, while the bottom graph is the average of the SPEC CINT95 benchmarks. As indicated by the relatively lower misprediction curve, PPM outperforms PAg, and the improvement of PPM is greater in IBS than in the SPEC CINT95. The small improvement in SPEC is due to its small miss rate (4.8%) in the BTB. This is because small miss rate implies little history information loss (occurred during misses); however, PPM is better than PAg only in cases when history information is not available, such as conflict misses and cold misses. The small BTB miss rate in the SPEC is because only a small number of distinct branches contribute to the vast majority of branch instances for most benchmarks [Sechrest96].

SPEC CINT95 average

Average BTB miss rate = 4.8%

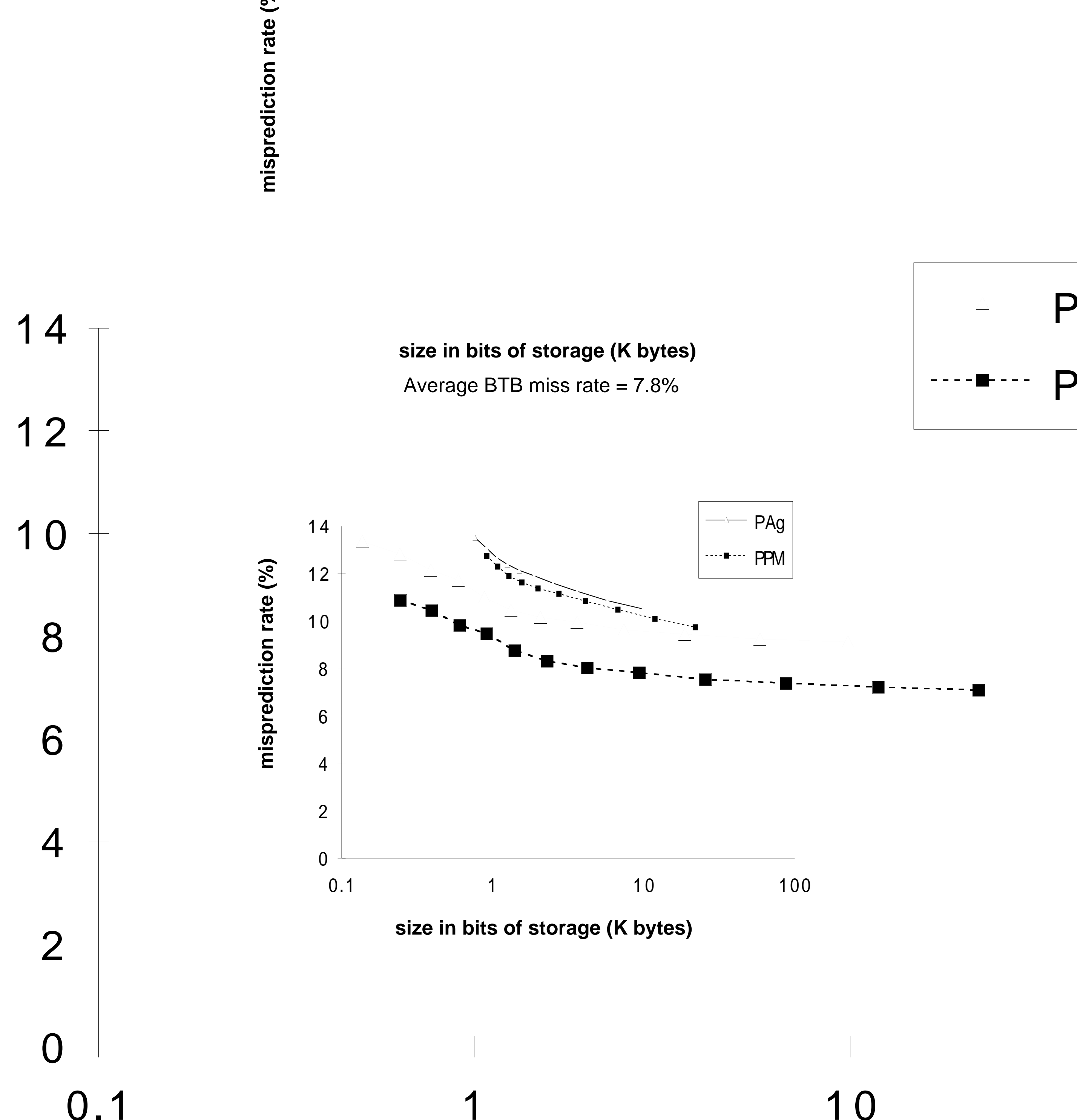
Figure 2.7: Misprediction rate for direct-mapped BTB with 1024 entries

---

To see the effect of a small BTB, we reduce the number of entries from 1024 to 128. As shown in Figure 2.8, PPM again performs better than the PAg scheme for both

Adobe’s PostScript Language Reference Manual, 2nd Edition, section H.2.4
says your EPS file is not valid, as it calls setpagedevice

IBS average



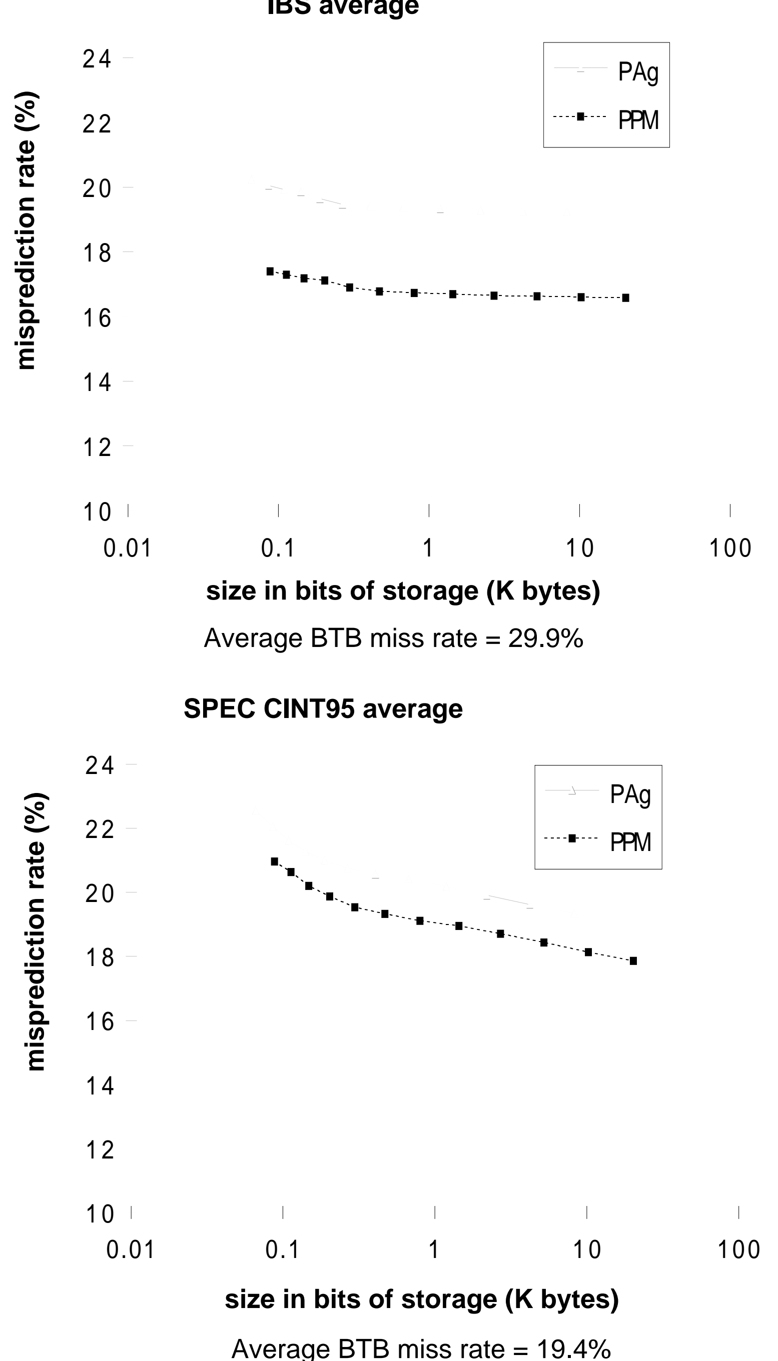


Figure 2.8: Improved accuracy of PPM predictor with a direct-mapped BTB wit 128 entries

---

benchmarks. Notice that, with a smaller BTB, the improvement of PPM over PAg is more pronounced. Also, as in the previous case, the improvement of PPM is greater in IBS than

in the SPEC CINT95. The improvement of PPM is more significant when the BTB miss rate (history information lost) is high.

As the results show, PPM performs better than two-level predictors. The improvement comes from a better mechanism for dealing with conflict (and cold) misses in which a set of Markov predictors rather than just the largest one are employed, as PAg does. If PPM cannot find a complete length of the branch history information, it reduces the length and search in the lower predictors. In the same situation two-level predictors use incorrect history information, which can lead them to index to the wrong counters. The accuracy decreases because not only is the wrong counter selected for prediction, but it is also incorrectly trained. PPM solves the conflict miss problem more gracefully than two-level predictors. This is why PPM performs better when the miss rate is pronounced.

To conclude this section, we briefly describe the implementation cost, and more details will be discussed in the next section. The PPM in our simulation uses 2-bit counters as the PAg does. The PPM implementation is essentially a set of PAg systems with history registers of length m, $m-1$, $m-2$, etc. Thus, the second-level table in the PPM implementation requires ($2^m + 2^{m-1} + 2^{m-2} + \ldots + 2^0 = 2^{m+1} - 1$) 2-bit counters. All together, this adds to about twice the number of bits required in the PAg system. This is reflected in Figure 2.7 and Figure 2.8.

2.6 Cost-effectiveness of PPM

Though a PPM needs more hardware than a PAg of the same history length, a PPM is still more cost-effective. We can see this by understanding the relationship of performance improvement to additional hardware requirement for both PPM and PAg. To enable a straightforward comparison of the hardware requirements of both predictors, the PPM in our simulator uses the same counters, 2-bit, as a PAg does. In this prototype, the hardware requirement of our PPM is about 2.5 times that of a PAg of the same history length. As shown in Figure 2.9, the additional hardware requirement comes from (1) the

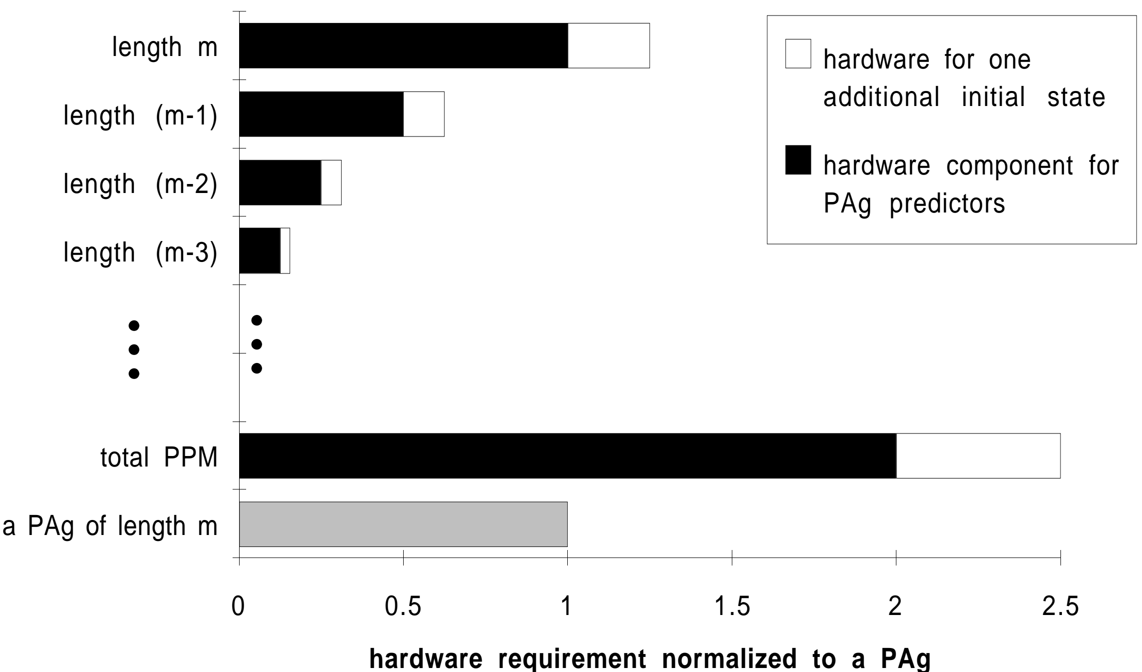


Figure 2.9: Comparison of hardware requirement of a PPM and a PAg of the same history length

---

additional set of PAg predictors with short history lengths and (2) one additional initial state for each counter within the predictors. With a PAg scheme, performance is improved by adding history bits to predictors; however, each additional history bit doubles the hardware requirement. When this increase is introduced, our PPM of history length *m* (order *m*) requires less hardware than does a PAg of history length *m*+2 (2.5 compared to 4) and slightly more hardware than a PAg of history length *m*+1 (2.5 compared to 2).

To show PPM is more cost-effective, we have to show that PPM is not only more accurate than a PAg of the same history length but also more accurate than a PAg of longer history. We have to do so because a PPM uses more hardware than a PAg of the same history length. Specifically, we have to show that a PPM of history length *m* is more accurate than a PAg of history length *m*+2.

In our simulations, the results indeed have shown that a PPM of history length *m* is generally more accurate than a PAg of history length *m*+2. As shown in Figure 2.7 and

Figure 2.8, for programs with numerous branches (like IBS) and for architectures with small branch-target buffers, a PPM of history length *m* does outperform a PAg of history length *m*+2, and even longer history lengths. Thus, in these cases, we may be able to replace PAg with PPM of shorter history length. In doing so, we reduce the hardware requirements while maintaining, and even improving, the prediction accuracy level.

Furthermore, we may be able to greatly reduce the hardware requirement of PPM by implementing a subset of PPM. This can be done by methodically eliminating smaller PAg predictors. Take a PPM of order *m* as an example: this PPM consists of PAg predictors of history lengths from *m* to 0. The hardware requirement of a PAg predictor doubles as its history length increases by 1. Therefore, the sizes of additional PAg of history lengths (*m*-1) to 0 form a geometric series, which add up to the size of the original PAg predictor of length *m.* Consequently, if we simplify the PPM by removing the PAg predictor of length (*m*-1), we reduce the extra hardware requirement of the small predictors alone by 50%. Similarly, if we remove two PAg predictors, those of lengths (*m*-1) and (*m*-2), the extra hardware requirement is reduced by 75%. Therefore, we have a method to significantly reduce the extra hardware requirement of PPM within an acceptable performance degradation. However, this method involves a trade-off between cost and performance. The most cost-effective configuration is the subject of a separate set of simulations.

In our discussion, an efficient hardware reduction technique is assumed: the use of minimal registers and finite-state machines, instead of shift-registers and counters. With minimal registers and finite state machines, we can customize the hardware required to the minimal needed by the PPM algorithm. We do this on two levels. Recall that a PAg predictor has two-levels. The first level is made of one or more shift-registers; while the second level is made of one or more sets of 2-bit counters. If we use the same hardware building blocks of PAg to implement a PPM, we introduce an expensive hardware expansion on the first level and superfluous increase of states on the second level. On the

first level, by introducing minimal registers instead of shift-registers, we can avoid increasing the number of shift-registers associated with the additional small predictors in PPM. On the second level, by using minimal registers and finite-state machines, we can accommodate the required number of states in counters without adding superfluous hardware, in the form of redundant and unnecessary states. Specifically, we no longer are constrained to the exponential hardware increase of binary counters, wherein we can only increase the number of states by a power of 2. For example, if we used counters with 4 states (2-bit counter) and determined that PPM requires an increase from 4 to 5, we would have to add hardware to accommodate 8 states (3-bit counters).

On the first level, PPM needs twice as many states as a PAg of the same history length because PPM needs to record the states of smaller predictors too. This extra state requirement can be solved by replacing m-bit shift-registers with (m+1)-bit registers and a shared finite-state machine. This is possible because the additional bit in the register doubles the number of states represented and, hence, satisfies the requirement of PPM. More specifically, instead of using current model of shift-registers, we can use plain registers just for the purpose of recording the states, and a shared final-state machine to map the transitions among these states. Since this finite-state machine is shared by all registers, the overhead is very small. Therefore, the extra hardware is essentially the one extra bit in each register, which makes the extra hardware requirement in the first level very small.

Similarly, on the second level, we can also use registers and a finite-state machines instead of the counters to reduce the hardware requirement. Compared to the original 2-bit counter (which has 4 states), PPM only needs one additional state to represent the initial state for each counter. However, if we use counters as building blocks, we can only increase the number of states by replacing a 2-bit counter with a 3-bit counters. In doing so, we increase the number of states from 4 to 8. Since PPM only needs 5 states, 3 out of these 8 states are wasted. Therefore, it is much more economic to combine several

counters and replace them with minimal registers. For example, we can implement three "5-state counters" with one 7-bit register. Recall that the combination of all possible states in three 5-state counters is 5^3 (125), and one 7-bit register can represent 2^7 (128) states. Therefore, one 7-bit register can accommodate the all the combination of states in three 5-bit counters and replace them. By using a minimal register, we only use 7 bits to represent three 5-state counters; whereas, it would take 9 bits if we use 3-bit counters to implement them. In term of states wasted, in this example, we reduce from original 3 per 8 (37.5%), to 3 per 128 (2.4%). For the transition among the states, we can use a single shared finite-state machine to map the transitions of all registers.

2.7 Discussion of further improvement

Compared to the field of branch prediction, data compression is a mature field that has been well studied for decades. For example, two-level branch predictors were proposed in the early 1990s, while in data compression a superset of them (PPM) had been proposed and studied in the 1980s. The potential benefit of applying data compression techniques to branch prediction is readily apparent in the similarity of predictors used in both methods. Because the predictors serve the same purpose in both fields, they can be used interchangeably. As shown in Figure 2.10, predictors used in branch prediction are only a very small subset of predictors developed in data compression. By exploring the opportunity to borrow techniques from data compression, we may be able to improve branch prediction.

Although two-level predictors may perform close to optimal, as our previous experiments suggest there is still some room to improve the prediction accuracy. Consider the three components: the predictor, the information processor, and the source.

2.7.1 Improvement of the predictor

Optimal accuracy has not yet been achieved by two-level predictors since they are

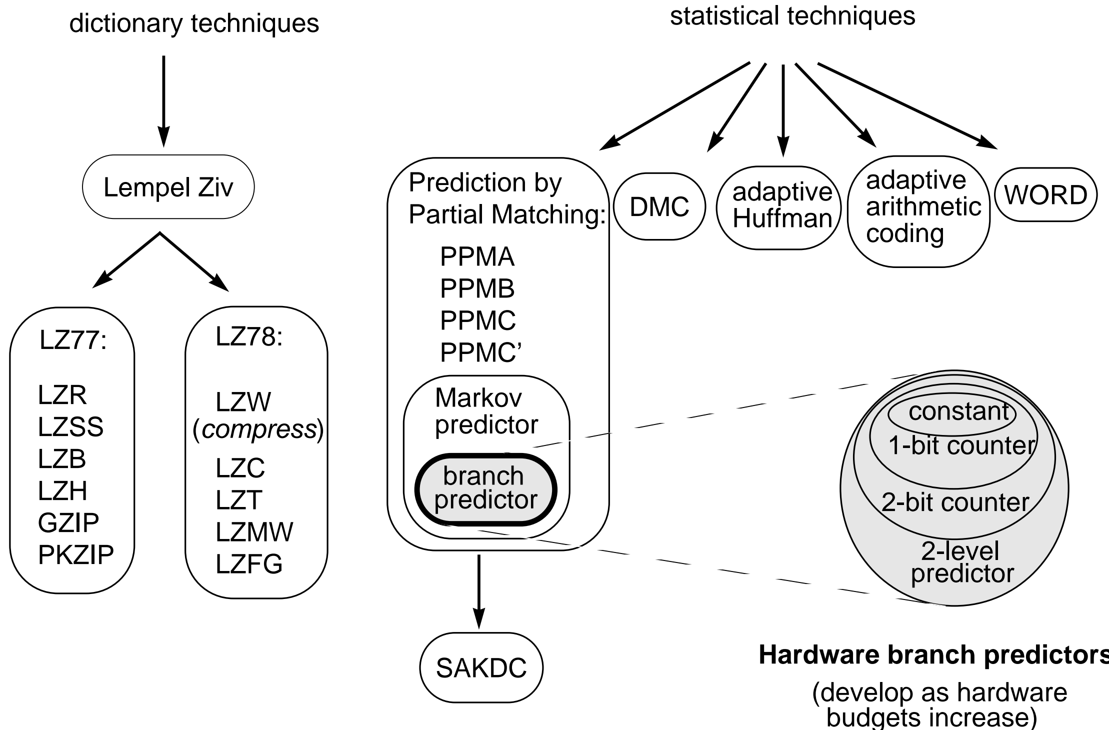


Figure 2.10: Branch predictors as a subset of predictors used in data compression

The areas highlighted in gray indicate the hardware branch predictors designed for microprocessors; the rest represent the predictors developed in data compression. We can see that the branch predictors are only a small subset of the predictors in data compression.

only a subset of optimal predictors. In addition, even if optimal predictors are available, it is still not clear how fast they can achieve optimal prediction accuracy. Therefore, we may further improve the predictor in the following ways.

2.7.1.1 Implementation of full-fledged optimal predictors

We will further demonstrate in Section 3.4 that two-level predictors have not achieved optimal accuracy using a Quicksort program whose optimal branch predictability can be analyzed exactly [Mudge96]. Currently, due to the limitation of hardware resources, it may not be cost-effective to implement full-fledged optimal predictors.

However, as hardware budgets increase with technology advancement, more features of optimal predictors can be added to achieve higher prediction accuracy. Examples would be the use of *n*-bit counters instead of 2-bit counters and the use of a variable length history instead of a fixed length. In the meantime, the design of predictors is an optimization under hardware constraints. Economic design and careful handling of details are required since every percent of improvement is important.

2.7.1.2 Design of other optimal predictors

There are several optimal predictors with different costs and levels of efficiency. Furthermore, depending on the type of application, an optimal predictor may have different levels of efficiency [Bell90]. For example, the Lempel-Ziv predictor (found in Unix *compress*) and PPM are both optimal predictors. While the Lempel-Ziv predictor has a faster prediction speed, PPM has higher accuracy in general. Yet in the long run, they can both achieve optimal accuracy. Therefore, depending on the application and the speed constraint, we may prefer one to the other.

2.7.1.3 Design of efficient non-optimal predictors

A non-optimal, yet efficient, predictor may have higher accuracy than an optimal predictor in some cases. This happens when programs end or change behavior too early for an optimal predictor to reach its highest accuracy. Therefore, though an efficient non-optimal predictor can never reach maximum accuracy, it may achieve higher accuracy in short or fast-changing programs. An example in data compression would be Dynamic Markov Compression (DMC) [Bell90].

2.7.1.4 Improvement of the information processor

Even with optimal predictors, we can still increase accuracy by improving the information processor. Good information selection, encoding, and dispatching can extract

the essence of branch behavior and, hence, improve prediction accuracy. In particular, this information processing is important since the predictor does not know the meaning of its input. Even using the same predictor, different information processing can result in prediction schemes with varied accuracy. Information describing branch behavior includes: branch address, branch outcome, operation code, target address, hint bits, and statistics from previous runs. How to best exploit and represent this information still remains to be studied. Examples of prediction schemes that attempt to improve the information processor are the *gshare* scheme [McFarling93] and the path correlation scheme [Nair95b].

2.7.2 Improvement of the source

We can fundamentally improve the predictability of the branches by changing the source and, thereby, their behavior. A more predictable source can be derived by adding algorithmic knowledge and run-time statistics from test-runs. The goal is to decrease the entropy of the source by making the outcomes of branches more unevenly distributed. An example is code restructuring with profiling information [Calder94, Young94].

2.8 Conclusions and further work

In this chapter, we establish the connection between data compression and branch prediction. This allows us to draw techniques from data compression to form a theoretical basis for branch prediction. In particular, we show that current two-level adaptive branch predictors are approximations of an optimal predictor, PPM. Based upon this theoretical basis rather than just simulation results, we can now have a reasonable degree of confidence in the performance of two-level predictors. Although two-level predictors are close to optimal if unlimited resources are available, PPM can still outperform two-level predictors when branch-target buffers are small. This is because PPM has better mechanisms for handling misses.

To illustrate directions for further improvement, we introduce a conceptual model, which consists of three components: a predictor, an information processor, and a source. For the predictor, we can borrow the rich set of predictors developed in data compression and apply them to branch prediction. However, since PPM is optimal, it is unlikely that significant improvement can be made by improving the predictor alone, except for the cases noted. Therefore, to further increase branch prediction accuracy, the focus should be on improving the information processor and the source.

CHAPTER 3

FURTHER EXAMINATION WITH OPTIMAL ALGORITHM AND EXACT ANALYSIS

Having shown that PPM can be successfully applied to branch prediction in Chapter 2, we further examine the performance of another popular optimal algorithm from data compression: Lempel-Ziv algorithm (found in Unix *compress*). Moreover, to calibrate the performance of various branch prediction schemes, we first derive the theoretical limit of predictability of an exact analyzable program, and then use this limit to measure the performance of each scheme. More detail discussion is shown in [Mudge96].

3.1 Description of Lempel-Ziv predictor

The Lempel-Ziv algorithm is one of the most popular and well-studied universal compression/prediction algorithms in the field of data compression. It was originally designed for text/image compression and was theoretically proven to achieve optimality asymptotically. More recently, it has further been applied to prefetch data from the disk memory and has been shown to be optimal. Although the original and more familiar algorithm is a word-based algorithm, an equivalent character-based version also exists.

This equivalent character-based version of the Lempel-Ziv universal compression/prediction algorithm has a structure that inherently favors prediction, therefore it is easy to construct a predictor from this version. This is because the algorithm consists of two parts: a predictor and an arithmetic encoder, and we can readily use the predictor part as it is. The predictor builds up a probabilistic model based on the outcomes seen so far, while the arithmetic encoder produces the output of minimum length based on the model derived from the predictor. Since the arithmetic encoder is so effective that the output produced is

close to optimal, the design of good compression relies on an accurate predictor which predicts the future correctly. For prediction purposes, only the predictor part of Lempel-Ziv universal compression/prediction algorithm is used, and we call it the Lempel-Ziv predictor.

To illustrate how the Lempel-Ziv predictor works, we first give an overview and then show with a concrete example in the next paragraph. The predictor predicts the future outcome based on the outcomes seen so far. The predictor builds up a tree-like data structure to make a prediction. The basic component of the tree is a node, which is a set of counters that keep count of all outcomes in each possible category. The predictor traverses from the root (the top node) down toward a leaf (a node without children). It makes a prediction as it traverses from one node to another. When the predictor reaches a node, it predicts next outcome based on the nodes immediately following the current node. As it reaches the bottom, it creates a new leaf, then it returns to the top (root) and repeats the same process. It predicts the next outcome to be the outcome that has been seen most frequently so far; or equivalently, the path taken most frequently so far. The predictor then moves to a new location based on the actual outcome. It predicts randomly when it encounters a tie in the outcome frequency, or when it reaches the bottom of the tree where no past information is available.

Figure 3.1, illustrates a specific example of how the Lempel-Ziv predictor predicts future outcomes based on the input, which is the history of outcomes seen so far. Assume the two possible outcomes are 0 and 1 (representing the outcomes of a branch, not taken or taken), and the input outcome sequence is 0, 1, 0, 1, 0, 1, 1. We show step-by-step how the predictor builds a tree-like data structure and predicts the next outcome. Steps 1 through 8 in Figure 3.1 show how the data structure after each input is observed and incorporated into the predictor.

To explain the detail of the prediction process, notations used in Figure 3.1 are described as follows. The complete process is divided into 8 steps, where each step shows

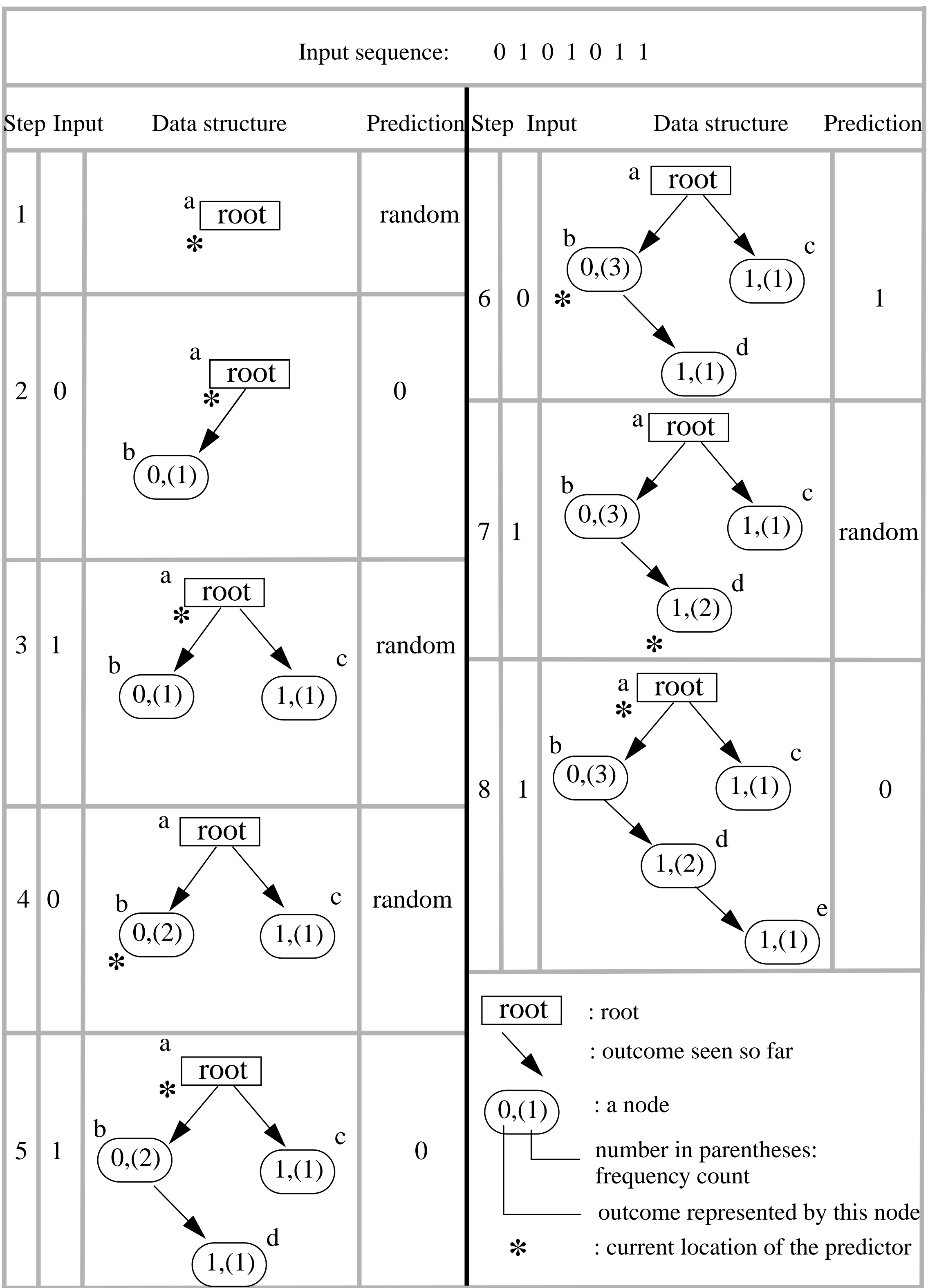


Figure 3.1: Example of a Lempel-Ziv predictor

(1) the input, (2) the data structure after the predictor has observed and incorporated each input, and (3) the prediction of the next outcome. Here, the input is the actual branch outcome seen so far, and it does not depend on the results of predictions. The input can be 0 or 1 representing the outcome of a branch, not taken or taken. The prediction is a guess of the next outcome (or input in the next step) made by the predictor, and the goal of the prediction is to match the next input as frequently as possible. Incorrect predictions result in recovery penalties causing a time delay in program execution. The rectangle in each step in Figure 3.1 represents the root, which is the starting node for the prediction process. Arrows indicate the outcomes seen so far at a particular node. An oval is a node representing an actual outcome and is labeled with alphabetic letters. Inside a node, the first number is the branch outcome represented by this node, and the second number (inside the parentheses) indicates the frequency count for this node. An asterisk indicates the location of the predictor after seeing the input.

In step 1, the predictor starts with only the root *a*. Since it has no past information, it predicts randomly. This prediction is a guess of the next branch outcome (input in step 2). A correct prediction can reduce the stalls due to branches; while an incorrect prediction requires misprediction recovery which causes additional delay. Overall prediction accuracy is recorded by comparing each prediction with the actual branch outcome.

In step 2, upon seeing input 0, the predictor creates a node *b* corresponding to that input, 0, since it has never seen a 0 before. It also increments the frequency count of this node *b* to 1. Because the predictor just created a new node, it moves back to the root *a* and starts over. It then predicts the next outcome to be a 0, because the root has only one node *b* (the predictor has only seen a 0 so far).

Similarly, in step 3, as the predictor sees an input of 1, it creates a node *c* for that input, 1, because outcome 1 has never occurred before. It also increments the frequency count of the newly created node *c* to 1. Since the predictor just created a new node, it moves back to the root and starts over. The predictor predicts randomly for the next branch

outcome since it encounters a tie; the frequency count of node *b* is the same as that of node *c*, i.e., once for both nodes.

In step 4, the input is a 0. Since the predictor has seen a 0 before (represented by the node *b*), it just increments the frequency count of node *b* to 2 and moves to node *b*. When the predictor moves to a node, indicated by an asterisk sign, it can only predict based on the nodes (representing past information) immediately following the current node. Since there is no node (past information) following node *b*, the predictor predicts the next outcome randomly.

In step 5, upon seeing the input 1, the predictor creates a new node *d* for that input, 1, since it has never seen a 1 at the current node *b*. Because it just created a new node, it moves up to the root *a* and starts the same process over again. Based on the nodes immediately following the root, the predictor predicts the next input to be 0, because the frequency count of node *b* is greater than that of node *c*.

Similarly, steps 6 through 8 follow the same process described above.

3.2 Implementation details and consideration

To implement the algorithm, there are a few parameters that need to be determined. Two of the most important are the size of memory allowed for the algorithm and the maximum size (number of distinct states) of the counters used in each node. The size of memory allowed for the algorithm is directly related to the resources available in the system. The more memory allocated for the algorithm, the more accurate the prediction is. Since we want to observe the best possible performance of the algorithm, we allow the algorithm to have as much memory as it needs. In other words, we do not impose a fixed limit on the size of memory, but let the algorithm build its data structure freely.

The second parameter is the maximum size (number of distinct states) of the counters used in each node. The size of the counter mainly affects the adaptivity of the algorithm, i.e., how fast the algorithm can respond to the changes in input data. A small

GAg two-level predictor

history length	compress	espresso	eqntott	sc	xlisp	Average
9	87.17	94.90	98.55	95.32	88.32	92.852
15	89.43	96.39	98.65	96.96	96.17	95.522
20	89.98	96.65	98.69	97.35	96.87	95.907

'GAg'-Lempel-Ziv

compress	espresso	eqntott	sc	xlisp	Average
84.56	93.37	99.08	95.19	96.02	93.645

Table 3.1: Prediction accuracy of GAg style of two-level predictor and Lempel-Ziv on SPECInt92 programs

counter records less information based on more recent history, then it predicts based on this history. Consequently, a small counter can detect and adapt to the change in input faster because it only reflects recent history. This is good for predicting fast changing data as in our case. In addition, a small counter also consumes less system resources. In our implementation, we set the maximum of the counter to be 4. When any counter reaches the maximum value, we divide (rescale) the values of all counters by half. By doing this, the ratio among the counters remains the same and the counters do not overflow.

3.3 Simulation results

Table 3.1 compares the simulation results of a Lempel-Ziv predictor and a two-level predictor (GAg). Both predictors take outcomes from all branches as input and use only one predictor. Based on this preliminary results, the accuracy of the Lempel-Ziv

predictor roughly corresponds to that of two-level predictors with history length between 9 and 15. However, the memory usage of a Lempel-Ziv predictor is much larger than that for a two-level predictor. In addition, the data structure needed for Lempel-Ziv predictor is dynamic and, thus, is not easy to implement in hardware. Therefore, the Lempel-Ziv predictor may not be an ideal candidate for branch prediction. Yet more extensive experiments are needed to justify the cost-effectiveness of the Lempel-Ziv predictor—in particular, the performance under the constraint of limited memory. In addition, we would like to further test the performance in other configurations.

3.4 Verification with an exactly analyzable program—Quicksort

The predictability of branches in some programs can be analyzed exactly, providing a provable limit to branch predictability. The analysis follows the approach commonly found in the concrete analysis of algorithms with the conditional branches being the object of interest. We have chosen Quicksort algorithm [Sedgewick92] to illustrate this point. Then use this exactly analyzed predictability to calibrate various branch prediction schemes.

3.4.1 Description of Quicksort

Quicksort is a divide-and-conquer algorithm. It selects an element from the array being sorted as pivot. Then the array is partitioned into left and right subarrays such that all elements in the left subarray are less than or equal to the pivot and all elements in the right subarray are greater than or equal to the pivot. Quicksort recursively partitions each subarray until the entire array is sorted. Different variations of Quicksort exist, and we have chosen one described in [Sedgewick92]. This implementation, shown in Figure 3.2, first picks the right end element to be the pivot. It also keeps two scan pointers that initially point to the left end element and the next-to-rightmost element respectively. The left pointer scans to the right until an element greater than the pivot is found. Similarly, the

```
/** function to swap two elements **/
void swap(itemType array[], int i, int j)
{
  itemType t = array[i];
  array[i] = array[j];
  array[j] = t;
}


void quicksort(itemType array[], int left, int right)
{
   int left_pointer, right_pointer; itemType pivot;

   if(right > left)
   {
       /**assign the right end element as pivot**/
      pivot = array[right];

       /**set the initial positions of two pointers**/
      left_pointer = left - 1; right_pointer = right;

       /**infinite loop to partition the array**/
      for(;;)
      {
        /**left_pointer scans for element greater or equal to pivot**/
        while ((array[++left_pointer]<pivot)&&(left_pointer<=right));

         /**right_pointer scans for element less or equal to pivot**/
        while ((array[--right_pointer]>pivot)&&(right_pointer>=0));

         /**stop if two pointers cross**/
        if (left_pointer > right_pointer) break;

        swap(array, left_pointer, right_pointer);
      }
      swap(array, left_pointer, right);
      quicksort(array, left, left_pointer-1);
      quicksort(array, left_pointer+1, right);
    }
}
```

Figure 3.2: A Quicksort program and its two comparison branches

---

right pointer scans to the left until an element less than the pivot is found. Then the two elements that stopped the pointers are swapped. When two scan pointers cross, the pivot and the element pointed by the left pointer are swapped. Now the pivot is in its final sorted position and it partitions the original array into right and left subarrays. The same process repeats with the right and left subarrays recursively.

Here we only consider the two branches that compare elements pointed to by pointers with the pivot value (the two while-statements printed in bold face in Figure 3.2).

These two branches form the kernel of the algorithm and are hard to predict: their outcomes depend heavily on the distribution of the input data set. The other branches in the program are essentially 100% predictable if enough past branch outcomes and computation time are provided.

3.5 Predictability of branches in Quicksort

We assume that the n numbers to be sorted are distinct, and that each possible initial ordering is equally likely.

It is well known that each subarray of each iteration is in random order, i.e., each possible ordering is equally likely [Sedgewick92]. The expected predictability for a subarray varies according to the number of elements. It has also been shown that the overall performance of Quicksort coincides, for large enough arrays, with the performance in one iteration of the algorithm on a sufficiently large array.

At any branch, our prediction of whether the program will branch or not depends on whether the new element being examined is more likely to be greater than or less than the pivot. Suppose we have compared j elements to the current pivot, of which i have been greater. Since we have assumed that all orderings are equally likely a priori, the conditional probability that the next element is greater than the pivot is simply $(i+1)/(j+2)$ as the following conditional probability argument shows: Compare the new element to the j numbers examined so far plus the pivot. It can be greater than from 0 to $j+1$ of these numbers, with each possibility being equally likely. Of these $j+2$ possibilities, $i+1$ mean the new element is greater than the pivot.

So the optimal prediction algorithm maintains a running count of the proportion of elements examined so far that are greater than the pivot, and compares this quantity to 1/2 to decide which way to predict the next branch. Equivalently, if a majority of the elements so far have been greater than the pivot, we predict that the new element will also be greater, and vice versa. (We guess randomly in the case of a tie.)

This scheme is optimal, and its prediction success rate will approach 75% from below as n becomes large. Coffey has shown [Mudge96] that we need only estimate whether the pivot is above or below the median, and we can do this with arbitrarily high accuracy from the first $\sqrt{n}$ computations for $\sqrt{n}$ large enough. The predictions made while this estimate is being formed make up a negligible fraction ($1/\sqrt{n}$) of the total number of predictions. Thus the scheme's performance approaches that of the situation where the rank of the pivot is known a priori. If we let p = (rank of pivot)/n, then we have p uniformly distributed over (0,1) as n becomes large, and our success rate is max$(p, 1\text{-}p)$. Since the pivot is equally likely to have any rank, our expected success rate is $\int_0^{1/2}(1-p)dp + \int_{1/2}^{1} p\,dp = 0.75$.

As an aside, note that if we were attempting to compress the branch history of the Quicksort program, we would feed each symbol into an arithmetic coder that encoded according to the best estimate of the probability of the next symbol, and compress to $H(p)$ bits per decision, where $H(p) = p\log_2(1/p) + (1-p)\log_2(1/(1-p))$ is the binary entropy function. Over the whole Quicksort program (involving many pivots) we would compress to $\int_0^1 H(p)dp$ bits per decision almost surely. The integral is $1/(2ln2)$. However, we know that the program trace can be compressed to $\log_2(n!) \cong n\log_2 n$ bits, and no further, since each of the $n!$ orderings is equally likely a priori. Thus we conclude that Quicksort has almost certainly $(2ln2)n\log_2 n$ decisions on average. This matches the well-known estimate of the performance of Quicksort [Sedgewick92].

Quicksort also provides a simple example of the potential dangers of extrapolating prediction performance from one program run. Suppose the program we run consists of one iteration of the Quicksort algorithm (so that one pivot is chosen). Ten million numbers are to be sorted, and the right end element (the pivot) is higher than nine million of these. Experimentation would demonstrate that an optimal predictor would be to predict branch back always in the loop for `left_pointer`, as shown in Figure 3.2, and do not branch back always in the loop for `right_pointer`. This would achieve the limit of 90%

accuracy for this choice of pivot. However, there are two major problems. First, this figure gives a very misleading impression of the overall program predictability, since no algorithm can do better than 75% on the average. What has happened is that a pattern particular to this input data has been picked up by the predictor. Secondly, note that the predictor thus developed will perform very poorly on average, achieving only 50% accuracy (and would achieve arbitrarily close to 0% if a sufficient low pivot is chosen), and in particular the predictor developed is very poor compared to the optimal one. These problems exist even though the raw data set used is large (ten million operations).

3.6 Simulation results

In this section we use the limit and the optimal predictor derived in the previous section to measure the performance of various schemes: PPM, a two-level branch predictor, Lempel-Ziv, a one-bit saturating counter, and a two-bit saturating counter. Again, we only consider the two comparison branches in Quicksort, as shown in Figure 3.2. The results are shown in Figure 3.3: The optimal predictor described in the previous section can best approach the asymptotic limit, followed by PPM, the two-bit counter (which also happens to be the best of the two-level predictors in this example), Lempel-Ziv, and finally the one-bit counter. Note that PPM can most closely approach the optimal predictor. Here the optimal predictor described in Section 3.5 is designed specifically for Quicksort and would approach the 75% asymptotic limit if the given data sets are truly random and uniformly distributed.

For the two-level predictor, we examine all four of the schemes shown in Figure 2.5: a global history register with global pattern history table (GAg), global history register with per-address pattern history table (GAp), per-address history register with global pattern history table (PAg), and per-address history register with per-address pattern history table (PAp). Note that, in this particular example, the “per-address” is equivalent to “per-set” because we only consider two distinct branches. We also use 2-bit

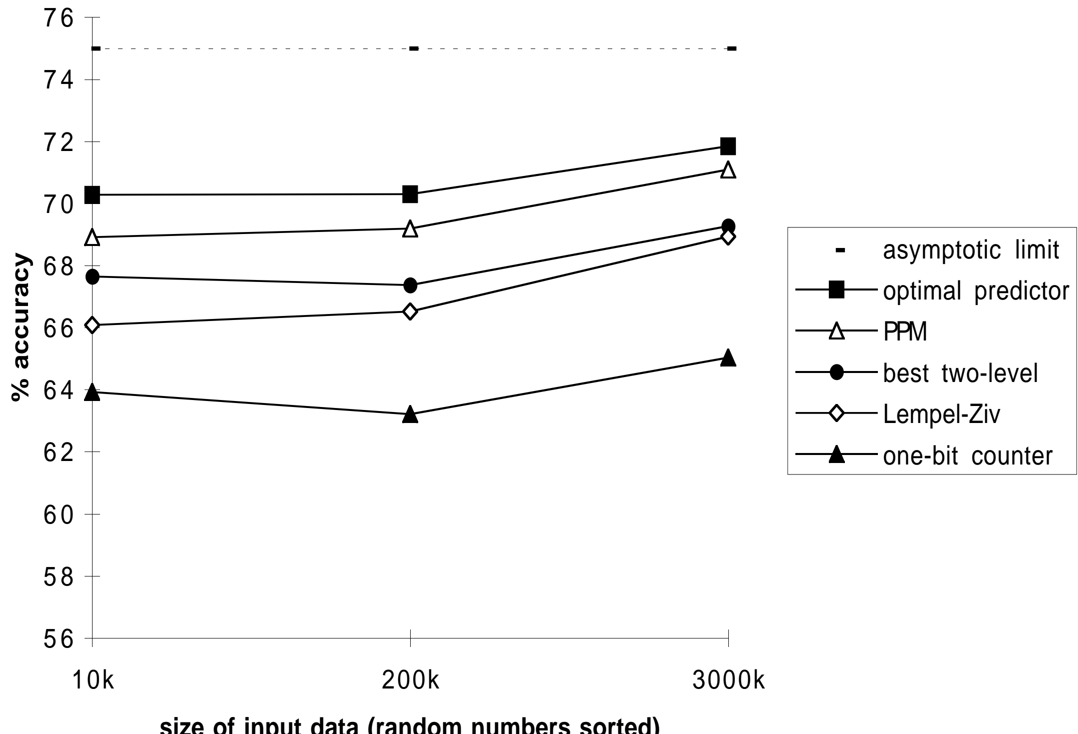


Figure 3.3: Comparison of prediction accuracy for Quicksort

This graph compares the prediction accuracy of different predictors for Quicksort. Note that we only consider the two comparison branches illustrated in Figure 3.2. The dashed line indicates the 75% asymptotic limit.

---

saturating up-down counters for the pattern history table as suggested in [Yeh92b]. The best results were obtained with a PAp scheme. It is plotted in Figure 3.3 as the line of closed circular bullets.

In practice only GAg or PAg schemes are implemented in hardware. GAp and PAp can quickly become unwieldy. In particular, a PAp scheme is usually too large to be practical but in the case of Quicksort where only two branches are under consideration, it is reasonable to consider for the purposes of a simulation. PPM still improves on the PAp scheme even though it is the most complex of the prediction schemes, that has the potential to capture the most information.

There are two reasons that attribute to the better performance of PPM over a two-level branch predictor. First, the maximum size of counters in PPM is larger (an adjustable

parameter) than two-level branch predictors and, thus, PPM can maintain more information. Second, as explained previously in Section 2.5.3, PPM has extra set of small predictors to handle cold start misprediction more efficiently.

3.7 Summary

In this chapter, we further examine the performance of another popular optimal algorithm, Lempel-Ziv algorithm. In addition, to calibrate the performance of various branch prediction schemes, we showed that the theoretical limit of predictability of branches in a Quicksort program is 75%, and then used this limit to compare with the performance of various schemes. We found that PPM can best approach this limit followed by the two-bit counter scheme (which also happens to be the best of two-level predictors), Lempel-Ziv, and finally the one-bit counter scheme. Based on our preliminary results, though the Lempel-Ziv predictor can offer decent accuracy, its memory requirement is larger than a two-level predictor and its data structure is dynamic, which is not easy to implement in hardware. Therefore, the Lempel-Ziv predictor may not be an ideal candidate for branch prediction.

CHAPTER 4

DESIGN OPTIMIZATION FOR HIGH-SPEED PER-ADDRESS TWO-LEVEL BRANCH PREDICTORS

4.1 Introduction

As we discussed in Section 2.7, although optimal predictors can be derived using techniques from data compression, the design of branch predictors still remains a cost-effective optimization problem due to implementation and budget constraints. In particular, optimal designs vary with target technology and hardware budget. To identify optimal designs, we need to consider all parameters in a branch predictor and evaluate their interaction.

In this chapter, we will show how the design style of optimal predictors changes due to fast clock rate, and how a comprehensive analysis can be done to find out the best design configuration (this work has also been published in [Chen97b]). We choose per-address two-level branch predictors for illustration, because they have been shown to be among the best predictors and have been implemented in current microprocessors. Among different predictors proposed, the per-address two-level branch predictor has been shown to be one of the best and has been implemented in the Intel Pentium Pro processor [MReport95b]. Typically, the two-level per-address predictor is coupled with a branch-target buffer (BTB) through the sharing of common tags [Yeh92a, Calder94b]. Both components benefit from tags and, thus, cost can be reduced by sharing. In particular, the tags enable high hit-rate set-associative design for the predictor and the BTB.

However, as the clock frequency of modern microprocessors continues to increase, the coupled set-associative design using tags may no longer be the best choice. This is

because set-associative designs require longer access time than direct-mapped designs and, thus, may become a critical path in a high clock rate microprocessor. Therefore, we re-evaluate and suggest an alternative tagless direct-mapped version of two-level per-address predictors [Yeh91].

A tagless direct-mapped per-address predictor can offer performance comparable to current implementations. Typically the tagless predictor does not have hit-rates that are as high as a set-associative design, however it offers two advantages. First, by removing tag storage, more resources can be allocated to the predictor and BTB to improve performance. Second, by decoupling the BTB from the predictor, the tagless design offers the flexibility to optimize the BTB and predictor individually. In particular, the predictor can have a different number of entries than the BTB. Thus, the BTB need only store taken branches instead of all branches [Calder94b]. Also note that the removal of tags does not prevent the identification of branch instructions, because branches can still be identified using predecoded information, which is already commonly employed in commercial microprocessors.

To justify the tagless implementation, we conduct performance evaluation and show that, for the prediction process, tagless predictors in general perform better, or no worse, than direct-mapped tagged predictors. To analyze the improvement, we break down the total errors into transitional-state and steady-state errors. We found that tagless predictors have lower transitional errors and, consequently, have higher performance. Moreover, the tagless predictor is simpler and faster than the tagged version.

To develop general design principles for optimal configurations, we exhaustively search the design space of tagless per-address predictors. Our study shows the sensitivity of the optimal configurations to various program characteristics. To conclude, we derive general principles for selecting the best parameters. When given a specific budget and benchmark suite, these principles can help designers to select the best configurations.

The rest of this chapter is organized as follows. In Section 4.2 we briefly review

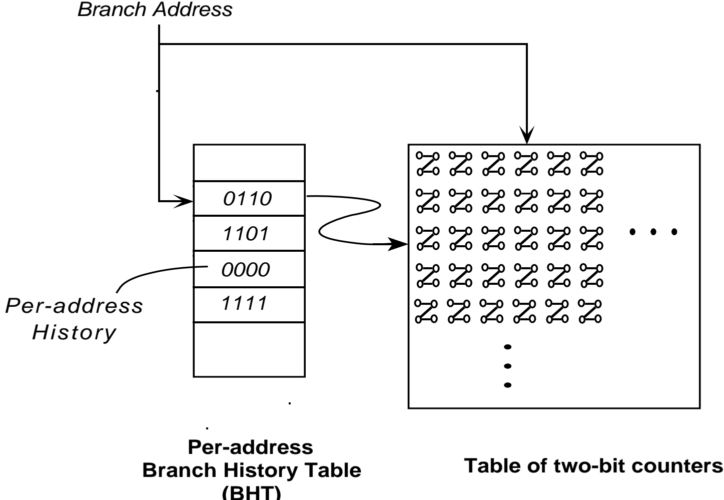


Figure 4.1: Schematic for a per-address two-level branch predictor

---

the per-address two-level predictor, and discuss the tagless per-address prediction scheme. In Section 4.3 we explain why the tagless scheme can have a better prediction accuracy than a traditional tagged scheme. Section 4.4 develops a cost analysis procedure to identify optimal tagless predictor designs. We present some concluding remarks with Section 4.5.

4.2 Per-address two-level branch predictors

The two-level per-address adaptive branch predictor is a variation of two-level branch predictors proposed by Yeh and Patt [Yeh91, Yeh92b]. As shown in Figure 4.1, a two-level per-address adaptive branch predictor consists of two tables. The first-level table, called the branch history table (BHT), has multiple shift-registers called branch history registers (BHRs). Each of these registers is used to record past branch outcomes for a single static branch. The branch outcome patterns recorded in the first-level table are then used to index a set of counters in the second level. The column index into the counters is usually some part of the address of the branch being predicted. Although there are many options for the counters, the best performance has been observed when the counters are

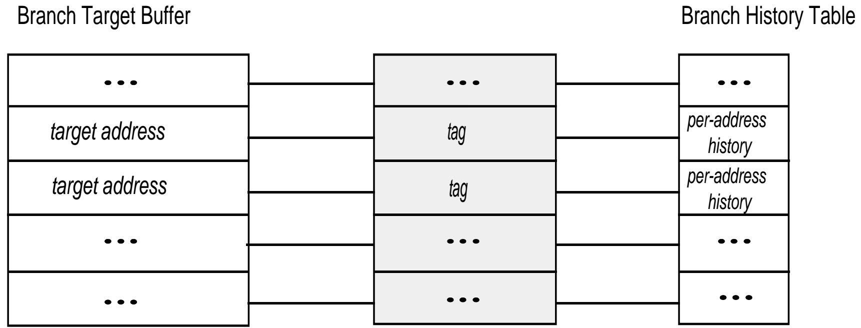


Figure 4.2: Tagged per-address two-level branch predictor

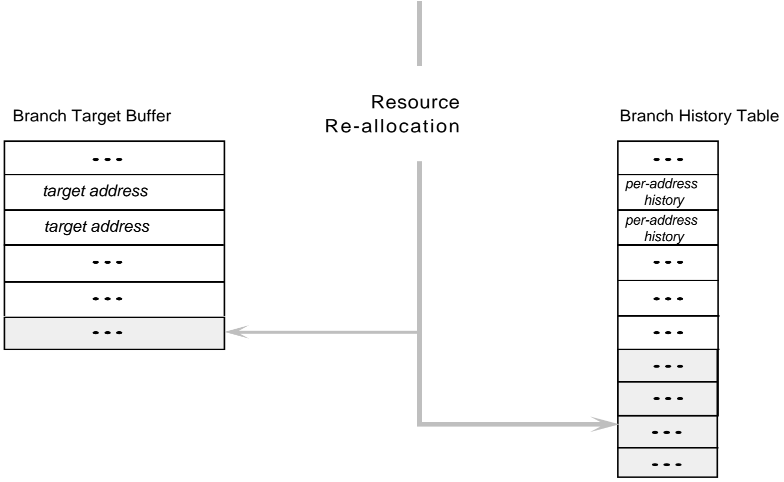


Figure 4.3: Tagless per-address two-level branch predictor

---

two-bit saturating up-down counters [Nair95a], and this fact was analyzed by Chen et al. [Chen96a].

Since the counters are typically organized as a two-dimensional array, there can be many configurations for the second-level table. If a configuration has multiple rows and columns, then it is generally referred to as a PAs scheme according to the taxonomy by Yeh and Patt [Yeh92b]. If the table has a single column, it is a PAg scheme. If the table is a single row, the predictor is equivalent to the traditional two-bit counter scheme proposed by Smith [Smith81] because the counters are exclusively indexed by the branch address. This design space has been thoroughly studied by Sechrest et al. [Sechrest96].

4.2.1 Tagless implementation

In a typical two-level per-address scheme, the predictor is coupled with a branch target buffer (BTB) through the sharing of common tags [Yeh92a, Calder94b], as shown in Figure 4.2. This coupling of predictor and BTB is cost effective in that the predictor and BTB benefit from a single copy of the tags. The presence of tags also allows set-associative predictors, which provide a high hit rate for both predictor and BTB.

As we have noted, as the cycle time of microprocessors continues to decrease, the coupled-set-associative design using tags may no longer be the best choice. Set-associative implementations require a longer access time than direct-mapped designs, and may become a critical path in a microprocessor with short cycle times. In addition, the coupling of predictor and BTB degrades performance in two ways. First, the BTB needs to allocate space to record not-taken branches, since the predictor needs information for all branches. This wastes BTB resources [Calder94b]. Second, the number of history entries in the predictor is limited to be the same as the number of BTB entries, restricting the designer's freedom to fully explore the design space. As cycle times become shorter, we believe that decoupled, direct-mapped per-address schemes, as shown in Figure 4.3, deserve closer inspection.

As an aside, the removal of tags does not prevent the identification of branch instructions. Branch instructions can still be identified using predecoded information stored in the cache, which is already commonly employed in commercial microprocessors.

In a decoupled, direct-mapped per-address scheme design, tags for the branch predictor are redundant. Tags are crucial for a set-associative BHT to distinguish branches in the same set. In contrast, in a direct-mapped BHT, no such distinction is needed, so tags only affect the miss handling policy. More specifically, if there is a miss (or conflict), the predictor needs to decide whether to use the history from old branch, or flush the history

register and restart with some predefined "reset value." The former scheme does not need tags at all, and we refer it as the tagless branch predictor in this chapter. As an aside, we note that the tagless implementation of per-address predictors can be categorized as a per-set history scheme, according to [Yeh93], because several branches may share one history register.

In the next section, we will show that, because of better miss handling policy, the tagless scheme is actually superior to the direct-mapped tagged predictor.

4.3 Performance analysis for tagless predictors

In this section, we investigate how tags affect the prediction mechanism, excluding the hit rate factor. Tags have no effect when a branch hits in the history shift-register; tags only affect prediction accuracy when misses or conflicts occur. Tagged schemes can employ different miss handling policies which in turn can yield different prediction accuracies.

4.3.1 Miss handling policies

When misses occur, the branch history shift-register has the options of flushing (*flush*), or not flushing (*no-flush*) its old history contents, as shown in Figure 4.4. Two intuitive miss handling policies for tagless and tagged schemes are:

1. Tagless predictors: *no-flush* policy, do nothing, since tagless predictors do not have tags to detect a "miss."
2. Tagged predictors: *flush* policy, flush and reset the history to a default *reset value*. The old history is discarded, and the incoming new branch starts accumulating its self history. This miss handling policy has intuitive meaning and takes advantage of tags.

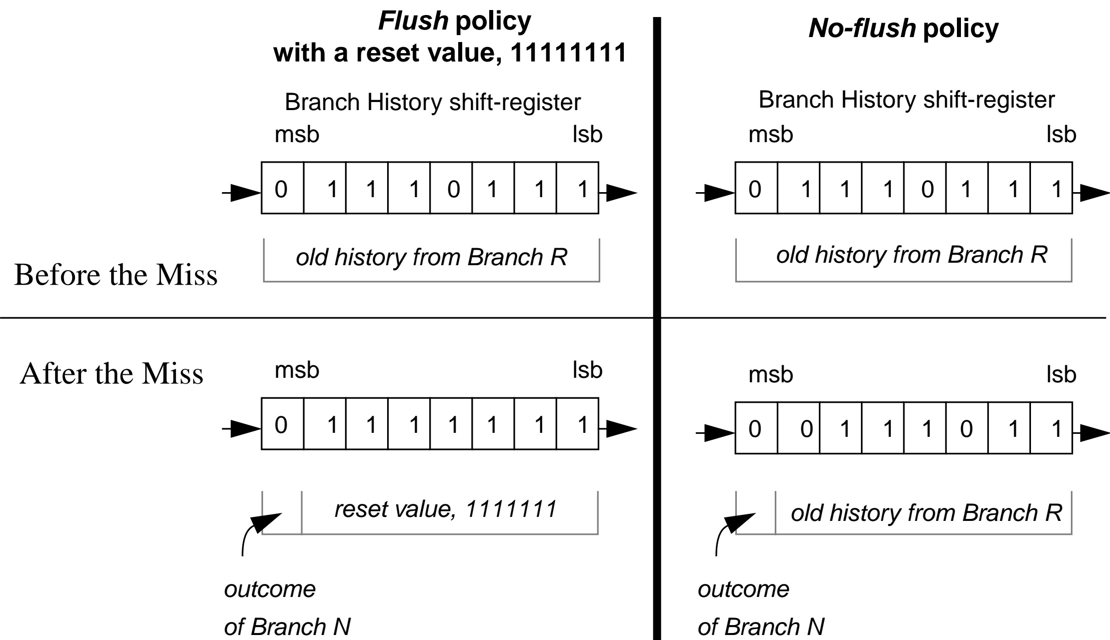


Figure 4.4: An example comparing the *flush* and *no-flush* policies

This example illustrates the difference between the *flush* and *no-flush* miss handling policies for a conflict miss in the branch history table (BHT). In this example, each BHT entry, i.e. the branch history shift-register, can store an 8-bit per-address history of the branches that index it; 0 represents not-taken and 1 represents taken. *R* represents an old branch being replaced, and *N* represents the new incoming branch. As illustrated, the incoming branch, *N*, happens to index the same shift-register that *R* does, so a miss occurs. The *flush* policy changes the register content to all 1's and shifts in the new outcome, which is 0, after outcome of *N* is resolved. In contrast, the *no-flush* policy simply shifts in the new outcome, 0, without flushing the old history of *R*.

4.3.2 Simulation methodology

To fairly compare the tagged and the tagless predictors, the best tagged predictor must be determined and used for comparison. We exhaustively simulated all possible 256 reset values for 8-bit history patterns and sampled 256 reset values for 14-bit predictors to find the best reset value for each benchmark. Since the best reset value is different for each benchmark, we present results using the best value for each benchmark[1], instead of

1. As an aside, we noticed that the best reset value is the least frequently occurring history pattern in the benchmark, because it causes least interference in the normal prediction process. This "best" reset value differs for each benchmark.

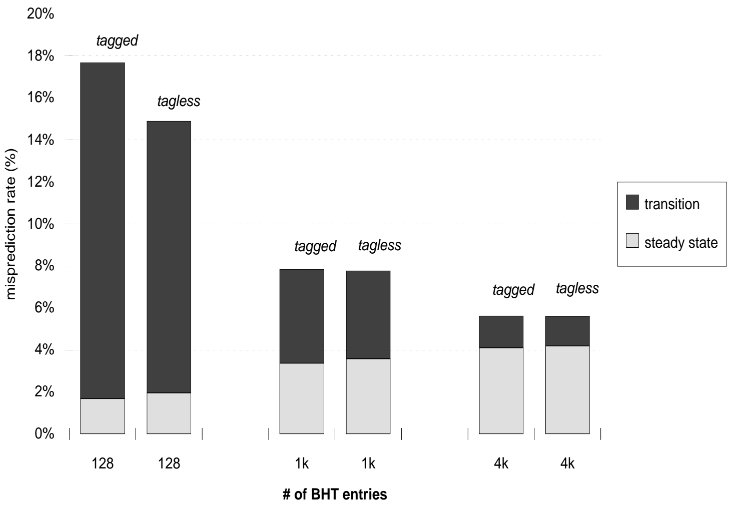


Figure 4.5: Misprediction rate of PAg with 8-bit history length for IBS

---

applying one common value for all. This creates an unreachable upper bound for the performance of the tagged predictor.

To focus on the prediction mechanism, we use direct-mapped schemes for both tagless and tagged predictors to isolate the effect of hit rate. For simplicity, we select PAg predictors of 8-bit and 14-bit history for our comparisons.

To assess the performance of tagless and tagged predictors, we conduct a trace-driven simulation. As input for the simulation, we use the Instruction Benchmark Suite (IBS) benchmarks [Uhlig95] and the SPEC CINT95 benchmarks [SPEC95] for our simulation. The branch statistics for both benchmark suites are summarized in Table 2.4.

4.3.3 Simulation results

Figure 4.5 shows the average misprediction rates for tagless and tagged predictors for the IBS benchmarks. PAg predictors with 8-bit history are used in this simulation. Since the *y*-axis represents the misprediction rate, a lower bar indicates better performance. The *x*-axis represents the number of history shift-registers. Within each pair of bars, the left bar represents the tagged predictor, and the right bar represents the tagless

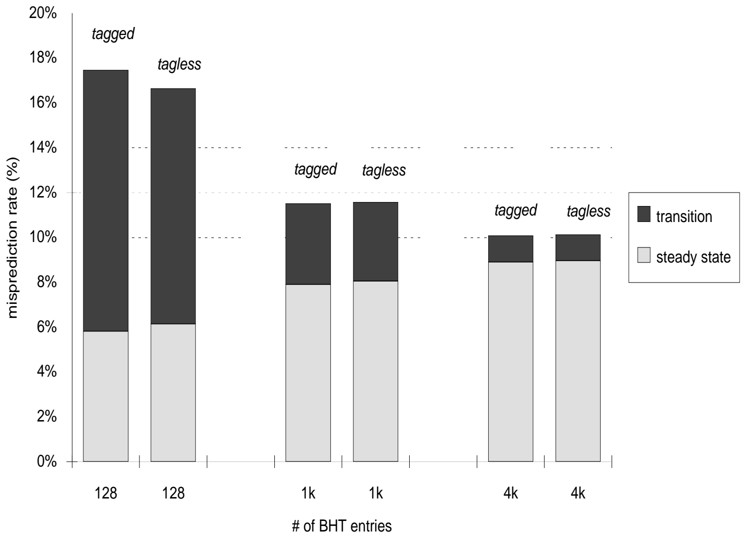


Figure 4.6: Misprediction rate of PAg with 8-bit history length for SPEC CINT95

---

predictor. The meaning of the gray and dark components will be explained in Section 4.3.4. As shown in Figure 4.5, the tagless predictor outperforms the upper bound of tagged predictor in a 128-entry branch history table, while both predictors have similar misprediction rates in 1K- and 4K-entry configurations. Similar conclusions can be drawn for SPEC CINT95 benchmarks, as Figure 4.6 shows.

When the length of history is increased from 8 bits to 14 bits, the misprediction rates for IBS are shown in Figure 4.7. In this case, the tagless predictor outperforms the upper bound of the tagged predictor for both 128 and 1K history shift-registers configurations, and they perform very closely to each other in 4k configuration. Similar conclusions can be drawn for SPEC CINT95 as Figure 4.8 shows.

Detail simulation data for each benchmark and configuration are shown in Table 4.1. We can see that the tagless predictor not only has better average results than the tagged predictors, but also has better performance for most of the benchmarks. For ease of comparison, the cells highlighted in gray indicate that the tagless predictor has better performance than the tagged predictor. Also notice that in the few benchmarks that tagless

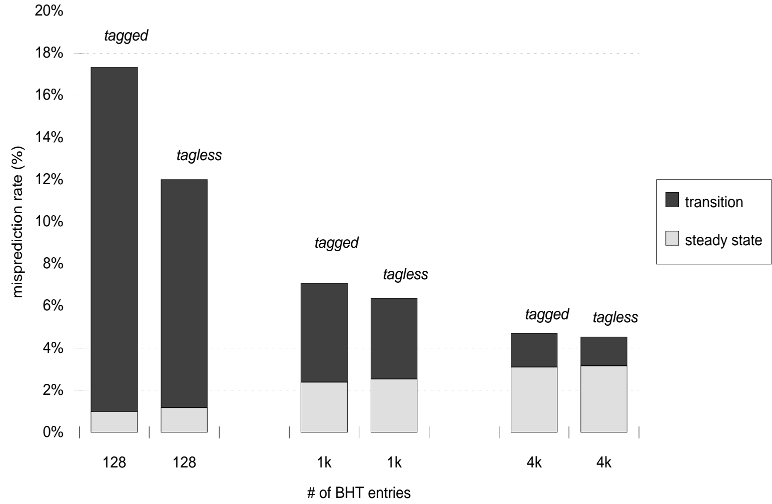


Figure 4.7: Misprediction rate of PAg with 14-bit history length for IBS

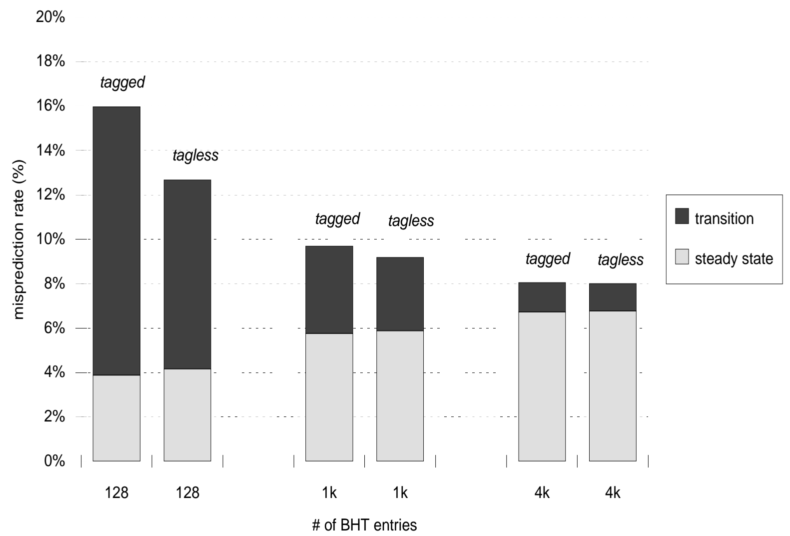


Figure 4.8: Misprediction rate of PAg with 14-bit history length for SPEC CINT95

---

predictors perform worse, the difference is usually less than 1%. To show the variation among different reset values, we also include the misprediction rates for the best (*tagged-best*) and the worst (*tagged-worst*) reset values for tagged predictors. We can see that the difference between the best and worst reset values is usually less than 1%.

In conclusion, the tagless predictors perform better when the number of entries in

Misprediction Rate (%)	128-entry DM BHT			1k-entry DM BHT			4k-entry DM BHT		
8 bit	tagged-best	tagged-worst	tagless	tagged-best	tagged-worst	tagless	tagged-best	tagged-worst	tagless
groff	17.05	17.44	12.84	6.51	6.98	6.70	5.00	5.25	4.82
gs	17.89	18.28	16.62	7.92	8.37	7.05	5.29	5.71	4.88
mpeg_play	16.33	16.77	14.54	8.27	8.68	8.17	7.99	8.29	6.61
nroff	13.27	13.64	9.97	4.77	5.05	4.98	3.06	3.22	4.11
real_gcc	21.77	22.38	18.83	11.57	12.06	11.21	8.85	9.17	8.89
sdet	11.11	12.09	13.12	6.68	7.62	7.55	4.63	5.24	4.91
verilog	19.45	19.99	15.13	7.49	8.12	6.65	4.20	4.64	4.53
video_play	18.87	19.53	18.13	9.08	9.88	9.88	8.08	8.57	6.21
14 bit	tagged-best	tagged-worst	tagless	tagged-best	tagged-worst	tagless	tagged-best	tagged-worst	tagless
groff	16.89	16.98	11.25	5.95	6.09	5.84	4.29	4.34	3.99
gs	17.54	17.66	11.69	6.91	7.13	5.11	4.14	4.25	3.50
mpeg_play	15.52	15.94	12.20	7.22	7.52	6.72	6.89	7.01	5.40
nroff	12.92	13.02	8.28	4.18	4.29	4.15	2.34	2.38	3.35
real_gcc	21.25	21.40	16.91	10.67	10.77	10.10	7.86	7.93	7.90
sdet	11.06	11.16	10.38	6.29	6.43	6.09	4.06	4.16	3.99
verilog	19.25	19.33	11.31	6.98	7.09	5.56	3.54	3.62	3.70
video_play	18.49	18.62	14.11	8.02	8.25	7.44	6.71	6.90	4.48

Misprediction Rate (%)	128-entry DM BHT			1k-entry DM BHT			4k-entry DM BHT		
8 bit	tagged-best	tagged-worst	tagless	tagged-best	tagged-worst	tagless	tagged-best	tagged-worst	tagless
compress	11.02	12.40	11.01	10.14	10.15	10.14	10.14	10.15	10.14
gcc	28.08	28.79	25.77	15.59	16.17	14.68	11.41	11.90	11.41
go	27.57	28.76	28.88	23.41	24.43	24.46	22.62	23.17	21.61
ijpeg	10.60	10.91	10.70	10.23	10.35	10.24	10.16	10.23	10.11
li	18.17	19.56	11.64	9.07	9.39	9.08	8.27	8.27	8.80
perl	19.97	20.82	18.90	9.84	10.96	9.94	7.52	8.11	7.84
vortex	25.89	25.92	9.63	5.40	5.49	2.47	2.33	2.40	0.98
14 bit	tagged-best	tagged-worst	tagless	tagged-best	tagged-worst	tagless	tagged-best	tagged-worst	tagless
compress	9.52	9.94	9.09	8.62	8.63	8.63	8.62	8.63	8.62
gcc	27.54	27.71	21.56	14.42	14.55	12.81	10.02	10.12	9.92
go	25.88	26.24	26.28	21.01	21.35	22.01	19.90	20.14	18.85
ijpeg	8.51	8.62	8.61	8.07	8.10	8.07	7.99	8.00	7.94
li	11.44	11.60	7.18	6.17	6.22	6.02	5.34	5.34	5.87
perl	18.65	18.93	10.05	7.41	7.66	5.07	4.31	4.44	4.13
vortex	25.87	25.88	6.04	5.31	5.33	1.72	2.17	2.18	0.70

Table 4.1: Detail misprediction rates for tagged and tagless predictors

---

branch history table is small or when history length is long; tagless predictors have comparable performance as tagged predictors in other configurations. Furthermore, tagless predictors are both simpler and cheaper.

4.3.4 Analysis using transitional-state and steady-state error

To explain the superior performance of tagless predictors, we broke down the total error into transitional-state error (black portion of the bars), and steady-state error (gray portion of the bars), shown in Figure 4.5 to Figure 4.8. Since tagless and tagged predictors differ in the miss handling policy (*flush* or *no-flush*), which affects the transitional state, we classify the error into these two categories to identify the sources of prediction error.

At any time, each of the history shift-registers, i.e. each BHT entry, is either in a transitional state or in a steady state. In a transitional state, the history of a branch does not fill up the entire history shift-register. In other words, only part of the history information belongs to the current branch and the rest of the history information is either part of the reset value (flush policy), or history left from the replaced branch (no-flush policy). The transitional state occurs on the first few references right after a miss, and is, in a sense, similar to “cold starts” for caches. On the other hand, the steady state is reached when a history shift-register is filled up with branch outcomes exclusively from the current branch.

For example, an 8-bit branch history shift-register is in a transitional state during the first 8 references right after a miss, since it takes 8 references (branch outcomes) to update and fill up the history shift-register. From then until the next miss occurs to the same shift register, the branch history shift register is in its steady state.

We observed that the tagless predictors have less transitional error than the tagged predictor. The transitional-state error is due to the partially correct history which is likely to index to a wrong 2-bit counter, resulting in incorrect prediction. As can be seen in Figure 4.5 to Figure 4.8, the transitional-state errors for tagless predictors are smaller for configurations with a small branch history table (128 entries), and long history length configurations (14 bits).

To explain why tagless predictors have smaller transitional errors, we examine the

miss handling process of tagged predictor. Every time a miss occurs, a tagged predictor flushes the old contents in the history shift-register, resets it to a default value, then starts accumulating the history information from the new branch. However, if misses occur too often, the content in the history shift-register will be flushed constantly and, thus, never reaches a steady state. In this case, during the transitional state, the 2-bit counter indexed by the reset value and the counters indexed by the next few subsequent branch outcomes are used frequently, creating a "hot spot" and thus resulting in large transitional-state error. This situation gets worse when history length is long, because it takes longer to reach steady state.

In contrast, a tagless predictor does not flush old history information when misses occur. On a miss, depending on the characteristics of the previous branch, old history information may not always be harmful. For example, if a mostly-taken branch (e.g., loop branch) is replaced by another mostly-taken branch, the old history information would be the same as that of new coming branch and, thus, resulting accurate prediction. In another situation, consider the case that a frequently occurring branch *X* is only briefly interrupted by another branch *Y*. When execution returns to branch *X*, most of the old history information would still belong to *X*, which helps the prediction. This also explains why tagless predictors perform better when history length is long.

The steady-state error is essentially independent of the miss handling policy, and hence prediction accuracy for steady state should be almost the same for both tagless and tagged predictors. Indeed, the steady-state errors (shown as the lower gray bars in Figure 4.5 to Figure 4.8) are about the same for both predictors. The prediction accuracy during the steady state is high, because all the history used for prediction belongs to the current predicting branch. Therefore, the difference between tagless and tagged schemes lies in the error during the transitional state.

In summary, tagless predictors can have better overall results than tagged predictors, in addition to its simpler implementations with more design flexibility.

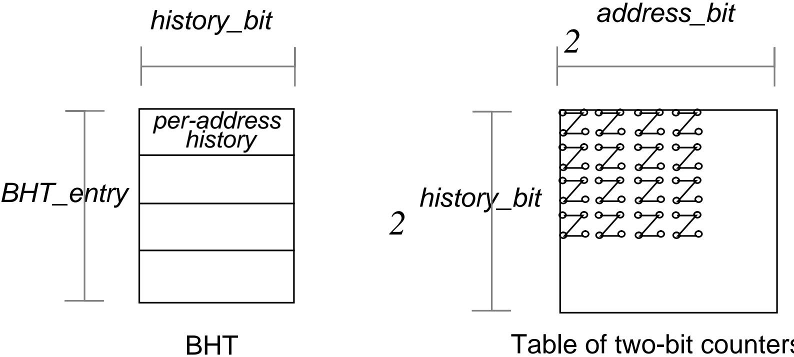


Figure 4.9: Illustration for the three parameters of per-address scheme cost function

---

4.4 Cost-benefit analysis for tagless predictors

After having shown the effectiveness of tagless per-address two-level predictors, we present a cost-benefit analysis for a wide range of configurations in its design space. There have been some previous studies for per-address schemes [Yeh92b, Yeh93, Sechrest96]. However, their work mainly focused on the design trade-off for the second-level table, while we incorporate the first-level table cost for a complete analysis. We examine hardware budgets ranging from 512 bytes to 16K bytes. The three parameters considered in our design space are: the number of entries in branch history table (BHT), the number of address bits indexing 2-bit counters, and the number of history bits in the branch history registers. These three parameters are labeled as *BHT_entry*, *address_bit*, and *history_bit*, respectively; see Figure 4.9 for a pictorial representation. The estimated cost in bits for the tagless per-address scheme is given as follows,

$$\text{Cost} = (BHT_entry) * (history_bit) + 2^{(\,history_bit \,+\, address_bit \,+\, 1\,)}$$

Based on this cost function, we can derive equal-cost contour lines for a fixed number of branch history entries in the BHT. Note that history bits are inherently more expensive than address bits, because history bits require extra resources (BHT) to record them. Figure 4.10 and Figure 4.11 show the equal-cost contour lines for 128 and 8K BHT

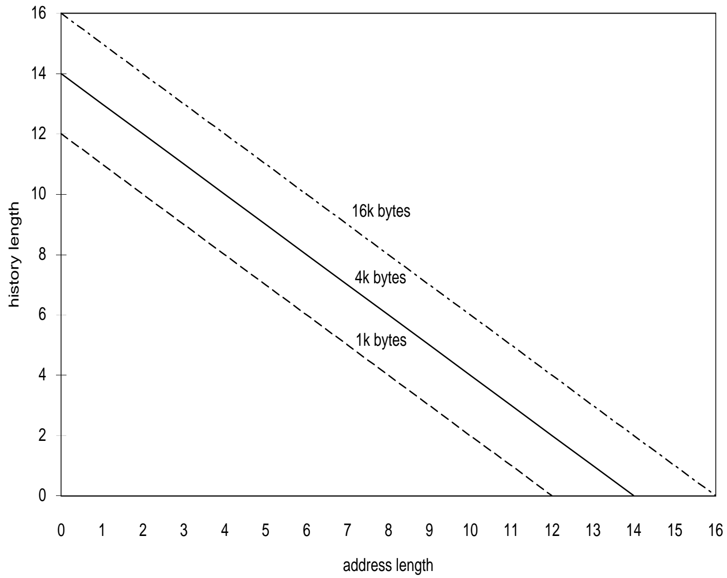


Figure 4.10: Equal-cost contours for 128 branch history entries

---

entries respectively.

In Figure 4.10, equal-cost contours are diagonal straight lines. This implies that, when the budget is fixed and the BHT entries are few, substituting one address bit with one history bit will incur almost no extra cost. In other words, the costs of each history bit and address bit are almost equal.

However, when the number of BHT entries is large, as shown in Figure 4.11, the equal-cost contours are almost parallel to the *x*-axis for small budgets. This implies that with the same budget, we can have more address bit than history bits. This is because, when the number of BHT entries is large, the cost of each additional address bit is insignificant to that of each additional history bit.

4.4.1 Cost/performance analysis

Figure 4.12 shows the optimal points for different budgets and configurations for the SPEC CINT95 benchmarks. Various configurations with the same budget are grouped as a clusters of bars, where each bar represents the best point for a fixed number of BHT

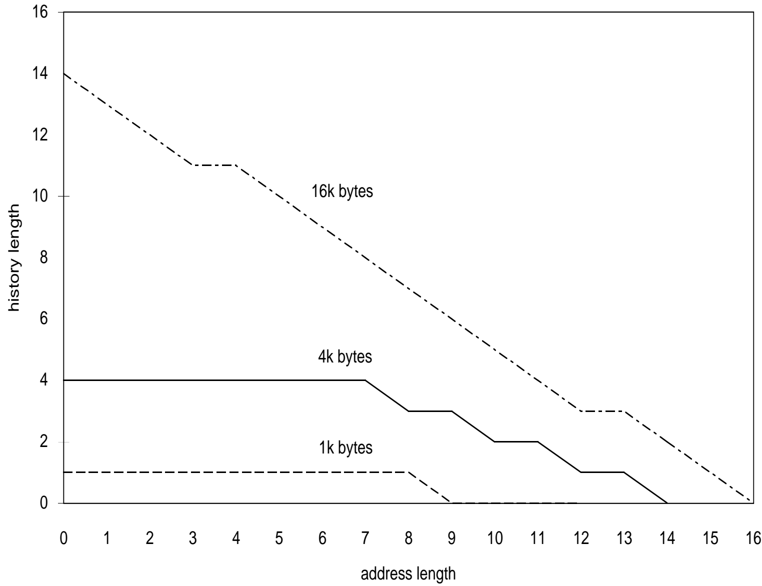


Figure 4.11: Equal-cost contours for 8k branch history entries

---

entries. The two-bit counter scheme is shown as a line. The best per-address two-level predictor consistently outperforms the 2-bit counter scheme.

However, for the IBS benchmarks (Figure 4.13), the two-bit counter scheme outperforms the per-address two-level predictor with small budget (512 byte). This is because IBS have relatively large number of static branches to be distinguished and predicted. These branches can be distinguished and predicted using either address bits or history bits (patterns). History bits require extra resources (BHT) to record them, while address bits can be obtained from the program counter and, thus, are essentially free. Consequently, when the budget is small, history information cannot effectively distinguish large number of static branches, because there are not enough resources to build a large BHT to record history information. In this particular case, the two-bit counter scheme can outperform a two-level predictor, because it does not need any history. However, as budgets increase, the two-bit counter scheme improves only a little. In contrast, the per-address two-level predictor improves and outperforms the two-bit counter scheme.

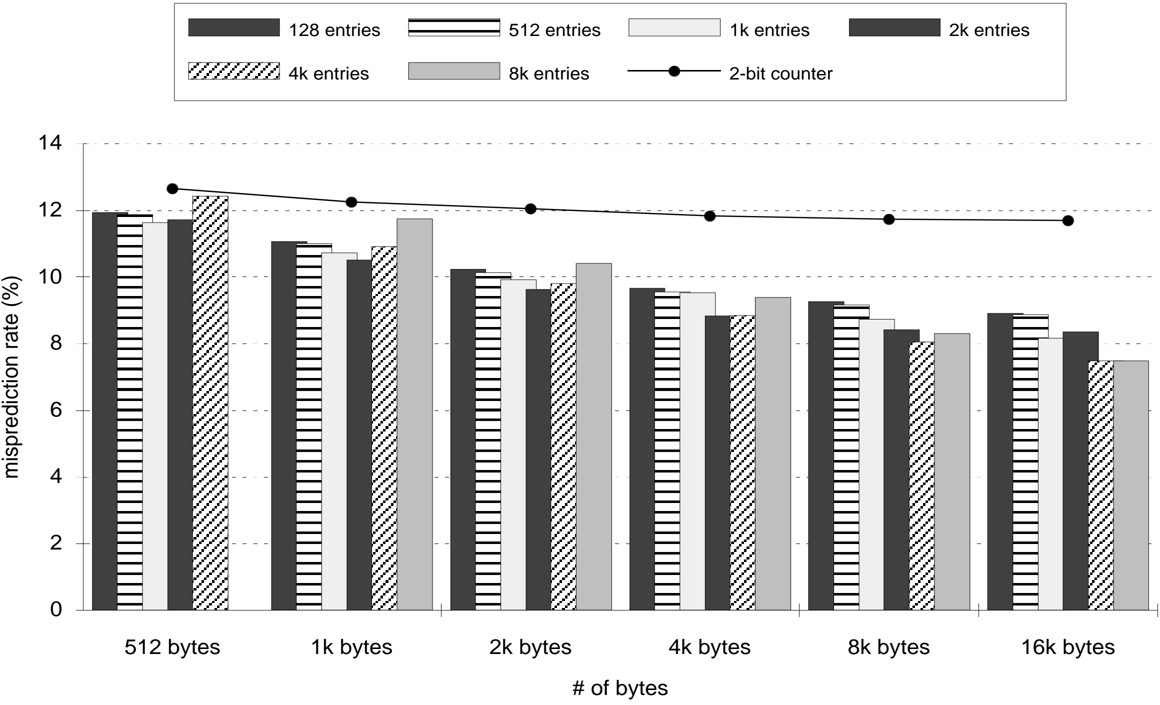


Figure 4.12: Misprediction rate vs. budget for SPEC CINT95

To study the optimal designs for two-level predictors, we plot the misprediction rates for different budgets and numbers of BHT entries, shown in Figure 4.14 and Figure 4.15. The *x*-axis indicates the number of entries in the BHT, and the *y*-axis indicates the misprediction rate. The dotted horizontal lines represent two-bit counter schemes (2bc). Each curve line in the figure represents a fixed budget, ranging from 512 bytes to 16k bytes. In addition to the number of BHT entries labeled in the *x*-axis, the optimal configuration for each budget is labeled with its two other parameters, formatted as (history bits, address bits).

When the budget is small, the address bits are very important. This is because, as previously explained, address bits are cheaper than history bits since they do not need extra resources for recording. Therefore, when the budget is small, it is relatively more efficient to use address bits instead of history bits. This fact can be verified for SPEC, shown in Figure 4.14. Labeled as the right number in each parenthesis, the number of

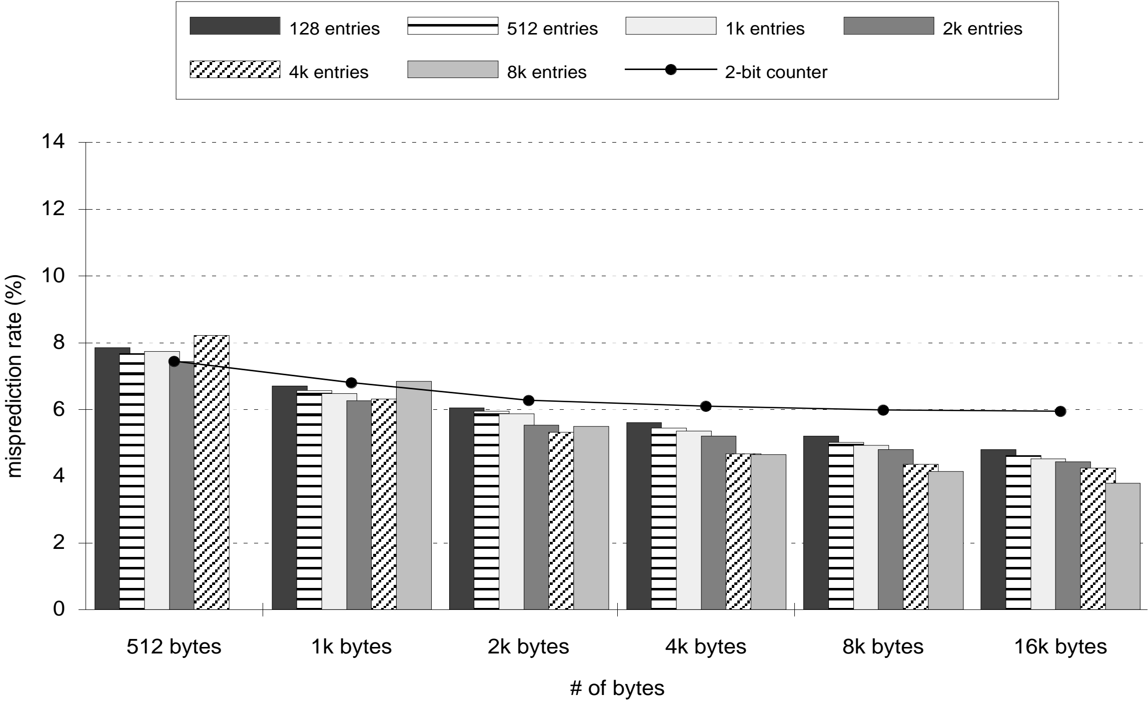


Figure 4.13: Misprediction rate vs.budget for IBS

---

address bits for the optimal configuration is very large when the budget is small. IBS exhibits similar behavior, as shown in Figure 4.15. The only difference is that the number of address bits is even larger for IBS, since IBS has more static branches to be distinguished than the SPEC benchmarks.

However, as budgets increase, the importance of address bits rapidly decreases and is replaced by history bits. At first, it may seem to be counter-intuitive that, when budgets increase, the number of address bits decreases, instead of increasing correspondingly or at least remaining constant. This is because the increase in history bits (patterns) can also replace the function of address bits, which is to distinguish branches. Therefore, as the budget increases, the history bits gradually takes place of address bits and become more important. This trend can be seen for both SPEC and IBS, shown in Figure 4.14 and Figure 4.15. Note that the number of address bits in SPEC decreases more rapidly because they contain less static branches.

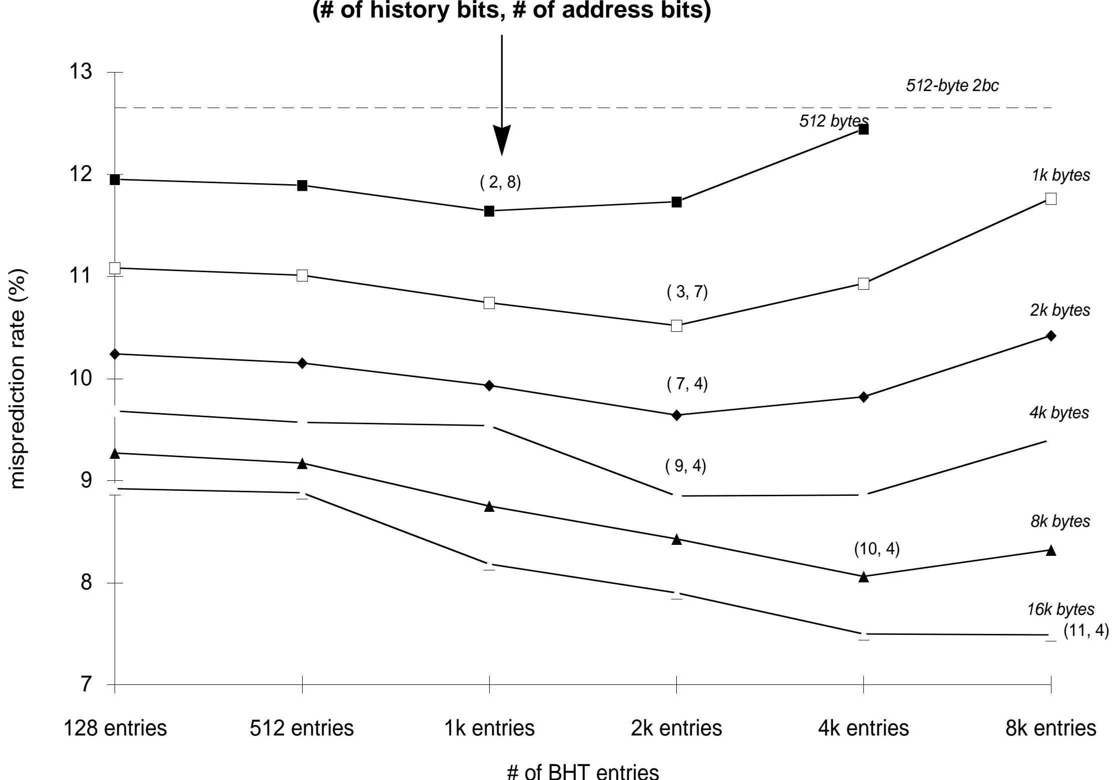


Figure 4.14: The optimal configuration for each budget in SPEC CINT95

Each optimal configuration is labeled with the number of history bits and address bits.

The importance of history bits quickly becomes dominant as budgets increase. In addition to distinguishing static branches like address bits, history bits can distinguish and better predict different patterns within the same branch. Thus, history bits can offer additional benefit over the address bits. This benefit is particularly important for branches that are hard to predict. Moreover, this benefit becomes more significant as budgets increase, since more resources can then be allocated to record history information. This trend of increasing address bits can be verified in both the SPEC and IBS benchmarks, shown in Figure 4.14 and Figure 4.15. As the optimal points move downward (budgets increase), the number of history bits increases (left numbers in the parentheses). We also notice that the number of history bits increase faster for SPEC, because branches in SPEC are relatively harder to predict [Sechrest96].

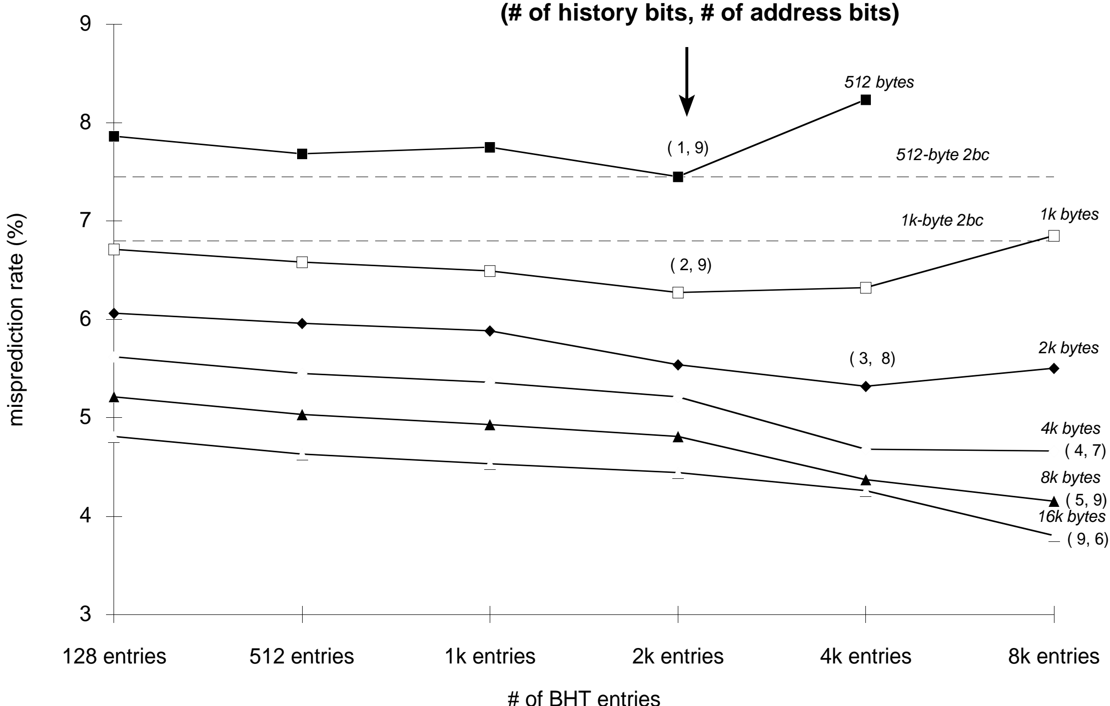


Figure 4.15: The optimal configuration for each budget in IBS

Each optimal configuration is labeled with the number of history bits and address bits.

Also note that as budgets increase, the number of entries in the branch history table also increases with the number of history bits. This trend can be explained by the increasing importance of history information. As described in the previous paragraph, history information becomes important when the budget is large. However, to effectively increase history information, increasing the number of history bits alone is not enough. Although by adding more history bits, more history information can be recorded in a longer branch history register, this information can easily be replaced and lost due to misses. Therefore, the number of entries in the BHT must be increased accordingly to improve the hit-rate. Since the number of BHT entries affects the hit rate, the number of entries needed is mostly determined by the number of static branches in the benchmarks. To see this trend, notice that the optimal configuration gradually moves toward the right along the *x*-axis (more entries in branch history table) as they move downward from one

curve to another (greater budget), as shown in Figure 4.14 and Figure 4.15.

As an aside, throughout the design budgets we examined, ranging from 512 bytes to 16KB, the tagless PAg scheme was never an optimal configuration. The tagless PAg is not cost-effective because it only uses expensive history bits, instead of relatively cheap address bits.

4.4.2 Design principles

The principles for designing tagless per-address predictors can be summarized as follows. When the budget is small, address information should be emphasized first. In other words, the number of address bits should be much larger than history bits (e.g., 8 address bits versus only 2 history bits). The number of address bits is determined by and proportional to the number of static branches in the benchmarks.

As the budget increases, address bits should decrease accordingly; at the same time, resources should be allocated for more history bits as well as the number of branch history entries. Concerning the rate at which address bits should be replaced with history bits: more aggressive replacement should be adopted when branches are hard to predict, or when the number of static branches is small. For example, when the budget is small, the SPEC benchmarks achieves the best performance when the number of address bits is 4 times the number of history bits. However, when the budget is large, the best performance is achieved when the number of address bits is reduced to less than half the number of history bits.

In addition to increasing with the budget, the number of branch history entries is determined by the number of static branches in the benchmarks. Large number of branch history entries should be allocated if the number of static branches is large, e.g., 4k to 8k entries are needed for the benchmarks we examined.

When the numeric values for design parameters are needed, computer architects can first measure the statistics of the targeted benchmarks, which including the number of

static branches and the misprediction rate for a baseline predictor within the budget. Then, to fine-tune the design parameters, these statistics can be compared to those of IBS and SPEC CINT95, as shown in Table 2.4, Figure 4.14, and Figure 4.15. Depending on how close the statistics are to those of IBS and SPEC, the parameters we have provided can either give a good estimate or, at least, significantly reduce the design space.

4.5 Conclusion

In this chapter, we have shown that the best design of branch predictor may change with technology and the actual design process requires a comprehensive analysis. We first evaluated the benefits of tagless per-address two-level branch predictors and then examined the design principles and cost/performance trade-off for a system with such a predictor.

To illustrate the benefits of tagless per-address predictors, we argued that they have faster access time, lower power, and simpler implementation than tagged predictors. At the same time, tagless predictors can offer performance comparable to the traditional tagged predictors by allowing additional resources to be allocated to the predictor and BTB and allowing these two components to be optimized as separate entities.

We also showed that the prediction accuracy of tagless predictors is better than direct-mapped tagged predictors. By characterizing the sources of prediction errors, we demonstrated the tagless predictor outperforms the direct-mapped tagged predictor due to the better capability of handling transitional-state branches.

We further evaluated cost and performance trade-off across a wide range of the design space. Equal-cost contour is the criteria for determining the best configuration. Based on the simulation results from the SPEC CINT95 and IBS benchmarks, we concluded that the number of address bits indexing into the second level table is the most important parameter when the available budget is small (e.g., 8 bit address bits versus only 2 history bits). However, the importance of address bits quickly drops as the budget

increases. With a larger budget, history bits and the number of branch history entries should increase accordingly, but the number of address bits should be reduced. In addition, we noticed that the PAg scheme is never an optimal configuration over the budgets and configurations we examined.

Finally, we present a set of design principles for tagless per-address two-level predictors. First, we can measure the statistics of target benchmarks, which include the number of static branches and the misprediction rate for a base configuration. Then, we can compare these statistics with those from IBS and SPEC. The quantitative data collected from IBS and SPEC can provide a rough idea of how an optimal implementation should be. By carefully examining the interaction among different parameters, we also outlined the principles on how to fine-tune these parameters for better design.

CHAPTER 5

IMPROVING INSTRUCTION FETCH BANDWIDTH AND I-CACHE PERFORMANCE USING DATA COMPRESSION

5.1 Introduction

In addition to branch prediction, data compression can improve instruction fetching rate in other aspects as well, such as instruction fetch bandwidth and instruction cache performance. Instruction fetch bandwidth and the performance of instruction caches are becoming increasingly important as microprocessors keep speeding up. To sustain a high execution rate in microprocessors, more instructions are needed in each cycle. Moreover, there is a trend referred to as “code bloat” caused by the growth in application binaries and increasing use of the operating system, which further put pressure on both instruction fetch bandwidth and instruction caches [Uhlig95a].

Data compression is an ideal candidate to alleviate these problems because it can reduce the effective code size. However, most data compression algorithms are too complicated to implement in hardware, and the overhead of decompression can easily outweigh the benefits from compression. Therefore, a very simple and fast compression algorithm is needed and, to be efficient, this algorithm should exploit inherent characteristics unique to programs.

In this chapter we present a straightforward technique for compressing the instruction stream for programs that overcomes the above mentioned limitations (this work has also been published in [Chen97c]). After code generation, the instruction stream is analyzed for frequently used sequences of instructions from within the program’s basic blocks. These patterns of multiple instructions are then mapped into single byte opcodes.

This constitutes a compression of multiple, multi-byte operations onto a single byte. When compressed opcodes are detected during the instruction fetch cycle of program execution, they are expanded within the CPU into the original (multi-cycle) sequence of instructions. We only examine patterns within a program's basic block, so branch instructions and their targets are only minimally affected by this technique allowing compression to be decoupled from compilation.

5.2 Intrinsic compressibility (redundancy) in programs

Compilers are universally used for program development, due to the complexity of managing the large applications developed today. However, despite the sophistication of the optimization process, the code generated by compilers can be sub-optimal and wasteful of program space, in part because compilers expand common syntax program fragments into machine instructions using a common set of mapping templates.

Even today, hand tuned assembly code is developed for those program fragments which have been found, by profiling, to execute frequently. Since programs spend most of their time in a few sections of code (the 90/10 locality rule [Hennessy96]), these hand-tuning for small sections of the program can have important performance payoffs.

5.2.1 Patterns

Compilers usually generate code using a *Syntax Directed Translation Scheme* (SDTS) [Aho86]. Syntactic source code patterns are mapped onto templates of instructions which implement the appropriate semantics. Consider, a simple schema to translate a subset of integer arithmetic:

```
expr ->  expr'+' expr
    {  emit( add, $1, $1, $3 );
       $$ = $1;
    }
```

```
expr ->  expr'*' expr
   {  emit( mult, $1, $1, $3 );
      $$ = $1;
   }
```

In these patterns, the expression subtrees on the right hand side of the productions return registers which is used by the arithmetic operation. The register number holding the result of the operation ($1) is passed up the parse tree for use in the parent operation. These two patterns would be reused for all arithmetic operations throughout all generated programs. The only difference in instruction sequences would be the register numbers used in the arithmetic operations.

More complex actions (such as translation of control structures) generate more instructions, albeit still driven by the template structure of the SDTS.

Although the code generation scheme outlined above is part of one method of translation (namely, pattern matching code generation), this reasoning applies to other techniques for instruction selection and code generator. In general, the only difference in instruction sequences for given source code fragments at different points in the object module are the register numbers in arithmetic instructions and operand offsets for *load* and *store* instructions. As a consequence, object modules are generated with many common sub-sequences of instructions. There is a high degree of redundancy in the encoding of a program.

Object Oriented Programming (OOP) languages encourage the increase of redundancy in programs. *Information hiding* is one organizational strategy (among many) used for OOP. The implementation of an object is hidden within the private namespace of the class, with member functions used as the interface to the object. Often, these member functions are simple access routines which reference private member data structures. These short code sequences are also pattern templates, similar to the SDTS of a compiler.

5.2.2 Instruction fetch bottleneck

Although most work has been done in optimizing data fetch behavior, it is clear that instruction fetch is also a significant bottleneck for high performance computing. An I-cache miss will stall the processor, while the instruction memory access bandwidth limits the rate at which new instructions can be delivered to the processor.

A recent study [Uhlig95a] examined instruction fetch and observes that "code bloat" caused by the growth in application binaries, with each new version, and increasing use of operation system services is resulting in larger and larger load modules that put increasing pressure on instruction caches.

A study at DEC [Perl96] also showed a similar result. When executing an SQL server on a DEC Alpha 21064 processor, instruction cache misses alone require twice the bandwidth than the actual pin bandwidth of the processor. Similarly, when executing the same program on an Alpha 21164 processor, instruction cache misses still generate traffic exceeding the processor pin bandwidth.

The instruction compression technique presented in this chapter can effectively solve the above mentioned problems. Because this scheme reduces the size of programs, it effectively increases the effective size of an I-cache for the same program fragment (a code stream is dynamically expanded within the CPU rather than redundantly occupying precious cache lines). Moreover, since fewer bytes are transferred from the I-cache to the CPU, the effective instruction fetch bandwidth is increased.

5.3 Description of the compression technique

Given the high degree of redundancy of instruction streams, an effective compression of redundant sequences should improve instruction fetch behavior. In this section, we describe our instruction compression technique.

After code generation and register allocation, we analyze the generated code

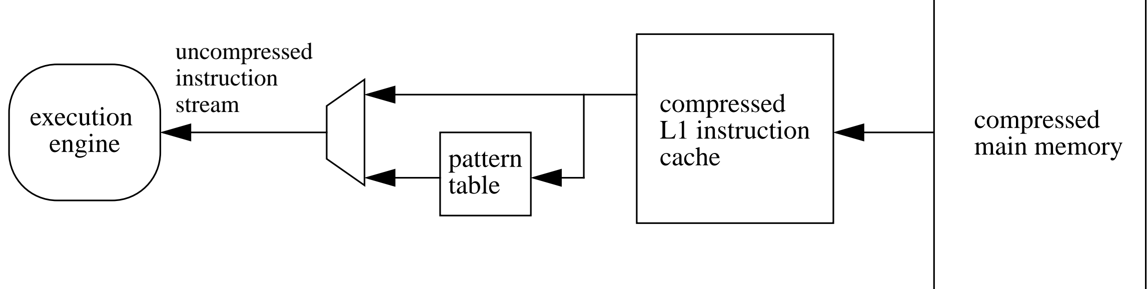


Figure 5.1: Organization of the compression scheme

stream to search for patterns. Patterns are built from the bit patterns of sequences of machine instructions. The pattern search is made over instruction sequences *within* all basic blocks of the program[1] (excluding branches). The maximum length of patterns is limited to 8 instructions for practical reasons. A frequency count is maintained for each unique pattern. The resulting set of patterns serves as the basis for program compression.

We also annotate each instruction with its execution frequency established through profiling. This profile information is accumulated with each pattern usage occurrence in the program.

After the pattern set has been constructed, a simple tiling procedure is applied to the program. Those patterns with the highest frequency of usage are encoded as one byte, and the original sequence of instructions for the pattern is replaced by this opcode. Incidence counts for patterns which overlap the selected pattern are subtracted.

The original instruction sequence is saved in a table in the CPU (see Figure 5.1). During instruction fetch, the instruction decoder checks the opcode of the incoming instruction. If the opcode indicates an uncompressed instruction, then instruction decode and execution proceeds in a conventional fashion. When the decoder encounters a compressed instruction, the entire sequence of instructions is retrieved from the ROM and

1. A basic block starts at a label (or after a branch operation) and ends with a branch operation (or another label).

	Benchmark	Static Instructions	Basic blocks
Integer	compress	2,260	223
	gcc	351,936	29,054
	go	62,380	8,877
	ijpeg	47,988	2,161
	li	16,752	1,167
	perl	87,172	3,252
	vortex	142,640	12,187
Floating point	applu	13,064	534
	fppp	12,848	334
	hydro	9,528	1,018
	su2cor	11,056	875
	swim	2,064	178
	tomcatv	1,308	128

Table 5.1: Instruction counts

dispatched through the execution pipeline sequentially. Instruction fetch from the I-cache is avoided until the sequence completes.

We have incorporated the decode table as a ROM in our CPU model. However, if the cost of loading this table is not high (with respect to the execution of the associated thread), then this table could be part of the state of the process.

5.4 Simulation results

To measure the impact of compression on dynamic behavior, we conducted trace-driven simulations. As input for the simulation, we use SPEC CINT95 and CFP95 benchmark suites [SPEC95]. The statistics of traces from the SPEC benchmarks are summarized in Table 5.1. The table shows the number of instructions and basic blocks of each program.

We do not currently have a compiler that generates code to support this

optimization. Therefore, we simply used the output generated by a existing compiler. All benchmarks were compiled with the DEC C compiler and DEC fortran compiler using -O optimization flag. Because code was not generated to take advantage of this peephole optimization, we believe that the performance information presented in this chapter represent a “worst-case” optimization capability for this compression technique.

To collect profile information and instruction traces, we used ATOM [Eustace95]. The profile information for the programs was attached to each basic block, and patterns were constructed from resulting assembler listing. Since we assume that a pattern set can be viewed as a part of the process state, we used the same programs for “training” as for simulation.

The instruction cache we simulated was a direct mapped implementation. The I-cache capacity measured was 2K through 32K. We measured caches with line sizes of 4 and 8 words (16 and 32 bytes). We looked at the traffic from the CPU to the first level cache.

Since we have compressed programs into a single byte opcode, we have made the necessary assumption that instruction words are not aligned.

In examining the results of our simulations, we were interested in two distinct machine behaviors:

1. The number of bytes fetched by the CPU from the level 1 I-Cache.
2. The miss ratio for the I-cache for different cache sizes.

These two sets of values were collected and compared for both compressed and uncompressed instruction traces.

The byte fetch demands provides an index of the demands upon the I-cache from the CPU. By comparing compressed and uncompressed programs, we can get a measure of the reduction in requirements caused by compression.

The I-cache miss ratio is a measure of the relative effectiveness of cache utilization by compressed and uncompressed instruction streams.

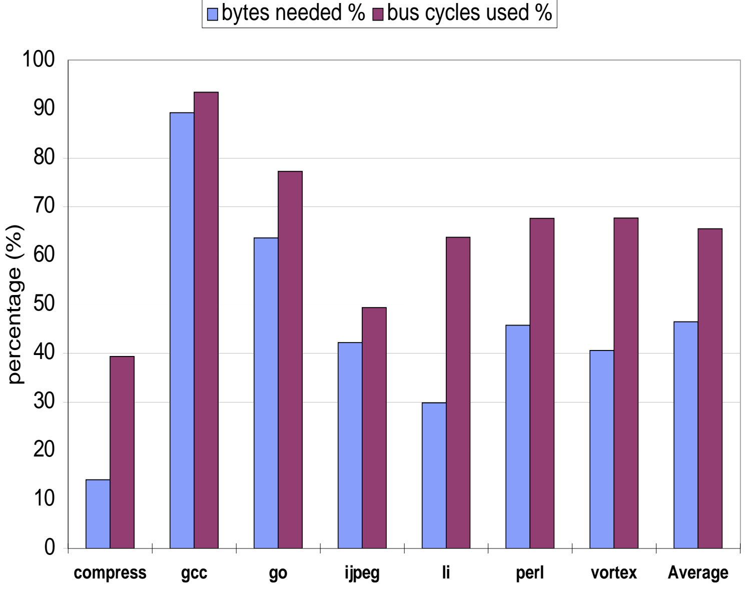


Figure 5.2: Bytes needed and bus cycles used for SPEC CINT95 benchmarks

---

5.4.1 I-cache fetch behavior

We compared the I-cache fetch requirements of programs with compressed instruction sequences against those with uncompressed streams. Figure 5.2 and Figure 5.3 show the results. Figure 5.2 shows the results of the *integer* programs from the SPEC benchmarks, while Figure 5.3 shows the *floating point* program set.

Each chart is organized to show the relative performance of each benchmark in its compressed implementation against the uncompressed version. The value of 100% indicates the fetch costs incurred by the uncompressed version of the program, while the bars measure cost in the compressed implementation.

Associated with each program are two cost bars. The left bar indicates the number of compressed bytes required by the CPU for program execution when we use 128 patterns. Here the value 100% indicates the total number of bytes needed by the CPU to execute the uncompressed program ($\text{total instructions} \times 4$). This compressed number is a measure of the minimum number of bytes required. However, this number may be too

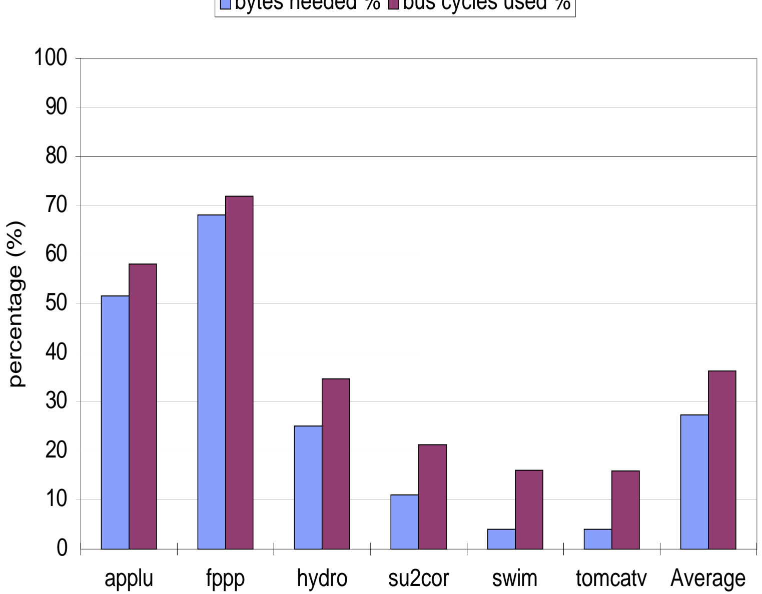


Figure 5.3: Bytes needed and bus cycles used for SPEC CFP95 benchmarks

---

optimistic because each I-cache fetch may result in multiple bytes delivered to the CPU (determined by the width of the bus between the CPU and I-cache). Therefore, we have also measured the number of bus cycles required for instruction transfer; this is the right bar for each program. We have provided the two views mainly for comparison. To compute the bus cycles, we assume each cycle 4 consecutive bytes (one instruction) can be fetched from the I-cache and a perfect I-cache is used. Each cache fetch less than 4 bytes is padded with no-ops and still takes one cycle to complete. Although it appears that the *byte-fetch* measure may be too optimistic an assessment of I-cache fetch behavior, any machine that uses a pre-decode phase [MReport95a] or has queue to hold incoming instructions would more likely approximate the byte fetch behavior recorded with the left bar.

The last entry of both charts represent the average performance of all programs in the associated benchmark set (integer or floating point).

Looking at the integer programs, we can see that the number of bytes fetched by

the CPU from the I-cache is, on average, less than 50%. Even assuming that each fetch from the I-cache requires a 4 byte transfer, on average we have reduced access to the first level cache by 35%.

Looking at the floating point benchmarks, the impact on I-cache traffic becomes more pronounced. Overall, the instruction fetch traffic to the I-cache is reduced by over 70%. If we assume a full, 4 byte transfer is required for each instruction fetch, we have still reduced the I-cache access requirements by 65%

5.4.2 I-cache miss behavior

The floating point benchmarks from SPEC are all very small. Their code segments are often small enough to fit entirely into even a small I-cache. This eliminates I-cache misses for all but the smallest I-cache configurations.

For this reason, we will concentrate upon examining the I-cache miss behavior of the integer benchmarks.

Consider the SPEC integer benchmarks. The average miss ratios for these benchmarks are shown in Figure 5.4. When programs are compressed with 128 patterns (that is, the 128 most active instruction sequences are compressed to single opcodes), the average miss ratio is less than the miss ratio for an uncompressed program utilizing a cache with twice the capacity. Even when only 32 patterns are used, the overall I-cache rate is reduced. This configuration achieves about half the miss rate reduction shown with 128 patterns.

The miss ratios for each individual benchmark are shown in Figure 5.5. Most of the benchmarks follow the same trend as described above, the notable anomaly is the *li* benchmark shown in Figure 5.5. In some configurations of the *li* benchmark, the compressed program has higher miss ratio than the uncompressed version. This occurs due to thrashing in the caches which are all direct-mapped. For example, if two frequently used instructions happen to map to the same location in a direct-mapped cache, the overall

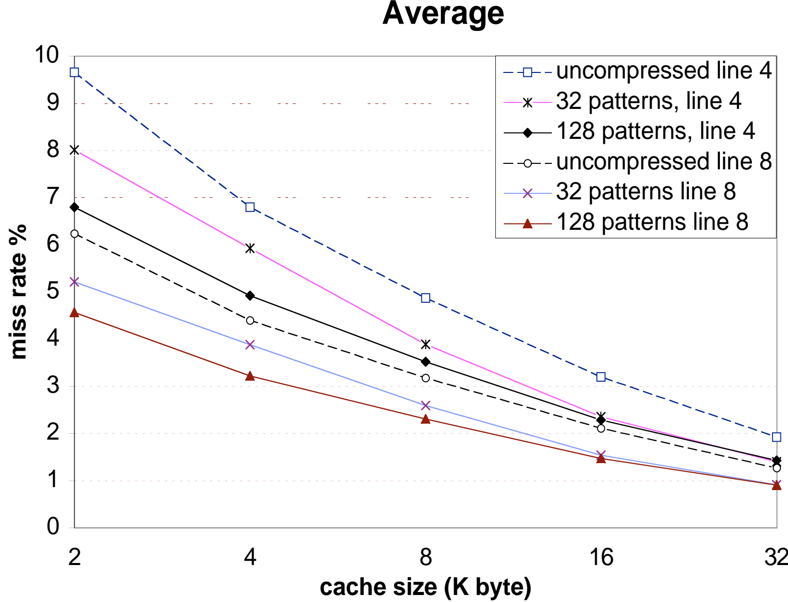


Figure 5.4: Average miss ratio for SPEC CINT95 benchmarks

---

miss ratio can increase due to the frequent swapping of these two instructions (if these two instructions are used alternatively). For similar reasons the benefit of compression is minimal in the case of *gcc* benchmark.

5.4.3 Pattern table sizes

As discussed in Section 5.3 above, instructions in each basic block are profiled to find the frequently used sequences (patterns). The original sequence of instructions is saved in a table in the CPU which is accessed during instruction fetch and decode to retrieve the original sequence of instructions. Since the table occupies space in the CPU, it is desirable that its size be small.

Table 5.2 shows the size of the pattern table (remapping table) for each application n integer benchmarks, and a list of possible pattern ranges. For all pattern sets less than 32, the pattern table size is less than 1K. When 128 patterns are used, the pattern table is in the range of 1K to 4K.

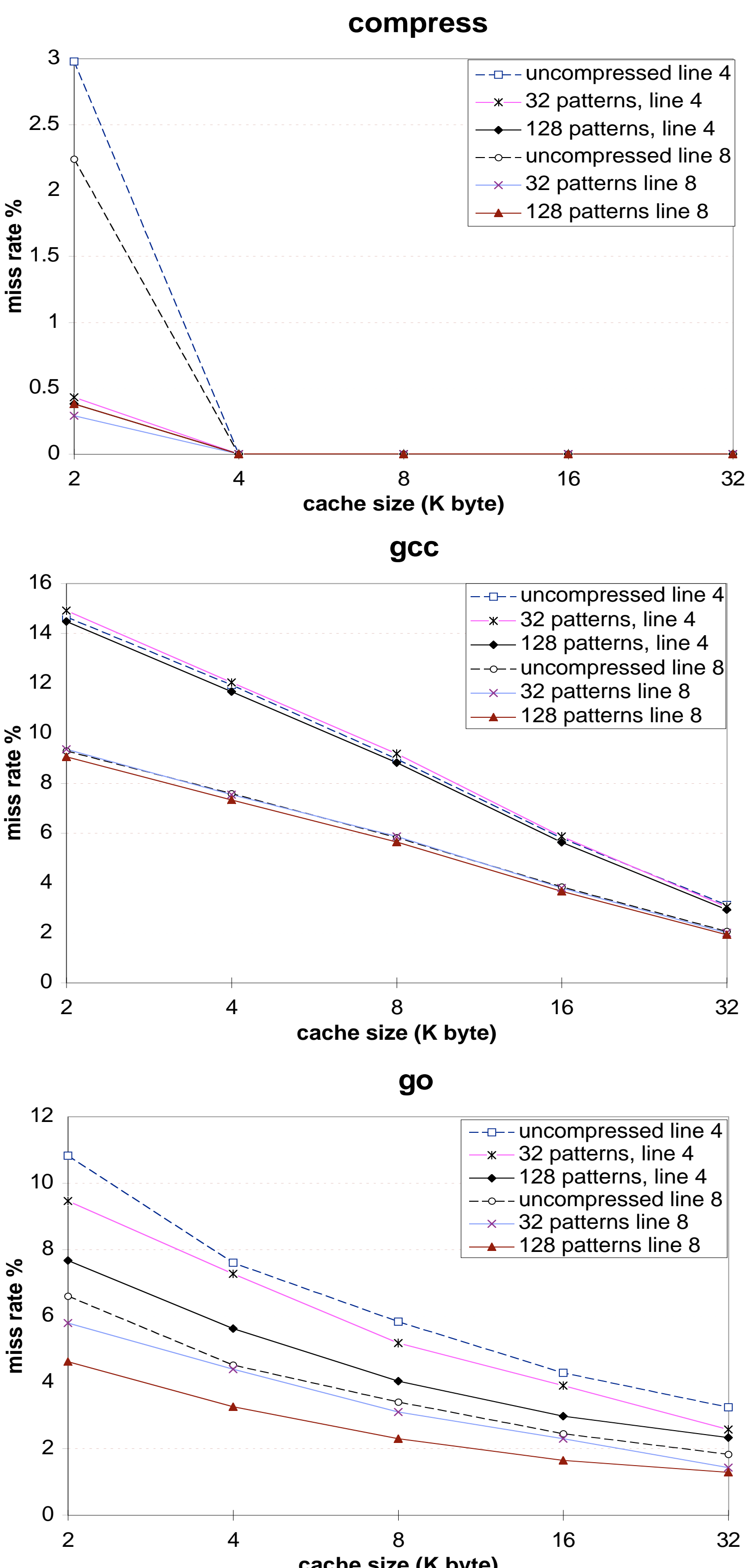


Figure 5.5: miss ratios for each individual benchmark

(note that the miss rate scaling factor changes)

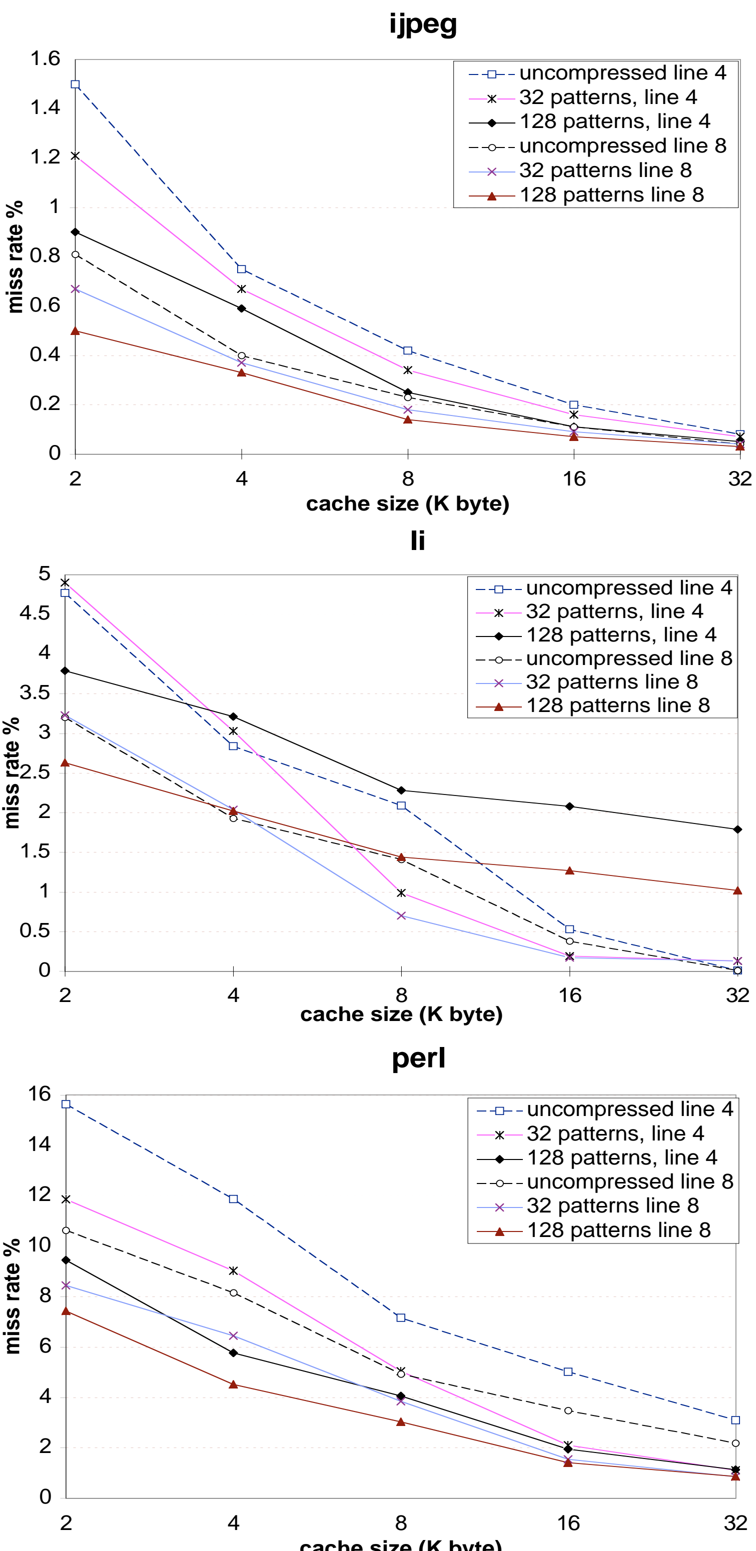


Figure 5.5: miss ratios for each individual benchmark

(note that the miss rate scaling factor changes)

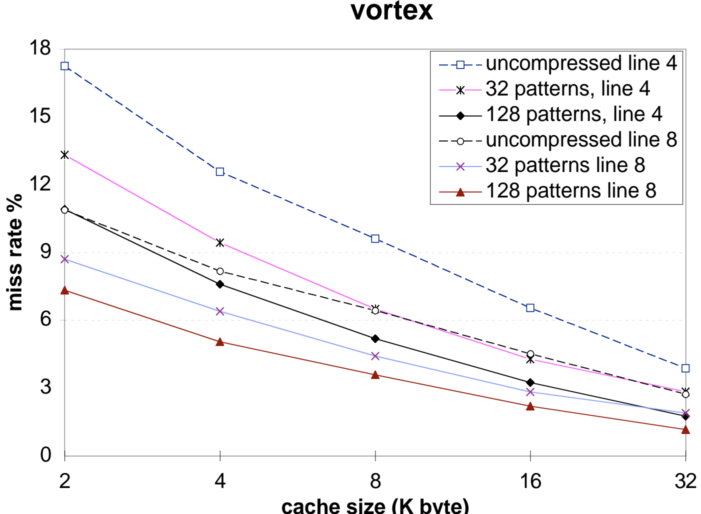


Figure 5.5: Miss ratios for each individual benchmark

(note that the miss rate scaling factor changes)

	Number of Patterns				
	8	**16**	**32**	**64**	**128**
compress	228	448	836	1,400	1,776
gcc	172	348	780	1,332	2,208
go	156	340	648	1,248	2,372
ijpeg	196	392	820	1,792	3,776
li	204	348	564	824	1,132
perl	208	376	704	1,176	1,788
vortex	196	292	592	980	1,748

Table 5.2: Pattern table sizes in bytes

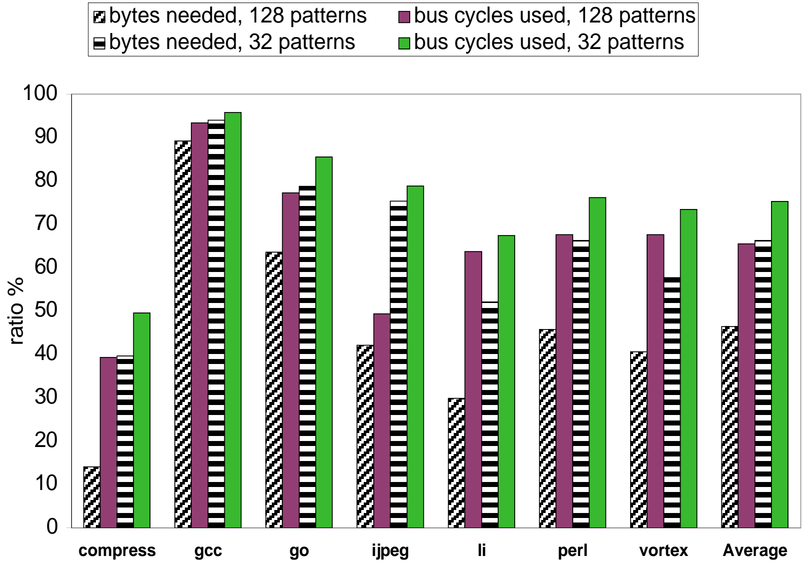


Figure 5.6: Comparison of the compression effects with 32 patterns and 128 patterns

Figure 5.6 shows the impact of the increase in pattern counts upon the *byte-fetch* behavior. Shown are the performances for the integer benchmarks using 32 patterns and 128 patterns. For each benchmark, 4 bars are shown. Again, we include the case where transfers between cache and CPU must be in multiples of 4 bytes to model the behavior of a word-wide bus.

- 128 Patterns: Byte fetch requirements

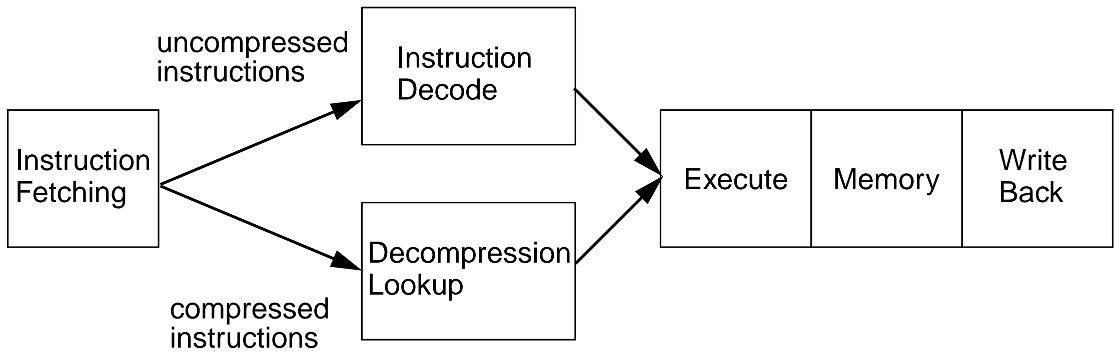


Figure 5.7: A five-stage pipeline using predecoded information to reduce the delay of decompression lookup

- 128 Patterns: Byte fetch, assuming a 4byte bus
- 32 Patterns: Byte fetch requirements
- 32 Patterns: Byte fetch, assuming a 4byte bus

On average, the 32 pattern implementation reduced the I-cache fetch requirements by 35%.

5.5 Discussion of implementation issues

To reduce the latency of the extra decompression lookup, we can store predecoded control information in the pattern table rather than the plain uncompressed instructions. Using this predecoded information, we can save one decoding cycle for compressed instructions and, thus, make up for the cycle needed to lookup the decompression table. The predecoded information is similar to microcode and hence its length is determined by implementation details. To illustrate this process, a 5-stage pipeline for executing compressed and uncompressed instructions is shown in Figure 5.7. All uncompressed instructions execute normally through the upper path, while all compressed instruction execute through the lower path. After the instruction fetching stage, compressed instructions enter a decompression lookup stage, during which predecoded information is obtained, and thus eliminating the need of an instruction decode stage.

To increase the object-code compatibility between processor generations, we can

add the predecoded information incrementally to the dictionary each time we first decode a dictionary entry. Then we can cache the predecoded information locally for future use. This is modeled on the approach proposed in [Conte96] to support VLIW object-code compatibility and the approach used in the Andrew File System [Silberschatz94]. In this way, a program is not constrained by machine dependent information and can executed across different processor generations.

When this compression scheme is applied to a multi-tasking environment, we can associate a process identifier to each entry in the dictionary, to allow each program to have its own dictionary. This addition of process identifiers eliminates the need to flush the dictionary on context switches. The overhead of dictionary management should be small since the dictionary only has 32 to 128 direct-mapped entries.

5.6 Related research

In [Wolfe92], Wolfe and Chanin propose a compression scheme for embedded microprocessors. Their scheme expands instructions on refills after a cache miss. Since the cache resident version of the program is uncompressed, the miss rates are not improved. Their scheme also incurs other penalties. Programs are compressed using a Huffman encoding which complicates decompression by requiring address pattern tables and extra translation hardware to maintain the relationship between the compressed and the uncompressed address spaces. A cache miss needs to be able to compute where in the compressed program the missing cache line resides.

An interesting contrast to our scheme is the Trace Cache [Rotenberg96]. The Trace Cache increases the effective bandwidth by combining information from multiple basic blocks. On the other hand, our scheme increases the effective bandwidth by condensing the information in each single basic block. The fundamental difference in the two schemes is that the Trace Cache captures information about program flow *dynamically*, while our compression technique is *static*; all compression is discovered during compilation. Thus,

the two approaches are orthogonal. They might be applied in conjunction to achieve a greater benefit.

5.7 Conclusion and future research

The instruction stream represents a significant bottleneck to high performance execution. In this chapter we have outlined a technique for reducing a program's instruction stream by compressing frequently encountered instruction sequences into single byte opcodes.

This compression technique has been examined for both static and dynamic models. When the technique is applied for static pattern incidences [Bird96, Lefurgy97], it reduces the overall program size to about 60%. Although the compression technique is adversely affected by heavy usage of large register sets, which will result a wider range of instruction bit patterns, it can effectively compress both CISC and RISC programs.

In the study presented in this chapter, we are interested in the impact of instruction stream compression on the I-cache behavior for a given architecture. We do not have a compiler that is organized for this optimization, so we used the output of an optimizing compiler for the Alpha RISC architecture.

Despite the fact that the compiler was not tuned to exploit instruction compression, we were able to reduce both the I-cache byte fetch requirements and the I-cache miss rates for the integer programs from the SPEC benchmarks.

We believe that the impact of instruction stream compression should be higher when we incorporate this optimization technique more directly into the compiler. This technique could be integrated either by rebuilding the code generation phase (most specifically, register allocation), or by performing a peephole optimization pass over the generated instructions (possibly reassigning registers so as to increase the common pattern count and hence increase pattern incidences).

Since we are constructing patterns based solely upon the bit patterns of individual

instructions, the use of a large member of registers by a compiler has a negative impact upon pattern incidence counts. And yet, although using only a very small register set size will increase pattern incidences, it will likely have an adverse effect on overall program execution. It is important, therefore, to identify strategies that both lead to high performance code sequences, and aid in the generation of compact code. At least two strategies might be employed.

It is possible for the compiler to establish machine “idioms” (much like a VECTOR opcode). The registers, instructions, and operand offsets for the operands to these “meta-operations” would need to be fixed.

Another possible implementation could use a smaller set of patterns (hence a smaller pattern table), but limit its scope to a subset of the executing program. If the size of the pattern table could be kept small (on the order of 100 bytes), it would be advantageous to update this table at different program phases. The table reload cost would be offset by the execution time savings due to reduce I-cache misses.

Although the pattern tables in our study are not large, the sizes of these tables for 128 pattern implementations are probably too large to permit incorporating them as a component of program state (due to the high cost of loading them upon thread dispatch). We believe either of the above two implementation schemes could reduce the pattern tables to manageable proportions.

Throughout our studies, we have presumed a single instruction stream model for program execution. However, it naturally extends to the VLIW execution model. In this case, each line of the pattern table could hold one or more VLIWs for parallel dispatch. Couple with trace scheduling [Ellis85], an entire high-incidence trace could be represented as a single opcode.

CHAPTER 6

PREFETCHING USING BRANCH PREDICTION INFORMATION

6.1 Introduction

Instruction prefetching is an important technique for closing the gap between the speed of the microprocessor and its memory system. As current microprocessors become ever faster, this gap continues to increase and becomes a bottleneck, resulting in the loss of overall system performance. To close this gap, instruction prefetching speculatively brings the instructions needed in the future close to the microprocessor and, hence, reduces the transfer delay due to the slow memory system. If instruction prefetching can predict future instructions accurately and bring them into the instruction cache in advance, most of the delay due to the memory system can be eliminated.

In this chapter we propose an efficient instruction prefetching scheme that makes use of current advanced branch prediction mechanisms that are often already part of the architecture. The branch predictors are built into current microprocessors to reduce the stall time due to instruction fetching and, in general, can achieve prediction accuracy as high as 95% for SPEC benchmarks [SPEC95]. With such high prediction accuracy, the instructions needed in the future can also be predicted accurately and be prefetched in advance. Furthermore, this approach is inexpensive because it applies and shares the existing branch predictors with little additional hardware cost.

We will show that prefetching based on branch prediction (BP-based prefetching) can achieve higher performance than a cache of 4 times the size in all the benchmarks and configurations we examined. In addition, BP-based prefetching outperforms other

hardware instruction fetching schemes, such as next-*n* line prefetching and wrong-path prefetching, by a factor of 17-44% in stall overhead. BP-based prefetching achieves its performance by speculatively running ahead of the execution unit at a rate close to one basic block per cycle. With the aid of advanced branch predictors and a small autonomous fetching unit, this type of prefetching can accurately select the most likely path and fetch the instructions on the path in advance. Therefore, most of the prefetches are useful and can fetch instructions before they are needed by the execution unit.

The chapter is organized into five sections. In Section 6.2 we introduce some related prefetching schemes and the ideas behind BP-based prefetching. Section 6.3 describes our simulation environment and the benchmarks used. In Section 6.4, we present BP-based prefetching simulation results and provide some qualitative analysis. In Section 6.5 we discuss some implementation issues.

6.2 Description of prefetching schemes

6.2.1 Related prefetching schemes

The concept of a Look Ahead Program Counter (LA-PC) has been proposed by Chen and Baer [Chen95]. This is a pseudo-program counter that runs several cycles ahead of the regular program counter (PC). The LA-PC is then used to look up a Reference Prediction Table to prefetch data in advance. Although the concept is in some ways similar to our proposed scheme, the LA-PC scheme is much more conservative; it only advances one instruction per cycle and is restricted to be, at most, a fixed number of cycles ahead of the regular PC. Furthermore, the studies in [Chen95] focused on data prefetching rather than instruction prefetching, and did not evaluate the effects of speculative execution, multiple instruction issue, and the presence of advanced branch prediction mechanisms. Some other data prefetching schemes extending their work can be found in [Liu96, Pinter96].

Another scheme, the next-*n* line prefetching scheme [Smith82], prefetches the next *n* sequential cache lines following the current program counter. This scheme is effective be-

cause it exploits the characteristic that most programs tend to execute sequentially by bringing the sequential lines in advance.

A third scheme, wrong-path instruction prefetching has been proposed by Pierce et al. [Pierce96]. This is an effective prefetching scheme combining next-line prefetching with the prefetching of all control instruction targets. The wrong-path scheme prefetches along both paths of a branch instead of simply prefetching along the predicted correct path, and the branch targets are computed during the normal decoding stage. The wrong-path scheme is based on the observation that programs eventually execute instructions along the not taken or "wrong path." Wrong-path prefetching has been shown to be more accurate and cost-effective than hybrid schemes [Smith92], which are table-based schemes.

6.2.2 Branch prediction-based prefetching

In the instruction prefetching scheme we propose, the prefetching unit is an autonomous state machine, which speculatively runs down the instruction stream as fast as possible and brings all the instructions encountered along the path. When a branch is encountered, the prefetching unit predicts the likely execution path using the branch predictor, records the prediction in a log, and continues. In the meantime, the execution unit of the microprocessor routinely checks the log as branches are resolved and resets the program counter of the prefetching unit if an error is found.

The extra hardware needed for prefetching is very little: just an additional program counter (PC), a return stack, an adder, and some miscellaneous logic. More details about the implementation will be discussed in Section 6.5. The branch predictors discussed in this chapter are two-level branch predictors and return address stacks, which are already found in various commercial microprocessors, such as the Intel Pentium Pro and the DEC Alpha 21264 [MReport95b, MReport96]. Figure 6.1 shows a block diagram of the conceptual organization of BP-based prefetching scheme.

Figure 6.2 shows the detailed operation of our proposed prefetching scheme. Initially, the program counter (PC) of the prefetching unit is set to be equal to the PC of

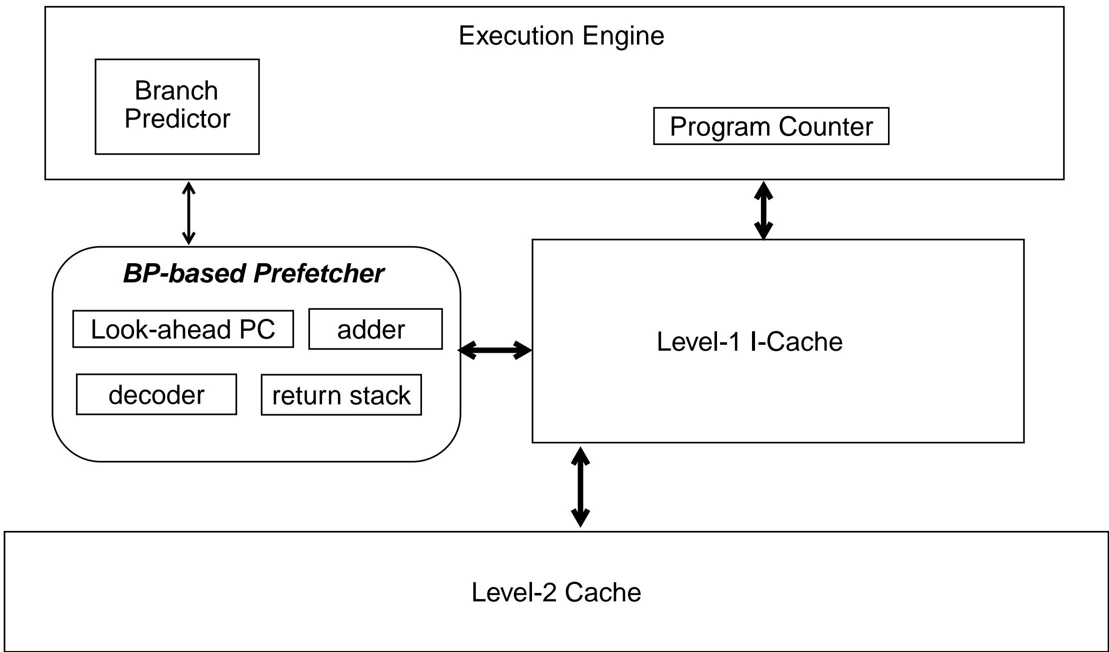


Figure 6.1: Conceptual organization of BP-based prefetching scheme

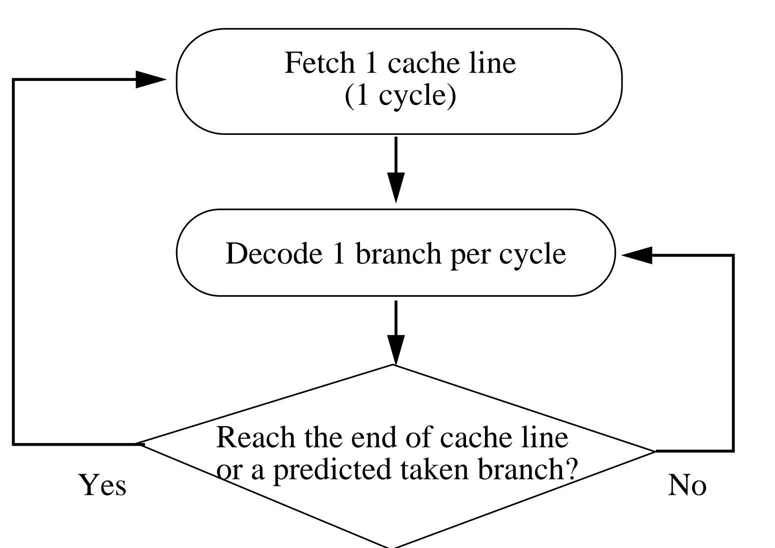


Figure 6.2: Flowchart of BP-based prefetching

the execution unit. Then the prefetching unit spends one cycle to fetch the desired cache line.

The prefetching unit examines an entire cache line as a unit, and quickly finds the

first branch (either conditional or unconditional) in that cache line using existing predecoded information or a few bits from the opcode. This searching of branches can be done very fast if the opcodes for branches are encoded appropriately. During the same cycle, the prefetching unit also predicts and computes the potential target for the branch as following.

For each branch, the prefetching unit spends one cycle to predict and compute the potential target of the branch. Depending on the type of the branch, the potential target is generated by three different prediction mechanisms: First, for a subroutine return branch, its target is predicted with a return address stack, which has high prediction accuracy [Kaeli91]. The prefetching unit has its own separate return address stack. Second, for a conditional branch, the direction is predicted with a two-level branch predictor and the target address is computed with the dedicated adder in the same cycle. A dedicated adder is used instead of a branch target buffer, because the branch may be encountered for the first time and thus will not yet be recorded in the target buffer. Also note that the two-level branch predictor used in the prefetching unit has its own small branch history register but shares the same expensive pattern history table with the execution unit (a *gshare* scheme is used—see Section 6.4). The prefetching unit only speculatively updates its own branch history register, but does not update the pattern history table. Third, for an unconditional branch, its direction is always taken and its target is calculated using the same adder used for conditional branches. However, for a branch on register, the prefetching unit stalls and waits for the execution unit because this type of branch can have multiple targets.

Depending on the predicted directions of the branches, different cache lines are prefetched. When a branch is predicted to be taken, the cache line containing its target is prefetched; otherwise, the prefetching unit continues to decode and predict the next branch in the cache line. The prefetching unit continues to work on the next branch until the end of the current cache line is reached, then the next sequential cache line is prefetched. The entire process is repeated again for the newly prefetched cache line.

To verify the predictions made, when a branch is predicted, the predicted outcome is recorded in a log. This log is organized as a first-in-first-out (FIFO) buffer. When the execution unit resolves a branch, the actual outcome is compared with the one predicted by the prefetching unit, which is recorded in the log. If the actual outcome matches the one predicted, the item is removed from the log. However, if the actual outcome differs from the one predicted, meaning the prefetching unit has gone down a wrong path, then the entire log is flushed and the PC of the prefetching unit is reset to the PC of the execution unit. In addition, it is also necessary to reset the contents of the branch history register and the return address stack of the prefetching unit to those of the execution unit.

We must guarantee the prefetching unit always stays ahead of the execution unit to prefetch new instructions. Violation of this condition is detected when the execution unit resolves a branch but the log is empty at that time. If this occurs, we need to reset the PC and branch history information of the prefetching unit to those of the execution unit.

Finally, we also enhanced our proposed prefetching scheme with the next-*n* line prefetching. This was done by prefetching the next *n* cache lines following the PC of the prefetching unit rather than from the PC of the execution unit. This enhancement is very effective because it reduces the delay by fetching the sequential cache lines in advance. These next-*n* line prefetches are assigned with the lowest priority, and are executed when the bus is free (not used by the execution unit or the prefetching unit).

6.3 Simulation environment

6.3.1 Simulation of speculative execution

Due to the speculative nature of prefetching, the normal trace-driven simulation is not enough to capture the behavior of the microprocessor that we are interested in, because it records the actual execution path of a program, and, thus, only contains the instructions executed by the program. However, if there is incorrect speculation, the prefetching

activity may bring redundant instructions not used by the program. These redundant instructions are not recorded in the traces, yet they are important in the evaluation of the pollution effect in the memory system. Therefore, to evaluate the effect of incorrect speculation and pollution, we add an instruction-fetching engine based on a design first developed in [Pierce96]. This instruction-fetching engine enables us to fetch any instruction in the program. This engine is implemented in the following way: it first disassembles the binary program to get all the instructions. Then the engine reads all the instructions and keeps them in an internal data structure for future access. When the engine receives an address requesting for an instruction, it searches in the data structure and returns the corresponding instruction. With this addition, the instructions examined are not limited to the ones recorded in the trace-driven simulation; the redundant instructions due to incorrect speculation can also be accessed and included in the final simulation.

To simulate speculative execution, our final simulation combines both the trace-driven simulation and the instruction-fetching engine. The traces from the trace-driven simulation are used to guide the correct execution path of the program, while the instruction-fetching engine is responsible for the speculative behavior of prefetching activities. More specifically, our simulated microprocessor will execute the correct instructions from the traces; our prefetching mechanism will guess the instructions needed in the future and fetch them speculatively using the instruction-fetching engine. In this way, we can correctly model the execution of a program as well as the speculative behavior of prefetching.

6.3.2 Description of benchmarks

To assess the performance of the BP-based prefetching, we used the SPEC CINT95 benchmarks [SPEC95] to measure its performance against other prefetching schemes. However, some of the benchmarks in SPEC CINT95 have very small instruction

SPEC CINT 95 benchmarks			
Benchmarks	conditional branches	prediction accuracy	total instructions
gcc	26,521,090	92.18%	191,548,351
go	17,873,434	84.15%	136,898,927
li	25,008,567	93.45%	248,490,436
perl	39,714,631	96.61%	365,938,737
vortex	27,792,013	98.72%	282,462,328

Table 6.1: Statistics of the benchmarks used

Input to the SPEC95 benchmarks was a reduced input data set; each benchmark was run to completion.

footprints, such as *compress* and *ijpeg*, hence, we exclude these benchmarks because they hardly miss at all for the cache sizes examined. The statistics of traces from the SPEC CINT95 are summarized in Table 6.1.

6.3.3 Hardware assumption

For the execution engine, we assume instruction fetching is the only source of stalls. We simulated a 4-issue machine and the instruction fetching stops only at the boundary of a cache line or a branch. Under this model, all instructions are executed within one cycle after they are fetched. This simplified assumption adds more pressure to the instruction prefetching.

For the memory system, we assume 1 cycle access time for a level-1 instruction cache hit, and 6 cycles for a miss. We also assume a perfect level-2 instruction cache, so it will always have the instructions needed.

The branch predictor used in all the following simulation is a variation of two-level dynamic branch predictor, *gshare* [McFarling92]. We have selected the *gshare* branch predictor as the basis of comparison, because it has been shown, in general, to perform better than other variations of two-level branch predictors due to lower aliasing rate. In *gshare*, the global history is XORed together with the low-order address bits of a branch to

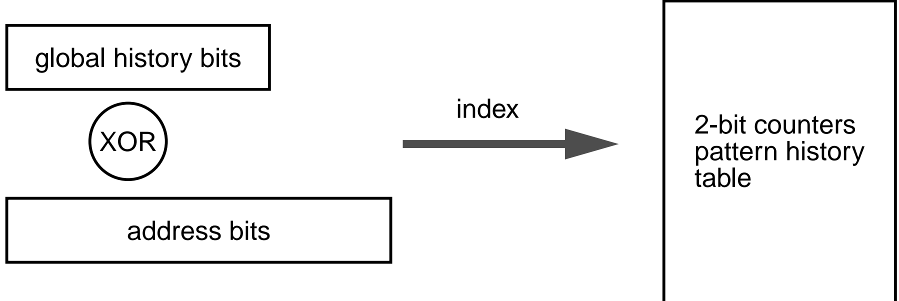


Figure 6.3: A *gshare* branch predictor

form an index, as shown in Figure 6.3. This index is then used to select a 2-bit saturating up-down counter from a pattern-history table to make a prediction. Depending on the sign bit of the selected 2-bit counter, the branch is either predicted as taken or not taken. The configuration used in our simulation has 15 address bits and 9 global history bits. This predictor has a hardware cost of about 8K bytes of storage.

6.3.4 Bus arbitration policy

We assume the bus to the level-2 cache can only take one request per cycle, so a bus arbitration policy is needed between the execution engine and the prefetching unit. All requests are serviced in a prioritized order: First, requests from the execution engine are serviced, since these requests result from cache misses and directly affect the total execution time. Second, prefetches based on branch prediction are serviced, because they are more accurate than sequential prefetches. Finally, sequential prefetches are serviced when neither of the above is present, otherwise they are postponed and stored in a first-in-first-out queue.

To avoid redundant bus traffic, all requests are compared against requests in transit on the bus, and any duplicated requests are canceled.

6.4 Simulation results and analysis

In this section, we compare our scheme with the best of next-*n* line prefetching scheme, and wrong path prefetching. For the next-*n* line scheme, we examined the performance of next 1 to 4 lines, and selected the best configuration, next-2 line, as representative. We also plotted the best configuration for BP-based prefetching, the basic BP-based scheme (a look ahead program counter) plus the sequential next-2 lines of the look ahead PC, denoted as BP-2 in the legends.

The metric we use to measure the performance is the total execution time in cycles. To measure how much improvement can be achieved, we further compute the percentage of stall time (stall overhead), as follows:

$$\text{stall overhead \%} = \frac{\text{total execution time} - \text{perfect execution time}}{\text{perfect execution time}} \times 100$$

Here the perfect execution time assumes a perfect cache with zero miss rate. Therefore, this perfect execution time is the best possible lower-bound we can ever achieve.

In Figure 6.4 to Figure 6.8, we compare the performance of the best configuration of next-*n* line prefetching, wrong path prefetching, and the best configuration of BP-based prefetching.

Figure 6.4 shows that BP-based prefetching outperforms both next-*2* line prefetching and wrong path prefetching in all benchmarks and cache sizes examined. The *y*-axis indicates the stall overhead, hence a lower bar indicates a better scheme. Averaging across benchmarks, BP-based prefetching is better by a factor of 17-32% in stall overhead than next-2 line prefetching, and by a factor of 34-44% than wrong-path prefetching.

To study the sources of improvement, we measure the number of prefetches generated per 100 instructions. In Figure 6.5, two cache configurations are shown for each benchmark: small (4K) and large (16K). Each bar indicates the total prefetches generated by each scheme, and these prefetches are further classified into three categories. Helpful

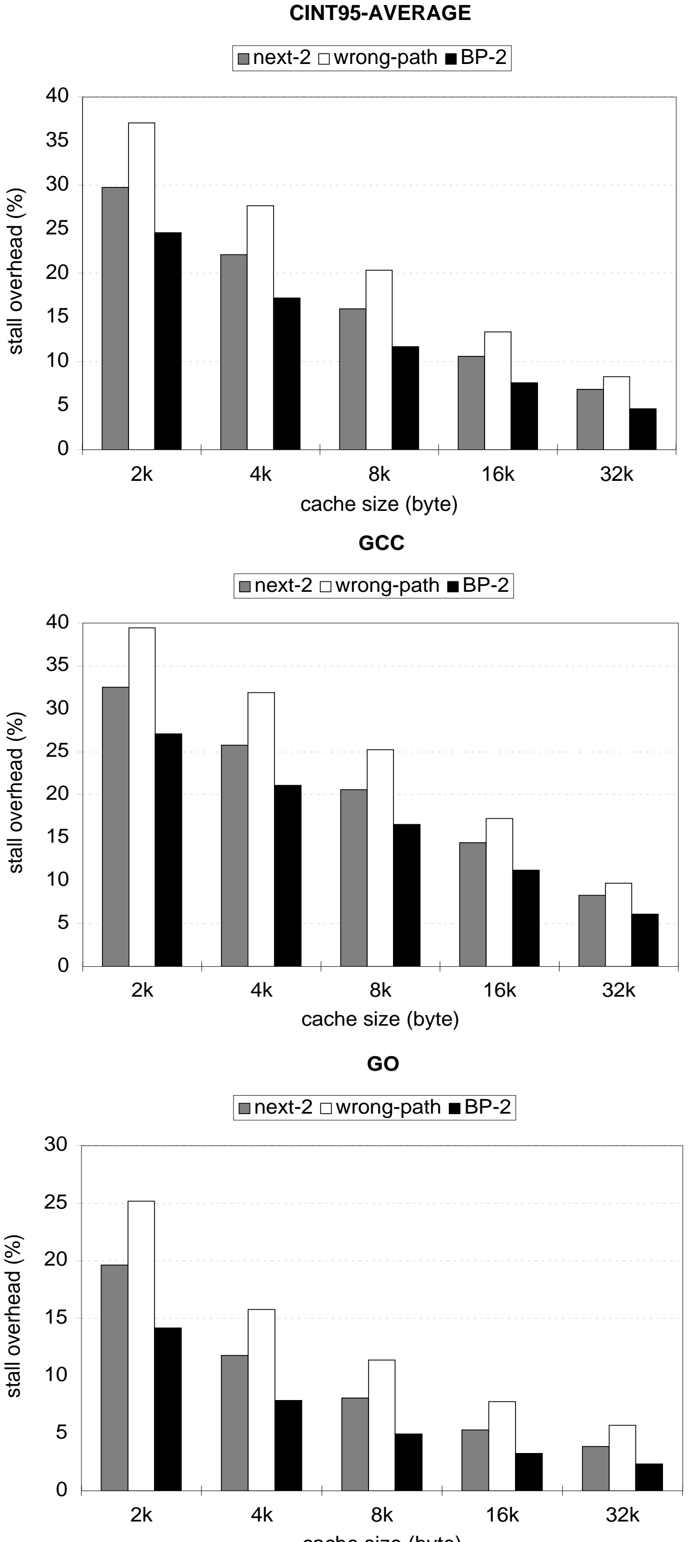


Figure 6.4: Performance measure: stall overhead for different schemes

Stall overhead measures the extra execution time needed over a perfect cache with zero miss rate.

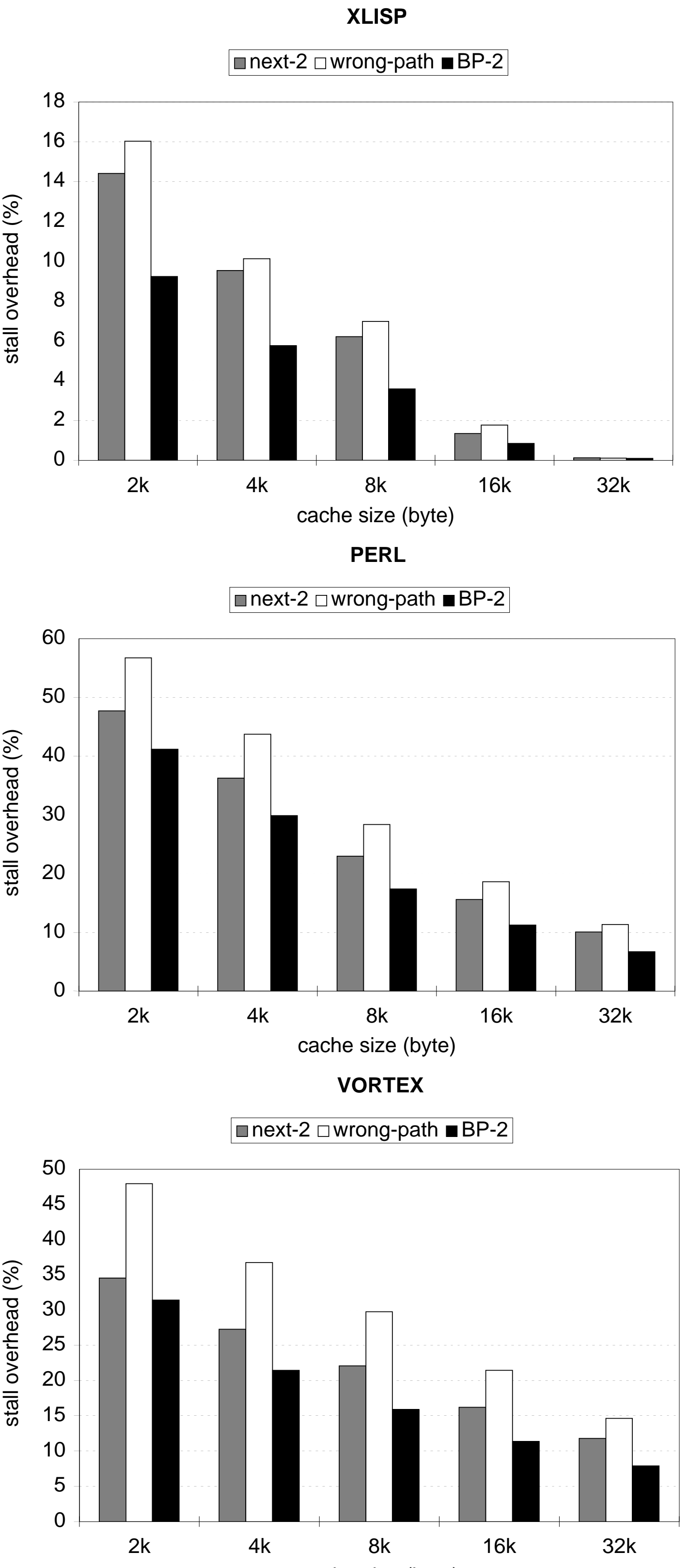


Figure 6.4: *(Continued)*

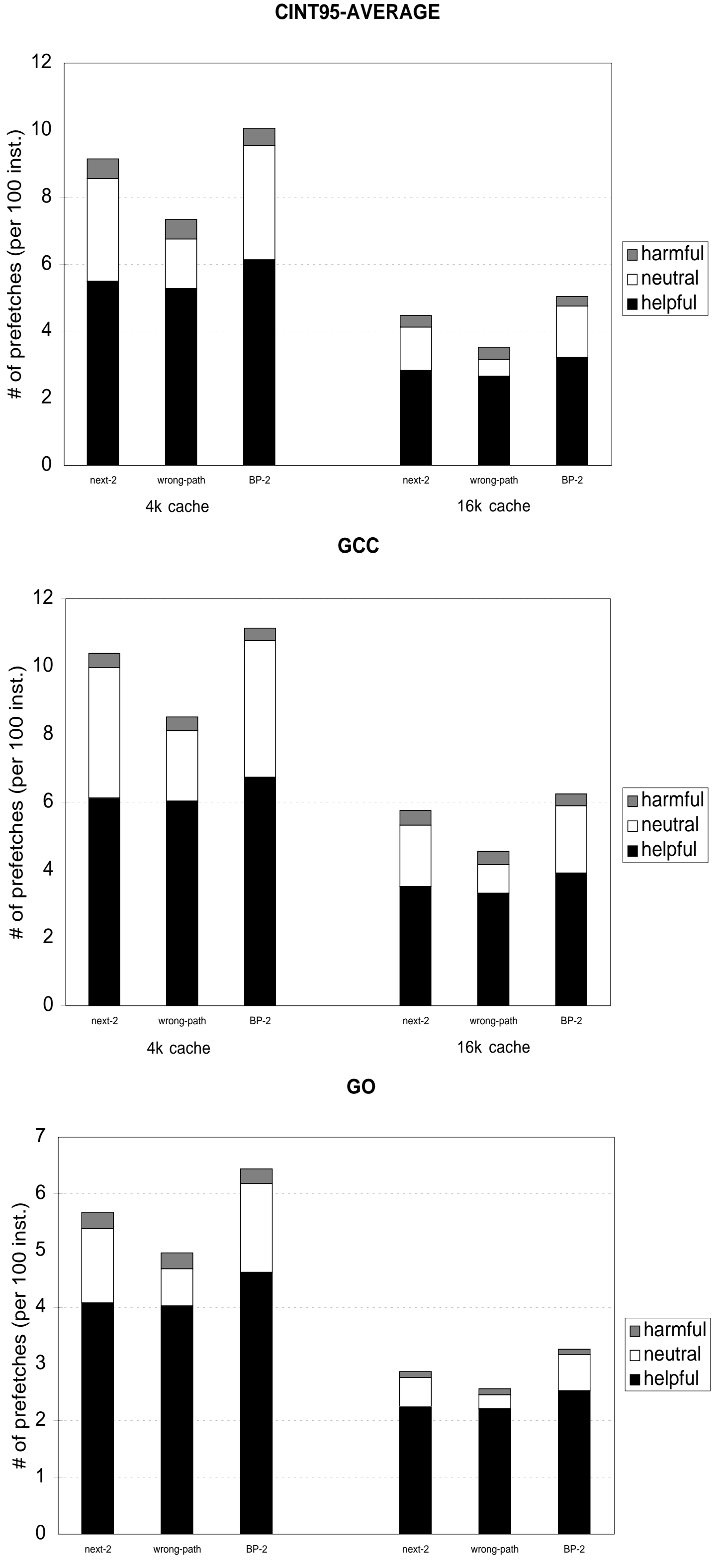


Figure 6.5: Total prefetches generated in each scheme and the classification of these prefetches

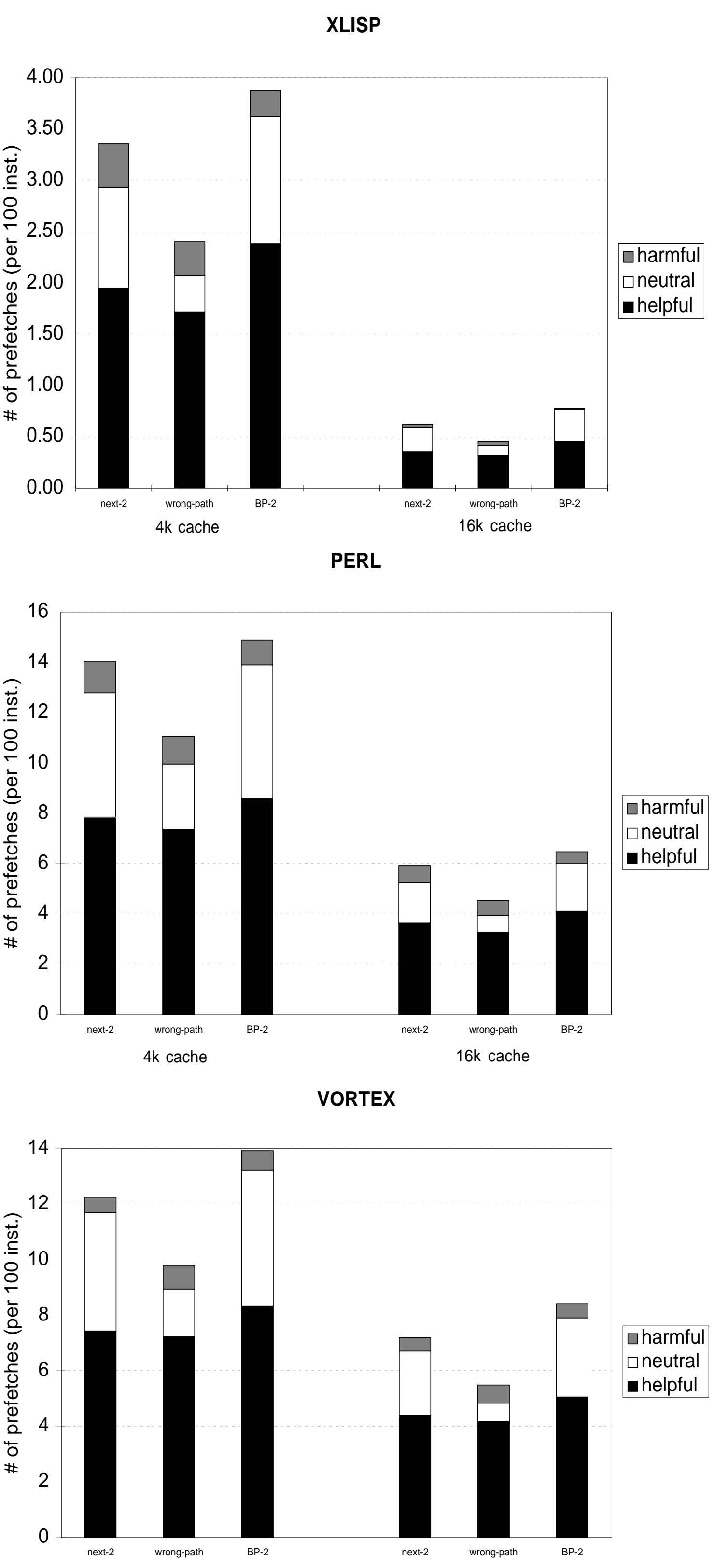


Figure 6.5: *(Continued)*

prefetches (shown in black) are the prefetches that are actually used by the program and, hence, improve the total execution time. Neutral prefetches (shown in white) are the prefetches that are not used by the program, yet they do not cause any harmful effects either. These neutral prefetches occur when prefetched instructions replace stale instructions that are no longer needed by the program and will soon be replaced by new instructions anyway. Harmful prefetches (shown in gray) are the ones that replace useful instructions (pollute the cache), and, hence, lower the performance. These harmful prefetches cause misses that would not occur in a cache without prefetching.

In Figure 6.5, we can see that BP-based prefetching has more helpful and more total prefetches than other schemes. These extra helpful prefetches improve the total execution time. Also note that the portion of harmful prefetches in BP-based prefetching is slightly smaller than other schemes, and this fact helps to improve the execution time too.

To analyze the nature of useful prefetches, we further classify the useful prefetches into two categories: prefetches causing hits, and prefetches reducing miss penalties, as shown in Figure 6.6. The prefetches causing hits are prefetches early enough such that instructions are already in the cache by the time the program needs them (hit in cache). These prefetches completely reduce the penalty to zero. On the other hand, the prefetches reducing miss penalties are also correct prefetches, but they are not generated early enough. By the time the program needs the instructions, these prefetches have not brought the instructions into the cache yet (hit in transfer). Therefore, there are still some penalties associated with these prefetches, but the penalties are smaller than normal cache misses.

As shown in Figure 6.6, BP-based prefetching has more prefetches causing hits than other schemes (hit in cache, shown in black). This means that BP-based prefetching is able to generate useful prefetches earlier, in addition to generating more useful prefetches, leading to better performance.

Figure 6.7 shows the impact of prefetching on the overall memory traffic on the bus to level-2 cache. The *y*-axis represents the number of requests sent through the bus per

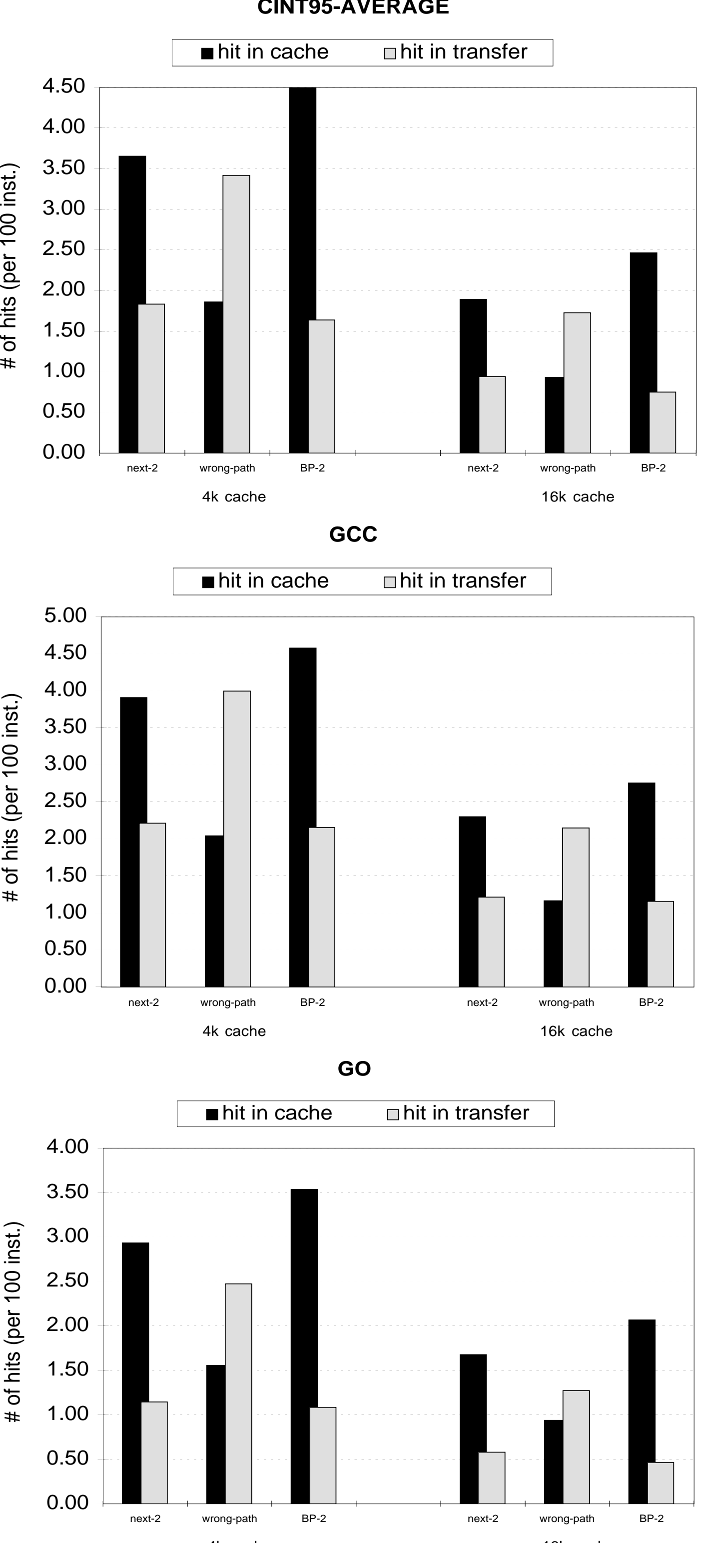


Figure 6.6: Further classification of useful prefetches

Early prefetches generate hits in the cache, and late prefetches generate hits being transferred.

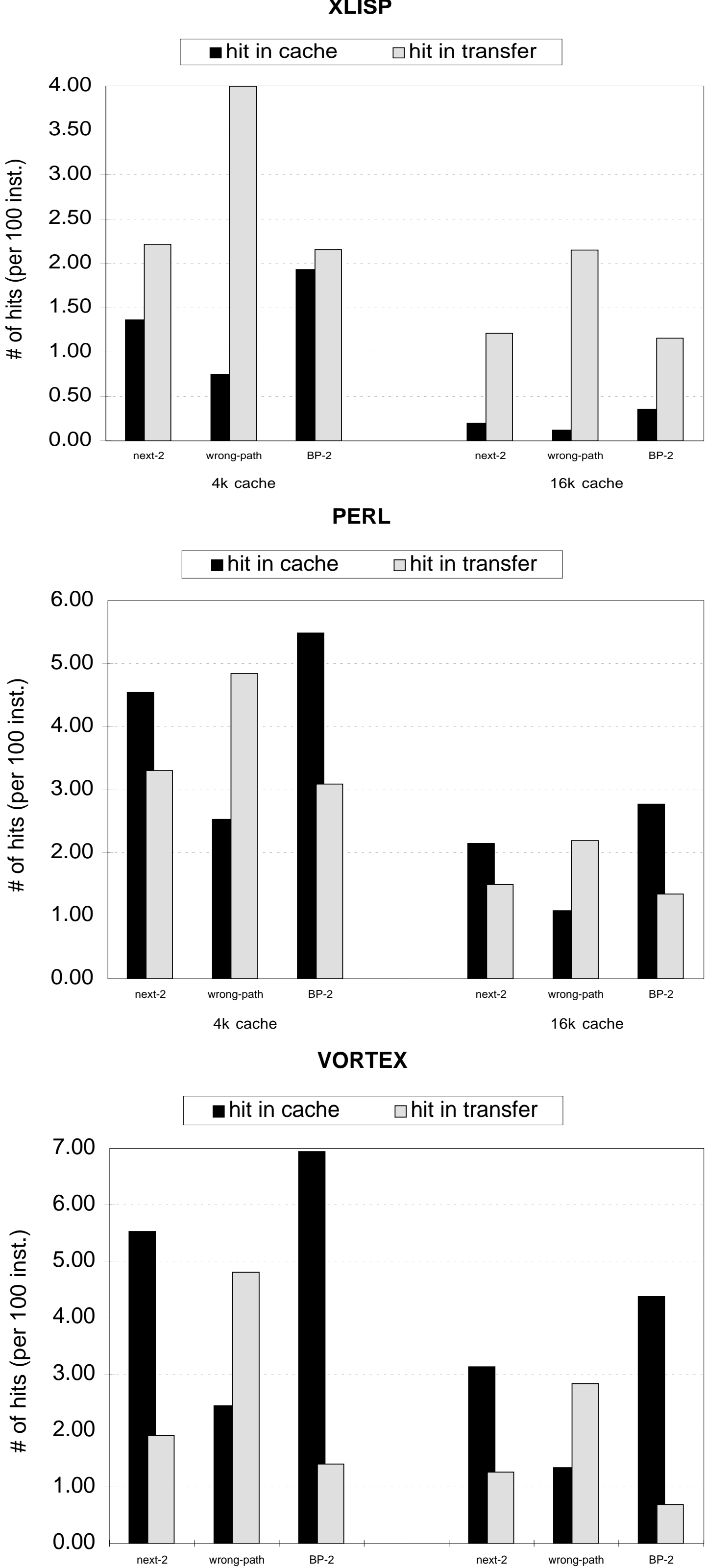


Figure 6.6: *(Continued)*

CINT95-AVERAGE

next-2 wrong-path BP-2

bus traffic (# of requests per 100 inst.)

cache size (byte)

2k 4k 8k 16k 32k

GCC

next-2 wrong-path BP-2

bus traffic (# of requests per 100 inst.)

cache size (byte)

2k 4k 8k 16k 32k

GO

next-2 wrong-path BP-2

bus traffic (# of requests per 100 inst.)

cache size (byte)

2k 4k 8k 16k 32k

Figure 6.7: Bus traffic for different schemes.

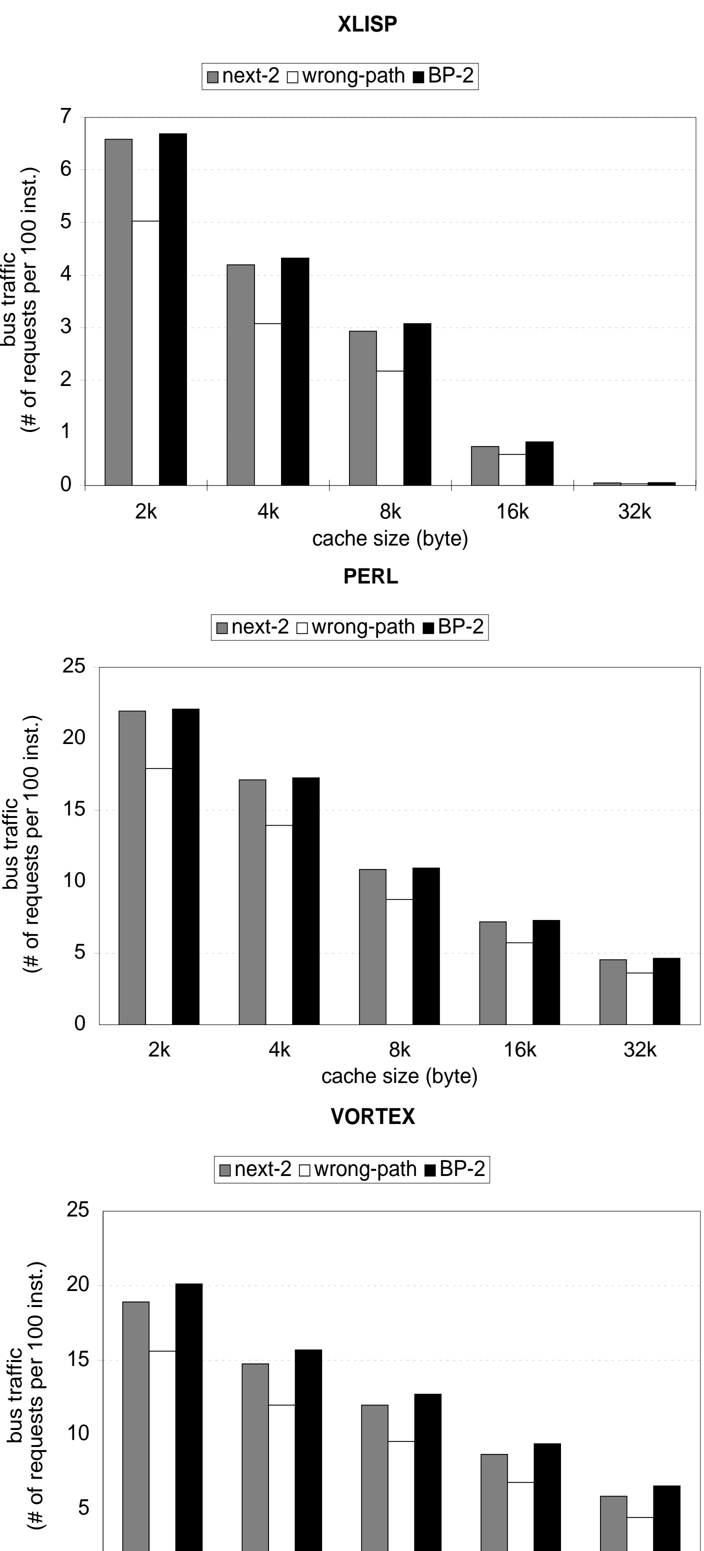


Figure 6.7: *(Continued)*

100 instructions (including the requests for misses generated by the execution unit), so a lower bar indicates less traffic. Even though BP-based prefetching generates more prefetches, the total bus traffic is only slightly higher than next-2 line prefetching. This is because most of the prefetches are helpful in BP-based prefetching and, thus, reduce the miss requests generated by the execution unit. Therefore, BP-based prefetching is very economical in bus traffic by being selective and accurate.

Figure 6.8 shows the utilization of the bus to the level-2 cache. The *y*-axis represents the percentage of bus utilization, which is computed as: (total bus busy cycles)/(total execution cycles). Since the total execution time for BP-based prefetching is shorter than other schemes, so the percentage looks slightly higher compared to the data shown in Figure 6.7. Averaging across benchmarks, the bus utilization of BP-based prefetching is about 28%.

Finally, to get the big picture on the performance of BP-based prefetching, the total execution time of BP-based prefetching is compared with a plain cache without prefetching. As shown in Figure 6.9, a cache with BP-based prefetching achieves lower execution time than a cache of 4 times the size. In particular, a 2K BP-based prefetched cache even outperforms a 16K cache without prefetching.

To gain further understanding, we compare BP-based prefetching scheme with other schemes. Unlike table-based schemes, the BP-based prefetching scheme is able to reduce the first time compulsory misses by pre-computing target addresses. This ability to independently compute target addresses eliminates the need for expensive tables and the awkward situation of having no initial history information the first time. Furthermore, using advanced dynamic branch predictors offers much more accurate target-prediction than simple table-based schemes. Therefore, BP-based prefetching, like wrong-path prefetching, is more effective than table-based prefetching schemes.

By reducing stalls on taken branches, BP-based prefetching also outperforms wrong-path prefetching. Our prefetching scheme runs ahead of the real program counter, allowing it to compute and fetch targets in advance, even before they are requested by the

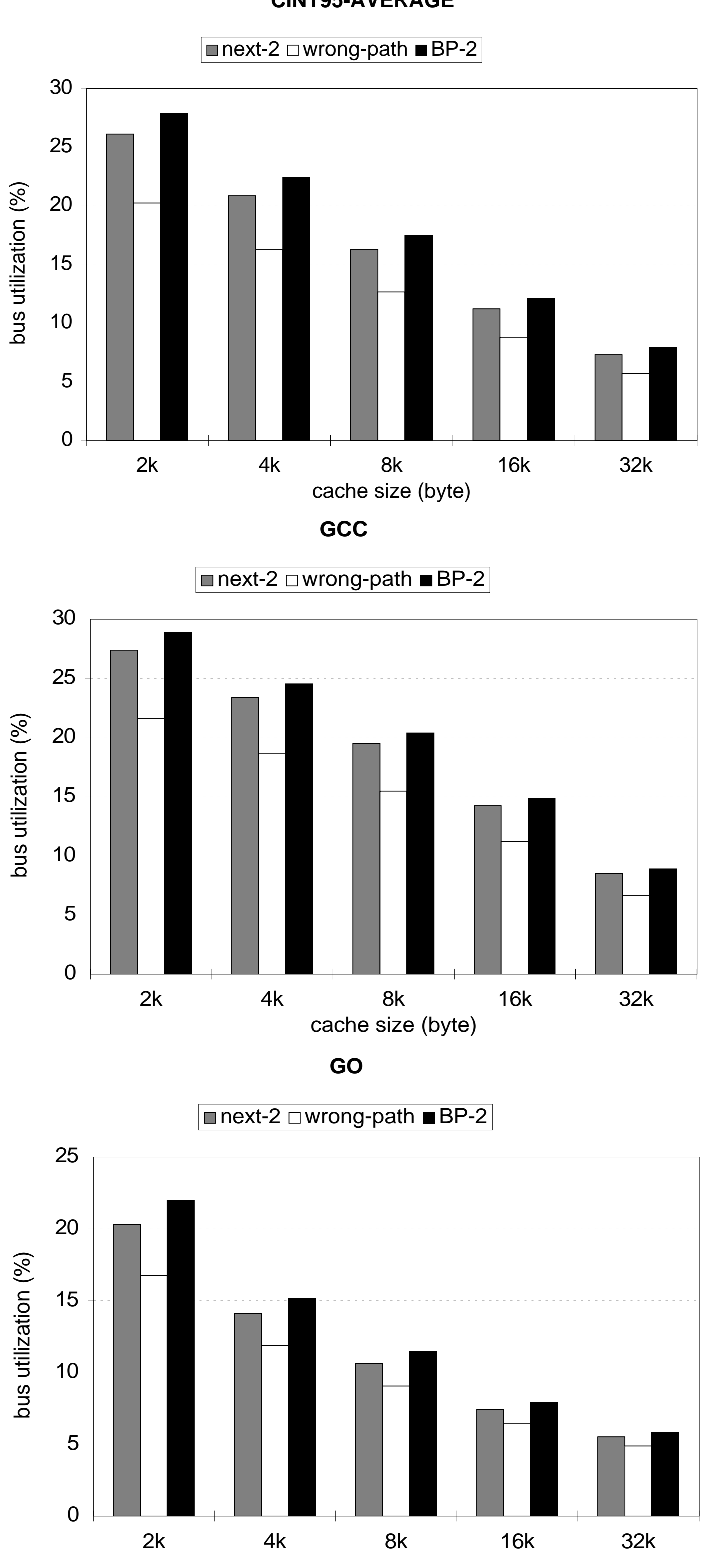


Figure 6.8: Percentage of utilization for the bus to level-2 cache for different schemes.

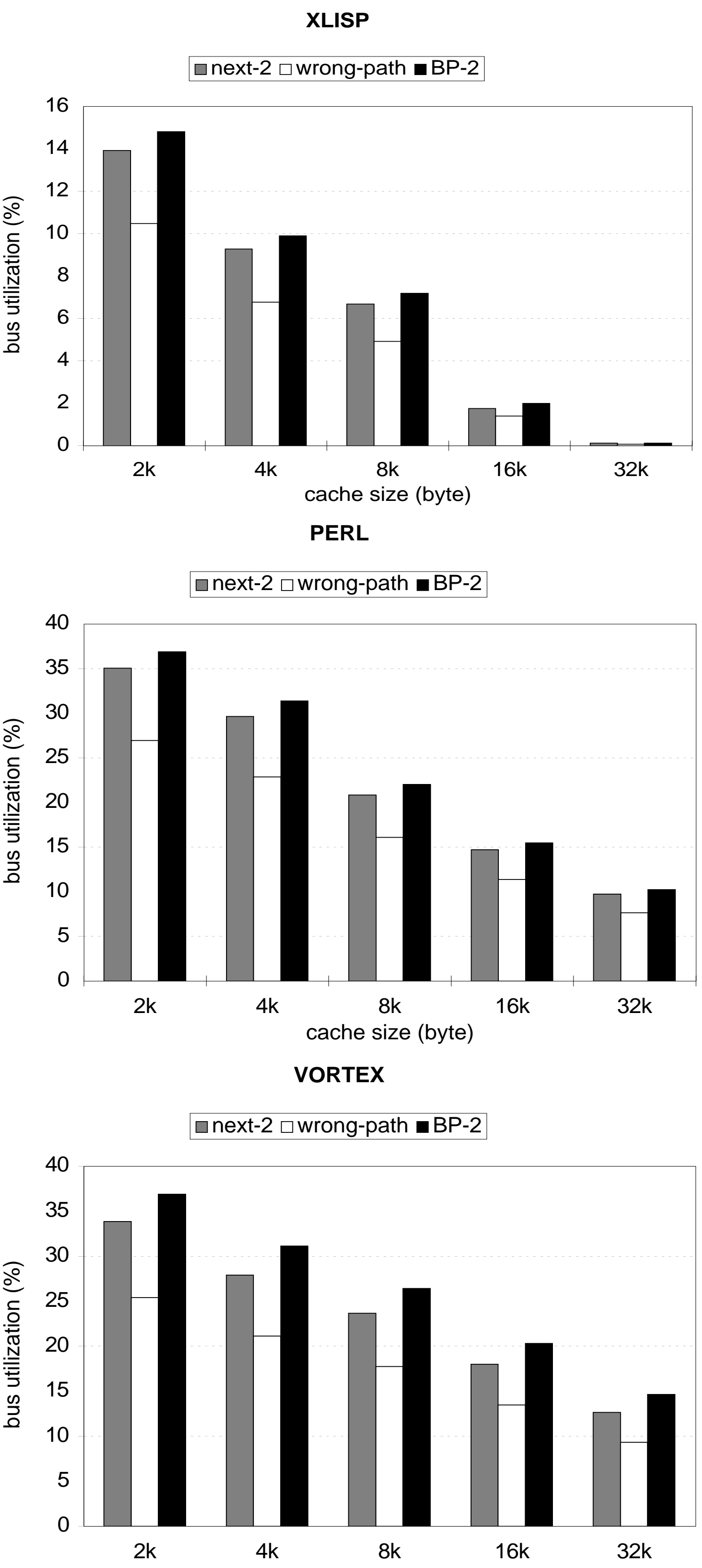


Figure 6.8: *(Continued)*

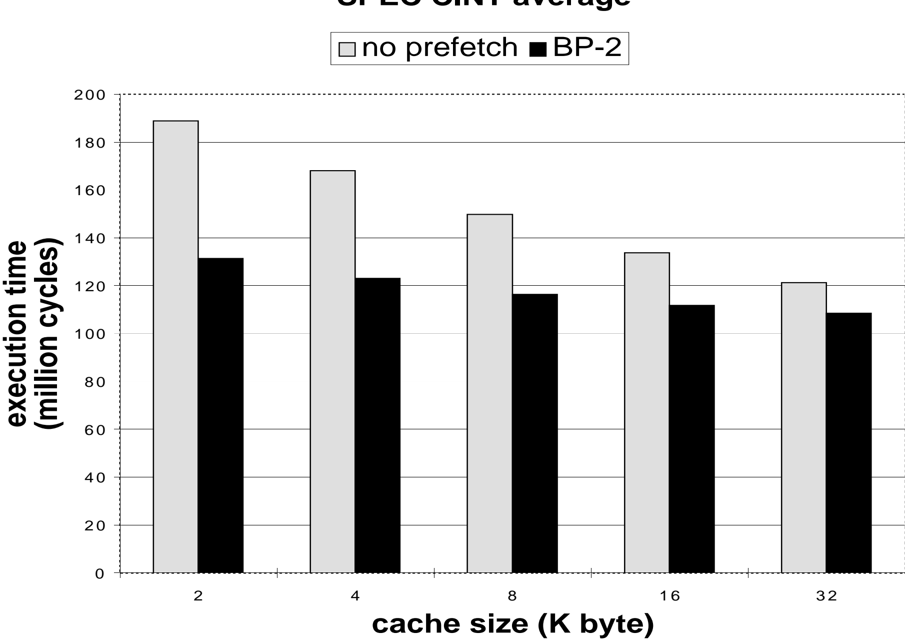


Figure 6.9: A cache with BP-based prefetching achieves lower execution time than a plain cache of 4 times the size

---

execution unit. In contrast, wrong-path prefetching calculates target addresses at decode stage, too late to produce any useful prefetches when branches are actually taken. In addition, BP-based prefetching executes deeper down the speculative path than wrong-path prefetching, and this deep speculation helps to reduce the transfer latency.

From the above analysis, we can see the benefit of BP-based prefetching lies in the ability to run ahead of the real program counter. BP-based prefetching can run ahead of the execution unit because the average length of basic blocks is more than 4 instructions. It can advance almost one basic block per cycle (if it is pipelined), while the execution unit can advance at most 4 instructions per cycle.

6.5 Discussion of implementation issues

Although conceptually we consider the prefetching unit as a separate piece of hardware, we can implement the prefetching unit with the existing fetching unit in an

actual hardware implementation. We can implement our prefetching by keeping two modes in the fetching unit: a regular fetching mode and a prefetching mode. In the regular fetching mode, instructions are fetched into the instruction buffer just like normal execution, while in the prefetching mode, instructions are only fetched into a temporary buffer. Note that cache misses are also serviced in the prefetching modes in order to prefetch instructions into the cache.

This combined implementation works as follows. During the regular fetching mode, if the fetching unit is stalled (either because the instruction buffer is full or the number of outstanding branches exceeds a maximum limit), the fetching unit switches to the prefetching mode and continues fetching aggressively. In the prefetching mode, as instructions are consumed by the execution engine, instructions are move from the temporary buffer into the instruction buffer. During the prefetching mode, the fetching unit switches back to the fetching mode when a branch misprediction occurs or when both the instruction buffer and temporary buffer are empty (the execution unit catches up).

This implementation works because the fetching unit and the "conceptual prefetching unit" do not actively fetch new instructions from the cache at the same time, only one of them is active at any moment. Therefore, we can combine both of them into the same fetching unit. More specifically, if the "conceptual prefetching unit" does not make a misprediction and stays on the correct path, the fetching unit does not need to fetch from the cache but simply follows the path of the conceptual prefetching unit. On the other hand, if the conceptual prefetching unit makes a misprediction (goes into a wrong path), this misprediction will be detected when the branch is resolved. In this case, the look ahead program counter of the conceptual prefetching unit is reset and synchronized with the program counter of the fetching unit. This combined implementation minimizes the hardware requirement and eliminates the need of additional ports to the branch predictor and the instruction cache.

Another key feature of our prefetching unit is the ability to find the earliest

instruction and predict the target in one cycle. A possible hardware implementation is illustrated in Figure 6.10 and is explained as follows.

Figure 6.10 shows a possible implementation of the decoding logic for an 8-instruction cache line (the implementation for longer cache line size can be extended similarly). To accelerate the decoding speed, the instructions are subdivided into two groups of four, similar to the principle of a carry-look-ahead adder. For simplicity, we only show one group in Figure 6.10. Each instruction has a predecoded bit to indicate whether it is a branch or not. These bits go through a priority encoder to determine the earliest branch (or the first non-zero position). The output from the priority encoder is then used to form the line offset and to select the displacement from the earliest branch instruction. Then, these two numbers are added to generate the target address.

In the meantime, the direction of the branch can be predicted in parallel. The starting address of this cache line is known before decoding, and this address is XORed with the global history to generate the row index to access the branch pattern history table in the *gshare* predictor. Using this row index, a row of 8 counters is selected. Note that the column index is being generated in parallel by the priority encoder, and should be available before the output from the row of counters is valid. Then this column index is used to select the right counter from the row and to predict the direction of the branch.

Finally, depending on the type of the branch, the target is either selected from the adder or from a return address stack.

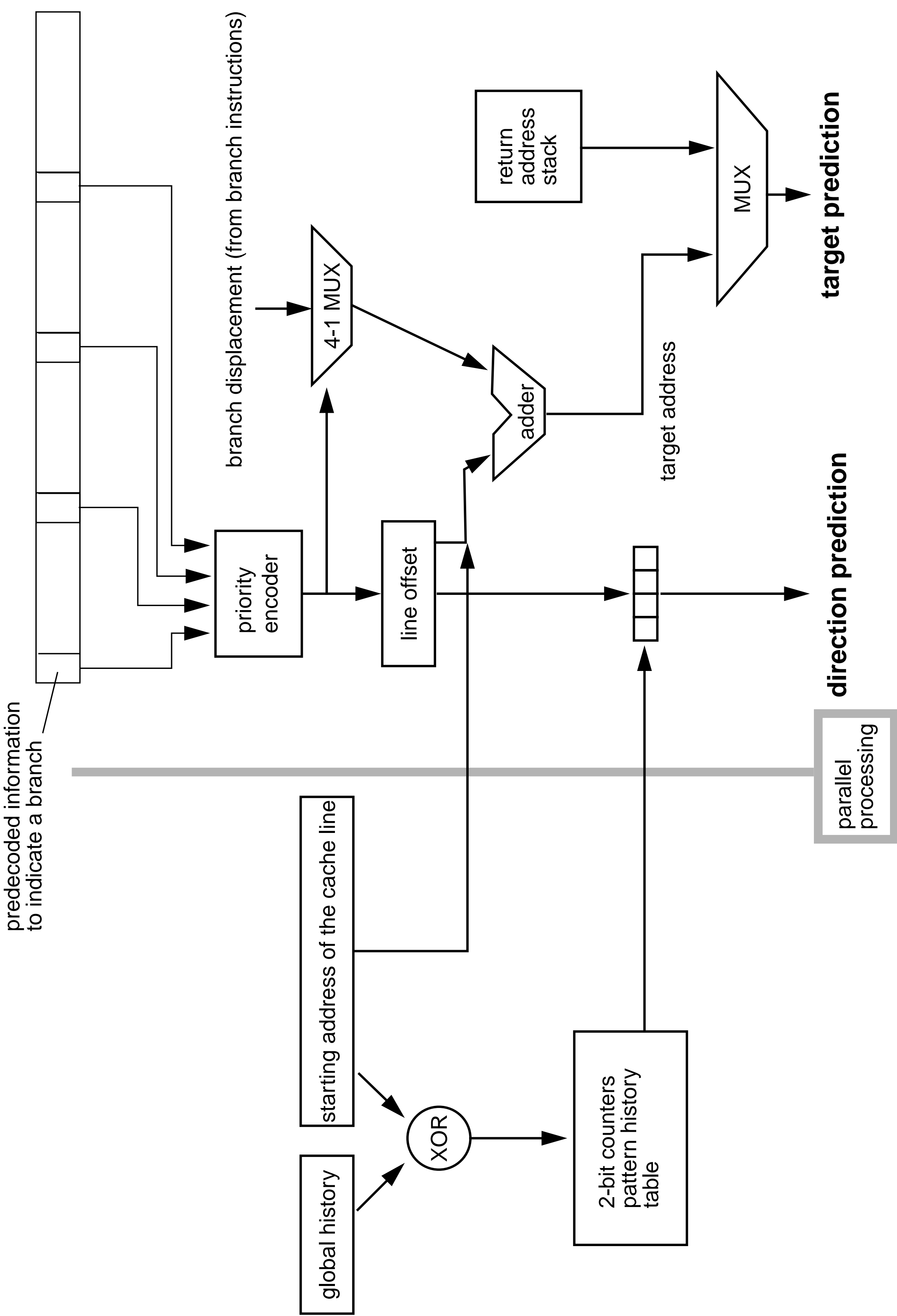


Figure 6.10: A possible implementation of branch prediction-based prefetching

CHAPTER 7
CONCLUSIONS

This dissertation has shown that there are a number of opportunities to apply data compression to improve various important issues for fast instruction fetching: branch prediction, instruction fetching bandwidth, and instruction-cache performance. Then using these as basic building blocks, a novel prefetching scheme, branch prediction-based prefetching, was proposed to further improve the performance of microprocessors.

In Chapter 2, we established the connection between data compression and branch prediction. This provides a new global perspective for branch prediction, and allows us to draw techniques from data compression to form a theoretical basis for branch prediction. In particular, we have shown that current two-level adaptive branch predictors are approximations of an optimal predictor, PPM. This theoretical basis, rather than just simulation data, can provide us a high degree of confidence in the performance of two-level predictors. In addition, an improvement in branch prediction using PPM was proposed that reduces the effects of transients due to cold starts and context switches.

A conceptual model of branch prediction was introduced, which consists of three components: a predictor, an information processor, and a source. For the predictor, we can borrow the rich set of predictors developed in data compression and apply them to branch prediction. However, since PPM is optimal, it is unlikely that significant improvement can be made by improving the predictor alone. To further increase branch prediction accuracy, the focus should be on improving the information processor and the source.

In Chapter 3, we further examine the performance of another widely used optimal compression algorithm, Lempel-Ziv. Although the Lempel-Ziv algorithm is very popular in the field of data compression and it also provides reasonable branch prediction

accuracy, its adaptive structure makes it hard for efficient hardware implementation. Based on our preliminary simulation results, Lempel-Ziv predictors may not be as cost-effective as two-level branch predictors. However, for a given particular technology and budget, more extensive experiments may be needed to assess its final effectiveness.

To calibrate the performance of various prediction schemes, we showed that, for some simple programs, the theoretical branch predictability can be analyzed using approaches found in the concrete analysis of algorithms. To illustrate this point, we analyzed Quicksort program and demonstrated that its limit of branch predictability is 75%. We then use this provable limit to evaluate the performance of various branch prediction schemes. We found that PPM can best approach this limit, followed by the two-level branch predictor, the Lempel-Ziv predictor, and finally the one-bit counter.

Although optimal predictors can be derived from data compression, efficient hardware implementation of branch predictors varies with technology and still needs careful analysis. In Chapter 4, we used per-address two-level branch predictors to illustrate how a comprehensive analysis can be done. As the clock rate increases, we argued that tagless per-address predictors may be more attractive than tagged predictors because of faster access time, lower power, and simpler implementation. In addition, by removing the tag, tagless predictors allow more resources to be allocated to the predictor and BTB, and also allow these two components to be optimized individually. We further showed that tagless predictors outperform direct-mapped tagged predictors due to better accuracy in transitional states.

Using equal-cost as criteria, we evaluated cost and performance trade-off across a wide range of the design space. Based on our simulation results, we notice that the number of address bits indexing into the second level table is the most important parameter when the available budget is small. However, the importance of address bits quickly diminishes as the budget increases. With a larger budget, history bits and the number of branch history entries should increase accordingly, but the number of address bits should be reduced.

We concluded with a set of design principles for tagless per-address two-level predictors. First, we can measure the statistics of target benchmarks, which include the number of static branches and the misprediction rate for a base configuration. Then, we compare these statistics with those from IBS and SPEC. The quantitative data collected from IBS and SPEC can provide a rough idea of how an optimal implementation should be. Finally, we can fine-tune the parameters using our principles.

In Chapter 5, we further applied data compression to alleviate the bottleneck of instruction stream fetching. We have introduced a technique to reduce instruction stream by compressing frequently encountered instruction sequences into single byte opcodes.

Despite the fact that the compiler was not tuned to exploit instruction compression, we were able to reduce both the I-cache byte fetch requirements and the I-cache miss rates for the SPEC benchmarks. The average bytes needed from level-1 cache were reduced by 50% for integer benchmarks, and 70% for floating point benchmarks. The average bus cycles needed to fetch instructions from level-1 cache were reduced by 35% for integer benchmarks, and 65% for floating point benchmarks. And a compression enhanced cache has a lower miss rate than a plain cache twice the size.

The impact of instruction stream compression should be higher when we incorporate this optimization technique into the compiler. This technique could be integrated either by rebuilding the code generation phase, or by performing a peephole optimization pass over the generated instructions. Since our patterns are strictly based upon the bit patterns, the varied use of registers by a compiler has a negative impact upon pattern incidence counts. And yet, although using only a very small register set size will increase pattern incidences, it will likely have an adverse effect on overall program execution. It is important, therefore, to identify strategies that both lead to high performance code sequences, and aid in the generation of compact code. Several possible strategies were discussed.

In Chapter 6, to further reduce cache miss rates, we presented an effective

instruction prefetching scheme, branch prediction-based (BP-based) prefetching, which applies branch prediction information to speculatively fetch instructions into the instruction cache. We should note that BP-based prefetching can achieve higher performance than a cache of 4 times the size. Examining other hardware instruction prefetching schemes, we see that BP-based prefetching is better by a factor of 17-32% in stall overhead compared to the best next-*n* line prefetching, and by a factor of 34-44% compared to wrong-path prefetching. BP-based prefetching is able to generate more useful prefetches than other schemes and generate them earlier. In addition, these prefetches are generated selectively, thus, the bus traffic and utilization are very close to the best next-*n* line prefetching.

In conclusion, we expect that ideas from compression and prediction will continue to provide new and interesting insights into, and solutions for, problems in computer systems and computer architecture in particular.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Aho86] Aho, A., Sethi, R. and Ullman, J. *Compiler: Principles, Techniques and Tools.* Addison-Wesley, 1986.

[Bell90] Bell, T. C., Cleary, J. G. and Witten I. H. *Text Compression.* Englewood Cliffs, NJ: Prentice-Hall, 1990.

[Bird96] Bird, P. and Mudge, T. An Instruction Stream Compression Technique. *Technical Report CSE-TR-319-96*, EECS Department, University of Michigan, November 1996.

[Calder94a] Calder, B. and Grunwald, D. Reducing branch costs via branch alignment. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1994, pp. 242-251.

[Calder94b] Calder, B. and Grunwald, Dirk. Fast & accurate instruction fetch and branch prediction. *Proceedings of the 21th International Symposium on Computer Architecture*, April 1994, pp. 2-11.

[Chang94] Chang, P., Hao, E., Yeh, T. and Patt, Y. Branch classification: a new mechanism for improving branch predictor performance. *Proceedings of the 27th Annual International Symposium on Microarchitecture*, November 1994, pp. 22-31.

[Chen95] Chen, T.-F. and Baer, J.-L. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, Vol. 44, No. 5, May, 1995, pp. 609-623.

[Chen96a] Chen, I-C. K., Coffey, J. and Mudge, T. Analysis of branch prediction via data compression. *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1996, pp. 128-137.

[Chen96b] Chen, I-C. K., Lee, C-C., Postiff, M. and Mudge, T. Tagless two-level branch prediction schemes. *Technical Report CSE-TR-306-96*, University of Michigan, 1996.

[Chen97a] Chen, I-C. K., Lee, C-C. and Mudge, T. Instruction prefetching using branch prediction information. *International Conference on Computer Design*, October 1997.

[Chen97b] Chen, I-C. K., Lee, C-C., Postiff, M. and Mudge, T. Design optimization for high-speed per-address two-level branch predictors. *International Conference on Computer Design*, October 1997.

[Chen97c] Chen, I-C. K., Bird, P. and Mudge, T. The impact of instruction compression on I-cache performance. *Technical Report CSE-TR-330-97*, University of Michigan, 1997.

[Cleary84] Cleary, J. G. and Witten, I. H. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, Vol. 32, No. 4, April 1984, pp. 396-402.

[Conte96] Conte, T. M., Sathaye, S W. and Banerjia, S. A persistent rescheduled-page cache for low overhead object code compatibility in VLIW architecture. *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996, pp. 4-13.

[Curewitz93] Curewitz K. M., Krishnan, P. and Vitter, J. S. Practical prefetching via data compression. *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, May 1993, pp. 257-266.

[Ellis85] Ellis, J. *Bulldog: A Compiler for VLIW Architectures*. ACM Distinguished Dissertation, MIT Press, 1985.

[Eustace95] Eustace, A. and Srivastava, A. ATOM: A flexible interface for building high performance program analysis tools. *Proceedings of the Winter 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems*, January 1995, pp. 303-314.

[Hennessy96] Hennessy, J. L. and Patterson, D. A. *Computer architecture a qualititative approach.* 2nd ed. San Francisco, CA: Morgan Kaufmann Publishers Inc. 1996.

[Kaeli91] Kaeli, D. and Emma, P. G. Branch history table prediction of moving target branches due to subroutine returns. *Proceedings of the 18th International Symposium on Computer Architecture*, May 1991, pp. 34-41.

[Krishnan94] Krishnan, P. and Vitter, J. S. Optimal prediction for prefetching in the worst case. *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1994, pp. 392-401.

[Kroeger96] Kroeger, T. M. and Long, D. D. E. Predicting file system actions from prior events. *Proceedings of USENIX Winter Technical Conference*, January 1996.

[Lee84] Lee, J.K.F. and Smith, A. J. Branch prediction strategies and branch target buffer design. *IEEE Computer*, Vol. 21, No. 7, January 1984, pp. 6-22.

[Lefurgy97] Lefurgy, C., Bird, P., Chen, I-C. and Mudge, T. Improving code density using compression technique. *Technical Report CSE-TR-342-97*, University of Michigan, 1997.

[Liu96] Liu, Y. and Kaeli, D. R. Branch-directed and stride-based data cache prefetching. *Proceedings of the International Conference on Computer Design*, October, 1996, pp. 225-230.

[McFarling93] McFarling, S. Combining branch predictors. *WRL Technical Note TN-36*, June 1993.

[Moffat90] Moffat, A. Implementing the PPM data compression scheme. *IEEE Transactions on Communications*, Vol. 38, No. 11, November 1990, pp. 1917-1921.

[MReport95a] Intel's P6 uses decoupled superscalar design. *Microprocessor Report,* Sbastopol, CA: MicroDesign Resources, February 1995, pp. 9-15.

[MReport95b] New algorithm improves branch prediction. *Microprocessor Report,* Sebastopol, CA: MicroDesign Resources, March 1995, pp. 17-21.

[MReport95c] Nx686 goes toe-to-toe with Pentium Pro. *Microprocessor Report,* Sbatopol, CA: MicroDesign Resources, October 1995, pp. 1-10.

[MReport96] Digital 21264 sets new standard. *Microprocessor Report,* Sebastopol, CA: MicroDesign Resources, October 1996, pp. 11-16.

[Mudge96] Mudge, T., Chen, I-C. K. and Coffey, J. T. Limits to branch prediction. *Technical Report CSE-TR-282-96*, University of Michigan, 1996.

[Nair95a] Nair, R. Optimal 2-bit branch predictors. *IEEE Transactions on Computers*, Vol. 44, No. 5, May 1995, pp. 698-702.

[Nair95b] Nair, R. Dynamic path-based branch correlation. *Proceedings of the 28th Annual International Symposium on Microarchitecture*, November 1995, pp. 15-23.

[Pan92] Pan, S-T., So, K. and Rahmeh, J. T. Improving the accuracy of dynamic branch prediction using branch correlation. *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, 1992, pp. 76-84.

[Perl96] Perl, S. and Sites, R. Studies of Windows NT performance using dynamic execution traces. *Proceedings of the USENIX 2nd Symposium on Operating Systems Design and Implementation*, October 1996.

[Pierce96] Pierce, J. and Mudge, T. Wrong-path prefetching. *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996, pp. 165-175.

[Pinter96] Pinter, S. S. and Yoaz, A. Tango: a hardware-based data prefetching technique for superscalar processors. *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996, pp. 214-225.

[Ross85] Ross, S. M. *Introduction to probability models*. London, United Kingdom: Academic press, 1985.

[Rotenberg96] Rotenberg, E., Bennett, S. and Smith, J. Trace cache: a low latency approach to high bandwidth instruction fetching. *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996, pp. 24-34.

[Sechrest96] Sechrest, S., Lee, C-C. and Mudge, T. Correlation and aliasing in dynamic branch predictors. *Proceedings of the 23rd International Symposium on Computer Architecture*, May 1996, pp. 22-32.

[Sedgewick92] Sedgewick, R. *Algorithms in C++*. Reading, Massachusetts: Addison-Wesley, 1992.

[Silberschatz94] Silberschatz, A. and Galvin, P. B. Operating system concepts. 4th ed. Massachusetts: Addison-Wesley, 1994.

[Smith81] Smith, J. E. A study of branch prediction strategies. *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981. pp. 135-148.

[Smith82] Smith, A. J. Cache Memories. *Computing Surveys*, Vol. 14, No. 3, 1982, pp. 473-530.

[Smith92] Smith, J. E. and Hsu, W.-C. Prefetching in supercomputer instruction caches. *Proceedings Supercomputing'92,* November 1992, pp. 588-597.

[SPEC95] SPEC CPU'95, *Technical Manual*, August 1995.

[Uhlig95a] Uhlig, R., Nagle, D., Mudge, T., Sechrest, S. and Emer, J. Instruction Fetching: Coping with Code Bloat. *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 345-356.

[Uhlig95b] Uhlig, R. Trap-Driven Memory Simulation, *Ph.D dissertation*. EECS Department, University of Michigan, Ann Arbor, MI, 1995

[Vitter91] Vitter, J. S. and Krishnan, P. Optimal prefetching via data compression. *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science*, October 1991, pp. 121-130.

[Witten94] Witten I. H., Moffat, A. and Bell T. C. *Managing Gigabytes*. New York, NY: Van Nostrand Reinhold, 1994.

[Wolfe92] Wolfe, A. and Chanin, A. Executing Compressed Programs on an Embedded RISC Architecture. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 81-91.

[Yeh91] Yeh, T-Y. and Patt, Y. Two-level adaptive training branch prediction. *Proceedings of the 24th Annual International Symposium on Microarchitecture*, November 1991, pp. 51-61.

[Yeh92a] Yeh, T-Y. and Patt, Y. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. *Proceedings of the 25th Annual International Symposium on Microarchitecture*, December 1992, pp. 129-139.

[Yeh92b] Yeh, T-Y. and Patt, Y. Alternative implementation of Two-Level Adaptive Branch Prediction. *Proceedings of the 19th International Symposium on Computer Architecture*, May 1992, pp. 124-134.

[Yeh93] Yeh, T-Y. and Patt, Y. A comparison of dynamic branch predictors that use two levels of branch history. *Proceedings of the 20th International Symposium on Computer Architecture*, May 1993, pp. 257-266.

[Young94] Young, C. and Smith, M. Improving the accuracy of static branch prediction using branch correlation. *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1994, pp. 232-241.

[Young95] Young, C., Gloy, N. and Smith, M. A comparative analysis of schemes for correlated branch prediction. *Proceedings of the 22nd International Symposium on Computer Architecture*, June 1995, pp. 276-286.