AN ANALYSIS OF VALUE PREDICTIBALITY AND ITS APPLICATION TO A SUPERSCALAR PROCESSOR

by

Yiannakis Thrasou Sazeides

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Electrical Engineering)

at the

UNIVERSITY OF WISCONSIN - MADISON

1999

AN ANALYSIS OF VALUE PREDICTIBALITY AND ITS APPLICATION TO A SUPERSCALAR PROCESSOR

Yiannakis Thrasou Sazeides
Under the supervision of Professor James E. Smith
At the University of Wisconsin–Madison

ABSTRACT

The central theme of this thesis is the predictability of data-dependence values. Value predictability is studied first at a fundamental level. The analysis shows for SPEC95 programs that:

- Values are amenable to prediction and that predictors that capture repeated behavior are necessary for high prediction accuracy.
- Value predictability is mainly dependent on the program internal structure and immediate values, not input data.

Value prediction and speculation is then applied to a superscalar processor to (a) determine its consequences on processor and value predictor design, and (b) evaluate its performance potential.

Speculative execution and timing models are defined to describe concisely the design space of a superscalar processor influenced by value prediction. It is shown that invalidation and

verification of dependent chains of instructions can be performed in "parallel", contingent on technology constraints. For predictor implementations, emphasis is given to the design of a context-based predictor capable of predicting repeated value sequences, and a hybrid predictor that combines context-based and stride prediction. The following issues are also discussed: (a) delayed updates, speculative updates, and the use of old versus current information for deterministic updates, (b) removal on branch mispredictions of the side-effects due to speculative updates, and (c) high bandwidth value prediction.

Performance simulations suggest that:

- Value prediction can reduce the instruction window size required to achieve a given level of parallelism.
- Value prediction has potential to improve performance on the average 3%–20% depending on the predictor, timing and confidence model used.
- Hybrid predictors are the most cost effective predictors for achieving a given prediction accuracy.
- There is a correlation between prediction accuracy and performance for a given predictor, i.e. improved accuracy translates to better performance.
- When comparing different predictors, accuracy is not a good indication of their performance relation.
- The predictors with the best performance are context-based and hybrid.
- Delayed-updating a context-based predictor does not affect significantly its correct prediction accuracy, it increases however the number of incorrect predictions with high confidence (i.e. mispeculations).
- Fast verification latency and accurate confidence are very critical to value prediction.

Contents

Abstract **i**

1 Introduction **1**

1.1 Motivation 1

1.2 Contributions 3

1.3 Overview 5

2 Related Work **6**

2.1 Prediction 6

2.1.1 Branch Prediction 7

2.1.2 Address Prediction 10

2.1.3 Value Prediction 11

2.1.4 Locality and Predictability 13

2.1.5 Other Prediction Applications 14

2.1.6 Predictability Sources 14

2.1.7 Text Compression 15

2.2 Microarchitectural Support for Value Prediction 17

2.3 Predictors and Pipelining Issues 18

2.4 Performance Trends with Value Prediction 19

3 Predictability of Values **21**

3.1 Motivation and Outline 21

3.2 Classification of Value Sequences 22

3.3 Data Value Prediction Models 23

3.3.1 Computational Predictors 24

3.3.2 Context-Based Predictors 26

3.3.3 An Initial Analysis 28

3.4 Simulation Methodology 30

3.5 Simulation Results 33

3.5.1 Predictability 33

3.5.2 Correlation of Correctly Predicted Sets 37

3.5.3 Value Characteristics 41

3.5.4 Sensitivity Experiments for Context-Based Prediction 44

3.5.5 Confidence for Value Predictions 45

3.6 Summary 47

4 Modeling Program Predictability **49**

4.1 Motivation 49

4.1.1 The Dynamics of Program Predictability 50

4.1.2 Chapter Overview 53

4.2 Predictability Model 53

4.3 Simulation Methodology 56

4.4 Simulation Results 60

4.4.1 Overall Results 60

4.4.2 Generation of Predictability 62

4.4.3 Propagation of Predictability 65

4.4.4 Termination of Predictability 67

4.4.5 Path Behavior 69

4.4.6 Predictable Contiguous Sequences 74

4.5 Branch Instructions 76

4.6 Summary and Discussion 78

5 Microarchitecture for Value Prediction 80

5.1 Motivation and Outline 80

5.2 Proposed Organization 81

5.3 Providing Predictions: Value Predictor, Prediction Register File and Prediction Renaming 85

5.3.1 Values: Valid, Speculative and Predicted 86

5.4 Wake Up and Selection with Speculative Execution 87

5.5 Branch and Memory Instructions 89

5.6 Recovery and Verification 90

5.6.1 Equality 91

5.6.2 Recovery 91

5.6.3 Verification 93

5.6.4 Verification Based on Retirement 95

5.6.5 Parallel Verification/Invalidation with Dedicated Hardware 97

5.7 Speculative Execution Models 101

5.7.1 Example Speculative Execution Models 106

5.8 Summary 108

6 Performance Potential with Perfect Value Prediction 111

6.1 Motivation and Outline 111

6.2 Methodology and Assumptions 112

6.3 Performance Results 114

6.3.1 Results for Perfect Configuration 115

6.3.2 Results for Real Configuration 121

6.3.3 An Analytical Perfect Value Prediction Performance Model 126

6.4 Summary 129

7 Value Predictors Implementations and Performance 131

7.1 Motivation and Outline 131

7.2 Design Space of Value Prediction Models 132

7.2.1 Computational Predictors 133

7.2.2 Context-Based Predictors 134

7.2.3 Hybrid Predictor: Context-Based Stride Predictors 141

7.2.4 Selected Predictor Configurations 143

7.3 Value Predictors Architecture with Pipelining Considerations 145

7.3.1 Delayed Updates 145

7.3.2 Misprediction Recovery 149

7.3.3 High Bandwidth Value Prediction 152

7.3.4 Selected Configurations 153

7.4 Results 155

7.4.1 Simulation Methodology 155

7.4.2 Performance 156

7.4.3 Predictability 160

7.4.4 Performance and Prediction Accuracy Trends for different Predictors and Cost 162

7.4.5 Context Functions and Table Utilization 165

7.4.6 Sensitivity of Performance to the Context-Function Shift Amount . . 166

7.4.7 Timing Models 168

7.4.8 Speculative Execution Models 171

7.5 Summary 174

8 Conclusions **177**

8.1 Summary 177

8.2 Future Work 179

Chapter 1

Introduction

1.1 Motivation

Performance has been a driving force for microarchitecture research since the advent of the computer. One of the popular approaches used to improve performance is the application of hardware and software transformations to a program to increase its **Instruction Level Parallelism (ILP)**. Fundamentally, true **dependences** limit the amount of ILP that can be extracted from a program. Two instructions exhibit a true dependence, or are simply ”dependent”, when an output of one of the instructions is an input operand of the other.

Dependences are typically divided into *control* and *data* dependences. Probably, the most primitive type of control dependence is the program counter (PC) dependence because every instruction depends on its predecessor’s output program counter. However, most instructions modify the PC in a trivial fashion, i.e. they increment it by the instruction length. This length is fixed for many architectures (e.g. RISC) and for others is a function of the instruction type. In either case the instruction length can be determined prior to execution. Processors effectively deal with the trivial PC–dependences by employing instruction caches that allow multiple consecutive instructions to be fetched simultaneously. This leaves control transfer instructions as the remaining control dependence problem.

For a control transfer instruction, the next PC may not be the PC of the next sequential instruction. And determining the next PC requires the instruction's execution. This consequently leads to serialization in the execution of control transfer instructions. **Prediction** and **speculation**

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11] have been proposed as a means for alleviating the impediments of control dependences by predicting the next PC of control transfer instructions and speculatively executing the instructions that follow the branch.

In addition to control dependences through the PC, there is also a serialization of execution due to dependences through register and memory locations. These data-dependences can also be predicted and can drive speculative execution[12, 13, 14, 15, 16, 17, 11, 18, 19, 20].

The predictability of data-dependence values, or **value predictability**, is the central theme of this thesis. This work is intended to demonstrate/present:

- *The amenability of values to prediction.* This is accomplished by proposing basic models for predicting values and determining their predictability limits and relative importance.

- *The robustness of value prediction.* This is demonstrated by showing that value predictability is mainly dependent on the program internal structure and immediate values, not input data.

- *New dimensions for processor design that value prediction creates.* A model for speculative execution is introduced to describe concisely the design space of a superscalar processor influenced by value prediction.

- *How pipelining affects value predictor design.* Parameters are introduced to describe the design space of value predictors in light of delayed updates, misprediction, and multiple predictions per cycle.

- *Performance and prediction accuracy trends for a superscalar processor using value prediction and speculation.*

We recognize that this work does not represent a complete treatment of the subject of value prediction. Rather it is our expectation that this thesis will be a step in the direction of better comprehending this phenomenon and will enable the development of methods for increasing processor performance.

1.2 Contributions

The main contributions of this thesis are described below.

Data Value Predictability: We identify two general prediction models for capturing the behavior of program values: computational and context-based models. The first is proposed for the prediction of value sequences that obey an arithmetic relation, whereas the latter is applicable to repeated value sequences. We establish levels of predictability for SPEC95 programs for two computational predictors proposed elsewhere in the literature and for a context-based predictor proposed in this thesis. The predictors used in the study are ideal in that they do not consider either cost or pipelining issues. It is demonstrated that if high value prediction accuracy is desired, it is necessary to consider models that capture repeated behavior.

Modeling Program Predictability: Three program predictability properties are identified: *generation*, *propagation* and *termination*. These properties are used to demonstrate that: (a) program predictability is mainly due to program control structure and immediate values, but not input data, (b) predictability is often terminated when all incoming information is predictable, (c) predictability of an event is often influenced by only a few predecessors, and (d) long sequences of instructions with both predictable inputs and predictable outputs are common.

Microarchitectures for Value Prediction: We present important microarchitectural features that may be needed for integrating value prediction and speculation in a high performance superscalar processor. The ramifications of having predicted and speculative values on the overall processor design are considered. Issue, invalidation and verification policies are identified as being central to value prediction. We examine their relation and present several options for their implementation. It is demonstrated that invalidation and verification of dependent chains of instructions can be performed in "parallel", contingent on technology constrains. We also propose a set of variables to describe concisely the parameters influenced by value prediction and speculative execution.

Value Predictor Implementations: We identify and discuss parameters for the design of several value predictors. Emphasis is given on the implementation of two new predictors: a context-based predictor, and a hybrid predictor that integrates context-based and stride prediction. The consequences of pipelining, misprediction and multiple instruction fetch and issue on the design of value predictors are also discussed.

Performance Trends: We determine performance trends when using perfect value prediction with perfect and real branch prediction. The data suggest that value prediction can reduce the instruction window size required to achieve a given level of parallelism.

Performance evaluation is also conducted for realistic value predictors, and accuracy and performance trends are established for several processor and predictor parameters. The results indicate that:

- value prediction has potential to improve performance: on the average 3%–20% for the SPEC95 benchmarks, depending on the predictor, timing, and confidence model used,
- hybrid predictors are the most cost effective predictors for achieving a given prediction accuracy,
- there is a correlation between prediction accuracy and performance for a given predictor, i.e. improved accuracy translates to better performance,

- when comparing different predictors, accuracy is not a good indication of their potential contribution to processor performance,
- the predictors with the best performance are context-based and hybrid predictors,
- delayed-updating a context-based predictor does not affect significantly its correct prediction accuracy; however, it increases the number of incorrect predictions with high confidence (i.e. mispeculations), and
- fast verification latency and accurate confidence are very critical to value prediction; significant performance reductions are observed when going from a zero to one cycle verification latency and from oracle to realistic confidence.

1.3 Overview

Chapter 2 reviews related work. Chapter 3 discusses value predictability models at a basic level and presents empirical results for a variety of ideal value predictors. A preliminary version of this work appeared in [16]. Chapter 4 presents a Predictability Model that is used to study and better understand program predictability phenomena. An early version of this work appeared in [21]. Chapter 5 presents a microarchitecture for value prediction. The design and complexity of the key components that may be required for value prediction and speculation are discussed. Chapter 6 reports the performance potential with perfect value prediction and establishes performance trends for different parameters such as width and window size. Practical implementations issues of value predictors are presented in Chapter 7. The chapter also discusses pipelining issues relevant to the organization of value predictors. The performance of last value, stride, context-based and hybrid predictors is also investigated. A subset of this work was reported in [22]. The thesis concludes with a summary and a discussion of future work.

Chapter 2

Related Work

In the following sections we briefly review general prediction methods, microarchitectural support for value prediction, implementation of value predictors, and value prediction performance studies. The main objective of the discussion is to survey and critique existing work.

2.1 Prediction

The earliest application of prediction in computer architecture, as far as we know, was made in the design of IBM computer 7030[1]. The 7030 employed a simple branch prediction strategy, always guess not taken, to enable instructions from the not taken path to be fetched. Another interesting heuristic suggested in [1] is the use of a "guess bit" in the opcode of branch instructions that encodes the likely branch direction. In the same work there is also a discussion about the divergence in processor and memory speed and the importance of mechanisms that will mitigate long memory latency. Interestingly enough, forty years later branches and memory operations still remain at the research forefront with branch and address prediction being two of the most extensive studied subjects.

In the following subsections we discuss prediction in general. An important aim of this discussion is to describe the similar and/or distinct features of value predictors to previously proposed predictors and illuminate the affinity of prediction to other related notions such as locality and compression.

2.1.1 Branch Prediction

Early branch resolution was one of the original motivations for prediction in microarchitecture. Static branch prediction was employed by early processors[1, 23]. The first dynamic branch prediction technique was proposed in combination with a static strategy that relied on branch opcodes[24]. A prediction was obtained by taking the exclusive OR of the static prediction for a branch with a bit indicating whether the static prediction for that branch was correct/incorrect the previous time. The bit was maintained in the instruction cache and was complemented depending on the prediction success/failure.

A purely dynamic prediction scheme that uses a table of saturating counters to capture the history of branches mapping to a table entry was proposed by Smith[2]. A subset of bits from the branch address are used to select which entry to use for prediction. Smith also examined a number of static heuristics that were based on instruction opcodes and showed them to perform more poorly.

Early determination of a branch target is important in reducing the delay for fetching instructions of a taken branch. A comprehensive study of branch target prediction was reported in [4]. A branch target buffer entry is used to hold the target address of a branch, often with other prediction information (such as a saturating counter[2]). The buffer is accessed using branch address bits, and when a branch is predicted taken, the target PC in the buffer entry is the next-PC.

Dynamic two–level prediction[5, 6, 25] enabled further increases in branch prediction accuracy by considering both the address of the predicted branch and the pattern(history) of recent branch outcomes in selecting the next prediction - earlier schemes used only the address. Mcfarling proposed the *gshare* predictor which takes the exclusive OR of the branch address and the previous history to form an index into a prediction table. This was aimed at reducing aliasing and increasing accuracy[7]. *Aliasing* describes the phenomenon where two unrelated events map to the same table entry. Aliasing can therefore be constructive or destructive to accuracy. *Agree* and *bi-mode* predictors are examples of recent dynamic predictors aimed at reducing destructive aliasing effects[26, 27].

Mcfarling[7] also proposed the use of hybrid predictors. The combining predictor in [7] consists of two predictors, a single level table similar to the one in [2] and a two–level gshare[7] predictor. There is also a selector table that is accessed using bits from the branch address. The value read from the selector table summarizes the recent accuracies of the two predictors and is used to choose the prediction from the more accurate of the two predictors. Evers et al.[28] proposed a general organization of a hybrid predictor that can combine any number of predictors. These hybrid predictors use small finite-state machines to guide selection between different predictors and appear to have the best prediction accuracy for a given cost[28].

Path-based prediction is a dynamic prediction approach aimed at reducing table aliasing. It accomplishes this by using the history of recent branch addresses[9]. The premise of this approach is that address history is more indicative of the program's execution path than the branch direction history. The predictor organization is based on two–level tables. A distinct feature of path–based predictors is the hash function used to reduce the address history to a prediction table index. In [9] a hash function that concatenates the least significant bits from recent path addresses is used. Jacobson et al.[29, 30] conducted a more thorough study of path-based prediction when applied to task and trace prediction. Tasks[31] and traces[32] are instruction groups that may consist of many instructions. An important difference between Jacobson's work as compared with previous work is that by predicting tasks and traces, a single prediction can possibly predict multiple branches. The organization used in this work was also based on a two-level table.

Indirect jump prediction is the subject of a number of recent studies [10, 33]. The increased interest in indirect jumps is due to their higher frequency in the emerging object-oriented programming. The underlying predictor structure of these proposals is a two-level predictor that has great similarity to path-based predictors.

Profile-directed static branch prediction was evaluated in a number of studies[34, 35, 36]. Fisher and Freudenberg[35] demonstrated that profiled branch behavior persists across different data inputs. Young and Smith[8] demonstrated that static path-based correlation can increase accuracy at the expense of code duplication. Mahlke and Natarajan[37] proposed the use of profile-based synthesized dynamic branch prediction. The technique is static in that an instruction is inserted in the code to calculate a branch prediction, and dynamic in that run-time values are used to compute the prediction. Deciding which instruction, where to insert it and what operands to use are based on profiling and other static heuristics. This is the first work, as far as we know, that performs branch predictions based on data values.

Basic studies aiming at understanding branch predictor behavior have appeared recently in the literature. A number of papers analyze the effects of aliasing and correlation for different two-level predictors [38, 39, 40]. In a rather theoretical study[41], the similarities between dynamic branch prediction models and prediction models used for text compression are illustrated. The best performing compression techniques, finite context models (*context-based prediction*), rely on the principle of learning events that follow a sequence of events and predict the same event when the sequence repeats[42] - this is the principle behind virtually all dynamic branch prediction techniques proposed to date. This is an important observation because finite context models are known to be among the best performing compression models in practice[42] and hence re-enforce the approach used for branch prediction.

2.1.2 Address Prediction

Numerous proposals have been made for tolerating increasing memory latencies. Some of these approaches rely on prediction – the ones of most interest to us. One block lookahead was one of the first prediction schemes proposed for prefetching[43]. Essentially it predicts that if line x is referenced in the cache then line $x+1$ will likely be referenced soon. The prefetch operation can be conditioned on whether x misses/hits in the cache. Several augmentations of look ahead prefetching were proposed in [44, 45]

A number of data prefetching strategies were proposed by Chen and Baer[46, 47, 48]. Their basic predictor was a stride predictor suited for constant and stride sequences. The stride predictor table was accessed by the program counter of the predicted instruction. Each table entry contained the last address generated and a stride - the difference between the two most recent addresses. The updating of the table entries is guided by a 2-bit state machine in each entry. A variety of improvements to the base strategy were proposed that included use of branch information to delineate loop nesting, and a lookahead program counter to enable early prefetches.

Eickemeyer and Vassiliadis[49] proposed the use of three strategies for dealing with load instruction latency, one of which was based on address prediction. They considered various predictors suited for constant and stride sequences. The best performing prediction strategy is the two-delta method where two strides (d1) and (d2) are maintained. The two delta prediction strategy is similar to the stride prediction of[46], however it uses the two deltas to guide replacement instead of using a 2-bit finite state machine. The one stride (d1) is always updated with the difference between the two most recent values, whereas the other (d2) is used for computing the prediction. When d1 occurs twice in a row it is used to update stride d2.

An interesting prediction scheme that does not rely on previous history was proposed in [50], where it is predicted that the operation required to calculate an effective address using two operands is a logical *or* instead of binary addition. This scheme is aimed at exploiting the behavior of base and offset memory operands that tend to have non-overlapping zero ranges.

Mehrotra and Harrison[51, 52] proposed a hybrid address prefetcher capable of prefetching both stride and indirect address sequences; an example indirect sequence is one generated by a linked list traversal. Indirect address sequences are generated by exploiting the encoding of load instructions with recurrences. The selection between different prefetching modes is guided by a small finite state machine.

Markov chain based prefetching was proposed in [53]. In this scheme, predictors are trained using the first level cache miss stream and learn up to four misses that follow a particular miss. A prefetch request is issued when the same miss is detected. This predictor has similar organization to two-level and path-based branch predictors (discussed in Section 2.1.1).

Software prefetching techniques rely on static models to establish regularity in address patterns and insert prefetch instructions. A number of software prefetching proposals consider prefetching for numeric codes(mainly stride sequences)[46, 47] and non-numeric code(pointer sequences)[54, 55].

A comparison of address and branch prediction techniques reveals similarities but also important differences. In both areas we observe the use of predictors that rely on previous history. However, the use of context-based predictors for address prediction has only been recently proposed[53] and, as yet, it is unclear whether good performance results. One other difference is that addresses exhibit arithmetic and indirection regularity amenable to prediction, something not known to be true for branch prediction.

2.1.3 Value Prediction

Until recently value dependences were considered a fundamental performance limitation. This view changed when Lipasti et al.[12] demonstrated that values exhibit locality; that is, they tend to produce/consume values from the same (often small) set, a large fraction of the time. Although locality by itself can be insufficient for predictability, its presence spurred the development of a number of value predictors that have accomplished increasingly higher prediction accuracy.

Lipasti et al.[12, 14] proposed a last value predictor as the means of exploiting the locality in combination with speculative execution. Last value prediction essentially predicts that the next value will be the same as the last. Value prediction with stride predictors was studied by Gabbay and Mendelson[13]. In [56] profiling is used to improve prediction efficiency by identifying the instructions that exhibit predictable stride and last value behavior and using hardware resources for predictable instructions. Last value and stride predictor table organizations are identical to proposed address predictors[46, 49]. Wang and Franklin[17] proposed a hybrid value predictor configuration that uses a stride predictor to capture constant and stride and a two-level predictor for sequences that exhibit locality up to four which are not stride predictable. The two-level predictor maintains n unique values per instruction. The selection among the different values is based on a unique index associated with each value (if 4 values are maintained a 2 bit index is used). A history of p such indices, maintained per PC, is used to access another table containing counts that indicate which of the values should be selected. The main limitations of the two-level scheme is that it can only exploit locality of up to four unique values, uses inefficiently a table entry when an instruction generates less than four unique values, and it can not share prediction information between different instructions. Fundamentally the proposed scheme may be limited to having few unique values per PC because an associative search is likely to be required to establish if the actual value is in the list of values maintained per PC.

A comparison of the predictors proposed for value prediction and existing branch and address predictors reveals that fundamentally no new prediction model was introduced for predicting values. Rather value predictors appear to be direct applications of predictor organizations used for branches and addresses. We note that context-based value prediction has not been considered before. In Chapter 3 we investigate the potential of context-based value prediction.

2.1.4 Locality and Predictability

An important point that needs to be made is that value locality and predictability are related but not identical notions. Locality is defined to be *"a program attribute that describes the likelihood of the recurrence of a previously seen value within a storage location inside a computer system*[12]. Locality is measured in terms of "history depth" by comparing the latest value in a sequence against the n most recent unique values in the sequence. Predictability of a sequence of values is measured in terms of a specific predictor. Therefore, locality and predictability are independent. For example, consider an instruction that writes *randomly* either 0 or 1 into a register. The sequence observed in the register will exhibit, with history-depth 2, a 100% locality although a history-based predictor will perform poorly. On the other hand, if the sequence consists of long 0 and 1 constant subsequences then it exhibits both predictability and locality. Finally, a sequence can be predictable but not exhibit locality. For example, a long stride sequence is highly predictable by a stride predictor, but exhibits no locally recurring values.

Another subtle but important difference between predictability and locality is that prediction inherently provides a mechanism that specifies a future event, whereas locality, in general, does not. Reuse is a recently proposed approach that essentially exploits locality at both inputs and output of an instruction using a table to look up results previously computed with the given inputs[57]. Reuse, as proposed, exploits locality but not predictability.

2.1.5 Other Prediction Applications

Numerous other applications of prediction in microarchitecture have been proposed. One of the most prevalent ones is confidence prediction. With confidence predictors, a prediction is made to indicate whether another predicted event is likely to be correct or not. Confidence was employed by various address predictors to reduce incorrect prefetches and misspeculation[46, 45, 52, 58, 59]. Similar confidence methods were used to reduce incorrect value predictions and mispeculations[12, 11]. A general class of confidence predictors based on branch predictor models was proposed in [60], and Grunwald et al.[61] analyzed a number of confidence predictors for different performance metrics. The organization of confidence predictors is similar to branch predictors discussed in Section 2.1.1. We consider confidence estimation for value prediction in Chapters 3 and 7.

Another prediction application is dependence prediction aimed at reducing mispeculation due to loads and stores executed out–of–order[18, 19]. Part of the proposed mechanisms, predicts whether a load instruction is dependent on an unresolved store. This is a two-way prediction and hence dependence predictors can also draw from a variety of work on branch prediction. Some other applications of prediction are to coherence protocols[62] and set associative caches[63, 64].

2.1.6 Predictability Sources

No systematic study has been performed to discover the fundamental causes of predictable behavior in programs. Some anecdotal causes of predictability are given in [12], for example, but the relative importance of each is not quantified. Understanding what causes value predictability is critical in asserting the soundness of approaches that rely on value prediction. For example if such an analysis determines that the predictability of programs depends heavily on input data values, this will suggest that value prediction may not be robust since the performance benefits may vary based on the input data. In Chapter 4, we introduce a model for studying program predictability, identify predictability sources, and conduct a detailed analysis of the behavior of predictability in programs.

A related work in [65] investigated the fundamental behavior of instruction repetition in programs. Instruction repetition occurs when multiple instances of an instruction use the same input operands and produce the same output[65]. Although the phenomena examined in [65] and Chapter 4 are different (see Chapter 2.1.4) the two studies used a similar model derived from the dynamic data dependence graph.

We note that the model in [65] can be used to study the behavior of predictability and conversely the model presented in Chapter 4 can be used to study instruction repetition. However, the model in Chapter 4 includes additional information. In particular, predictability labels are assigned to the tail and head of all arcs in the model. Furthermore arcs are divided into categories according to the number of instances of an instruction that are dependent on another instruction instance. The labels and the arc classification are useful in determining (a) different types of predictable behavior, and (b) capturing the influence of control dependences on predictability. In [65] control dependences were not considered.

The two studies also differ in the way they performed analysis. In Chapter 4 a predictable node or arc attributes its predictability to the earliest data predecessor(s) (arc(s) and/or node(s)) that there is a path to with all predictable labels. In [65] an instruction that is repeated owes its behavior to its earliest data predecessor(s). This means that predecessors that do not *repeat* can belong to the path(s) that reaches the earliest predecessor(s).

2.1.7 Text Compression

Compression is a subject formally studied in the area of information theory. The objective of compression is to minimize the amount of information required to represent a message. A message consists of a sequence of symbols from a finite alphabet. For example, the alphabet representing a text sequence usually consists of 7 bit ASCII codes. An essential part of the compression process is a prediction model that is used to assign probabilities for the different symbols that may occur in a message. In general, higher probability symbols are assigned shorter codes and low probability symbols longer codes. Several prediction models for compression have been proposed. As mentioned earlier, the best performing models in practice - and optimal under certain assumptions - are adaptive(dynamic) models that rely on past history(context) to predict the future. A recent study in [41] showed that essentially the best performing predictors used for text compression have great resemblance to predictors used for branch prediction.

Hammerstrom[66] used information theory to study the information content(entropy) of programs. He studied the information content of the address and instruction stream which revealed a high degree of redundancy. The redundancy observed by Hammerstrom was exploited recently by a proposal aimed at compressing the memory stream between processor and memory[67].

If one considers the extensive study of prediction models for compression and microarchitecture, it is probably safe to conclude that there is little opportunity for fundamental innovation in prediction models. Where we believe there is a lot of room for innovation and creativity is in identifying prediction applications for future processors, tuning predictor parameters for these applications and determining how to integrate prediction mechanisms in future processors for better performance. Chapter 3 considers the application of different prediction models to values at a basic level and Chapter 7 considers implementation issues of value predictors when applied to a superscalar processor.

2.2 Microarchitectural Support for Value Prediction

Value prediction so far has been applied to support speculative execution for superscalar[12, 13], trace[11] and very long instruction word[20] processors. All proposals are similar in having mechanisms to support:

- making predictions,
- verifying predictions, and
- recovering from mispredictions.

Lipasti et al.[15, 12, 14, 68] considered predicting either the input or output values of instructions using last and stride value prediction in combination with confidence estimation. A prediction is verified by comparing the predicted and computed values. If there is a mismatch, a recovery action is needed. Three recovery models are proposed: *Instruction Refetch*, *Instruction Reissue* and *Selective Instruction Reissue*. In instruction refetching all instructions following a mispeculated instruction are completely squashed, and instructions are refetched starting at the address that follows the mispredicted instruction. Instruction reissuing is identical to instruction refetch with the exception that following a value mispeculation, instructions can reissue from a buffer, they do not need to be refetched. Finally, selective reissuing squashes and reissues only the instructions that are dependent on the mispredicted instruction. Selective reissue requires a mechanism for tracking the dependences between instructions and a mechanism for communicating the state of these dependences (incorrect, correct etc.). For most of their studies, Lipasti et al. assumed a selective reissuing mechanism that is capable of simultaneously (during a single cycle) validating/invalidating multiple instructions even if they are on a dependence chain. However, they did not discuss the complexity or consequences of such a mechanism. A performance optimization proposed in [15] is to keep instructions with speculative operands in their reservation stations until they are verified. This enables squashed instructions to reissue faster since they do not need to be refetched.

Rotenberg et al.[11] studied the performance of context-based value prediction with selective reissue capability when predicting live-in register values and addresses for traces. Live-ins are input values to instructions not produced from other instructions in a trace. Similar to Lipasti et al., single cycle parallel validation capability is assumed[69]. However, for invalidation, a *serial selective reissue* scheme is used. This is based on the existing result and issue mechanisms of a processor. This work also did not address the design space or consequences of value prediction related policies.

Fu et al.[20] consider value prediction and speculative execution for a statically scheduled processor. The approach relies on profiling to determine which instructions to predict. Instructions are inserted in the code to perform the actual predictor access, verification, and recovery in the case of misprediction.

This thesis considers value prediction and speculation for dynamically scheduled processors. We discuss microarchitectural issues relevant to value prediction in Chapter 5. A detailed discussion of the design space of verification mechanisms and other related policies is presented. The discussion in Chapter 5 elucidates and formalizes some of the implicit (unstated/unclear) assumptions made in some of the above studies. Dimensions of the design space for value speculative execution processors are also identified and their ramifications are discussed. The performance potential of a multiple issue dynamically scheduled processor with perfect and realistic value predictors is analyzed in Chapters 6 and 7.

2.3 Predictors and Pipelining Issues

An important issue that has not yet been addressed adequately is how pipelining influences the design and performance of different predictors. In a real implementation with pipelining, it is possible for a predictor entry to be used for a prediction before the entry is deterministically updated from a previous prediction. Most published prediction results are obtained under the ideal assumption of immediate updates; that is, updating a predictor with the correct value immediately following the prediction. Other issues typically ignored in value prediction studies are updates from instructions that are squashed due to control misprediction and the recovery of prediction state prior to the misspeculation. Bandwidth issues also have not been addressed adequately - that is, how to organize predictors to provide multiple value predictions, possibly for the same instruction, in the same cycle[70].

Issues pertaining to pipelining and two-level branch predictors were briefly discussed in[71]. The issue of speculative updates[72] - that is updating a predictor with speculative state - was studied in a subsequent work where it was shown that speculative updates could attain roughly the same performance as with with immediate updates for a two-level branch predictor with global history. The same study indicated that with no speculative updates, the performance deteriorates significantly. Implementation issues for speculatively updated branch predictors with global history and return address stack were considered in [73]. Although speculative updates for global history predictors appear to be essential is not clear whether they are needed for local history predictors.

To our knowledge, no study has considered pipelining issues with respect to address and value predictors. We expect that such considerations will strongly influence the organization of predictors as well as their accuracy. In Chapter 7, we discuss implementation issues of value predictors and how pipelining affects prediction performance.

2.4 Performance Trends with Value Prediction

Most previous value prediction studies did not establish performance trends with varying issue widths and window sizes[12, 14, 15, 11]. The exception to the above is a study by

Gabbay and Mendelson which examined configurations with variable width but a fix window size[70]. In [70], is concluded that small fetch and issue widths limit the potential of value prediction since most instructions entering the instruction window already have their operands available, and thereby obviate any benefit due to prediction and speculation.

In this work, we perform a comprehensive analysis of the trends with perfect value prediction by varying issue width and window size for both real and perfect branch prediction. These results are the subject of Chapter 6. We also establish performance and prediction accuracy trends for four realistic value predictors. Two of these, context-based predictors and the hybrid context/stride predictor, have not been considered before. The effects on accuracy and performance of several processor and predictor parameters are considered. Examples of parameters considered are processor width/window size, predictor cost and configuration, timing, and confidence.

Chapter 3

Predictability of Values

3.1 Motivation and Outline

Value prediction in combination with value speculation is a promising approach for increasing processor performance[12, 13, 14, 11, 20]. Although value speculation may prove to be a successful approach in the long-run, we believe it is far from clear how to best exploit value predictability. Furthermore, we argue that to develop effective paradigms that utilize predictability it is paramount to identify models suited for value prediction and establish predictability limits for program values. Consequently, in this chapter we study the predictability of data values at a fundamental level. We first identify basic prediction models amenable to the prediction of program values and then establish predictability levels for representative value predictors. Two basic predictor models are defined: **Computational** predictors perform an operation on previous values to yield predicted next values. Examples we study are stride value prediction (which adds a delta to a previous value) and last value prediction (which performs the trivial identity operation on the previous value).

Context-Based predictors match recent value history (context) with previous value history and predict values based entirely on previously observed patterns. The study is somewhat idealized; for example, predictors costs are ignored and tables are updated immediately in order to more clearly understand limits of data value predictability. The ways in which data prediction can be used in a processor microarchitecture are not discussed in this chapter (see Chapter 5), so that we can concentrate in greater depth on the prediction process, itself.

3.2 Classification of Value Sequences

The predictability of a sequence of values is a function of both the sequence itself and the predictor used to predict the sequence. To better understand the behavior of predictors in later discussions we provide an informal classification of data sequences. Note the actual sources of predictability is the subject in Chapter 4. The following classification contains simple value sequences that can also be composed to form more complex sequences. They are best defined by giving examples:

Constant(C)	5 5 5 5 5 5 5...
Stride(S)	1 2 3 4 5 6 7 8...
Non-Stride(NS)	28 -13 -99 107 23 456...

Constant sequences are the simplest, and result from instructions that repeatedly produce the same result. Lipasti et al.[12] show that this occurs often. A stride sequence has elements that differ by some constant delta. For the example above, the stride is one, which is probably the most common case in programs, but other strides are possible, including negative strides. Constant sequences can be considered stride sequences with a zero delta. A stride sequence might appear when a data structure such as an array is being accessed in a regular fashion; loop induction variables also have a stride characteristic. Gabbay and Mendelson[13] show that stride sequences are common and predictors that capture them can increase value prediction accuracy.

The non-stride category is intended to include all other sequences that do not belong to the constant or stride category. This classification could be further divided, but we choose not to do so. Non-strides may occur when a sequence of numbers is being computed and the computation is more complex than simply adding a constant. Traversing a linked list would often produce address values that have a non-stride pattern.

Also very important are sequences formed by composing stride and non-stride sequences with themselves. Repeating sequences would typically occur in nested loops where the inner loop produces either a stride or a non-stride sequence, and the outer loop causes this sequence to be repeated.

Repeated Stride(RS)	1 2 3 1 2 3 1 2 3...
Repeated Non-Stride(RNS)	1 -13 -99 7 1 -13 -99 7...

Examination of the above sequences leads naturally to two types of prediction models that are the subject of discussion throughout the remainder of this chapter:

Computational predictors make a prediction by computing some function of previous values. An example of a computational predictor is a stride predictor. This predictor adds a stride to the most recent value in a sequence.

Context-based predictors learn the value(s) that follow a particular context - a finite ordered sequence of values - and predict one of the values when the same context repeats. This enables the prediction of any repeated sequence, stride or non-stride.

3.3 Data Value Prediction Models

A typical data value predictor takes microarchitecture state information as input, accesses a table, and produces a prediction. Subsequently, the table is updated with state information to help make future predictions. The state information could consist of register values, program counter (**PC**) values, instruction fields, control bits in various pipeline stages, etc.

The variety and combinations of state information are almost limitless. Therefore, in this study, we restrict ourselves to predictors that use only the program counter value of the instruction being predicted to access the prediction table(s). The tables are updated using data values produced by the instruction – possibly modified or combined with other information already in the table. These restrictions define a relatively fundamental class of data value predictors. Nevertheless, predictors using other state information deserve study and could provide a higher level of predictability than is reported here.

For the remainder of this chapter, we further classify data value predictors into two types – computational and context-based. We describe each in detail in the next two subsections.

3.3.1 Computational Predictors

Last Value Predictors perform a trivial computational operation: the identity function. In its simplest form, if the most recent value produced by an instruction is **v** then the prediction for the next value will also be **v**. However, there are a number of variants that modify replacement policies based on **hysteresis**. An example of a hysteresis mechanism is a saturating counter that is associated with each table entry. The counter is incremented/decremented on prediction success/failure with the value held in the table replaced only when the count is below some threshold. Another hysteresis mechanism does not change its prediction to a new value until the new value occurs a specific number of times in succession. A subtle difference between the two forms of hysteresis is that the former changes to a new prediction following incorrect behavior (even though the new behavior may be inconsistent), whereas the latter changes to a new prediction only after new behavior has been consistently observed. Last value prediction was first applied to values by Lipasti et al.[12].

Stride Predictors in their simplest form predict the next value by adding the most recent value to the difference of the two most recent values produced by an instruction. That is if $\mathbf{v}_{n-1}$ and $\mathbf{v}_{n-2}$ are the two most recent values, then the predictor predicts $\mathbf{v}_n$**=**$\mathbf{v}_{n-1}$**+(**$\mathbf{v}_{n-1}$**-**

$\mathbf{v}_{n-2}$). As with the last value predictors, there are important variations that use hysteresis. In [46, 58, 59] the stride is only changed if a saturating counter that is incremented/decremented on success/failure of the predictions is below a certain threshold. This reduces the number of mispredictions in repeated stride sequences from two per repeated sequence to one. Another policy, the **two-delta** method, was proposed in [49]. In the two-delta method, two strides are maintained. The one stride (*s1*) is always updated by the difference between the two most recent values, whereas the other (*s2*) is the stride used for computing the predictions. When stride s1 occurs twice in a row then it is used to update the prediction stride s2. The two-delta strategy also reduces mispredictions to one per iteration for repeated stride sequences and, in addition, only changes the stride when the same stride occurs twice - instead of changing the stride following mispredictions.

The first application for stride predictors was for data addresses in scientific codes by Chen and Baer[46]. A number of variants of address stride predictors were proposed that differ mainly in terms of the replacement policy used[49, 52, 58, 59]. Stride prediction was first applied to values by Gabbay and Mendelson[13].

Other Computational Predictors using more complex organizations can be conceived. For example, one could use two different strides, an “inner” one and an “outer” one – typically corresponding to loop nests – to eliminate the mispredictions that occur at the beginning of repeating stride sequences[48]. This thought process illustrates a significant limitation of computational prediction: the designer must anticipate the computation to be used. One could carry this to ridiculous extremes. For example, one could envision a Fibonacci series predictor, and given a program that happens to compute a Fibonacci series, the predictor would do very well. Going further down this path would lead to large hybrid predictors that combine many special-case computational predictors with a “chooser” - as has been proposed for conditional branches in [7, 28]. While hybrid prediction for data values is in general a good idea, a potential pitfall is that it may yield an ever-escalating collection of computational

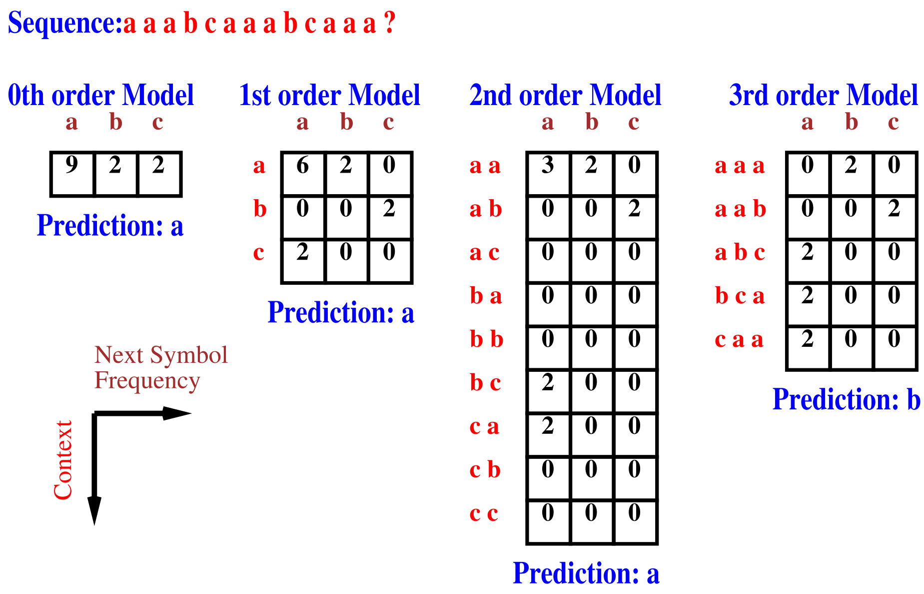

Figure 3.1: Finite Context Models

predictors, each of which predicts a diminishing number of additional values not caught by the others.

In this study, we focus on last value and stride methods as primary examples of computational predictors. We also consider the “potential” of hybrid predictors involving these predictors and the context-based predictors to be discussed in the next section.

3.3.2 Context-Based Predictors

Context-based predictors attempt to “learn” values that follow a particular context – a finite ordered sequence of previous values – and predict one of the values when the same context repeats. An important type of context-based predictors is derived from finite context methods(also known as predictors by partial matching) used in text compression[42].

Finite Context Method Predictors (fcm) rely on mechanisms that predict the next value based on a finite number of preceding values. An **order** k fcm predictor uses k preceding values. Fcms are constructed with counters that count the occurrences of a particular value immediately following a certain context (pattern). Thus for each context there must be, in general, as many counters as values that are found to follow the context. The predicted value is the one with the maximum count. Figure 3.1 shows fcm models of different orders and predictions for an example sequence.

In an actual implementation where it may be infeasible to maintain exact value counts, smaller counters may be used. The use of small counters comes from the area of text compression. With small counters, when one counter reaches the maximum count, all counters for the same context are reset by half. Small counters provide an advantage if heavier weighting should be given to more recent history instead of the entire history.

In general, n different fcm predictors of orders 0 to n-1 can be used for predicting the next value of a sequence, with the highest order predictor that has a context match being used to make the prediction. The combination of more than one prediction model is known as *blending* [42]. There are a number of variations of blending algorithms, depending on the information that is updated. *Full blending* updates all contexts, and *lazy exclusion* selects the prediction with the longer context match and only updates the counts for the predictions with the longer match or higher. To illustrate blending and lazy exclusion consider the scenario shown in Figure 3.1 using a predictor that blends all four models. The prediction, in this case, will come from the 3rd order model - predicting **b** - since it has the longest match **a a a**. According to lazy exclusion, if the next value is b then only the 3rd order model will be updated.

Other variations of fcm predictors can be devised by reducing the number of values that are maintained for a given context. For example, only one value per context might be maintained along with some replacement policy. Such policies can be based on hysteresis-type replacement policies as discussed above for last value and stride prediction and can be more suitable for hardware implementation of context-based predictors.

Sequence	Prediction Model					
	Last Value		Stride		FCM	
	LT	LD(%)	LT	LD(%)	LT	LD(%)
C	1	100	1	100	o	100
S	-	-	2	100	-	-
NS	-	-	-	-	-	-
RS	-	-	2	100(p-1)/p	p+o	100
RNS	-	-	-	-	p+o	100

Table 3.1: Behavior of various Prediction Models for Different Value Sequences

As far as we know, context-based prediction has not been considered for value prediction, though the last value predictor can be viewed as a 0th order fcm with only one prediction maintained per context. Note that virtually all dynamic branch predictors (see Chapter 2) strongly resemble context-based predictors[41] and hence practical implementations of context-based predictors should draw from the wealth of knowledge in branch prediction.

3.3.3 An Initial Analysis

At this point, we briefly analyze and compare the proposed predictors using the simple pattern sequences shown in Section 3.2. This analysis highlights important issues as well as advantages and disadvantages of the predictors to be studied. As such, they can provide a basis for analyzing quantitative results given in the following sections.

We informally define two characteristics that are important for understanding prediction behavior. One is the Learning Time (**LT**) which is the number of values that have to be observed before the first correct prediction. The second is the Learning Degree (**LD**) which is the percentage of correct predictions following the first correct prediction.

We quantify these two characteristics for the classes of sequences given earlier in Section

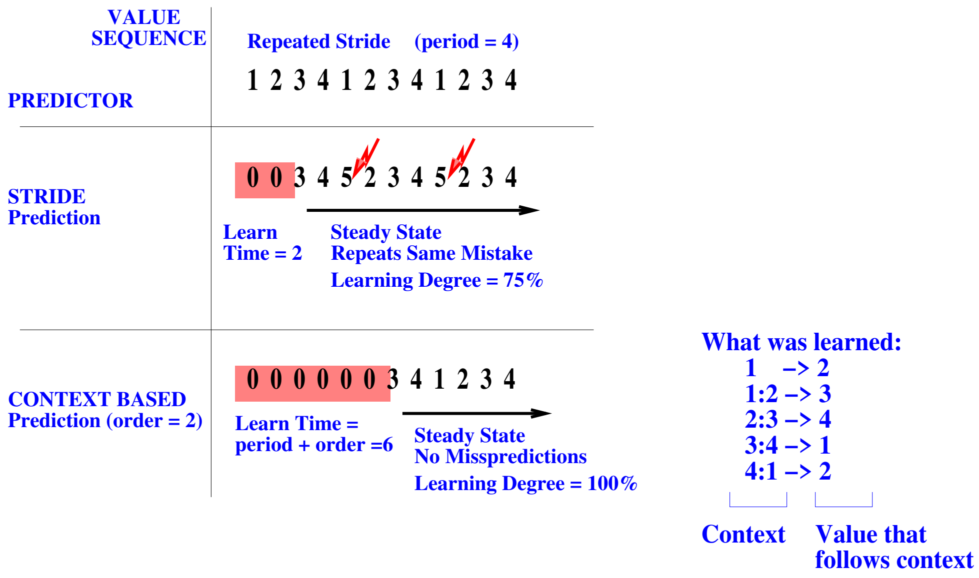

Figure 3.2: Computational vs Context-Based Prediction

3.2. For the repeating sequences, we associate a period (**p**), the number of values between repetitions. For context predictors, the order (**o**) of a predictor influences the learning time. We assume repeating sequences that occur with high frequency, and p is large and fixed.

Table 3.1 summarizes how the different predictors perform for the basic value sequences. Note that the stride predictor uses hysteresis for updates, so it gets only one incorrect prediction per iteration through a sequence. A row of the table with a "-" indicates that the given predictor is not suitable for the given sequence, i.e., its performance is very low for that sequence.

As illustrated in the table, last value prediction is only useful for constant sequences – this is obvious. Stride prediction does as well as last value prediction for constant sequences because a constant sequence is essentially zero stride. The fcm predictors also do very well on constant sequences, but an order o predictor must see a length o sequence before it gets matches in the table (unless some form of blending is used).

For (non-repeating) stride sequences, only the stride predictor does well; it has a very short learning time and then achieves a 100% prediction rate. The fcm predictors cannot predict non-repeating sequences because they rely on repeating patterns.

For repeating stride sequences, both stride and fcm predictors do well. The stride predictor has a shorter learning time, and once it learns, it only gets a misprediction each time the sequence begins to repeat. On the other hand, the fcm predictor requires a longer learning time – it must see the entire sequence before it starts to predict correctly, but once the sequence starts to repeat, it gets 100% accuracy (Figure 3.2). This example points out an important tradeoff between computational and context based predictors. The computational predictor often learns faster – but the context predictor tends to learn "better" when repeating sequences occur.

Finally, for repeating non-stride sequences, only the fcm predictor does well. And the flexibility this provides is clearly the strong point of fcm predictors. Returning to our Fibonacci series example – if there is a sequence containing a *repeating* portion of the Fibonacci series, then an fcm predictor will naturally begin predicting it correctly following the first pass through the sequence.

Of course, in reality, value sequences can be complex combinations of the simple sequences in Section 3.2, and a given program can produce about as many different sequences as instructions are being predicted. Consequently, in the remainder of the paper, we use simulations to get a more realistic idea of predictor performance for programs.

3.4 Simulation Methodology

We adopt an idealized approach for studying predictability of data dependence values. The reason for this choice is to remove microarchitecture and other implementation idiosyncrasies in an effort to develop a basic understanding of predictability. Hence, these results can best be viewed as bounds on prediction accuracy; it will take additional engineering research to develop realistic implementations.

We study the predictability of instructions that write results into general purpose registers (i.e. memory addresses, stores, jumps and branches are not considered). Prediction was done with no table aliasing; each static instruction was given its own table entry. Hence, table sizes are effectively unbounded. Finally, prediction tables are updated immediately after a prediction is made, unlike the situation in practice where it may take many cycles for the actual data value to be known and available for prediction table updates.

We simulate three types of predictors: last value prediction (**l**) with an always-update policy (no hysteresis), stride prediction using the 2-delta method (**s2**), and a finite context method (**fcm**) that maintains exact counts for each value that follows a particular context and uses the blending algorithm with lazy exclusion, described in Section 3.3.2. Fcm predictors are studied for orders 1, 2 and 3. To form a context for the fcm predictor we use full concatenation of history values so there is no aliasing when matching contexts. A stride predictor that uses a 2-bit saturating counter to guide replacement was simulated and was found to consistently underperform the 2-delta predictor; thus we do not present results for that predictor.

Trace driven simulation was conducted using the Simplescalar toolset [74] for the integer SPEC95 benchmarks shown in Table 3.2. For ijpeg the simulations used the reference flags with the following changes: compression.quality 45 and compression.smoothing_factor 45. The benchmarks were compiled using the simplescalar compiler with -O3 optimization.

For collecting prediction results, instruction types were grouped into categories as shown in Table 3.3. The abbreviations shown after each group will be used subsequently when results are presented. The percentage of predicted instructions in the different benchmarks ranged between 62%–84%. Recall that some instructions like stores, branches and jumps are not predicted. A breakdown of the dynamic percentages of predicted instruction types is shown in Table 3.4. The majority of predicted values are the results of adds and loads.

Benchmark	Input Flags	Dynamic Instr. (mil)	Instructions Predicted (mil)
compress	30000 e	8.2	5.8 (71%)
gcc	gcc.i	203	137 (68%)
go	9 9	132	105 (80%)
ijpeg	specmun.ppm	129	108 (84%)
m88k	ctl.raw	493	345 (70%)
perl	scrabbl.in	40	26 (65%)
xlisp	7 queens	202	125 (62%)

Table 3.2: Benchmarks Characteristics

Instruction Types	Code
Addition, Subtraction	AddSub
Loads	Loads
And, Or, Xor, Nor	Logic
Shifts	Shift
Compare and Set	Set
Multiply and Divide	MultDiv
Load immediate	Lui
Floating, Jump, Other	Other

Table 3.3: Instruction Categories

Type	com	gcc	go	ijpe	m88k	perl	xlis
AddSu	42.6	38.9	42.1	52.4	42.6	34.1	36.1
Loads	20.5	38.6	26.2	21.4	24.8	43.1	48.6
Logic	3.1	3.1	0.5	1.9	5.0	3.1	3.4
Shift	17.4	7.7	13.3	16.4	3.2	8.2	3.2
Set	7.4	5.4	4.9	4.2	15.2	5.6	3.2
MultDi	0.01	0.4	0.3	3.2	0.1	0.2	0.01
Lui	3.3	3.7	11.4	0.2	6.9	2.4	0.8
Other	5.7	2.1	1.3	0.3	2.1	3.3	4.8

Table 3.4: Predicted Instructions - Dynamic(%)

We collected results for each instruction type. However, we do not discuss results for the **other**, **multdiv** and **lui** instruction types. In the benchmarks we studied, the multdiv instructions are not a significant contributor to dynamic instruction count, and the lui and "other" instructions rarely generate more than one unique value and are over 95% predictable by all predictors. We note that the effect of these three types of instructions is included in the calculations for the **overall** results. For averaging we used arithmetic mean, so each benchmark effectively contributes the same number of total predictions.

3.5 Simulation Results

3.5.1 Predictability

Figure 3.3 shows the overall predictability for the selected benchmarks, and Figures 3.4–3.7 shows results for the important instruction types. From the figures we can draw a number of conclusions. Overall, last value prediction is less accurate than stride prediction, and stride

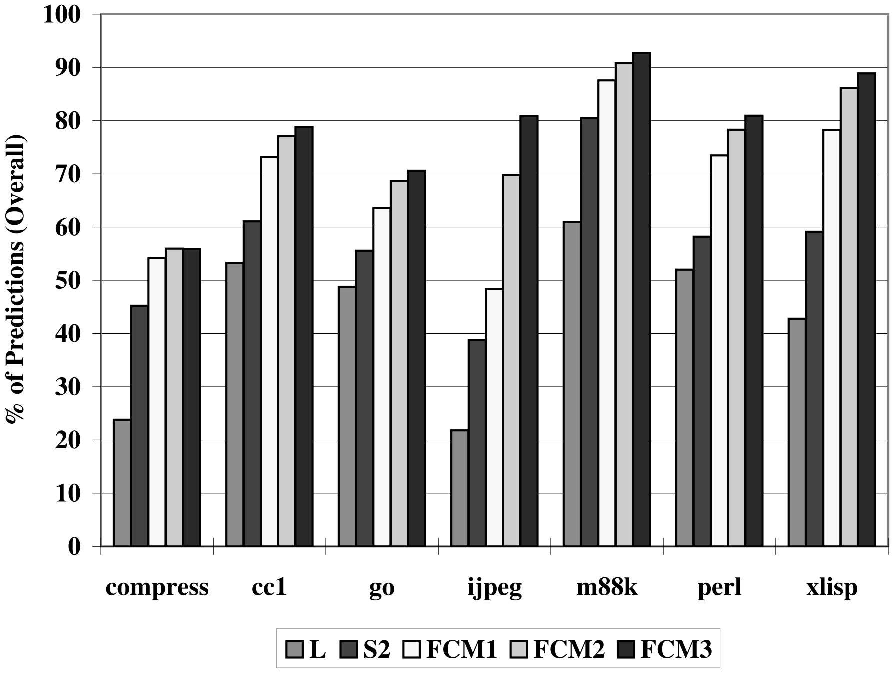

Figure 3.3: Prediction Success for All Instructions

prediction is less accurate than fcm prediction. Last value prediction varies in accuracy from about 23% to 61% with an average of about 45%. This is in agreement with the results obtained in [14]. Stride prediction provides accuracy of between 38% and 80% with an average of about 56%. Fcm predictors of orders 1, 2, and 3 all perform better than stride prediction; and the higher the order, the higher the accuracy. The order 3 predictor is best and gives accuracies of between 56% and over 90% with an average of 78%. For the three fcm predictors studied, improvements diminish as the order is increased. The effect on predictability with increasing order is examined in more detail in Section 3.5.4.

Performance of the stride and last value predictors varies significantly across different instruction types for the same benchmark. The performance of the fcm predictors varies less significantly across different instruction types for the same benchmark. This reflects the flexibility of the fcm predictors – they perform well for any repeating sequence, not just strides. In general, both stride and fcm prediction appear to have higher predictability for

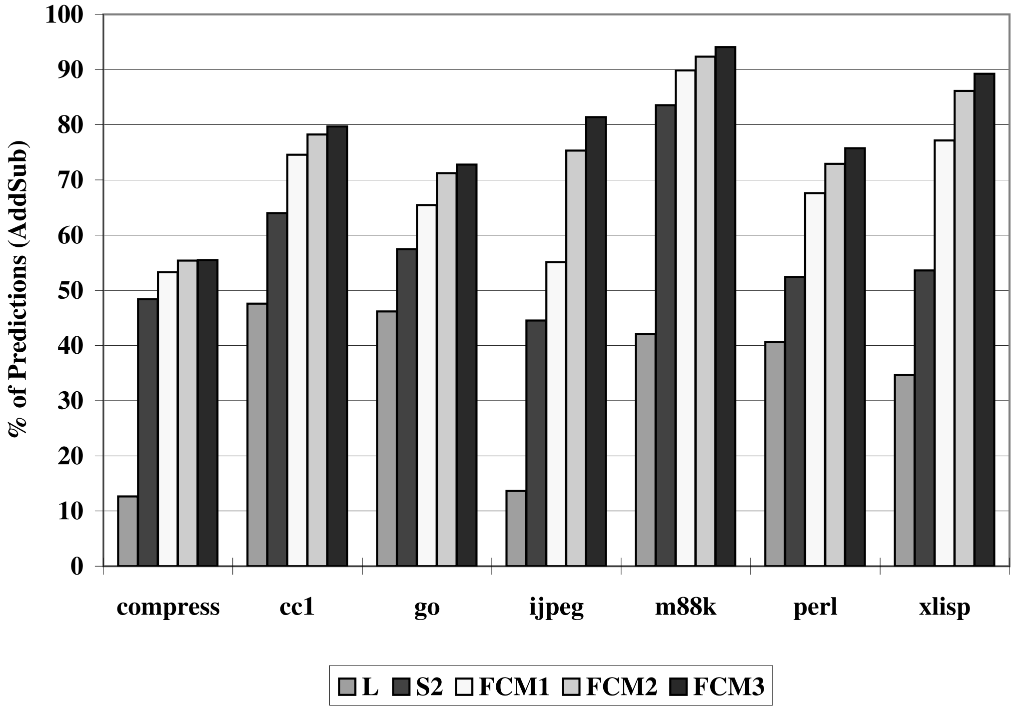

Figure 3.4: Prediction Success for Add/Subtract Instructions

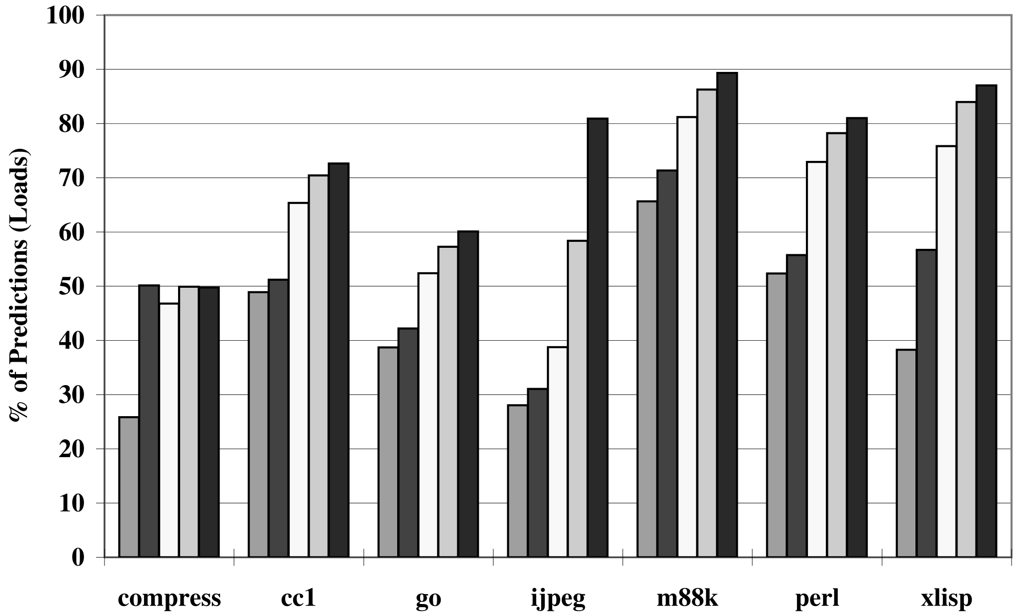

Figure 3.5: Prediction Success for Loads Instructions

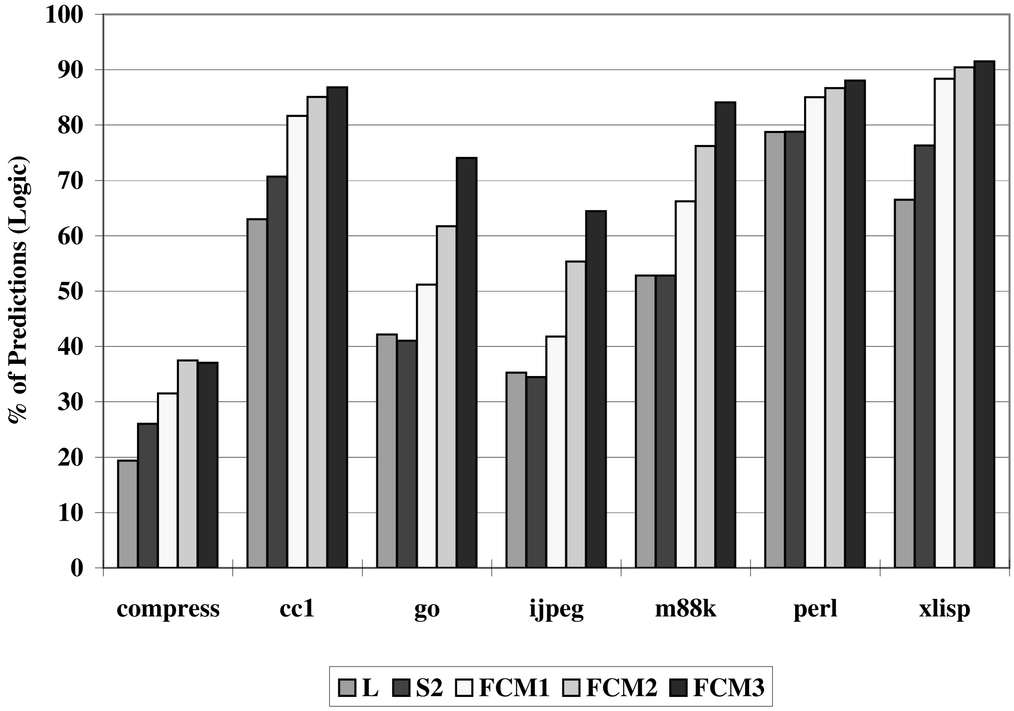

Figure 3.6: Prediction Success for Logic Instructions

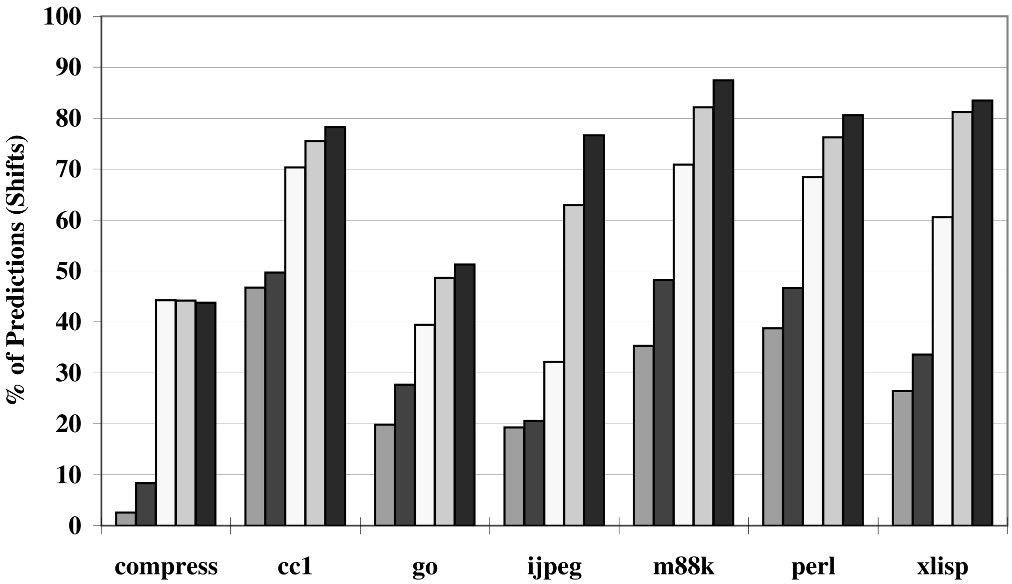

Figure 3.7: Prediction Success for Shift Instructions

add/subtract instructions than loads. Logical instructions also appear to be very predictable, especially by the fcm predictors. Shift instructions appear to be the most difficult to predict.

Stride prediction does particularly well for add/subtract instructions. But for non-add/subtract instructions the performance of the stride predictor is close to last value prediction. This indicates that when the operation of a computational predictor matches the operation of the instruction (e.g. addition), higher predictability can be expected. This suggests new computational predictors that better capture the functionality of non-add/subtract instructions could be useful. For example, for shifts a computational predictor might shift the last value according to the last shift distance to arrive at a prediction. This approach would tend to lead to hybrid predictors, however, with a separate component predictor for each instruction type.

3.5.2 Correlation of Correctly Predicted Sets

The results in the previous section essentially compare the sizes of the sets of correctly predicted values. It is also interesting to consider relationships among the specific sets of correctly predicted values. Primarily, these relationships suggest ways that hybrid predictors might be constructed – an actual implementation of a hybrid predictor that combines stride and context-based prediction is discussed in Chapter 7.

The predicted set relationships are shown in Figure 3.8. Three predictors are used: last value, stride (delta-2), and fcm (order 3). All subsets of predictors are represented. Specifically: **l** is the fraction of predictions for which *only* the last value predictor is correct; **s** and **f** are similarly defined for the stride and fcm predictors respectively; **ls** is the fraction of predictions for which *both* the last value and the stride predictors are correct but the fcm predictor is not; **lf** and **sf** are similarly defined; **lsf** is the fraction of predictions for which all predictors are correct; and **np** is the fraction for which none of the predictors is correct.

In the figure results are averaged over all benchmarks, but the qualitative conclusions are similar for each of the individual benchmarks. Overall, Figure 3.8 can be briefly summarized:

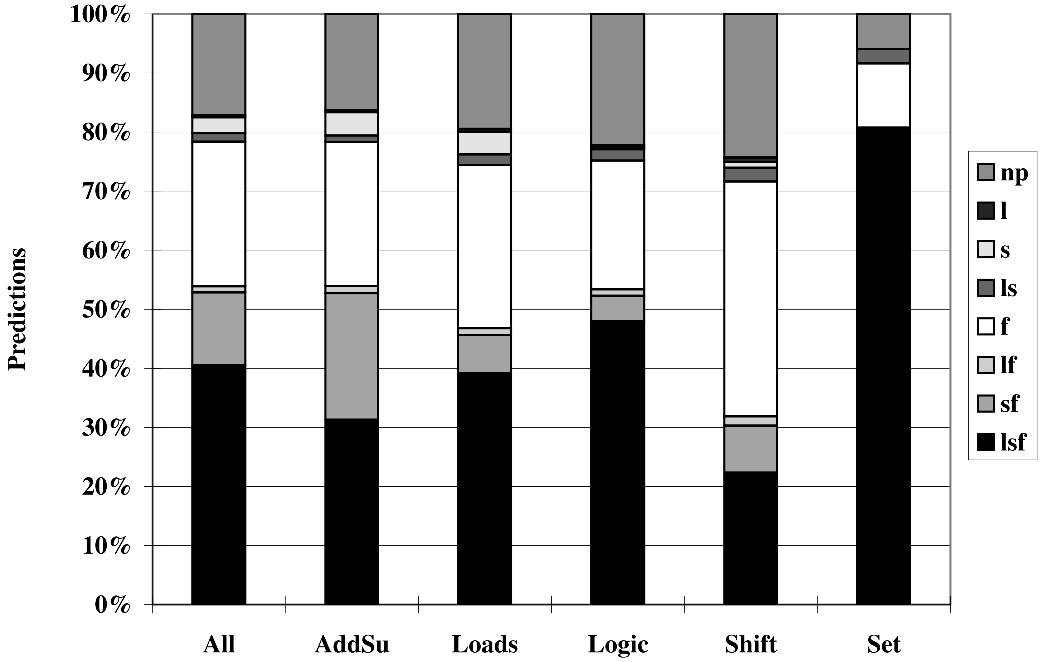

Figure 3.8: Contribution of different Predictors

- A small number, close to 18%, of values are not predicted correctly by any model.
- A large portion, around 40%, of correct predictions is captured by all predictors.
- A significant fraction, over 20%, of correct predictions is only captured by fcm.
- Stride and last value prediction capture less than 5% of the correct predictions that fcm misses.

The above confirms that data values are very predictable. And it appears that context-based prediction is necessary for achieving the highest levels of predictability. However, almost 60% of the correct predictions are also captured by the stride predictor. Assuming that context-based prediction is the more expensive approach, this suggest that a **hybrid** scheme might be useful to maintain high prediction accuracies at lower cost. That is, one should try to use a stride predictor for most predictions, and use fcm prediction for the remaining ones.

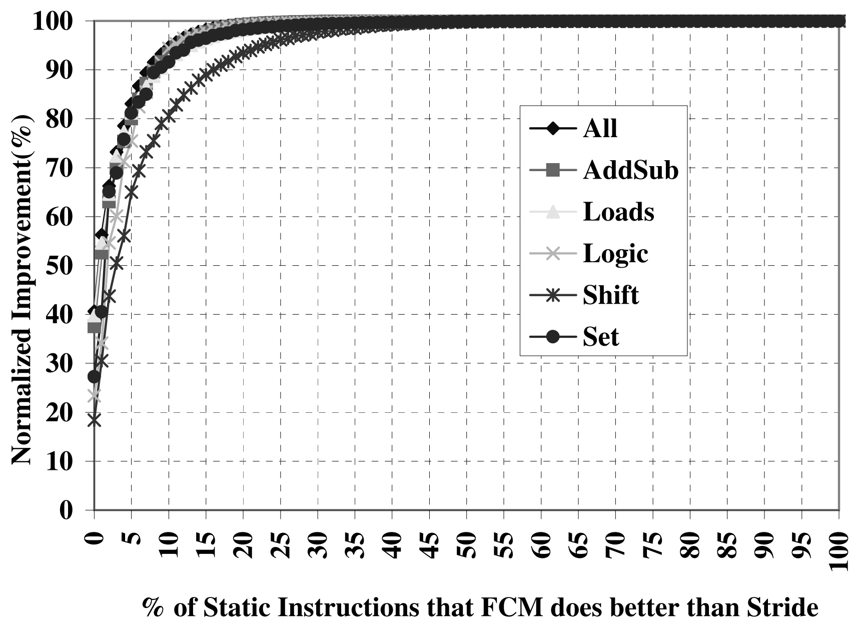

Figure 3.9: Cumulative Improvement of FCM over Stride

Note that the results suggest that a hybrid predictor with oracle selection among the predictors in Figure 3.8 can achieve a marginal improvement over fcm predictor (82% versus 78%). This suggests that hybrid implementations should be designed with care so that the fcm contribution is not significantly reduced.

Another conclusion is that last value prediction adds very little to what the other predictors achieve. So, if either stride or fcm prediction is implemented, there is no point in adding last value prediction to a hybrid predictor.

The important classes of load and add instructions yield results similar to the overall average. Finally, we note that for non-add/subtract instructions the contribution of stride prediction is smaller, this is likely due to the earlier observation that stride prediction does not match the functionality of other instruction types. This suggests a hybrid predictor based on instruction types.

Proceeding along the path of a hybrid fcm-stride predictor, one reasonable approach would be to choose among the two component predictors via the PC address of the instruction being predicted. This would appear to work well if the performance advantage of the fcm predictor is due to a relatively small number of static instructions that generate relatively few values.

To determine if this is true, we first constructed a list of static instructions for which the fcm predictor gives better performance. For each of these static instructions, we determined the difference in prediction accuracy between fcm and stride. We then sorted the static instructions in descending order of improvement. Then, in Figure 3.9 we graph the cumulative fraction of the total improvement versus the accumulated percentage of static instructions. The graph shows that overall, about 10% of the static instructions account for about 90% of the total improvement of fcm over stride prediction. For most of individual instruction types, the result is similar, with shifts showing slightly worse performance. The results suggest that improvements due to context-based prediction are mainly due to a relatively small fraction of static instructions.

With regard to the number of unique values generated by instructions that fcm predicts more accurately than stride, in Figure 3.10, we show the cumulative improvement as a function of the number of values generated by these instructions. The results indicate that about 70% of the improvement is due to instructions that generate less than 256 values. If practical implementations of large value tables is not feasible this suggests that it may be necessary to limit prediction by the fcm predictor to a subset of instructions that generate few values. We note that this conclusion can be pessimistic in that it assumes all unique values produced by an instruction have similar influence on predictability. A more detail analysis for instruction value behavior is given in Section 3.5.3.

Overall, the above indicate that a practical implementation may need to rely on a hybrid approach with filtering of the instructions predicted by the fcm component. We discuss implementation of value predictors - including hybrid - in Chapter 7.

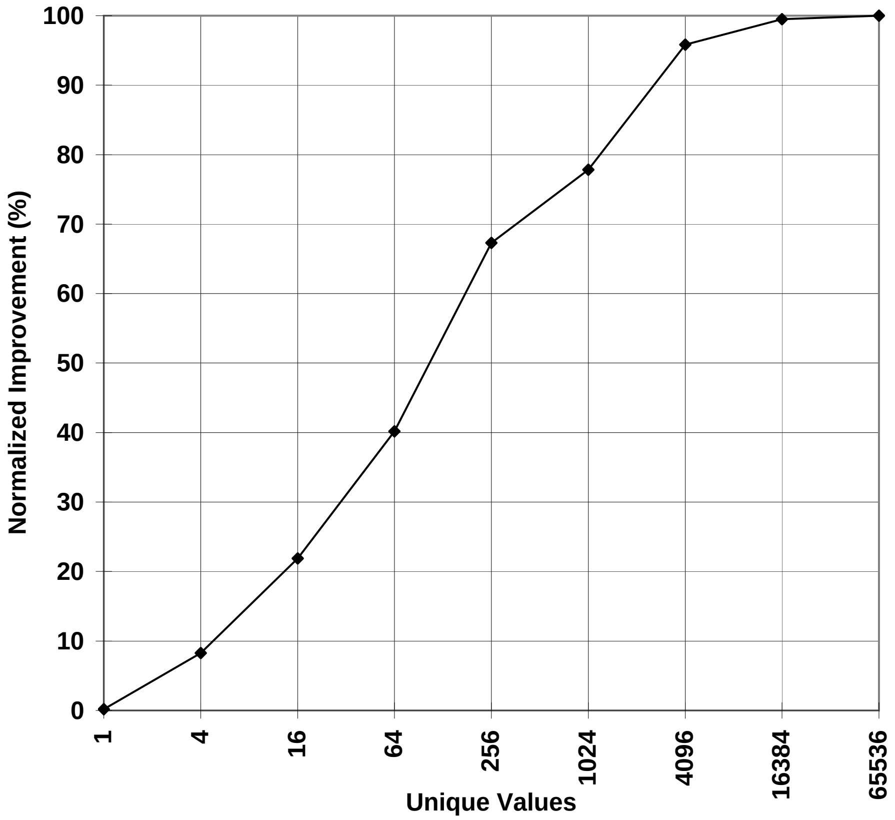

Figure 3.10: Improvement of FCM over Stride as a Function of Unique Values Generated

3.5.3 Value Characteristics

In this section we study the value characteristics of instructions. In particular we report on the number of **unique** values generated by predicted instructions. The overall numbers of unique values could give a rough indication of the numbers of values that might have to be stored in a table.

In the left half of Figure 3.11, we show the number different values produced by percentages of static instructions (an **s** prefix). In the right half, we determine the fractions of dynamic instructions (a **d** prefix) that correspond to each of the static categories.

From the figure, we observe:

- A large number, $\geq$50%, of static instructions generate only one value.

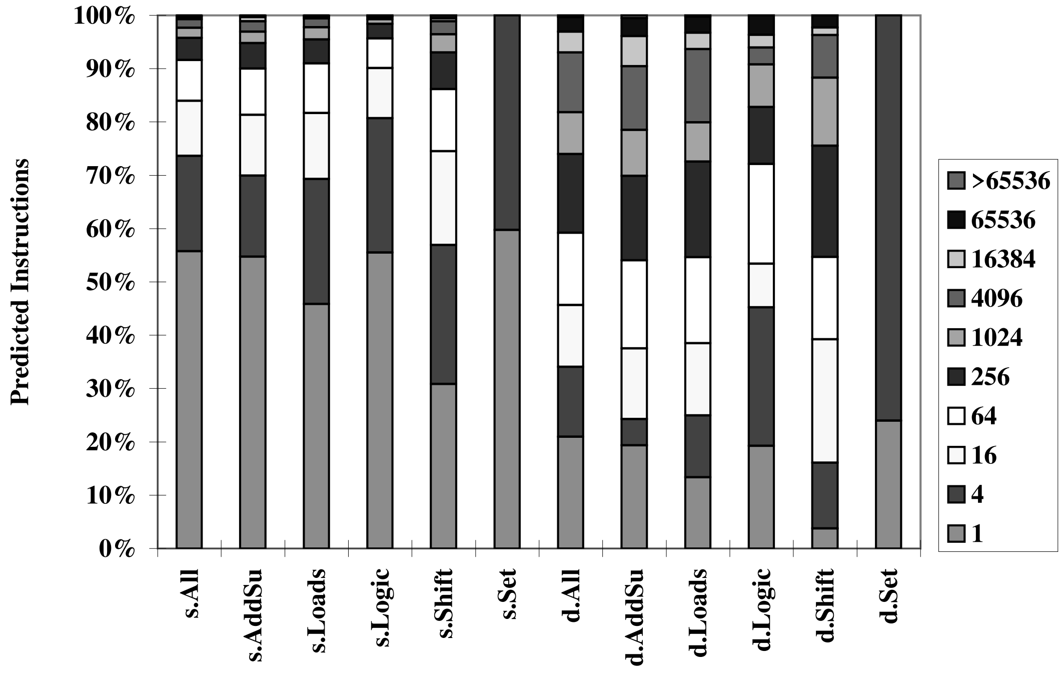

Figure 3.11: Values and Instruction Behavior

- The majority of static instructions, $\geq$90%, generate fewer than 64 values.
- The majority, $\geq$50%, of dynamic instructions correspond to static instructions that generate fewer than 64 values.
- Over 90% of the dynamic instructions are due to static instructions that generate at most 4096 unique values.
- The number of values generated varies among instruction types. In general add/subtract and load instructions generate more values as compared with logic and shift operations.
- The more frequently an instruction executes, the more values it generates.

The above reenforces an earlier observation (Section 3.5.2) that a relatively small number of values would be required to predict correctly the majority of dynamic instructions using context-based prediction – a positive result.

From looking at individual benchmark results (not shown) there appears to be a positive correlation between programs that are more difficult to predict and the programs that produce more values. For example, the highly predictable m88ksim has many more instructions that produce few values as compared with the less predictable gcc and go. This would appear to be an intuitive result, but there may be cases where it does not hold; for example if values are generated in a fashion that is predictable with computational predictors or if a small number of values occur in many different sequences.

Previously seen Values: The number of unique values generated by a program can be used to determine an upper bound on the correct prediction rate of context-based predictors. A context-based predictor never predicts a value it has not previously seen, so each unique value is guaranteed to produce a misprediction.

If we assume there is no sharing of values among instructions, then the number of unique values per instruction represents the lower bound for misprediction. The ratio of the difference between the total number of values and the number of unique values generated by the predicted instructions, to the total number of times the instructions execute is shown in Table 3.5. The results suggest that the number of unique values generated are small as compared to the number of times instructions execute. However, a small number of unique values does not directly imply high predictability, on the contrary, in our experimentation the results show that one of the benchmarks with the smaller number of unique values (go) was one of the most difficult to predict.

A comparison of the correct predictions by context-based predictors (Figure 3.3) and the upper bounds for correct prediction in Table 3.5, shows that the obtained predictability for the different benchmarks is within 5–28% of the upper bounds.

Type	comp	gcc	go	ijpeg	m88k	perl	xlisp
All	74.5	94.2	98.4	96.9	98.9	98.2	99.8
AddSu	72.7	93.0	98.3	95.4	98.7	97.0	99.7
Loads	70.2	92.6	98.2	99.0	97.9	98.5	99.7
Logic	74.1	98.8	99.8	95.9	99.9	99.3	99.6
Shift	78.9	98.2	97.3	98.0	99.8	98.9	99.9

Table 3.5: Upper Bound to Correct Prediction

3.5.4 Sensitivity Experiments for Context-Based Prediction

In this section we discuss the results of experiments that illustrate the sensitivity of fcm predictors to input data and predictor order. For these experiments, we focus on the gcc benchmark and report average correct predictions among all instruction types.

Sensitivity to input data: We studied the effects of different input files and flags on correct prediction. The fcm predictor used in these experiments was order 2. The prediction accuracy and the number of predicted instructions for the different input files is shown in Table 3.6. The fraction of correct predictions shows only small variations across the different input files. We note that these results are for unbounded tables, so aliasing affects caused by different data set sizes will not appear. This may not be the case with fixed table sizes.

In Table 3.7 we show the predictability for gcc for the same input file, but with different compilation flags, again using an order 2 fcm predictor. The results again indicate that variations are very small.

Sensitivity to the order: experiments were performed for increasing order for the same input file (gcc.i) and flags. The results for the different orders are shown in Figure 3.12. The experiment suggests that higher order means better performance but returns are diminishing with increasing order.

File	Predictions (mil)	Correct (%)
jump.i	106	76.5
emit-rtl.i	114	76.0
gcc.i	137	77.1
recog.i	192	78.6
stmt.i	372	77.8

Table 3.6: Sensitivity of 126.gcc to Different Input Files

Flags	Predictions (mil)	Correct (%)
none	31	78.6
-O1	76	75.3
-O2	121	76.9
ref flags	137	77.1

Table 3.7: Sensitivity of 126.gcc to Input Flags with input file gcc.i

3.5.5 Confidence for Value Predictions

To maximize gains when using prediction for speculative execution, it is imperative to have high prediction accuracy and infrequent misspeculation. Misspeculation can be mitigated with the use of confidence mechanisms[60], and they are probably essential for value speculation. In this section we report results for a static confidence measure for value prediction. Figure 3.13 shows the cumulative correct prediction corresponding to a given fraction of dynamic instruction count overall benchmarks. The results were obtained by accumulating the correct prediction of static instructions sorted in ascending order according to their misprediction rate. The static confidence measure is shown for the last value, the stride and different order fcm predictors.

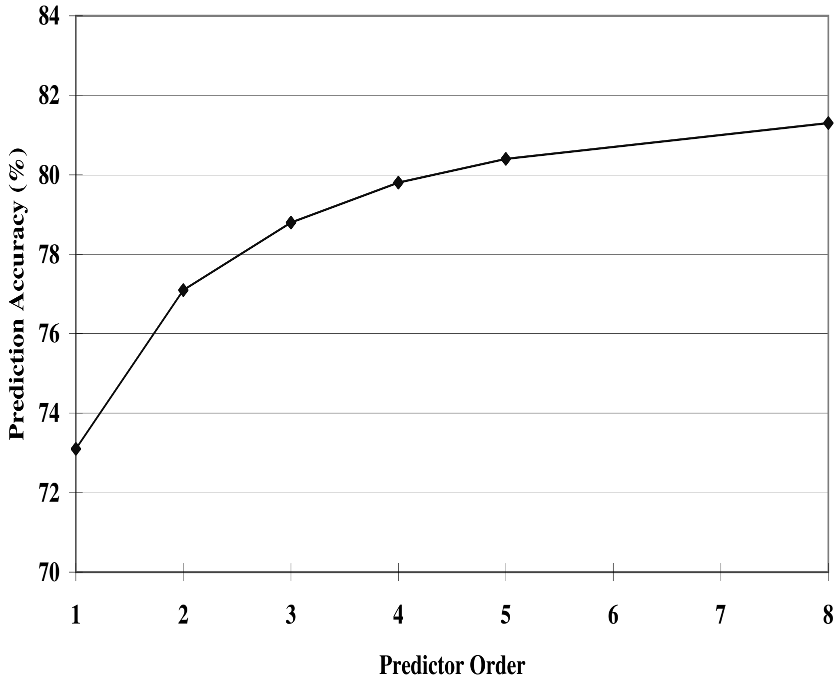

Figure 3.12: Sensitivity of 126.gcc to the Order with input file gcc.i

The results suggest that a large fraction of correct predictions can be isolated with a small fraction of incorrect predictions. For example for fcm order-3 a 6% misprediction can be incurred when predicting 60% of the instructions. This is an encouraging result since it was shown for branches[60] that dynamic techniques can improve significantly over static confidence. Dynamic confidence for value prediction is discussed in Chapter 7.

Another interesting observation is that a significant fraction of predictability is due to instructions that are 100% predictable (15%-30% depending on the predictor). Some of these are usually due to long constant sequences and hence captured by all predictors. Most importantly, it appears that the difference in the prediction accuracy across predictors is due to instructions that have less than perfect predictability.

This is verified in Fig. 3.14 that shows the fraction of predicted instructions with a given prediction accuracy for last value, stride and fcm order-3 predictors. For example, fcm order-3 predicts about 17% of the predicted instructions with accuracy between 81%–90%. The

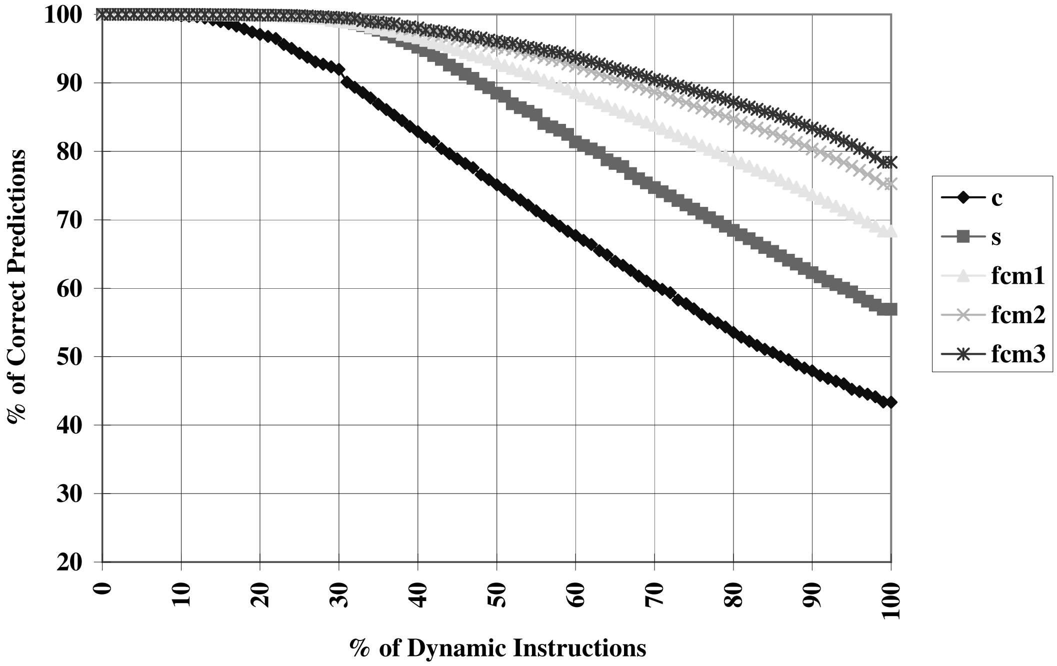

Figure 3.13: Static Value Prediction Confidence

figure indicates an important difference between computational and context-based predictors. That is for computational predictors most correct predictions are due to highly predictable instructions (over 90% predictable) whereas for context-based prediction, significant contributors to correct prediction are instructions with predictability ranging from 60%–90%. This suggests that a classification for fcm predictors will need to allow instructions with low thresholds of predictability to be predicted for a significant improvement to be observed over a computational predictor.

3.6 Summary

We considered representatives from two classes of prediction models: (i) computational and (ii) context-based. Simulations demonstrate that values are potentially highly predictable. Our results indicate that context-based prediction outperforms previously proposed computational prediction (stride and last value), and that, if high prediction correctness is desired con-

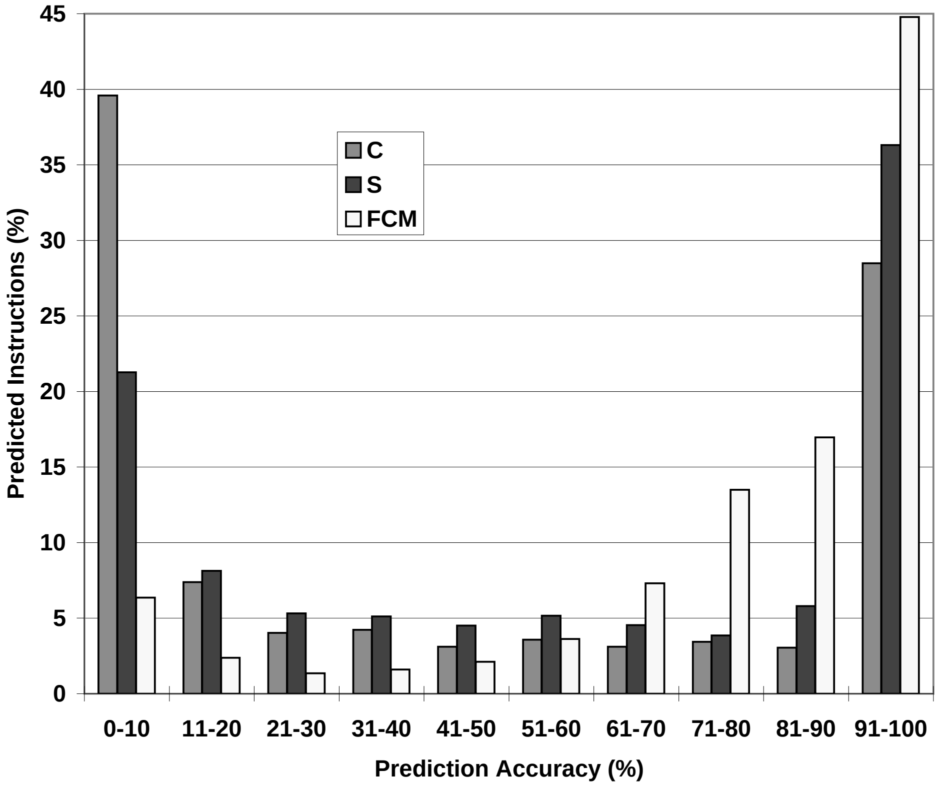

Figure 3.14: Distribution of Prediction Accuracies

text methods probably need to be used either alone or in a hybrid scheme. The obtained results also indicate that the performance of computational prediction varies between instruction types indicating that performance can be further improved if the prediction function matches the functionality of the predicted instruction. Analysis of the improvements of context-based prediction over computational prediction suggest that about 10% of the instructions that generate relatively few values are responsible for over 90% of the improvement. With respect to the value characteristics of instructions, we observe that the majority of instructions do not generate many unique values. The number of values generated by instructions varies among instructions types. This result suggests that different instruction types need to be studied separately due to the distinct predictability and value behavior.

Chapter 4

Modeling Program Predictability

4.1 Motivation

The need for higher levels of instruction level parallelism is pushing high performance processor implementations toward widespread use of prediction and speculation. Ultimately this could lead to substantially new processing paradigms. However, it may be premature to start reaching for radically new paradigms at this point – first some basic research and understanding of program predictability is required. Data, addresses, and control interact in complex ways, and these interactions should be understood before prediction can be fully exploited for increased performance.

We propose a model for studying program predictability. This model, or related models, should prove useful for:

- Understanding the underlying phenomena that lead to predictability.
- Finding relationships among computations that may point to more accurate and more efficient predictors.
- Identifying critical points for prediction; i.e. places where prediction and speculation

may have greater payoff.

- Eventually identifying new microarchitecture paradigms *based* on prediction rather than using it as just an add-on.

We do not achieve all these goals in this chapter – we only take a first step. We concentrate primarily on model development and study the first of the above items in some depth. However, we include discussion suggesting possible directions for pursuing the others. Data value prediction is the primary focus, but we also include control prediction because the interactions between data and control are key for effective prediction and speculation. Further extensions to address and dependence prediction are clearly possible, but we do not pursue them here.

4.1.1 The Dynamics of Program Predictability

Because predictability is based on program behavior, we informally introduce some of the key concepts using a code example. Then, we make some general observations about program behavior that will provide a basis for understanding predictability phenomena.

Fig. 4.1 shows a frequently executed code sequence taken from the *invalidate_for_call* function in the SPECINT95 benchmark *126.gcc*. The code sequence tests bits in a mask that correspond to 64 machine registers; consequently, it is a loop that executes 64 iterations. Beside each instruction is a regular expression that describes the sequence of values produced by the instruction each time the function is called. For instruction 6, the v_i correspond to values not essential to the example. Note that the use of regular expressions is for illustrative purposes, program sequences do not necessarily have to be regular expressions – at least not compactly represented ones.

In the example, register \$6 is initialized by adding the value 0 to itself outside the loop (\$0 = 0, by definition in the instruction set we are using). Each time through the loop, the value

	OpCode	Operands	Values Produced
0	add	\$6,\$0,\$0	**0**
LL1:			
1	srl	\$2,\$6,5	$\mathbf{(0)}^{32}\mathbf{(1)}^{32}$
2	sll	\$2,\$2,2	$\mathbf{(0)}^{32}\mathbf{(4)}^{32}$
3	addu	\$2,\$2,\$19	$\mathbf{(0x1002f8b0)}^{32}\mathbf{(0x1002f8b4)}^{32}$
4	lw	\$2,(\$2)	$\mathbf{(0x8000bfff)}^{32}\mathbf{(0xffffffff)}^{32}$
5	andi	\$3,\$6,31	$\mathbf{(0,1,..,31)}^{2}$
6	srlv	\$2,\$2,\$3	$\mathbf{v}_0,\mathbf{v}_1,..,\mathbf{v}_{62},\mathbf{v}_{31}$
7	andi	\$2,\$2,1	$\mathbf{(1)}^{14}\mathbf{(0)}^{1}\mathbf{(1)}^{1}\mathbf{(0)}^{15}\mathbf{(1)}^{33}$
8	beq	\$2,0,LL2	$\mathbf{(NT)}^{14}\mathbf{(T)}^{1}\mathbf{(NT)}^{1}\mathbf{(T)}^{15}\mathbf{(NT)}^{33}$
LL2:			
9	addiu	\$6,\$6,1	**1,2, .. ,64**
10	slti	\$2,\$6,64	$\mathbf{(1)}^{63}\ \mathbf{(0)}^{1}$
11	bne	\$2,0,LL1	$\mathbf{(T)}^{63}\ \mathbf{(NT)}^{1}$

Figure 4.1: Example Code from 126.gcc

in register \$6 is incremented by one from instruction 9. This means that the values in register \$6 form the sequence 0,1,2,3,...64. This sequence would be predictable by a stride predictor – i.e. the values in the sequence differ by a constant. After the second value in the sequence, a typical stride predictor would recognize the stride and start making correct predictions. Hence, assuming stride prediction, predictability has been *generated* at that point. And this predictability is *propagated* by each of the successive executions of instruction 9.

Instruction 1 also uses the values in register \$6. It shifts each of these by an immediate 5. The resulting sequence of 32 zeros and 32 ones is also largely predictable by a stride predictor (where the stride is 0). Hence, the predictability generated by the stride one sequence at the input of the shift (instruction 1) is propagated through to the output of the instruction. After 32 zero values, however, the output of instruction 1 will change to one. At that point the predictability *terminates*, but is almost immediately re-generated when the sequence of ones is detected by the predictor. Going on, instruction 2 shifts the result of instruction 1 by an immediate 2. Hence, the outputs of instruction 2 are also predictable – the predictability generated by instruction 9 propagates still further. And predictability continues to propagate through instructions 3 and 4, as can be seen by their output sequences.

Predictability is also influenced by communication between instructions. Often, there is direct communication within a basic block. When this happens, the predictability of values as they are produced is essentially the same as the predictability of values when they are consumed. There are other cases, however, where control instructions separate the producer and consumer of a value; for example, if a value is produced outside a loop and consumed repeatedly inside the loop. When control flow separates producer and consumer, the predictability characteristics of the value sequences may differ.

We study program predictability by considering sequences of values consumed and produced by instructions as a program executes. We assume prediction mechanisms that monitor these sequences and attempt to predict next value(s) in the sequence based on previously observed values. Hence, when we speak of *predictability*, we mean the ability of a specified finite state predictor to predict the next value in a sequence. A related, complementary concept is *unpredictability*; however, we focus mainly on predictability. Note that we use a pragmatic definition of predictability: it is related to specific predictors we have chosen to model.

In summary, we take the view that value sequences have characteristic patterns that lead to predictability, and these characteristics can be propagated through instructions and along dependence paths. That is, predictable inputs often lead to predictable outputs. Furthermore, predictable patterns must originate somewhere, and, similarly, predictability can be terminated. It is the relationships among generation, propagation, and termination of predictability that we are interested in studying.

4.1.2 Chapter Overview

The chapter is organized as follows: Section 2 introduces the model for predictability. Section 3 describes the simulation methodology. Predictability definitions for generation, propagation, and termination and simulations results are presented in Section 4. In Section 5, the model is used to study relationships between data and branch predictability. Section 6 summarizes the results and concludes with a discussion of possible ramifications.

4.2 Predictability Model

Thus far, we have discussed predictability and value behavior in terms of static program representations, with sequences of values flowing through the static representation; e.g. the static code and value sequences in Fig. 4.1. One could continue this approach and define a model based on the static program image. If this is done, however, evaluation of "predictability" becomes difficult, because the average prediction accuracy for the sequence generated by a static instruction falls somewhere between 0 and 1. And one must decide at what level of accuracy a static instruction becomes "predictable". By using a dynamic model, each instance of an instruction is either predictable or not. So in terms of the dynamic instances there is no ambiguity regarding predictability. And, by collecting statistics for the entire dynamic execution, we are in effect achieving an averaging over the program.

The dynamic predictability model has two principal components, a predictor and a dynamic prediction graph (DPG). We discuss the DPG in this section and describe specific predictors in Section 4.3.

The DPG is an acyclic directed graph derived from the dynamic dependence graph. Most nodes in the DPG correspond to instances of executed instructions, and a static instruction appears in the DPG as many times as the number of times it executes. A second node type is used to represent *input data values*. That is, values that are not directly computed by the program; these could be statically allocated or be the result of a program input operation. To simplify figures, we generically label these nodes with a **D** for data. *Immediate* values specified in an instruction are considered to be part of the instruction; they are not passed in as data values and are therefore never predicted. However, when collecting data, we keep track of instructions using immediate data because these affect predictability significantly.

A directed arc connects nodes *u* and *v* if there is a true data dependence from dynamic instruction *u* to *v*. The in-degree of a node is determined by the number of instruction source operands, whereas the out-degree of a node is determined by the number of its dependent instructions. Note that for a load instruction the in-degree is the number of address operands plus the number of stores the load depends on.

The predictor component of the model is used to predict the result (output) of an instruction at the time it is produced and predict instruction inputs at the time they are consumed. Note that a result value that is fanned out to the sources of many dynamic instructions is predicted once when it is produced and many times when it is later consumed. Each arc in the graph is labeled with a pair <x,y>, x,y $\epsilon\{p,n\}$. The first element of the pair is associated with the tail of the arc and indicates whether the producing instruction's result was predicted (p) or not predicted (n) correctly. The second element of the pair indicates whether the consuming instruction's source operand was predicted correctly.

D (data) nodes only have output arcs. By definition, all out-arcs from data nodes are labeled <n,y>, y $\epsilon\{p,n\}$; i.e. an initial data item is inherently not predictable. But it may be predicted correctly when used as an input operand by other instructions.

Generation, propagation, and termination of predictability can all be defined in terms of the DPG; this is most easily done pictorially (in Fig. 4.2). We show immediate values (i)

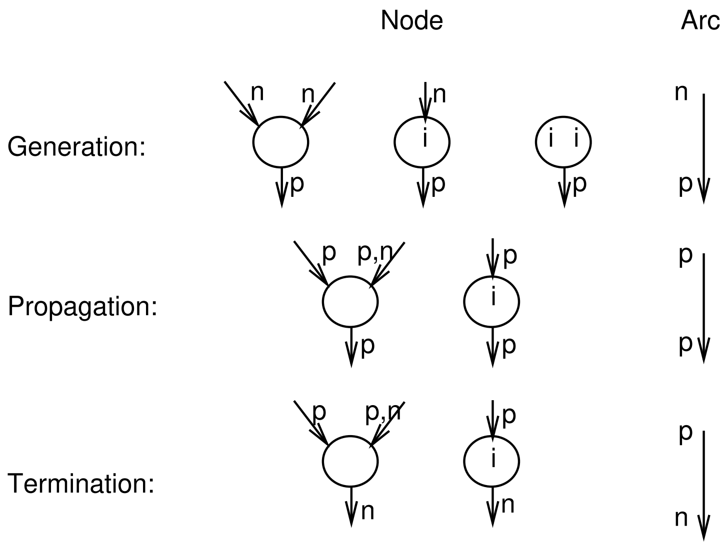

Figure 4.2: Predictability Definitions

inside nodes, as they are part of the instruction. Arc behavior is self-evident. Predictability is generated by a node if it has no correctly predicted inputs, yet the output is predicted correctly. Predictability is propagated by a node if it has at least one correctly predicted input and the output is predicted correctly. Predictability is terminated by a node if its output is not predicted correctly, but at least one of its inputs is. Immediate values may or may not be present in any of these cases.

To illustrate the parts of the prediction model defined thus far, Fig. 4.3 contains a portion of the DPG for the first three iterations of the example code in Fig. 4.1 with the arcs labeled using a stride predictor.

Instruction 1 has an immediate value 5 and takes a value from instruction 0 via register \$6. In the first instance of instruction 1, its output is predicted incorrectly – the predictor's state is such that it does not recognize a pattern in the input values. The first instance of instruction 2 is similar; non-immediate input and output are both predicted incorrectly. However, after the first instance, the inputs and outputs of both instructions become predictable because of generation on the arc between instructions 9 and 1, propagation through instruction 1, and propagation along the arc between instructions 1 to 2.

To illustrate memory operations, instruction 4 is a load instruction that reads a value that was stored earlier. Note that the value being read from memory is considered an input value to the load; the value placed in the result register is considered to be an output.

The model as presented does not directly capture the influence of control instructions on predictability (for example, branch and some jump instructions do not have any data dependent successors). Control instructions are important, however, and we attempt to model them – albeit in an *ad hoc* way. For branch and jump instructions, we depart slightly from a proper graph model by providing output arcs for branch directions and target PCs. However, these arcs do not connect to other nodes. This allows us to determine the influence of data predictability on control predictability.

To model the influence of control on data predictability, we focus on the way control flow affects the passing of data values between instructions. Because of control flow structures such as loops, a particular instance of a static instruction (corresponding to a single node in the DPG) may pass the same data value to multiple nodes in the DPG that all correspond to the same static instruction. When this happens, all such arcs passing the same data value are defined to be *repeated-use* arcs. If there is only one arc passing a value from a node to the instances of a given static instruction, then it is defined to be a *single-use* arc.

4.3 Simulation Methodology

Trace driven simulation was used to build the DPGs for all integer and representative floating point SPEC95 benchmark programs. The simulator is based on the SimpleScalar toolset [74]. Benchmarks were compiled using the gcc compiler provided with the toolset using -O3 optimization.

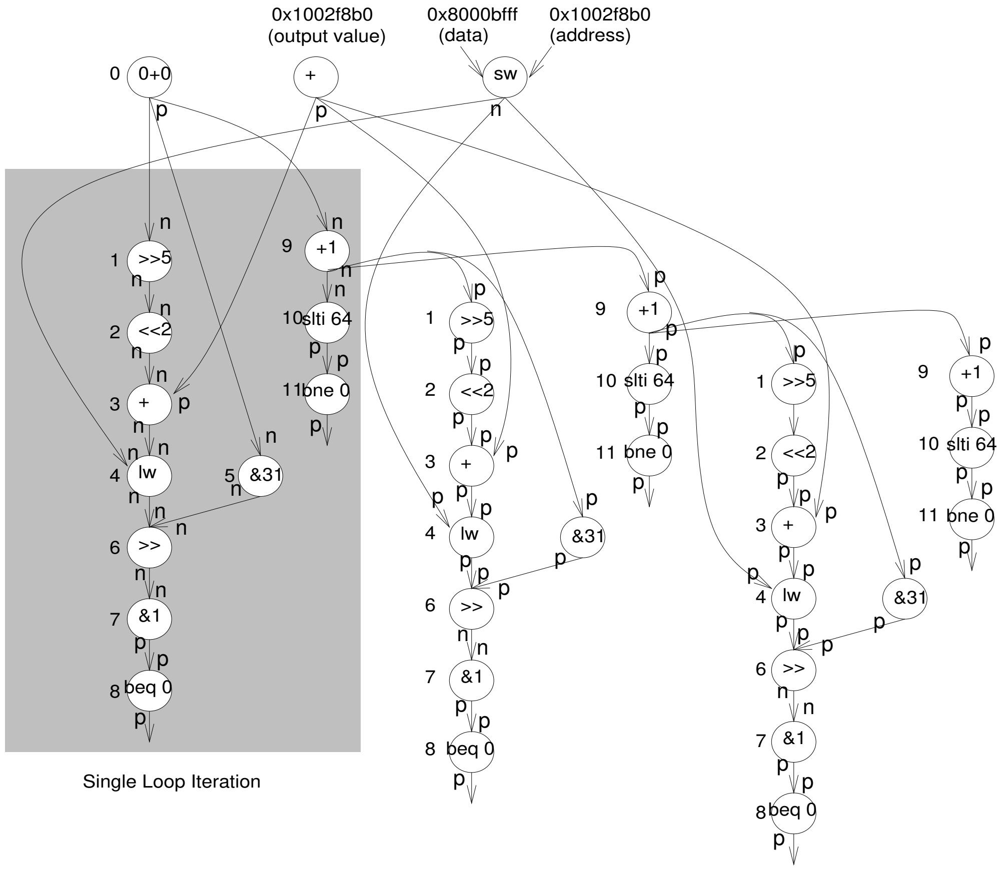

Figure 4.3: DPG for code in Fig. 4.1

DPG statistics are shown in Table 4.1. We note that the ratio of arcs to nodes is about 1.5 for integer benchmarks and about 1.7 for floating point benchmarks. This fraction is a rough indicator of the number of immediate instruction values. The fraction of D nodes is negligible; fewer than .03 percent. Most of the benchmarks have fewer than 1 percent arcs connected to D nodes; the largest fraction is 2.6 percent (for *m88ksim*).

Predictability behavior was studied for three predictors: last-value, stride, and context–based. We chose these three predictor types because the first two are commonly proposed, and because the third (with a large prediction table) has the best prediction performance we have been able to produce thus far.

Benchmark	Input Flags	Nodes (mil)	Arcs (mil)
compress	400000 e 2231	102	142
gcc	gcc.i	200	286
go	9 9	132	183
ijpeg	specmun.ppm	128	196
perl	scrabbl.in	39.8	59.0
m88ksim	modified train	119	161
vortex	modified train	101	149
xlisp	7 queens	199	292
applu	5x5x5 grid	60.0	99.6
fpppp	4 atoms	149	266
mgrid	32x32, 8 iter	119	219
swim	256x256 grid	199	326

Table 4.1: Benchmark Characteristics

The last-value predictor is based on one proposed in [12] with 2^{16} entries and a 2–bit saturating counter replacement policy. In general, it predicts that a value is the same as the last time. The counter provides hysteresis; for example, the prediction value is replaced when the counter indicates two bad predictions in a row.

The stride predictor is the *2-delta* predictor, first proposed for addresses in [49], with 2^{16} entries. This predictor uses hysteresis to replace the predicted stride only when a different stride appears twice in a row.

The context-based predictor is a version of the predictor proposed in Chapter 3. The predictor uses a two–level table, similar to two–level branch predictors [5], and is shown in Fig. 4.4. The first–level table has 2^{16} and the second level table 2^{20} entries. A first–level table is accessed with a truncated instruction program counter (PC). The first level entry holds the last four values (in hashed form) produced by the instruction(s) that reference that

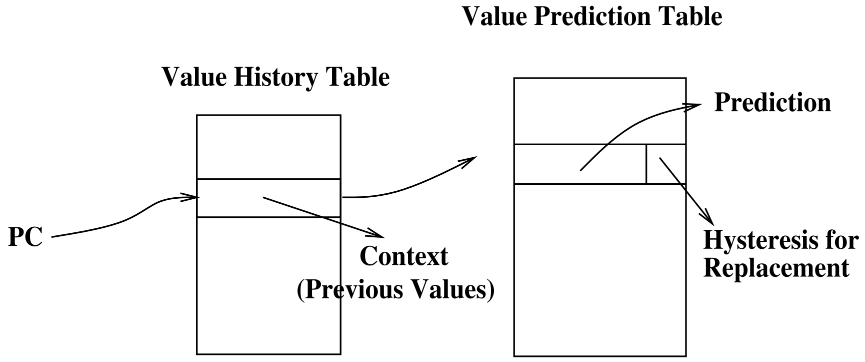

Figure 4.4: Two Level Context-Based Predictor

entry – there can be aliasing in the tables. These four values form a “context” that is used to access the second–level table. The second–level table contains a predicted next value and a 3-bit saturating counter to guide replacement. An important property of the context-based predictor is that the second–level table is shared. This does affect results and was done in part to reduce the memory required by our simulations. It also enables some instructions to benefit from the learned behavior of another instruction, although destructive interference is also possible. Sharing effects will be discussed when they affect results significantly.

Value predictors were used to predict all inputs and outputs of instructions with the exception of conditional branch directions for which a 64K entry gshare predictor is used [7]. The use of a branch predictor enables us to study the relation between branch and value predictability (Section 4.5). This particular branch predictor was chosen because it represents one of the better performers. An alternative would be a two-level predictor that more closely mirrors the structure of the context-based predictor.

For all three prediction methods, we used separate, but identical, predictors for instruction inputs and outputs to prevent direct input/output prediction “short circuits” involving the same dynamic instruction. But, for certain instruction types that always pass an input value through to the output, only the input is predicted and the same predictability is assigned to the output. This was done for memory instructions and register indirect jumps. As a result such instructions never generate predictability.

An important caveat is that the predictors are immediately updated following a prediction. Introducing delayed update timing would have imposed particular implementation idiosyncrasies that may have limited the scope of the results in other ways.

4.4 Simulation Results

4.4.1 Overall Results

In Fig. 4.5, we present the overall generation, propagation and termination behavior of nodes and arcs. To permit relative comparison between node and arc behavior, throughout the chapter the y-axes are expressed as a percentage of the total nodes and arcs. There is a bar for each predictor: last-value (**L**), stride (**S**), context-based (**C**). The graph only shows the average behavior for integer and floating point benchmarks. Averages are calculated as the arithmetic mean. Note that the sum of all arcs and nodes for each case is less than 100%. The missing portion corresponds to the fraction of nodes and arcs that propagate unpredictability; i.e. have all unpredicted inputs and outputs.

All benchmarks appear to follow similar trends, i.e. there is cross-program consistency regarding predictability. Context-based prediction accuracy is higher, as expected. Nodes and arcs tend to generate about the same amount of predictability, and their propagation is also similar – with arcs propagating slightly more than nodes. However, significantly more predictability is terminated at nodes than on arcs; reasons will be given later. Overall, propagation is the dominant predictability behavior. By adding the middle (darker) segments of the node and arc graphs, we see that on average 40%–65% of the nodes/arcs in the inte-

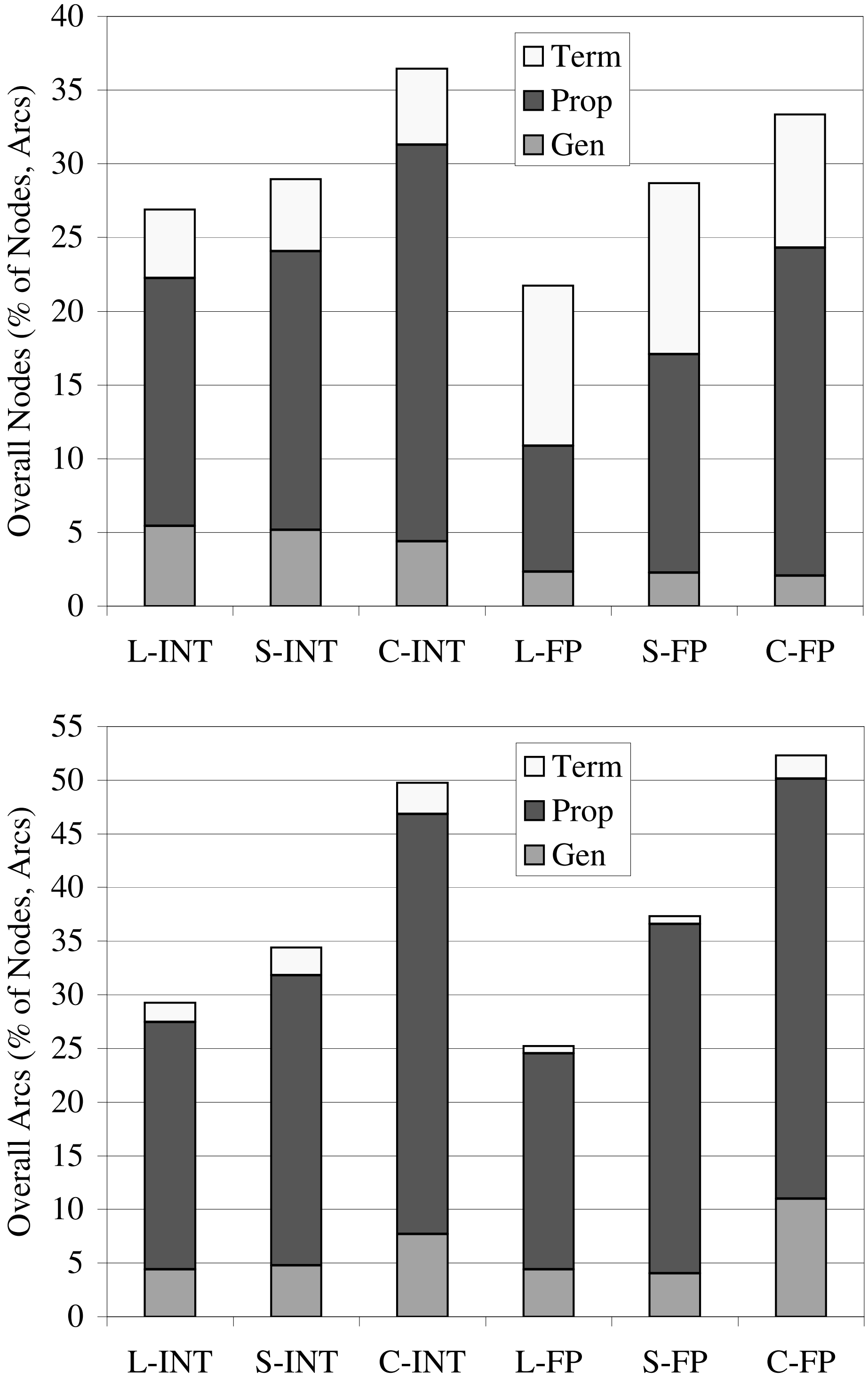

Figure 4.5: Overall Node and Arc Predictability

ger benchmarks and 25%–60% in the floating point benchmarks (depending on the predictor used) propagate predictability.

In the following three subsections, we provide more detailed results for predictability generation, propagation, and termination.

4.4.2 Generation of Predictability

We first consider generation of predictability. Although the generate nodes and arcs comprise a small fraction of all nodes and arcs, an important point is that their overall influence is bigger than their numbers: a single generate can affect the predictability of many instructions via propagation. This is explored further in Section 4.4.5.

Our first conclusion is that predictability is most commonly generated by iterative control flow (e.g. as in loops). In terms of the graph model, iterative control flow appears as repeated-use arcs that generate predictability. In terms of programs, these arcs occur frequently when an instruction outside a loop initializes a memory or a register with a value that is unpredictable at the time it is generated, yet, because of repeated-use inside the loop the value is predictable.

To support our first conclusion, refer to Fig. 4.6. Note that y-axes of both graphs are expressed in the same units: percentage of total nodes and arcs, but the upper graph is scaled to cover a smaller range. In the arc generation graph, the bottom three segments all correspond to repeated-use arcs. With stride and last-value prediction, these three segments account for much more generation than any of the other segments; for context-based prediction, single-use arcs (discussed below) provide about as much. We divided the repeated-use arcs into three segments for more detailed understanding of the phenomena at work.

The bottom segment, denoted <**w1:n,p**>, occurs when a static source instruction producing the value executes only once during the entire lifetime of the program. We call this repeated-use *write-once* control flow. An example of program behavior that can lead to write-once generation is the use of a global data pointer.

The second segment, denoted <**rd:n,p**> , occurs when the source data value comes from program input data (data nodes). We call this *repeated-input use* control flow. Reads from statically allocated arrays can lead to repeated-input use predictability. Generation due to write-once and repeated-input use is important because their corresponding values remain invariant for a long time (often for the entire program execution) and offer potential for static/dynamic specialization.

The third segment from the bottom includes all the other cases of repeated-use, i.e. where the source instruction is executed multiple times, and is denoted as <**r:n,p**>.

Intuitively, repeated-use arcs would most often result in last-value predictability, and in the integer benchmarks this is borne out. Stride prediction and context-based prediction include last-value prediction as a special case, yet they provide similar generation for repeated-use arcs. On floating point benchmarks, context-based prediction does provide more generation, but this is mainly caused by programs that repeatedly scan the same array of input data. Context-based predictors generate predictability at these points.

The second conclusion regarding generation is that single-use arcs also contribute significantly to generation, especially for context-based prediction. These arcs are denoted <**1:n,p**> in the bottom graph of Fig. 4.6. In terms of program structure, these often occur when the producing and consuming instructions are in different basic blocks, and they are separated by some kind of "filtering" conditional branch that converts the unpredictable producer sequence into a predictable consumer sequence. A simple case occurs when a branch tests a relatively hard-to-predict sequence for some constant value. Any consuming instructions on the true path of the branch will see a very predictable (i.e. constant) sequence. Context-based prediction works better than last-value or stride for single-use because of cases where filtering branches test for a range of values. Constructive sharing in the prediction table also helps context-based prediction for these cases.

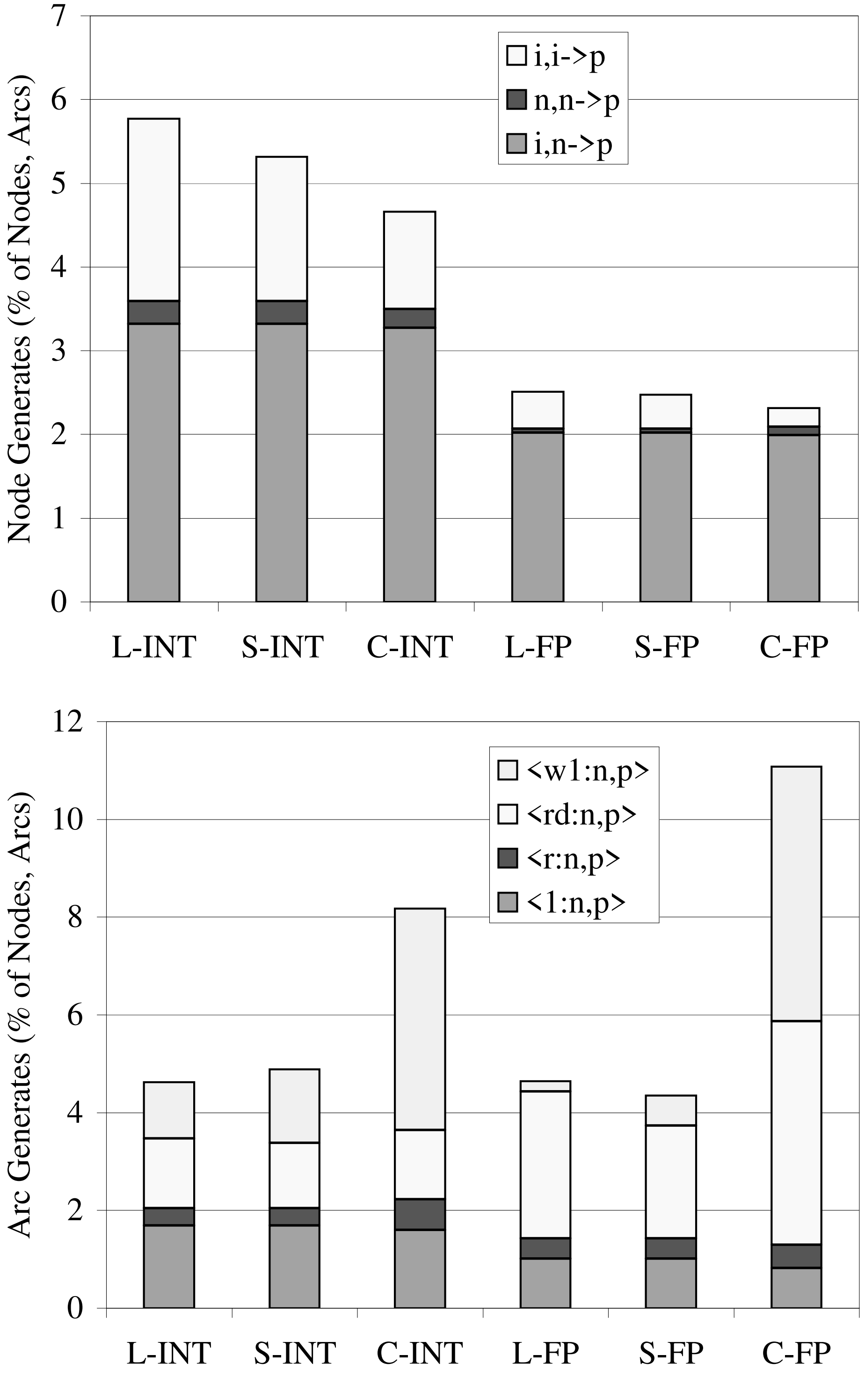

Figure 4.6: Node and Arc Generation

The third conclusion regarding generation is that instructions with all immediate inputs are responsible for most of the predictability caused by instructions. Results for node predictability generation are shown in the top graph of Fig. 4.6. Instructions with only immediate values are denoted as **i,i–> p** . In terms of programs, these occur for load immediate instructions or when instructions initialize a register (typically to zero).

Generation also occurs at some nodes when there is at least one unpredicted input. These cases are denoted with **n,n–>p** and **i,n–>p** in the graph. The majority of these, 70%-95%, depending on the predictor used, are due to branch, compare, logical, and shift instructions. These operations often produce few unique output values and often produce the same value for many consecutive executions. That branches and compares have few output values is obvious. Logic operations often use masks that select a small number of bits, and shifts often have large shift counts that clear out large number of bits (see instruction 1 in Fig 4.1). Often the shift count or logical mask is an immediate, so the majority of the time one of these instruction generates predictability, it involves at least one immediate value.

For node generation, *mgrid* stands out; it has almost no generation at nodes because very few instructions in this benchmark have immediate inputs.

4.4.3 Propagation of Predictability

Most propagation occurs on single-use arcs, denoted as <**1:p,p**> in the bottom graph of Fig. 4.7. This should be expected because single-use arcs include all the dependences between instructions in the same basic block.

Propagation through repeated-use arcs, denoted as <**r:p,p**>, occurs more often in the floating point benchmarks and happens in doubly nested loops where the output value of an instruction in the outer loop is predictable and an inner loop makes repeated-use of that value. Propagation due to <**w1:p,p**> arcs is rare and shown for completeness.

Propagation through nodes most often occurs when all inputs are predictable. These cases

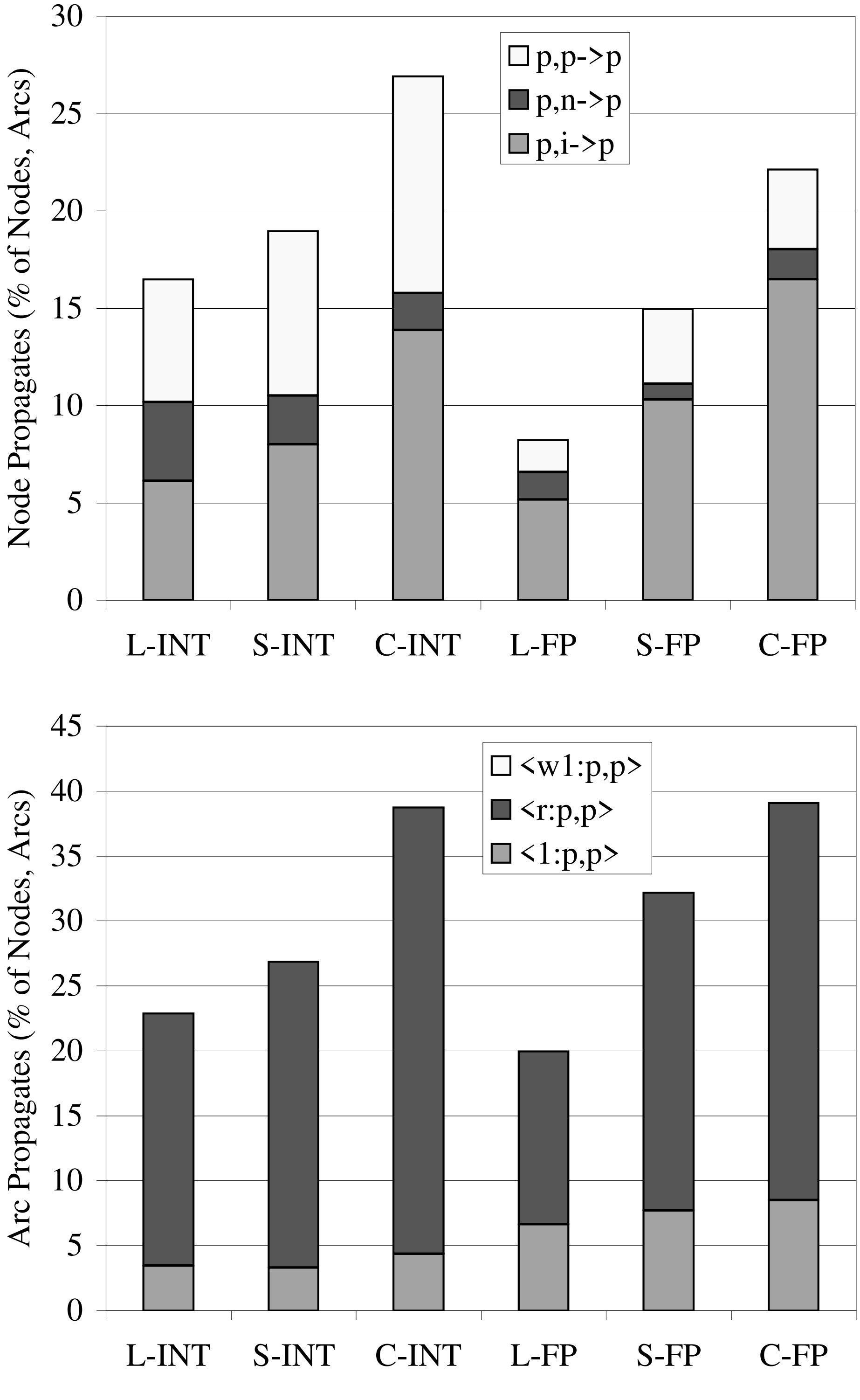

Figure 4.7: Node and Arc Propagation

are denoted **p,p–>** **p** and **p,i–>** **p** in the top graph of Fig 4.7. This is expected behavior – if all inputs of an instruction are predictable, the output is often predictable as well.

Memory instructions are responsible for most of the nodes that propagate predictability and have an unpredictable input (denoted as **p,n–>p**). These propagates occur when data is predictable but the register used to calculate the address is not.

4.4.4 Termination of Predictability

Most termination of predictability happens at instructions where a predictable input is combined with an unpredictable input, and the instruction output is unpredictable. These are shown in the top graph of Fig. 4.8, and are denoted **p,n–>n**. Additional analysis revealed that this is caused primarily by memory instructions with predictable addresses but unpredictable data. The remainder of the **p,n–>** **n** terminations are mostly due to integer or floating add instructions.

Another significant cause of termination occurs because of control flow and is similar to the "filtering" described for the generation of predictability. However, in this case the filtering terminates predictability. That is, the producer and consumer instructions may be in different basic blocks with a conditional branch in between. Even though the producer instruction has a predictable output, the conditional branch only allows the consumer to see some proper subset of the producer instruction's outputs. Because only a subset is observed at the consumer, they may be much less predictable. In terms of the graph model, these are single-use arcs and are denoted as <**1:p,n**> in the lower graph of Fig. 4.8.

Control flow termination also occurs when different instances of the same static instruction consume values produced by different static instructions. The values produced by some of the static instructions are predictable, but their combination may not be. This suggests predictors that combine values with the PCs of producing instructions. Termination due to <**w1:p,n**> arcs is negligible and shown for completeness.

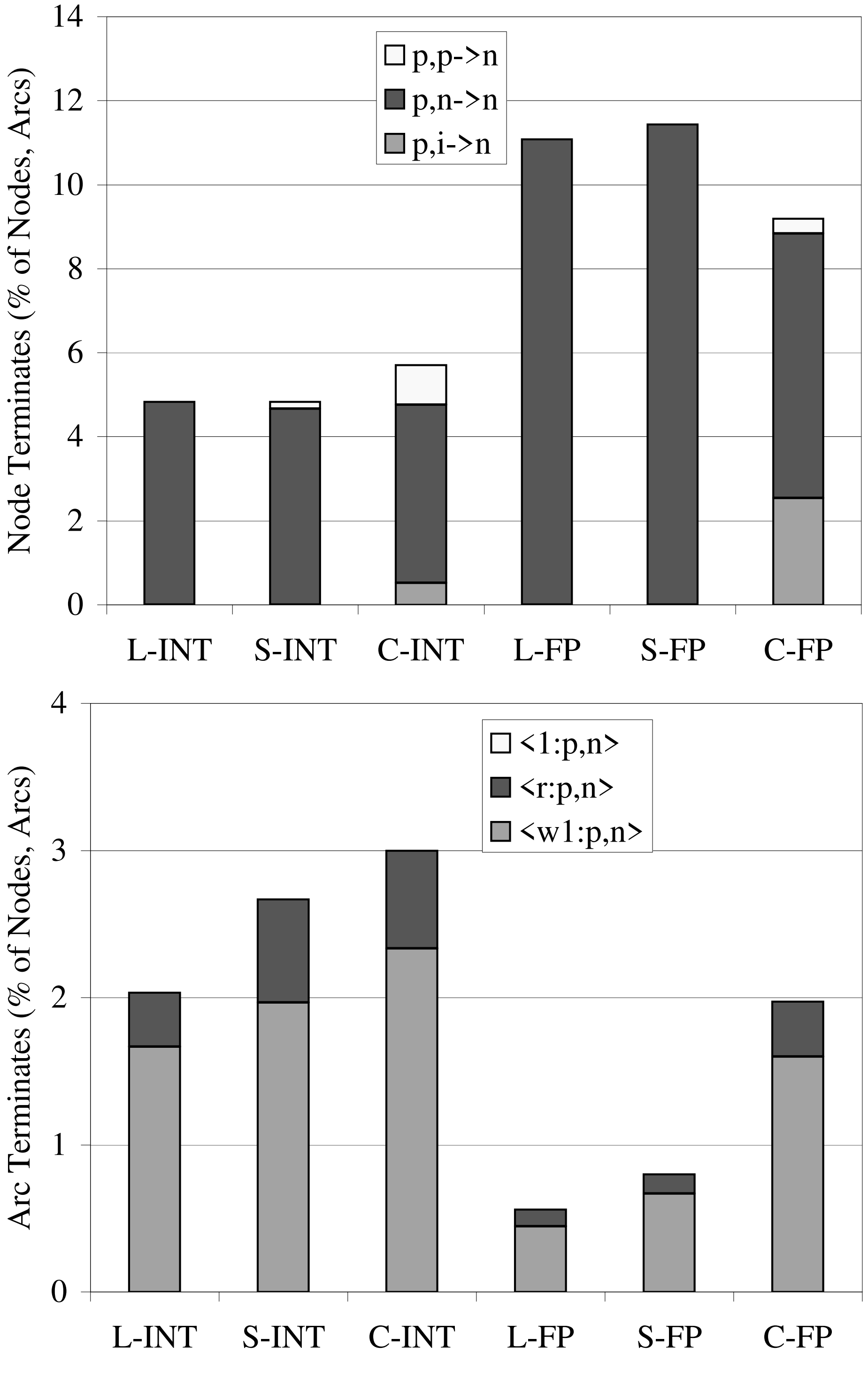

Figure 4.8: Node and Arc Termination

Turning back to nodes, one would expect termination when all instruction inputs are predictable to be rare. And for last-value and stride prediction, they are indeed rare. These cases are denoted **p,p–>n** and **p,i–>n** in Fig. 4.8. For context-based prediction, however, **p,p–>n** and **p,i–>n** nodes are much less rare. This is often due to the limited context history length maintained in the predictor. Consider the sequence that consists of the integers from 0-9, repeatedly. A context-based predictor would successfully predict these values if its history length is one. If, however, this sequence is the input to an AND instruction and the other input is a mask consisting a single 1 in the 4th bit position, the output sequence becomes 00000000110000000011... A context-based predictor with history length one will now yield some mispredictions each time through the repeating sequence. Further analysis (not detailed here) shows this often happens with compare, logical, shifts, and branch instructions. This suggests that when predicting the output values of these instructions, it may be beneficial to correlate with (possibly predicted) input data values of the same or preceding instructions.

4.4.5 Path Behavior

Thus far, we have analyzed the three major components of predictable behavior separately. It is perhaps more interesting to consider their inter-relationships. In particular, we focus on predictable paths. A *predictable path* begins at a generate node or arc, and includes only propagate nodes and arcs. An interpretation of paths is that all predictable values along a path owe their predictability to the original generator(s). Or, conversely, a specific generator may influence a number of predictable values "downstream".

We conjecture that paths indicate potential correlation among values. That is, a predictable value upstream probably correlates well with a predictable value downstream along a predictable path. This can potentially be used to aid in prediction. The distance between the two may indicate how early a correlating value is available, and the paths flowing into a given node may indicate the upstream values that should be used for correlation.

Note that all subpaths of a path are considered as paths. This leads us to analyze all the predictable paths, and for each determine the generating node or arc at its beginning. These, in effect, are the sources of predictability.

In doing the path analysis, we combine some of the generation classes. In particular we consider the following six major classes of generators:

C, control flow, including both <**r:n,p**> and <**1:n,p**>.

D, input data, <**rd:n,p**>.

W, write-once, <**w1:n,p**>.

I, nodes, all immediate inputs , **i,i–>p**.

N, nodes, all inputs unpredictable, **n,n–>p**.

M, nodes, mixed immed. and unpredict. inputs, **n,i–>p**.

Results given in Fig. 4.9 are for all three predictors and are averages over the integer benchmarks. The graphs show the percentages of propagating arcs and nodes on predictable paths beginning at the various classes of generators. The top graph in the figure shows the overall contributions from each generator type. Note that in the top graph, a given arc or node can be included more than once if it is on paths from more than one generator class. The results indicate that the dominant mechanism influencing predictability is control flow. For context-based predictors, generation at single and repeated-use arcs initiate predictable paths that include 45 percent of all arcs or nodes in the DPG. Instructions with only immediate inputs are the second most important generator. With context-based prediction, they initiate paths comprising about 30 percent of the DPG graph.

On the other hand, the predictability influenced by program input data (D) is relatively small. This leads us to the general conclusion that most predictability is caused by program structure and internal data values, i.e. control flow and immediates, and not by program input data.

The bottom graph in Fig. 4.9 shows the contributions of specific combinations of gen-

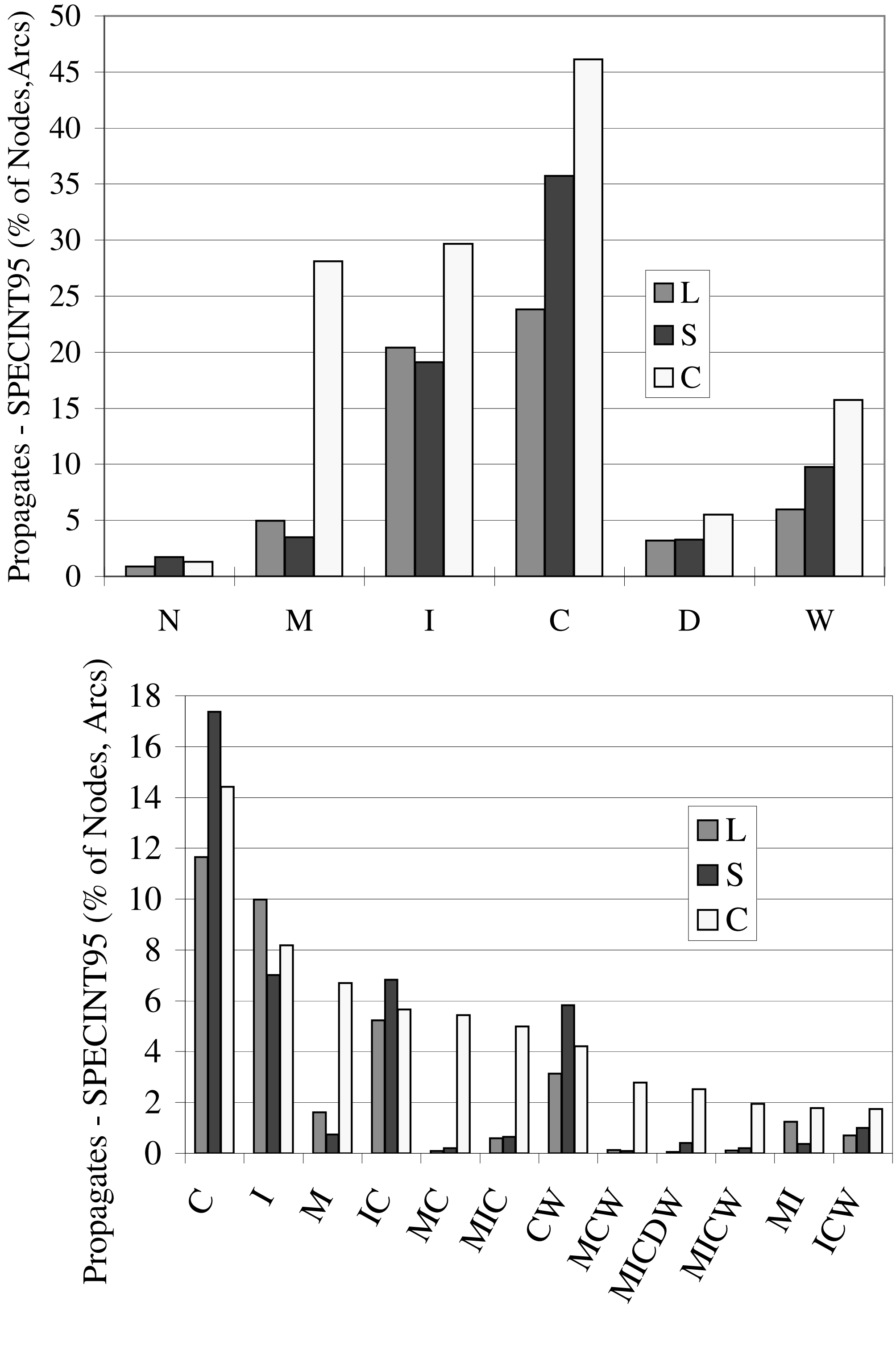

Figure 4.9: Overall and Combinations Contribution of Generates to Propagation

erators; these are the top 24 combinations when sorted with respect to their set sizes for context-based prediction. In this graph, a given arc or node is only counted once. The set labeled "C" is the percentage of arcs and nodes that are influenced only by control flow generation. This is the largest set for all three predictors, containing roughly 12 to 17 percent of the arcs and nodes, depending on the predictor. The set "I" are influenced only by **i,i–>p** generators; i.e. instructions that have not data inputs. These account for 10 percent of the arcs and nodes with last-value prediction, and suggest the use of some form of specialization or dynamic constant propagation. The set "CI" are influenced by both control flow and immediate generators. For two of the three predictors, CI is the third largest set, and underscores the importance of control flow and immediates. For the context-based predictor, CI is the fourth largest set. The set of "M", nodes with mixed unpredictable and immediate inputs is slightly larger; this again indicates the influence of immediate values.

We now expand our analysis to predictable trees. That is, a generator node or arc is at the start of multiple predictable paths, which, collectively, form a tree. For this analysis, we concentrate on a few specific benchmarks.

Fig. 4.10 illustrates some of the characteristics of the predictability trees for the benchmark *126.gcc* using the context-based predictor. The curve marked "trees" shows the lengths of the longest paths within predictable trees. This is a cumulative graph, so, for example, about 90 percent of the generates form trees with longest paths containing 8 or fewer propagating nodes and arcs. This graph indicates that *most* of the predictable trees are relatively shallow. However, a few of the trees are very deep, and include a large number of the propagate nodes and arcs. This is shown in the second curve labeled "aggregate propagation". Aggregate propagation is the total number of nodes and arcs in all trees. This graph shows, for example that 80 percent of aggregate propagation is due to trees with longest path length 256 or more. Hence, relatively few generates influence a large proportion of the predictability. This suggests that predictors that correlate on a few points of generation can be used in

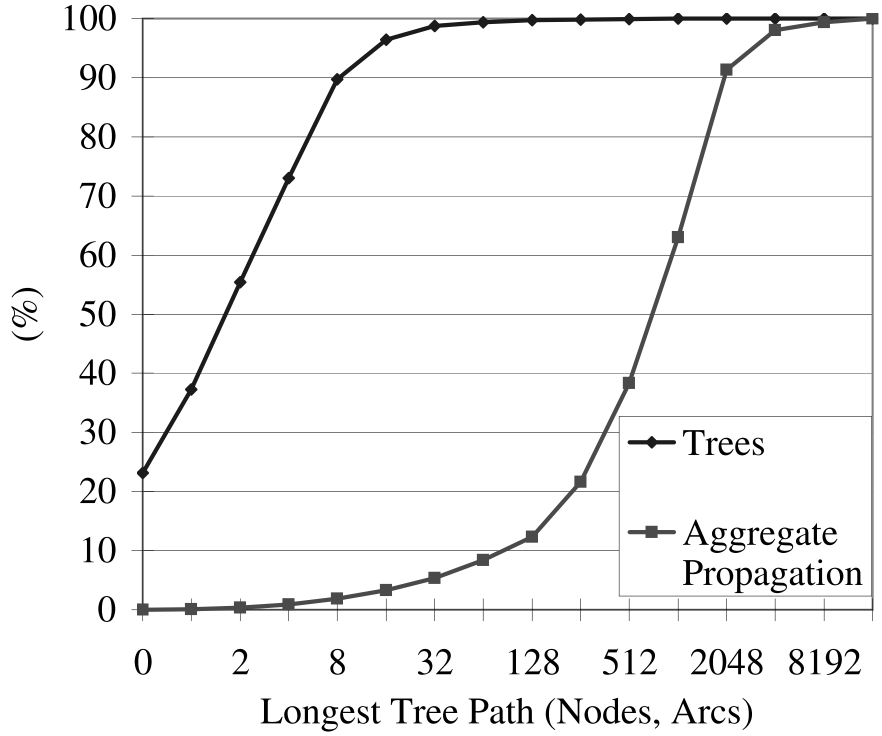

Figure 4.10: Longest Tree Path and Aggregate Propagation

making a large proportion of predictions.

Taken collectively, the predictability trees are intermingled to form the predictable regions. That is, a given propagate node or arc can belong to multiple trees. Hence, one can start at a propagate node or arc and trace back to generator nodes and arcs, and get additional information. The top graph in Fig. 4.11 shows the number of generates that influence a given propagate for the benchmarks *129.compress*, *099.go*, and *126.gcc*, with the context-based predictor. This is also a cumulative graph. The data indicate that about 70%–85% of the propagates are influenced by fewer than four generates. This suggests that the predictability trees are not highly intermingled. It also indicates that to predict a value by correlating with all of its upstream generators, relatively few correlating values would be needed.

The bottom graph in Fig. 4.11 shows the distance between a propagate and the earliest generate that influences it. That is, the longest distance along a propagating path from any generate to the propagate node or arc under consideration. This is also a cumulative graph. For a simple control flow program (loop dominated such as *compress*) about 50% of the propagate nodes/arcs are no more than 64 nodes and arcs away from the farthest generate that influences them. On the other hand, these lengths are greater for complex control programs, *go* and *gcc*, where about 50% of the propagates are influenced by a generate 1024 or more away. Such distances may indicate how early a correlated prediction can be made, i.e. when correlating on the earliest generator.

4.4.6 Predictable Contiguous Sequences

Although these results do not come from the graph model, because they are not based on dependences, it is interesting nevertheless to consider contiguous sequences of predictable instructions in the dynamic instruction stream. These are instructions where all inputs and outputs are correctly predicted. We feel this is of interest because an implementation using predictability may work better on contiguous sequences of instructions rather than single instructions.

Fig. 4.12 shows the number instructions (nodes) contained within instruction sequences of different lengths. This graph includes averaged benchmark data for all three predictors and integer benchmarks.

The data show that fairly long predictable sequences are common with all three predictors. For example, with the context predictor, 13% of the instructions are in blocks of length 9-16. By adding data points, it can be determined that 40% of the instructions, with the context-based predictor, are in sequences containing 9-256 instructions. Last value prediction is also interesting. The data show that many instructions occur in sequences where all instructions have the same inputs and outputs as their previous execution.

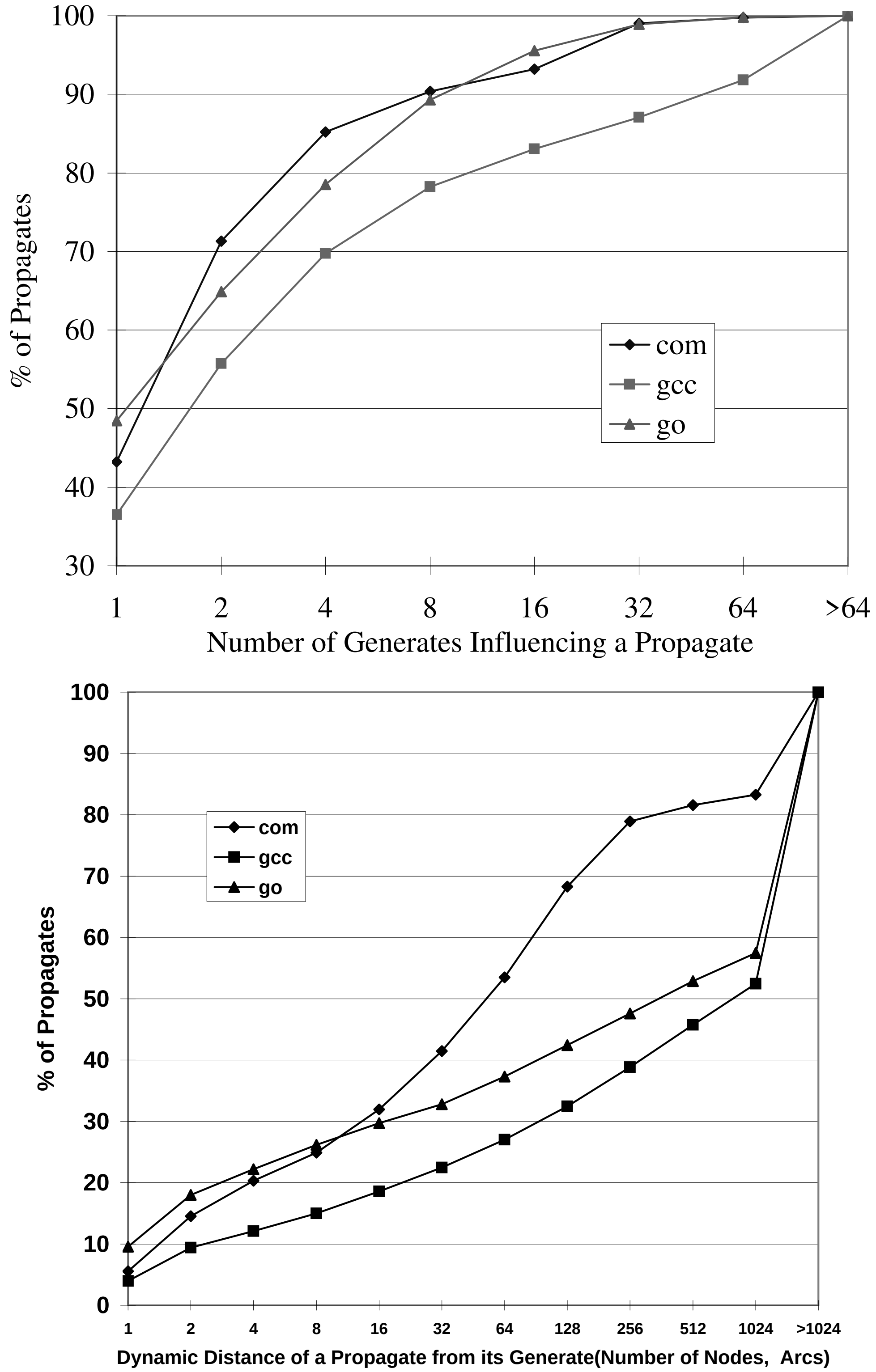

Figure 4.11: Number of Generates Influencing a Propagate and Propagation Distance

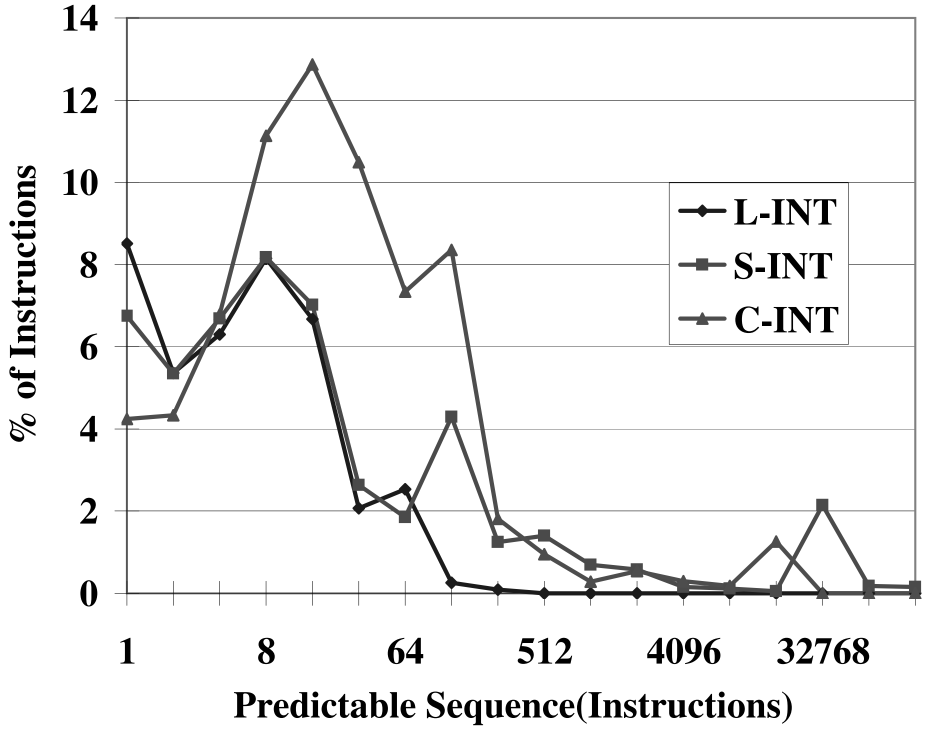

Figure 4.12: Predictable Sequence Length

4.5 Branch Instructions

In this section, we apply the predictability model to conditional branch instructions, and consider ways that data predictability may affect branch predictability.

Fig. 4.13 shows the predictability behavior of all branch instructions for the integer benchmarks. Recall that branch outputs are predicted with a 64K entry gshare predictor. The x axis of the graph is labeled using notation similar to that we have been using. Only branch instructions (nodes) are considered. Branches that generate or propagate predictability are labeled **x,y–> p**, where x and y are either n, p, or i. Mispredicted branches are labeled **x,y–> n**. The overall prediction accuracy of gshare was 93%.

Many of the branch nodes propagate predictability; i.e. when their output is predictable they have at least one input that is value predictable. This is the case for 70%-82% of the

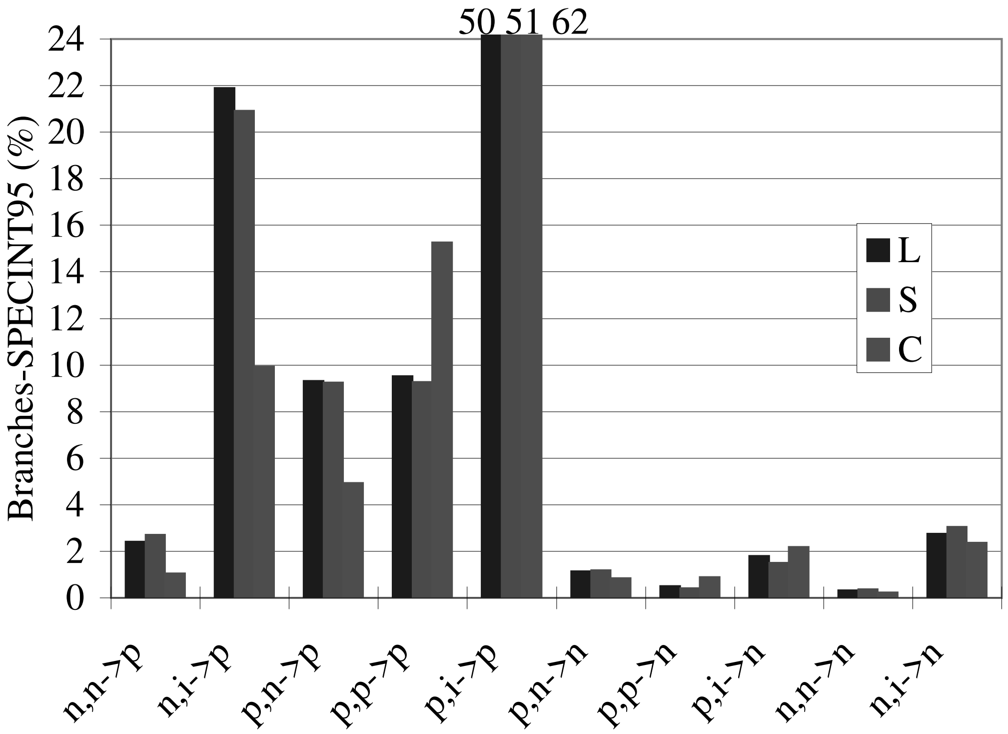

Figure 4.13: Branch Predictability Behavior

branches, depending on the predictor used. Only a small fraction of branches 1%–2% are predicted correctly when all inputs are unpredictable values.

It is perhaps more interesting to consider mispredicted branches. The data show that branch mispredictions when all inputs are unpredictable are relatively rare (fewer than 0.5% of all branches). Branches with only unpredictable inputs and immediate values account for about 2% of the branches. Slightly over half of the branch mispredictions, however, occur when all input values are predictable (**p,p–>n** or **p,i–>n**). This suggests that branch prediction can be enhanced by incorporating data values into the predictor in some form – for example, including input values from previous instances of the same static branch in a history register. Updating the history in a deterministic way would be significant design issue, however.

4.6 Summary and Discussion

We have developed a model for studying program value and control predictability. In terms of the DPG, we described some readily identifiable constructs that lead to predictability generation, propagation, and termination. The relative contribution of the different causes for each behavior is investigated using three different predictors: last-value, stride and context-based. First we studied the three categories of predictability in isolation. We then performed path analysis to determine which are the important sources of predictability. Finally, we considered branch predictability in terms of value predictability. Some of the more significant results are:

- Most predictability can be traced back to program control structure and immediate values. Program input data are a relatively unimportant source of predictability.

- The majority of generates are at the beginning of paths that propagate through fewer than 8 nodes and arcs. However, a very small fraction of generates originate very long paths and influence the majority of the predictability.

- Predictability is often terminated by unpredictable memory data or when a correctly predicted value is combined with an incorrectly predicted value in an instruction. Another significant way that predictability is terminated is through single-use control flow. That is, conditional branches between the producer and the consumer causes irregularity in patterns read from one or many predictable producers.

- Slightly over half of mispredicted branch instructions have predictable input values.

There are a number of possible applications of the model. One application is making better predictors. The model can point to cases where there should be correlation among data and control values. It may be advantageous to feed control information into data predictors and vice versa. Or data values corresponding to different instructions may be correlated; for example, the value at a generate point may correlate with values on paths that it originates. This suggests that occurrences of **p,p–>n** and **p,i–>n** can possibly be exploited to improve prediction accuracy. That is, one could perform output predictions by correlating on predecessor instructions' input values.

The model may also be used to explore new paradigms. For example the large number of **p,p–>p** and **p,i–>p** nodes and <**p,p**> arcs naturally suggest speculation and/or reuse/ memoization of **regions** with predictable nodes and arcs. Regions could possibly start execution early, and/or be precomputed or eliminated all together in hardware and/or software.

Finally, as we did this work, it became evident that unpredictability is as interesting as predictability. For example, in the branch part of the study, insight was gained by considering the behavior of the unpredicted branches. Similarly, study of unpredictable values may give insight into making them predictable; this remains for future research.

Chapter 5

Microarchitecture for Value Prediction

5.1 Motivation and Outline

Previous chapters demonstrated that values produced by instructions are amenable to prediction and established that value predictability is mainly dependent on the internal program structure and immediate values, not input data. In this chapter we try to exploit the observed value predictability for increasing the performance of superscalar processors. This represents an application of value predictability aimed at improving the performance of a current execution paradigm.

The main objective of this chapter is to present an organization for a high performance superscalar processor that supports speculation based on value prediction and discuss its critical components. An important part of this process attempts to parameterize the rather complex design space that is created when integrating value prediction into a superscalar processor. The performance of this organization will be investigated with ideal and realistic predictors in Chapters 6 and 7.

5.2 Proposed Organization

An organization of a superscalar processor without value prediction is shown in Fig. 5.1. This is an out-of-order execution core based on the Register Update Unit[75] which unifies issue resources (reservation stations[76]) and retirement resources (reorder buffer entries[77]). We will refer to this organization as the base–processor and to the unified issue/retirement structure as the instruction window. An entry in the instruction window will be called a reservation station(RS). Below we describe some of the key features of the base–processor.

As instructions are fetched, they are assigned entries in the instruction window according to the dynamic program order and remain in the window until they can be retired. Values in this organization can exist in the following locations: register file, instruction window and functional units. The register file maintains architected state and is only updated when instructions retire. Instructions read any available input values from the register file before entering a reservation station in the instruction window. Instructions can also receive values through a bypassing network from predecessors in the window[75]. Instructions with unresolved dependences monitor the results bus and capture their source operands as the producing instructions finish execution. When an instruction finishes execution and is the oldest instruction in the window, it can write its result in the register file or memory and release its entry in the window.

The important fields in a RS are shown below:

In1Ready	In1Tag	In1Value
In2Ready	In2Tag	In2Value
OutValue	Issued	Executed

The *ready* fields are used to indicate whether an input operand is available. The *tags* are used to specify which results an instruction is waiting for. The *value* field holds the actual input

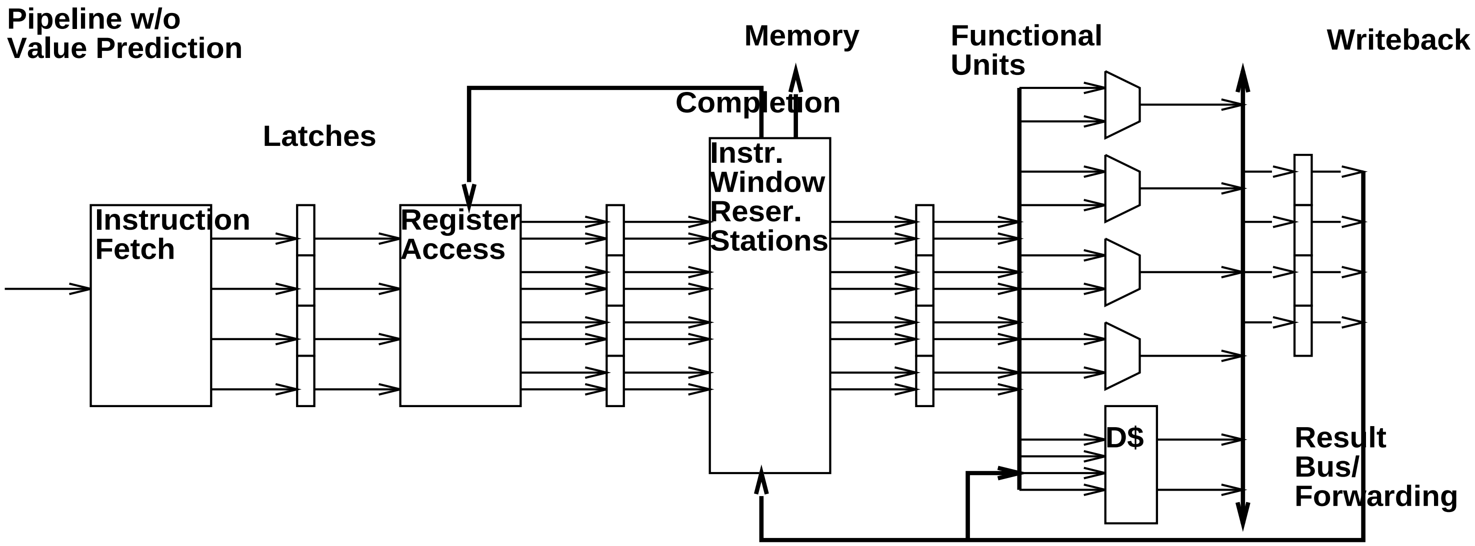

Figure 5.1: Pipeline Without Value Prediction

value. The *outvalue* stores the output of an instruction that is eventually used for updating the processor state when the instruction retires. The two binary fields *issued* and *executed* are used to guide issue and retirement.

Wake up and selection logic, in the instruction window, determines the instructions that get issued each cycle[78]. *Wake up* determines which instructions can be considered for issue in the next cycle. An instruction *wakes up* when its ready fields are set. Selection, based on a scheduling priority and resource availability, chooses which of these instructions issue in the next cycle.

Memory instructions are split into two operations: address generation and memory access. Loads are executed when all preceding store addresses in the instruction window are known and hence no memory dependence violations can occur. The only source of mispeculation in the base-processor is due to branch prediction; when a branch is mispredicted all subsequent instructions in the window are squashed and fetching starts from the new target address.

The base–organization is augmented to support value prediction as shown in Fig. 5.2. Essential to any organization that supports prediction are mechanisms that:

- provide predictions,
- verify predictions, and
- recover from mispredictions.

The new components that enable the above are the *value predictor*, *prediction register file*, *prediction renamer*, and *verifier*. This Chapter considers the design of these components, the ramifications of supporting value prediction in the proposed organization, and discusses different options for implementing value prediction related policies. Although our discussion is mainly for the above organization, we believe our conclusions are applicable to other organizations. The rationale is that the aforementioned value prediction components are likely to be necessary for any processor that supports value prediction and speculation.

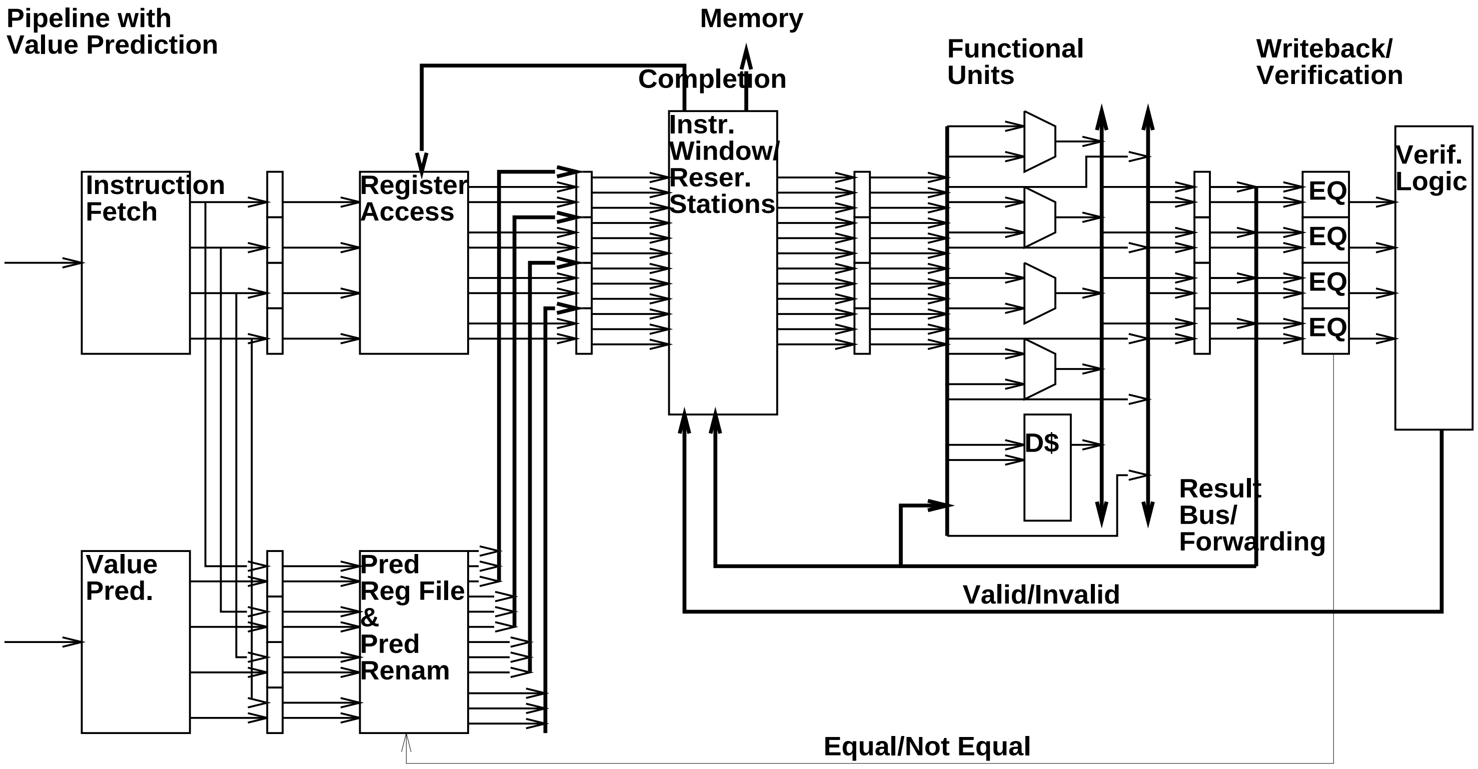

Figure 5.2: Pipeline With Value Prediction

5.3 Providing Predictions: Value Predictor, Prediction Register File and Prediction Renaming

The value predictor is used to provide predictions for **output values** of instructions. Outputs are preferred over inputs since, as was shown in Chapter 4, for non-memory instructions more often instructions have predictable outputs than inputs. We postpone the discussion of the implementation of the specific predictors we considered until Chapter 7. The predictor is accessed in the same pipeline stage the instruction cache is read, and obtains as many predictions as the fetch width of the processor (bandwidth and timing issues are also discussed in Chapter 7).

A prediction is used in two ways: it is forwarded to the inputs of dependent instructions fetched in the same cycle using *prediction renaming*, and is stored in a *prediction register file*(see Fig. 5.3). Each prediction register files entry holds the most recent prediction made for a given register. Valid values in the prediction register file are used by instructions in succeeding fetch cycles that depend on a predicted register.

A prediction register file access and prediction renaming are performed during the second pipeline stage while the register file is accessed. Prediction renaming is used to determine the most recent prediction that an input operand should use. The inputs for prediction renaming are current predicted values and confidence from the predictor, and predictions and their valid bits from the prediction register file.

The valid bit in the prediction register file is set/reset based on the confidence of predictions, and is reset by predicted instructions after they have executed and their outcome is not equal to the predicted value. This prevents instructions from using a mispredicted value and hence can reduce mispeculation. Both the prediction register file and prediction renaming are indexed with logical register names of instructions and hence can not be accessed before instructions are fetched.

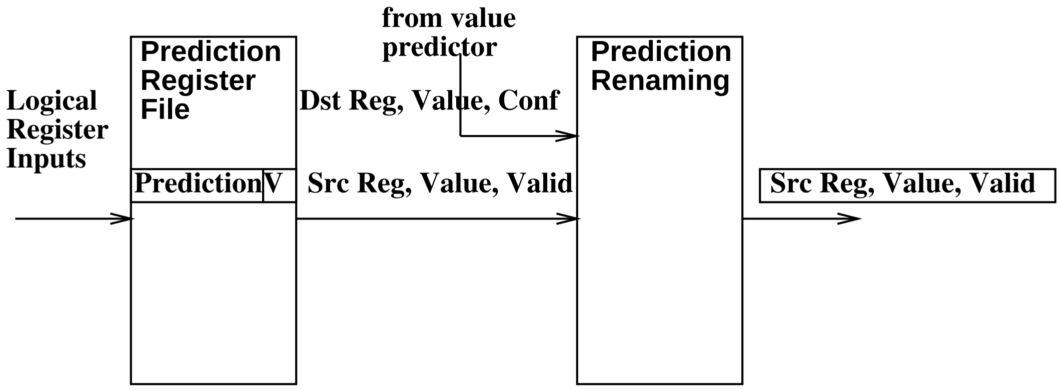

Figure 5.3: Prediction Register File and Prediction Renaming

Finally, valid predicted values are merged with architected state read from the register file as well as any values from completing instructions or instructions in the window, and written in the reservation stations. It is important to note that values from the predictor need to become available when prediction renaming is performed — this means that the access time for the value predictor can be multi-cycle.

We believe that the complexity of performing a prediction register file access and prediction renaming are comparable to register renaming[78] and hence does not introduce additional design complexity. We note that an alternative design – but logically equivalent – is to maintain predicted output values and valid bit information in the reservation stations. We choose to present the organization with the separate prediction register file and prediction renaming for conceptual simplicity.

5.3.1 Values: Valid, Speculative and Predicted

The use of value prediction introduces two additional types of values in the processor: *predicted* and *speculative*. A value is *predicted* if it is the one obtained directly from the value predictor, and is *speculative* if it is the result of computation(s) that involved a predicted value. An input value is *valid* if it is read from the architected file or it is the result of computation that involved only valid inputs. Therefore, with value prediction an input operand may be: speculative, predicted, valid, and invalid. Predicted and speculative values exist in the reservation stations and functional units but not in the architected register file.

Depending on the implementation of value prediction and speculation, the different types of values may require additional state to represent them. To accommodate the new value types in the proposed microarchitecture, the ready fields in the RS are expanded to two bits. Another required change is for checking the correctness of output predictions; a field is added in the RS to indicate whether the output of an instruction is predicted and another field is used to contain the predicted output value. The modified RS incorporating the above changes is shown below:

In1Ready	In1Tag	In1Value
In2Ready	In2Tag	In2Value
OutValue	Issued	Executed
PredictValue	Predicted	

The way the different flags are used to guide issue, invalidation, and verification policies are discussed in following sections.

5.4 Wake Up and Selection with Speculative Execution

The use of value–speculation introduces additional options in the design space of the issue logic. In this Section we discuss some of these options for wakeup and selection.

Wake Up: Without value–speculation an instruction monitors the result bus for its source tags. Tags are broadcast on the result bus by instructions that complete execution. An instruction can wake up if it has received the tags for all its input operands – this indicated by the ready flags being set in its RS.

With value–speculation, wake up may need to consider both tags and other information, such as whether the instruction already issued and the state for each input value (recall that an input can be invalid, speculative, predicted or valid). Wake up logic may also consider the confidence state of predicted and speculative values. The above creates a number of possibilities as to when to wake up instructions. One of them is considered next for the proposed microarchitecture.

The underlying method used for broadcasting tags with value-speculation can be identical to the one used for the base–processor: as instructions complete execution they broadcast their tags on the result bus(es). However in addition to the tags, instructions may need to broadcast their tag state. The state of a tag for an instruction is a function of its inputs and its predicted state. One such function is described in Section 5.6.5 - see *instruction verification state*. This state can be sent, along with the tags, on the results bus as instructions complete execution or come from the verification hardware (see Section 5.6.5). In this work, an instruction can wake up when its inputs are either valid and/or speculative and the instruction has not yet issued. The information as to whether an instruction has issued can be found in the issued field of its RS. In some of the models we consider, the issued field is required to be reset whenever an instruction is *invalidated*(see Section 5.6.5- selective invalidation).

We note that the above ensure instructions without predicted or speculative operands can wake up as fast as on the the base-processor.

Selection with speculative execution presents new trade-offs since instructions can be candidates for issue with predicted, speculative and/or valid inputs. It is therefore important to determine the priority order to be assigned among the different ready instructions based not only on their program order or type but also on information such as, but not limited to, how many speculative inputs an instructions has and confidence in those inputs.

For example, a selection scheme that is combined with a very good confidence estimation can ignore the speculative state information since most of the time predictions are correct. On the other hand, selection with a confidence estimator that often results in mispeculation may assign higher priority to all instructions with valid inputs and then consider instructions with speculative operands.

We feel selection for speculative execution is an important topic for research but will not be addressed in this thesis. In this work we consider a selection scheme that assigns highest priority to branch and load instructions and prioritizes the rest instructions based on dynamic program order - oldest first.

5.5 Branch and Memory Instructions

Branch instructions: With value prediction branch instructions can issue using speculative or deterministic values. This presents a performance tradeoff as to whether to resolve branches early (speculatively) or late (deterministically); this issue was recently addressed in [79]. There are two cases to consider with speculative branch resolution: (a) when the predicted and the speculative result are the same, and (b) when they are different. For the former case, speculative resolution offers no benefit. For the latter case, performance can be improved when the speculative resolution is correct and degraded when it is incorrect.

Combining speculative and deterministic resolution (*selective speculative resolution*) is another option. A possible mechanism for selecting between speculative or deterministic resolution can be based on learning and confidence. An implementation of selective speculative resolution for branches was explored recently by Rotenberg et al.[80].

In previous work [81, 15, 58, 82] branches were resolved speculatively only when they had correct predicted/speculative operands. Rotenberg[69, 11] resolved branches only when their operands were known to be non-speculative. Sodani et al.[79] studied the performance for two models in combination with value prediction: the first resolves branches whenever they are ready (even with value speculative operands), whereas the second waits for branch operands to be non-speculative before resolving the branch.

The design of effective speculative branch resolution mechanisms presents interesting possibilities for future research. In this work we choose to resolve branches only after their inputs are found to be valid and hence our performance numbers may be limited in this respect.

Memory Instructions: An interesting distinction between data dependences through registers and memory is that the former dependences can be established following register renaming, whereas the latter are not known until memory instructions have computed their addresses. Consequently not allowing memory instructions to issue out of order can degrade performance[18, 19] since establishing memory dependence relations between store and load instructions may require many cycles. To alleviate the effects of memory dependences, dependence prediction[18, 19] can be employed to predict whether a load depends on a store in the window, and address prediction can be used to accelerate the process of synchronizing aliased store–load pairs in the window, or accessing memory. This also represents an interesting direction for future research. A recent study [83] considers the combined benefits of address, value and dependence prediction.

In this work we choose to disallow load instructions from accessing memory speculatively. This may represent another limitation of our study from achieving higher performance.

In summary, our study may not exploit the full potential of value prediction since branch and memory instructions were disallowed from being resolved speculatively. Determining mechanisms that enable the effective exploitation of speculative resolution of branch and memory instructions represents an interesting direction for future research.

5.6 Recovery and Verification

The use of prediction and speculation requires mechanisms to:

- check predictions (**equality**),
- recover in the case of a value mispeculation(**recovery**), and
- inform direct and indirect successors of a correctly predicted instruction that their operands are valid (**verification**).

We first discuss the three mechanisms separately and then consider them in combination for presenting a few speculative execution models with different verification and recovery parameters.

5.6.1 Equality

Equality is responsible for determining whether a value is predicted correctly or incorrectly. Equality can be performed by comparing the predicted value against the actual value (**value equality**). As far as we know, value equality is the approach used for checking predictions by all proposals that rely on prediction and speculation. An alternative to value-equality is discussed in Chapter 8.

The latency for performing value equality is a critical performance parameter because it influences how quickly recovery or verification actions can be performed. This latency has direct ramifications on many processor parameters that will be discussed in Section 5.7. In the proposed microarchitecture (see Fig. 5.2) value equality is performed in the write-back/verification stage using comparators (denoted EQ).

5.6.2 Recovery

Recovery is responsible for removing(*invalidating*) the side-effects caused by a mispredicted instruction. There are two basic recovery models to consider: **complete invalidation** or **selective invalidation**[15]. Complete invalidation, invalidates all instructions in the window that follow the mispredicted instruction. This approach is employed by virtually all current processors for recovering from branch mispredictions. Complete invalidation can also be employed in the case of value mispredictions. This may be practical to implement and beneficial to performance if value mispredictions are rare. Value mispredictions can be made rare using confidence prediction[60] at the possible expense, however, of fewer confident predictions and hence lower performance gains.

An alternative to complete invalidation is selective invalidation where only instructions that depend, directly or indirectly, on the mispredicted instruction get invalidated. Selective invalidation is probably most effective if instructions remain in their reservation stations - or in a buffer from which they can issue - as long as they have speculative operands[15]. This enables instructions when they are invalidated to reissue faster since they do not need to be refetched.

Selective invalidation can be performed "serially", "parallel" or in some intermediate fashion. Serial invalidation means than an instruction can only invalidate its direct successors. The invalidated successors then invalidate their own successors. This process is repeated until all successors get invalidated. Serial invalidation is likely to yield lower performance because dependent chains of instructions will likely require multiple cycles to get invalidated while their successors may needlessly use resources. The main advantage of serial invalidation is that it can be implemented with only minor changes in many processors, based on their existing mechanism for transmitting values and tags of instructions that complete execution. Parallel invalidation means that a mispredicted instruction can invalidate simultaneously all its successors (direct and indirect). This scheme can be faster than serial invalidation, contingent on technology constraints (parallel invalidation is discussed further in Section 5.6.5). Examples of other recovery schemes include invalidating successors up to a certain dependence chain depth or limiting the number of instructions that participate in a given invalidation transaction.

Recovery latency is performance critical because it determines how soon after instructions get invalidated they can be considered for reissue. We will discuss the manifestations of recovery latency in Section 5.7 when we discuss speculative execution models. The choice for a particular recovery and invalidation model is expected to be influenced by the particular processor organization traits that make one recovery scheme more amenable than others. In this work we only consider selective invalidation in combination with parallel invalidation. In this scheme each time an instruction is invalidated, it resets its issue field in order to be consider for reissue. We choose these policies because they represent the ones with the highest performance potential.

5.6.3 Verification

Verification is responsible for informing the direct and indirect successors of a correctly predicted instruction that their operand(s) are valid. An instruction that executes using predicted or speculative inputs may not be able to release a resource that is assigned until it is verified. Fast verification is important, therefore, because it can reduce the time an instruction occupies a resource and leads to faster overall processing.

Verification can be performed serially, in parallel or in some intermediate form. Serial verification means that a correctly predicted instruction can only verify its immediate successors. The verified successors will then verify their own successors. This process will be repeated until all successors get verified. Serial verification may provide a performance advantage if it does not reexpose all dependences that were eliminated by prediction.

Parallel verification can be performed at least in the following two ways:

- based on the existing retirement hardware in place in most processors (discussed in Section 5.6.4), and
- using dedicated verification hardware that informs the successors of a predicted in-

struction that their operands are valid as soon as the predicted value is known to be correct(discussed in Section 5.6.5).

Verification based on retirement, as we show next, requires minimal hardware changes to the retirement mechanism that is already in place in most processors. However, instructions with speculative operands cannot be verified until they can retire. This may present a disadvantage as compared with a dedicated verification hardware, at least in the following three cases:

- when branches are not allowed to be resolved speculatively,
- when memory instructions are not allowed to be resolved speculatively, and
- when selective invalidation is used on a processor where the issue resources are separate from the retirement resources.

It may not be desirable to resolve branches using speculative data if this leads to incorrect branch resolution. Therefore when branches are not resolved speculatively and verification is based on retirement, a misprediction due to a branch may not be detected until a branch retires. This may come many cycles after the speculative branch operands become valid and hence reduce performance gains from value prediction. Similarly preventing memory instructions from being resolved speculatively can force them to wait many cycles after their operands become valid before accessing memory.

When value prediction is used in conjunction with selective invalidation, it may be better for instructions with speculative operands to remain in an issue resource (for example a reservation station) until they are verified[15]. With verification based on retirement, a speculative instruction will not be able to release its issue reservation station until retirement. This will present a disadvantage if the number of issue entries are smaller than the number of retirement entries (reorder buffer entries) because the retirement resources will be underutilized when instruction fetching is blocked by full issue resources. This problem is not present in organizations with unified issue and retirement structures like the microarchitecture we considered here. Such organizations, however, have their disadvantages since large unified structures could be complex and/or slow.

Consequently, when the only function of verification is to determine when an instruction can retire then dedicated hardware verification does not provide a performance advantage over verification based on retirement.

Similar to recovery, we expect the choice of a particular verification model to be influenced by specific processor characteristics. In this work because we do not allow branches and memory instructions to be resolved speculatively, we use the best performing option for verification: parallel verification using dedicated hardware (discussed in Section 5.6.5).

It is noteworthy that selective invalidation and verification perform similar functions - inform the direct and indirect successors of a predicted instruction about the correctness of the prediction. This observation is used in Section 5.6.5 in the design of a unit capable of performing parallel selective invalidation and parallel verification. We note however, that this is not necessary for a unit that performs verification to perform invalidation. For example, it is possible to perform parallel verification based on the retirement mechanism and serial recovery based on the result bus mechanism.

5.6.4 Verification Based on Retirement

A typical optimization in high performance processors is to allow instructions to complete out of order. This optimization introduces the problem of imprecise interrupts because instructions can retire out of order[77].

The reorder buffer[77] is a mechanism proposed to enforce in order retirement of instructions when instructions are allowed to complete out of order. The reorder buffer is a queue with finite number of entries where instructions are assigned an entry in sequential program order as they are fetched into the processor. An instruction can retire when it has *completed* execution and is at the head of the reorder buffer. If the retirement bandwidth, *j*, is greater than one, then the i_{th} instruction in the retirement list, where $i \leq j$, can retire if all previous *i - 1* instructions have completed. The retirement of an instruction is associated with the release of its reorder buffer entry. Retiring instructions as early as possible can affect performance because a full reorder buffer blocks instruction dispatch.

Central to retirement is the condition that determines when an instruction has completed execution. In a processor without prediction and speculation, an instruction is defined to have completed when it finishes execution.

Branch prediction requires an additional completion condition. That is, a predicted branch is defined to be complete if it has finished execution and its predicted target matches its computed target. Instructions that follow a branch (speculative instructions) that complete ahead of the branch cannot retire unless the branch completes. However, a branch misprediction can be detected before a branch is ready to retire and may need a recovery action which affects the function of the reorder buffer. Most current processors perform a complete squash following a branch misprediction which requires the resetting of the tail of the reorder buffer to the entry that follows the mispredicted branch.

With value prediction and speculation, the completion condition for predicted instructions is similar to predicted branches. An instruction is defined to be complete when it finishes execution and, if it is predicted, the prediction matches its computed value. Note that this is a sufficient condition for retiring multiple predicted instructions with any dependence relation (all independent, all in a dependence chain, etc). This is the case because if there retirement bandwidth *j*, which is greater than one, and the first *j* instructions in the reorder buffer are shown as complete, then by definition all predictions are correct which also means that (speculative) values computed using predicted operands are correct and consequently the instructions can retire.

The above demonstrates that even with value prediction, multiple instructions can be verified in parallel using the retirement mechanism existing in most current processors. The parameter that is performance critical is the latency for determining the completion condition of a predicted instruction. This requires a compare operation of the predicted value with the value produced by the instruction following its execution. If this latency is on the critical path it may delay the release of processor resources by predicted instructions.

5.6.5 Parallel Verification/Invalidation with Dedicated Hardware

The hardware discussed in this Section is capable of performing parallel verification and parallel selective invalidation. For brevity will refer to the combination of the two as verification. This is in principle similar to the hardware we assumed for verification in the proposed microarchitecture (shown in Fig. 5.2).

We propose that verification should be implemented following execution. Verification is preceded by an equality check. The equality check involves only instructions that just finished execution. Their computed outputs are compared against the predicted output (where a prediction was made). The result of the comparison is needed for performing verification in the same cycle. It may also be used for future verifications - if the instruction does not retire - and hence is saved in a field in the RS. This field indicates whether a comparison was performed and its outcome.

The second phase of verification involves not only the instructions that just executed but all instructions in the window. In this phase each RS determines the state of its output value based on the state of its inputs and broadcasts its output state so that dependent instructions can determine their state. A *simplified* verification logic and a verification example are described next. The example is aimed at illustrating that verification can be performed in "parallel," contingent on technology constrains. Parallel verification means that a chain of two or more true data dependent instructions can be verified/invalidated in fewer cycles than the maximum dependence chain length. This example illustrates one of many possible designs and may not be optimal. Also we note that this is not a complete design but rather an overview of the key aspects of verification logic.

To perform verification, we propose the use of a bus with as many lines as reservation stations. Each reservation station i has an output connection to a specific line, i. Each RS can read the output of any two line(s) in the verification network. In addition to the bus, we propose purely combinational validation logic associated with each RS that is used to determine the *verification state* for each instruction. The verification state is the information that is broadcast on the network. Fig. 5.4 shows the verification network and the verification logic required in a RS. Each RS reads the state of its predecessors by selecting their corresponding lines on the bus. As a result, there can be no cyclic dependences in this network since by definition an instruction cannot depend on its successor. The verification logic consists of five combinational units labeled A, B, C, D and E; below we describe their functionality.
The state held in the ready field of a RS (**present ready state**) for an input is used by some of the verification units, below its possible values are:

Valid, the input is produced by computations using only valid inputs, or read from register file.

Invalid.

Speculative, the input is predicted or speculative. An instruction may enter the RS with predicted input values, but before it issues it may receive a valid value or a speculative value from a predecessor. In both cases, we allow the forwarded value to overwrite the value in the field. If the value is valid, execution can proceed correctly. A speculative value can either be the same as the predicted value, in which case there is no change the status of the input value, or is different, in which case the instruction and its successors will be invalidated.

Unit A produces the **verification state of an input** using its *present ready state* and its *ver-*

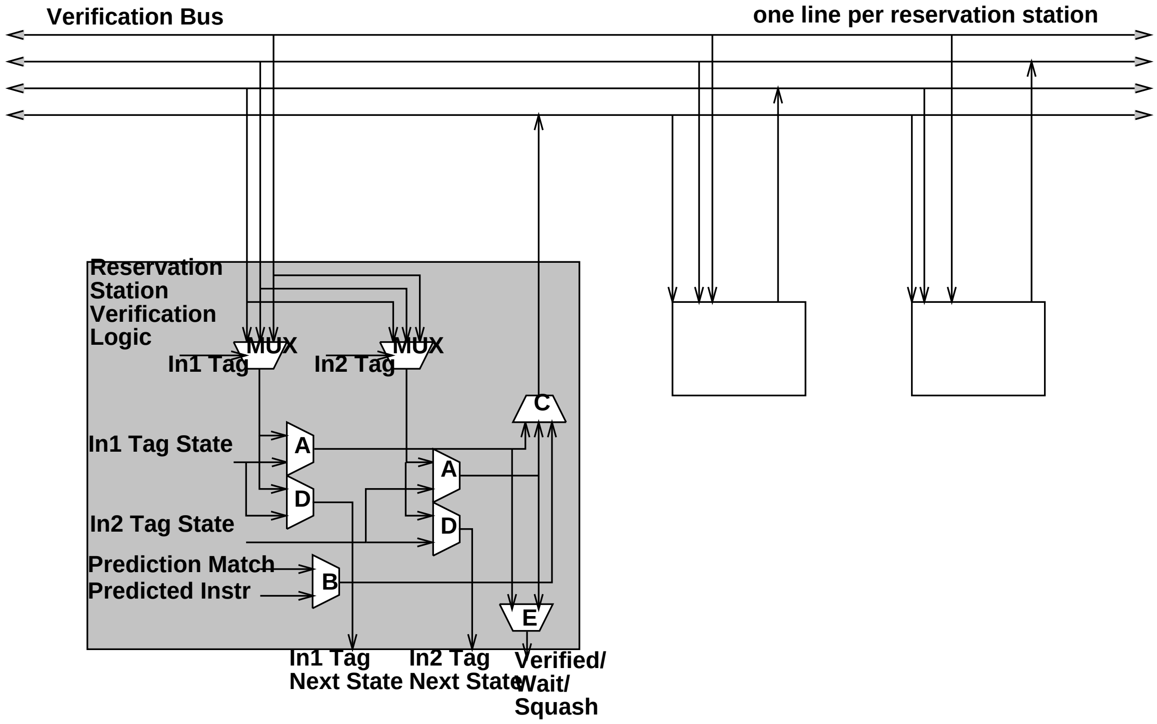

Figure 5.4: Verification Hardware

ification state from the bus. The verification state of an input is useful for determining the validity of the instruction in the RS:

Valid (srcV), instruction could be correct.

Incorrect (srcI), a predecessor instruction was predicted incorrectly and instruction is incorrect.

No Change (srcNC), instruction is either correct or incorrect.

Unit B is used for instructions with a predicted output to determine whether the prediction matches the *first* computed result. This is useful for resolving the validity of the predicted instruction. Only the first computed result is compared to the prediction because it is very unlikely that a predicted output is correct while the inputs were predicted incorrectly (Chapter 4). This is accomplished by resetting the predicted field in the RS when a misprediction is detected for an instruction. The predicted field is also reset when an instruction with a predicted output has executed, and during validation one of its inputs is found to be incorrect. Note that, alternatively, the predicted output could have been checked only when valid values are available, but we did not consider this option. The output values of B are:

Computation matches prediction (Match), instruction could be valid.

Computation does not match prediction (Not-Match), instruction is incorrect.

Don't care (DC), either this not a predicted instruction or the instruction was predicted but the first computed result did not match, or it was computed using an incorrect input. Note that when the output of this unit is either Match or Not-Match it means the instruction issued and executed only once.

Unit C is responsible for generating the **verification state for each instruction** and placing it on the bus. This unit can produce the following values:

Valid, the value produced/forwarded by this instruction is valid.

Incorrect, there is at least one mispredicted predecessor, and the value produced/forwarded by this instruction is incorrect.

Incorrect valid, this instruction output was mispredicted, but its most recent produced value is valid. This causes instructions that use the predicted value of this instruction to be invalidated. Note that direct successors of this instruction may wake up since the new value forwarded to them is valid.

No Change.

Unit D is used to determine the next ready state (the next value in the ready field of a RS) for an input operand. This is performed using its present state and the input from the verification bus.

Unit E is used to determine the next action for an instruction (retire, wait or invalidate).

The functionality of units A, C, D and E are shown in Tables 5.1-5.4. Note that the order of the inputs is immaterial. *Ex*, used in Table C, indicates whether an instruction has finished execution (this corresponds to the execution field in the RS), and *X* is used to indicate a don't care.

Fig. 5.5 shows examples of the verification of a chain of three dependent instructions. The examples assume in both cases that the inputs of the first instruction are valid. In the first example, the output prediction of the first instruction is incorrect and all subsequent instructions get invalidated. Note that the computed result of the first instruction is correct and the value broadcast on the bus indicates that the predicted value was incorrect, but the computed value is valid. Also note that the speculative results of instruction Y and Z match the prediction; however, these are values computed with an incorrect predicted value for instruction X and, as a result, are invalidated. In the second case, all predictions are correct and all instructions are valid.

An important property of the outlined verification logic is that instructions only wake up when a valid or speculative input arrives. Alternatively, an instruction could wake up each time it receives a new value; such a policy may lead to useless resource utilization when the value is the same as the predicted value or is produced using mispredicted values.

5.7 Speculative Execution Models

There have been several proposals in the literature for speculative execution based on values and addresses[14, 13, 59, 11, 79, 58]. However, all the models used in these studies, to support speculative execution, were described in an adhoc/cryptic manner, mainly due to lack of formalized variables that define such models. Furthermore, the manifestations of these models on the latency for performing operations such as releasing resources, reissue, etc, are partial or not addressed. This presents a number of problems since results obtained in these studies are difficult to assess and reproduce, and, most importantly, may not contribute in a better understanding of the design space for speculative execution. Based on the discussion in

Present Ready State	**Verification Bus**	**Output**
Valid	X	Valid (srcV)
Speculative	Valid	Valid (srcV)
Speculative	Incorrect	Incorrect (srcI)
Speculative	Incorrect Valid	Incorrect (srcI)
Speculative	No Change	No Change (srcNC)
Not Valid	X	No Change (srcNC)

Table 5.1: Unit A: Input Verification State

Unit B	**In1**	**In2**	**Output**
Match	srcV	srcV	Valid
Match	srcV	srcI	Incorrect
Match	srcV	srcNC	No change
Match	srcI	srcI	Incorrect
Match	srcI	srcNC	Incorrect
Match	srcNC	srcNC	No Change
Not-Match	srcV	srcV	Incorrect Valid
Not-Match	srcI	X	Incorrect
Not-Match	srcNC	X	Incorrect
DC and Ex	srcV	srcV	Valid
DC and Ex	srcV	srcI	Incorrect
DC and Ex	srcV	srcNC	No change
DC and Ex	srcI	srcI	Incorrect
DC and Ex	srcI	srcNC	Incorrect
DC and Ex	srcNC	srcNC	No Change
DC and not Ex	X	X	No Change

Table 5.2: Unit C: Instruction Verification State

Present Ready State	**Verification Bus**	**Next State**
Valid	X	Valid
Speculative	Valid	Valid
Speculative	Incorrect	Not Valid
Speculative	Incorrect Valid	Valid
Speculative	No Change	Speculative
Not Valid	Valid	Valid
Not Valid	Incorrect	Not Valid
Not Valid	Incorrect Valid	Valid
Not Valid	No Change	Not Valid

Table 5.3: Unit D: Next Ready State

In1	**In2**	**Verified/Action**
srcV	srcV	yes, if at head retire
X	srcI	no, invalidate
srcV	srcNC	no, wait
srcNC	srcNC	no, wait

Table 5.4: Unit E: Verification/Action

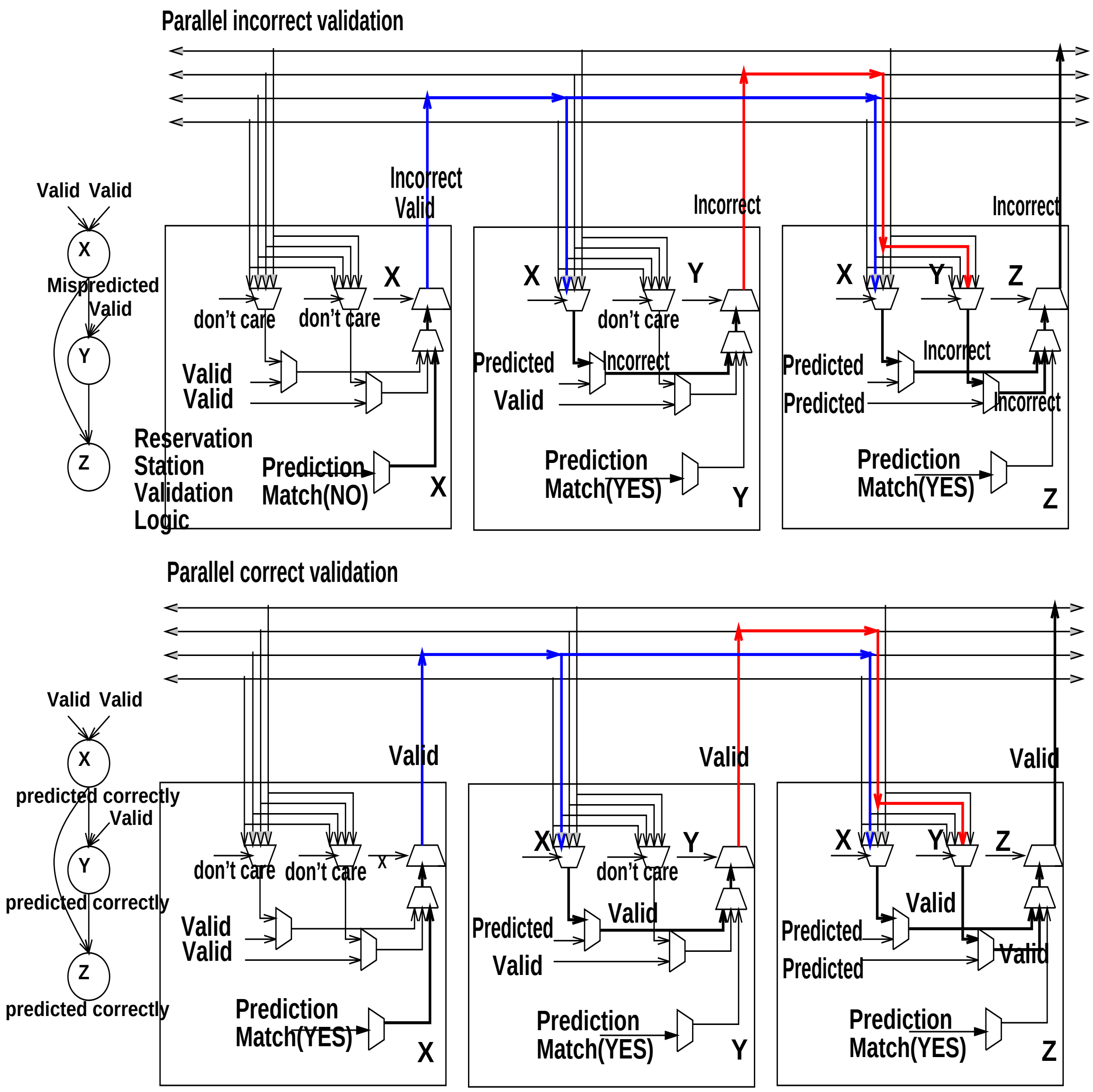

Figure 5.5: Verification Examples

this Chapter we propose combining the different variables that influence speculative execution under a single model called *speculative-execution model*. Consequently, when describing a speculative execution model the following information should be provided:

- a specific list of model variables and their values, and
- manifestations of a speculative execution model in terms of latency between different events.

Below we summarize the design space of speculative execution models discussed in this Chapter in terms of model variables and some of their possible values:

Model Variable	**Values**
WakeUp	tags, confidence state, speculative state
Selection	speculative state, ready state, confidence state
Branch Resolution	speculative, deterministic, selective
Memory Resolution	speculative, deterministic, selective
Recovery	complete, selective (serial, parallel, other)
Verification	serial, parallel (based on retirement, dedicated hardware)

A model manifests itself in terms of at least the following **latency variables** that describe the latency required between events influenced by speculative execution. The latency variables are defined from the end of the first event to end of the second event and should be given in terms of cycles:

Execution – Equality, latency required to determine if the prediction and a computed value are equal assuming equality is performed immediately after an instruction finishes execution.

Equality – Recovery, latency required for an instruction to be invalidated after it was determined that a predecessor's prediction was incorrect. This may not be a fixed number of cycles (for example, the latency of selective serial invalidation is a function of the dependence chain invalidated).

Equality – Verification, similar to the previous but for correct predictions.

Verification – Free completion resource, latency after an instruction is verified before it can release its reorder buffer entry.

Verification – Free issue resource, latency after an instruction is verified before it can release its reservation station entry.

Invalidation – Reissue, latency after an instruction is invalidated before it can reissue.

Verification – Branch, latency after verification of the inputs of a branch before a branch can issue.

Verification Address – Memory Access, latency after verification of a speculative address generation before issue to memory.

These latencies are not all relevant to all speculative execution models. The latency for branch and memory instructions is pertinent if these instruction types are not allowed to be resolved speculatively. We note that the latencies for the events need not be given separately, for example instead of reporting Execution – Equality and Equality – Verification with two separate values they can be combined as Execution – Equality – Verification and described by one value.

Although we do not claim that these are complete for describing speculative execution precisely we believe this a more systematic approach that can mitigate the aforementioned problems. In the next section, we present few speculative execution models and vary some parameters to illustrate how that influences execution.

5.7.1 Example Speculative Execution Models

In this section we consider some speculative execution models with the following parameters: instructions can wake up based on tags and tag state (Section 5.4); selection is based on instruction types and dynamic order and does not consider speculative state (Section 5.4); branches are always resolved deterministically (Section 5.5); memory instructions are not allow to access memory with speculative address (Section 5.5); recovery is based on parallel selective invalidation (Section 5.6.2); and verification is parallel (Section 5.6.3). Verification and selective invalidation are performed using a dedicated hardware similar to that presented in Section 5.6.5.

The different models are distinguished according to the values of the different latency variables. Three models, denoted **super, great, and good** are considered:

Latency Variable	Super	Great	Good
Execution – Equality – Invalidation	0	0	1
Execution – Equality – Verification	0	0	1
Verification – Free Completion Resource	1	1	1
Verification – Free Issue Resource	1	1	1
Invalidation – Reissue	0	1	1
Verification – Branch	0	1	1
Verification Address – Memory Access	0	1	1

When computation does not involve predicted values, all models have behavior identical to the base-processor. Recall that because we have a unified issue/resource structure, the latency to free issue and completion resources is the same. Also note that in the proposed microarchitecture, resources cannot be free earlier than a cycle following the completion of an instruction.

The *super* model is the most optimistic and the *good* model the most pessimistic. The difference between the *good* and the *great* is in verification latency, from one to zero cycle. The *super* model has zero cycle verification, zero cycle reissue latency, that can enable mispredicted instructions to execute early, and zero cycle latency to inform branch and memory instructions when their inputs become valid.

In Fig. 5.6, we illustrate the execution of three instructions using the three speculative execution models. The examples illustrate the behavior with correct and incorrect prediction. The most important observation is that execution behavior appears to be sensitive to the latencies. We can also observe that the smaller the latencies, the shorter the overall time require to finish execution. Another observation is that the differences for incorrect execution tend to be more pronounced as compared with the differences for correct execution. We note that for the *good* model, unlike *super* and *great* models, instructions with predicted outputs, but not predicted or speculative inputs, still need to go through verification.

The models represent a spectrum of designs with variable degree of optimism regarding the different latencies and are only a *few* of numerous possible models. We believe that exploring the design space of different speculative execution models and understanding their requirements is a promising area of future research. We evaluate the performance of the *super, great* and *good* models in (Chapters 5 and 6). The *great* model is the default model used in the simulations.

5.8 Summary

We presented key microarchitectural components that may be needed for integrating value prediction and speculation in a high performance superscalar processor. We considered the ramifications of having predicted and speculative values on the overall processor design. Issue, invalidation and verification policies were identified as central to value prediction. We examine their relation and presented several options for implementing them. It was demonstrated that verification of dependent chains of instructions can be performed in parallel, contingent on technology constrains. We also proposed a framework for more systematic description of speculative execution in terms of model parameters and latencies. Finally, some speculative execution models were presented to illustrate how the execution behaves with cor-

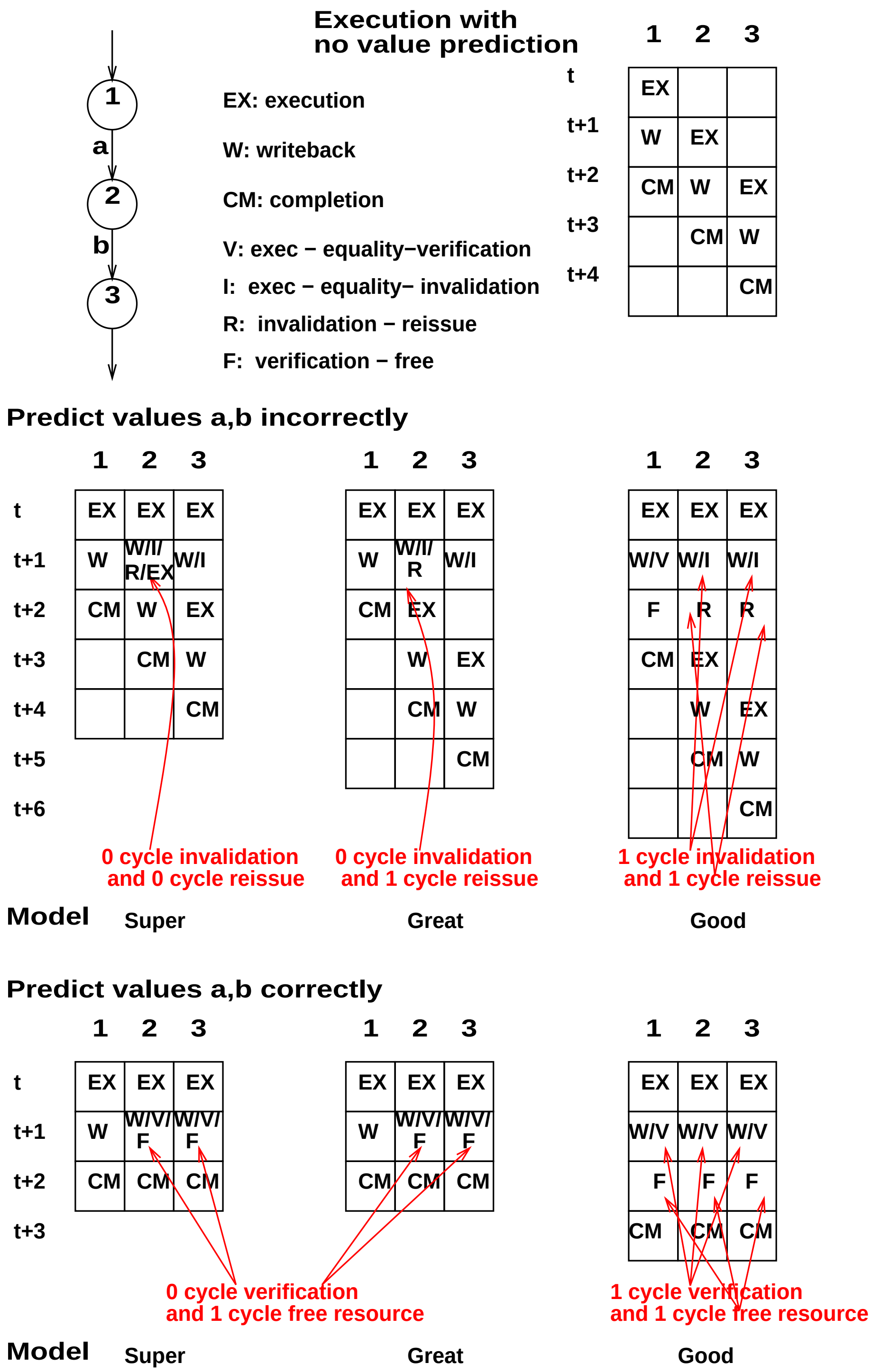

Figure 5.6: Execution example based on different Speculative Models

rect and incorrect predictions while important latency variables were varied. Our discussion reveals that the actual design of a microarchitecture with value prediction will present several new challenges to computer architects and designers because a plethora of options exist for implementing sensitive interelated policies that maximize performance without increasing complexity.

Chapter 6

Performance Potential with Perfect Value Prediction

6.1 Motivation and Outline

This chapter establishes performance limits when using value prediction and speculation for the microarchitectures discussed in Chapter 5. Determining the potential of an approach is useful in at least two ways: firstly, if the analysis reveals that there is potential, it justifies further research as to how such potential can be realized, and secondly, in the case there is little or no potential it can be an indication that the approach is not promising or the assumptions under which the potential was established were not favorable for improvements, and they need to be reconsidered. Bounds established for any approach are limited by the different assumptions made in the framework used to determine them. As a results bounds can be optimistic or pessimistic depending on the assumptions, and they should be carefully viewed as bounds within the framework and not as representative of the overall behavior of the approach[84].

Value prediction limits are important in this work for the above reasons and because they may enable us to establish important trends concerning the behavior of value prediction for different configurations. More specifically our analysis is aimed at determining:

- The performance improvement potential,
- Sensitivity to window size, issue width and branch prediction, and
- Configurations that may be interesting to study using realistic predictors.

In this Chapter we consider the performance potential with **perfect** value prediction, in Chapter 7 we examine performance with realistic predictors.

6.2 Methodology and Assumptions

We perform a simulation study for the microarchitecture presented in Chapter 5 using a modified out-of-order simplescalar simulator[74]. The simulation was conducted for two base configurations denoted *perfect* and *real*. In addition to the simulation results, we also present performance numbers obtained using an analytical model.

The perfect configuration uses oracle branch prediction, has an ideal fetch engine that can fetch from multiple basic blocks in the same cycle, and all memory references hit in the first level cache. The perfect configuration resources are not constrained; for each resource type there are as many resources as the issue width of the processor. All simple instructions take a single cycle to execute. Memory instructions require two cycles to execute, one to perform address generation and another to access the cache. Latencies for complex operations are listed in Table 6.1. A load/store queue with as many entries as the instruction window is used. Loads can receive a value from preceding store(s) in the window that write to the same address in a single cycle. Recall that load instructions are not allowed to issue to memory unless all previous store address are known (see Chapter 5).

Instruction	Latency
Int Mult	3
Int Div	20
Float Simple	2
Float Mult	4
Float Div	12
Float Sq	24

Table 6.1: Latency for Complex Instructions

Cache	Size	Configuration
L1 I$	64KB	4-way associative 32 byte block size, 1 cycle hit
L1 D$	64KB	4-way associative 32 byte block size 2 cycle hit
Unified L2	1MB	4-way associative 64 byte block size 12 cycle hit/36 cycle miss

Table 6.2: Cache Parameters for Real Configuration

The real configuration is different from the perfect in the following ways. Its branch predictor is gshare[7] that uses 16 bits of global branch history and the 16 lower bits of the PC. The branch predictor is immediately updated following a prediction. Unconditional branches are assumed to be predicted correctly always and conditional branch targets are predicted correctly if the branch direction is correct. The cycle after a branch misprediction is detected, fetching commences from the correct target. The number of memory ports is half the size of issue width. The cache parameters for the real configuration are listed in Table 6.2. Note that for the real configuration wrong path instructions are executed and their side-effects are modeled.

The performance of the two configurations was simulated with perfect value prediction and without value prediction, denoted **VP** and **NVP** respectively. When value prediction was used, for integer benchmarks we predicted only instructions that produce 32 bit values and are written to general purpose registers, whereas for floating point we predict both 32 and 64 value producing instructions. Simulations were performed for all integer and representative floating point SPEC95 benchmarks shown in Table 6.3. The Table also shows the fraction of dynamic instructions that qualified for prediction, ranged from 62% to 82% for integer and from 84% to 97% for floating point benchmarks. These fractions are important because they roughly indicate the set of dynamic instructions that value prediction can potentially speedup. The *great* parameters are used (described in Chapter 5) to model speculative execution when using value prediction. We simulated configurations that have equal fetch, issue and commit widths of 4,8 and 16 instructions for different window sizes. The configurations without value prediction are denoted when presenting results as **NVP-4, NVP-8 and NVP-16**, whereas the ones using value prediction are denoted as **VP-4, VP-8 and VP-16**. In addition to the results based on simulation we also present performance numbers obtained using an analytical model that highlight the performance limitations of the *great* model and processor configuration. We report and discuss the performance results in subsequent sections.

6.3 Performance Results

This section discusses the performance results for the different configurations. We first present results for the perfect configuration and then report the performance for the real configuration. Finally, we introduce an analytical model useful for establishing limits for a perfect value prediction. The results are presented in terms of instructions per cycle (IPC) and speedups.

Benchmark	Input Flags	Dynamic Instr (mil)	Instructions Predicted (%)
compress	400000 e 2231	103	70.5
gcc	gcc.i	203	67.3
go	9 9	132	78.7
ijpeg	specmun.ppm	129	82.0
m88ksim	scrabbl.in	120	70.6
perl	modified train	40	63.9
vortex	modified train	101	61.9
xlisp	7 queens	202	61.7
applu	5x5x5 grid	60	91.5
fpppp	4 atoms	150	83.5
mgrid	32x32, 8 iter	119	96.5
swim	256x256 grid	200	88.3

Table 6.3: Benchmark Characteristics

6.3.1 Results for Perfect Configuration

Integer Benchmarks Fig. 6.1 shows the performance and speedup obtained for the perfect configuration without value prediction and with perfect value prediction. The graphs contain data for three different issue widths of 4, 8 and 16 instructions. The speedup for a given width and window size is obtained using the performance of two configurations with and without value prediction with identical width and window size.

One of the most important observations is that with increasing window size, the configurations with value prediction achieve higher levels of parallelism faster than the equivalent configurations without value prediction. However as windows grow larger the improvements

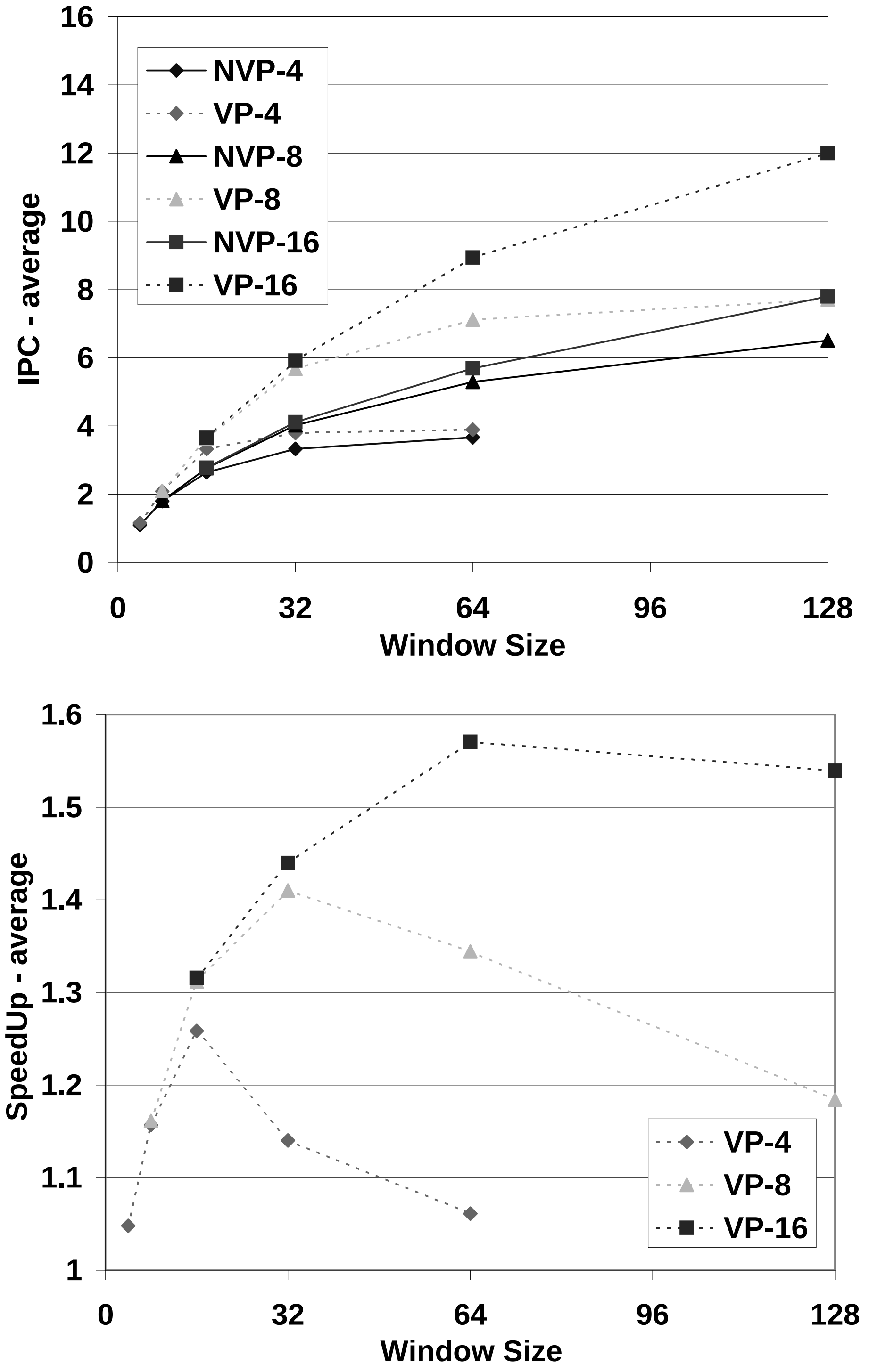

Figure 6.1: Performance for Integer Benchmarks

diminish. With smaller windows there are fewer independent instructions available, and as a result value prediction can be more beneficial to performance; as windows grow bigger more independent instructions become available which decreases the relative benefit of value prediction.

For example, for benchmark *go* (Fig. 6.2) with issue width 4 and window size 16, the performance without and with value prediction is 2.42 and 3.42 respectively, a 41% speedup. For the same issue width but window size 64, the performance is 3.69 and 3.97, corresponding to only 8% speedup. If the window size increases further the two configurations will likely achieve equal performance. This observation is valid for all issue widths and for all benchmarks. An interesting direction of research to which this observation leads, is to determine the potential of value prediction as a complexity effective approach. That is, use value prediction to reduce the window size required to achieve a given level of parallelism — we revisit this issue in the next section, using the real configuration and, in Chapter 7, with practical value predictors.

A related observation is that value prediction configurations with smaller widths can achieve, for some benchmarks, similar or higher performance as compared to a wider configuration. For example for benchmark *go* the performance of the configuration with value prediction, issue width 8 and window size 64, achieves better performance than the configuration with width 16 and window size 128, 7.5 versus 6.25 respectively. This may also point to the use of value prediction as a complexity effective approach for reducing width requirements.

Another significant observation is that the relative importance of value prediction increases with wider configurations. This is the case because wider configurations have greater likelihood of fetching data dependent instructions in the same cycle. If there are not enough independent instructions in the window, such dependences can result in issuing fewer than the maximum possible instructions which limits the maximum attainable performance. Value prediction can improve performance by predicting the value of a dependence between in-

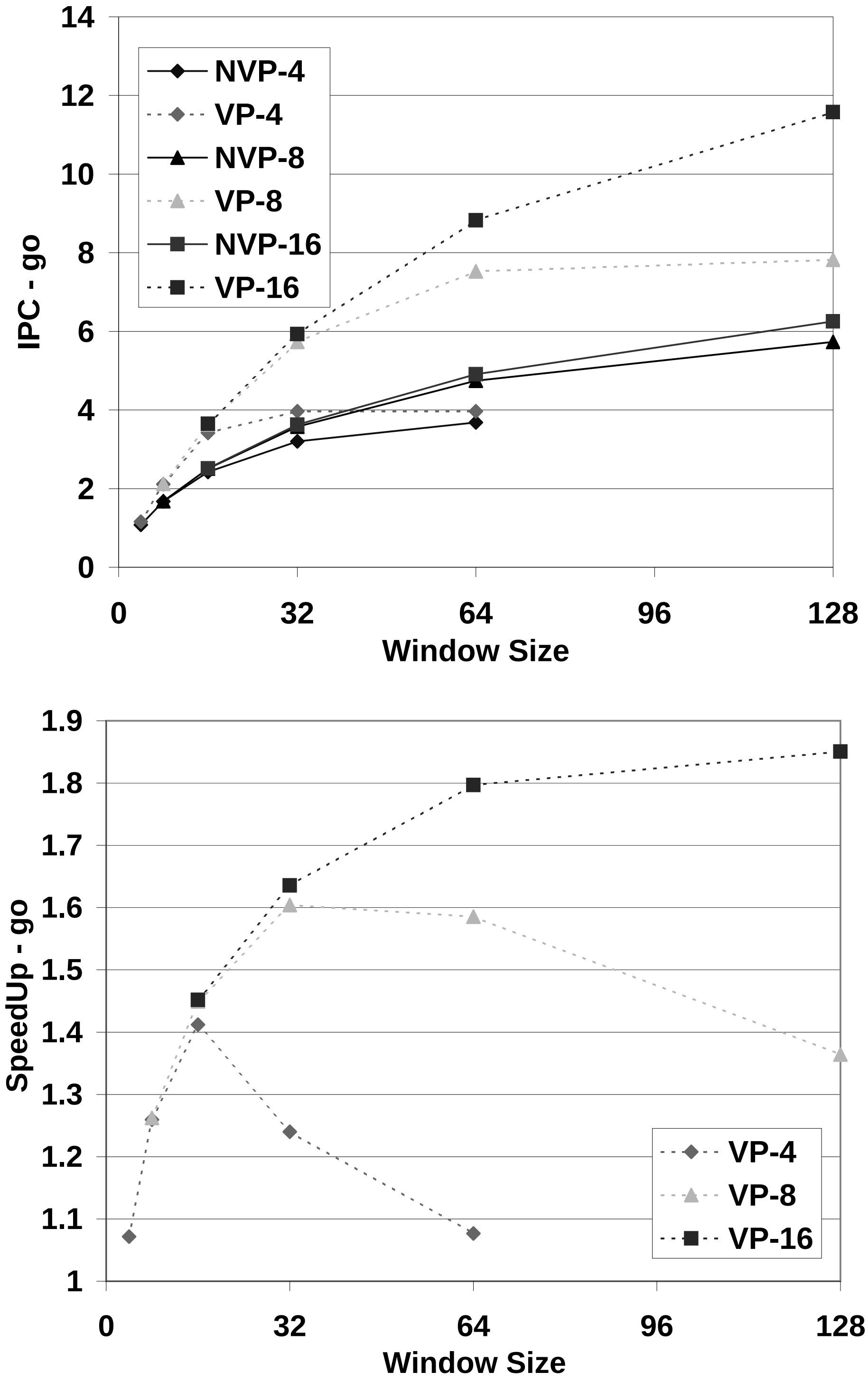

Figure 6.2: Performance for go

structions and allowing such instructions to proceed to execution in parallel. For example, *go* using value prediction, with issue width 4 has a maximum speedup of 1.41, whereas, for issue width 16, its maximum speedup is 1.85. This observation is in agreement with results reported in [70]. However even for wider configurations the improvements diminish with increasing window size.

We note that the difference in the speedups among the different benchmarks can be attributed to individual program characteristics, such as, but not limited to, how many instructions get predicted (see Table 6.3), dynamic instruction distance[70] and memory dependence behavior[18](recall the base microarchitecture does not allow loads to issue before stores with unresolved addresses). With regard to memory instructions, we note that an interesting direction for future research will be to establish the potential for a more aggressive disambiguation model and/or address prediction. An initial study addressing these issues was presented in[83].

Finally, the data show that even with perfect branch and value prediction and increasing window size the maximum IPC is not attained (Fig. 6.1). For issue widths 4 and 8, the performance with value prediction gets close to maximum, however, for issue width 16, the difference is more pronounced. This is due to the assumptions for the base configuration, such as a two cycle memory operations (address generation + cache access), not predicting addresses and forcing loads to wait for all previous store addresses before they can issue. We interpret this as an indication of the importance of using address and dependence prediction for large issue widths.

Floating Point Benchmarks The results for floating point benchmarks are shown in Fig. 6.3 for configurations with width of 8 instructions. The data indicate higher improvements as compared to integer benchmarks. One of the main reasons for this behavior is the larger fraction of instructions that qualify for value prediction in floating point benchmarks as compared to the integer. For example, for *mgrid* about 97% of the instructions (see Table 6.3) qualify

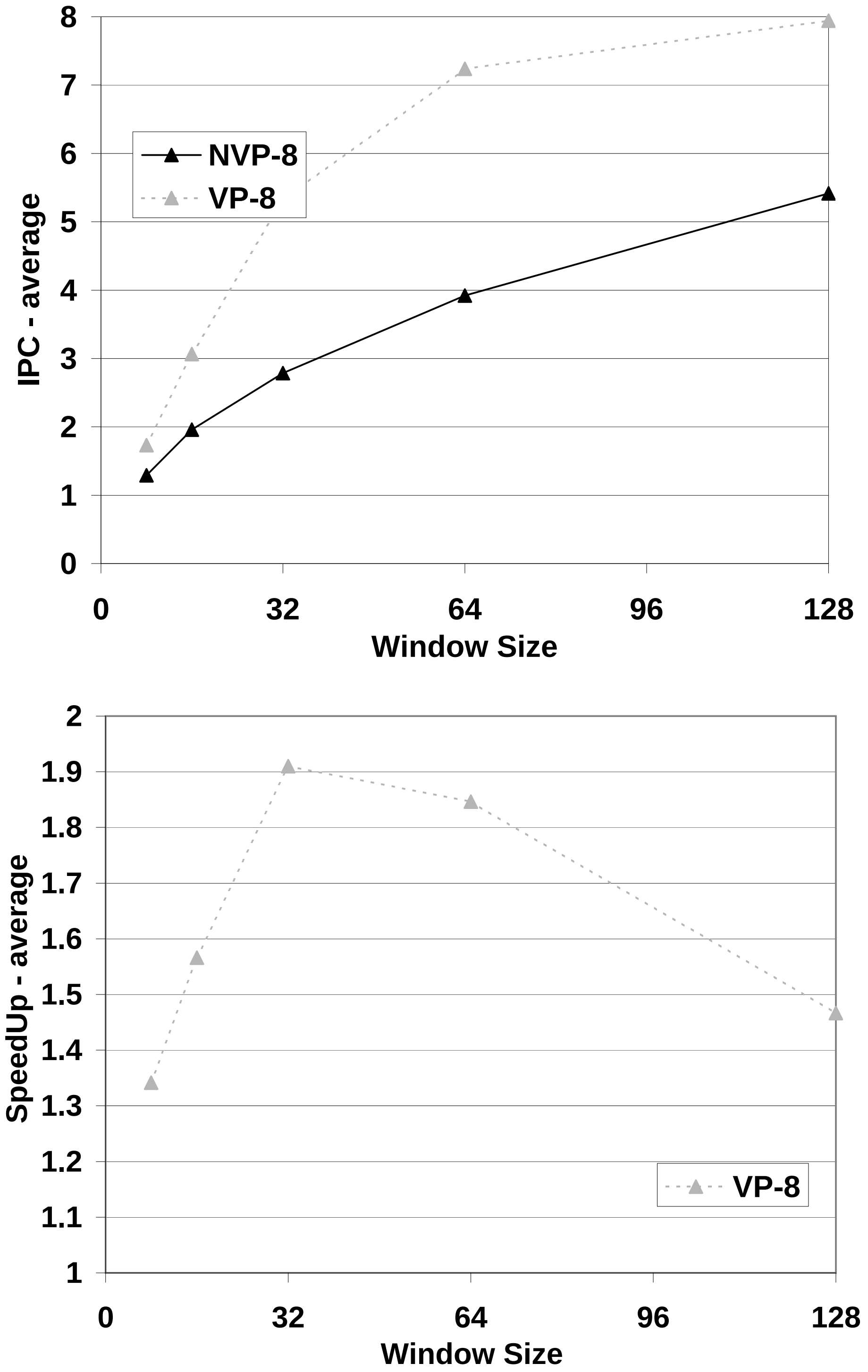

Figure 6.3: Performance and SpeedUp for Floating Point Benchmarks

for prediction. A study that limited prediction to 32 bit values revealed that a significantly smaller fraction of instructions get predicted (14% in the case of *fpppp*) and the performance benefits were drastically less. This suggests that to increase the performance potential for floating point benchmarks, 64 bit values may need to be predicted.

The above observations were based on the *perfect* configuration with perfect branch and value prediction. In the next section, we consider the performance for the *real* configuration that has memory hierarchy, limited memory ports and real branch prediction. One of our aims is to determine if the observations made in this section persist.

6.3.2 Results for Real Configuration

Figs 6.4-6.5 show the performance and speedup without and with perfect value prediction for the real configuration. A comparison of the results with the real and perfect configurations reveals that the real configuration has lower, yet significant, potential for performance improvement. The comparison also reveals that the two follow similar trends. However for benchmarks *compress*, *go*, and somewhat for *gcc* (Fig. 6.6 shows only *go* which is representative of this behavior), there is a difference among the two sets of results with regard to having diminishing speedups with increasing window size (an observation that holds for perfect branch prediction). For these three benchmarks, we observe a less pronounced drop – almost constant in some cases – in the speedup with increasing window size. The reason for this behavior is that performance saturates beyond a certain window size for these benchmarks and as a result the relative difference is also constant. Saturation is caused by branch misprediction which limits the *effective* window size. Effective window size for a given run of a program on a specific configuration is defined to be the minimum value between the configuration's window size and *average number of instructions between branch mispredictions* for that run. The average number of instructions per misprediction is the ratio of the instructions per cycle to the branch misprediction rate. The aforementioned benchmarks have

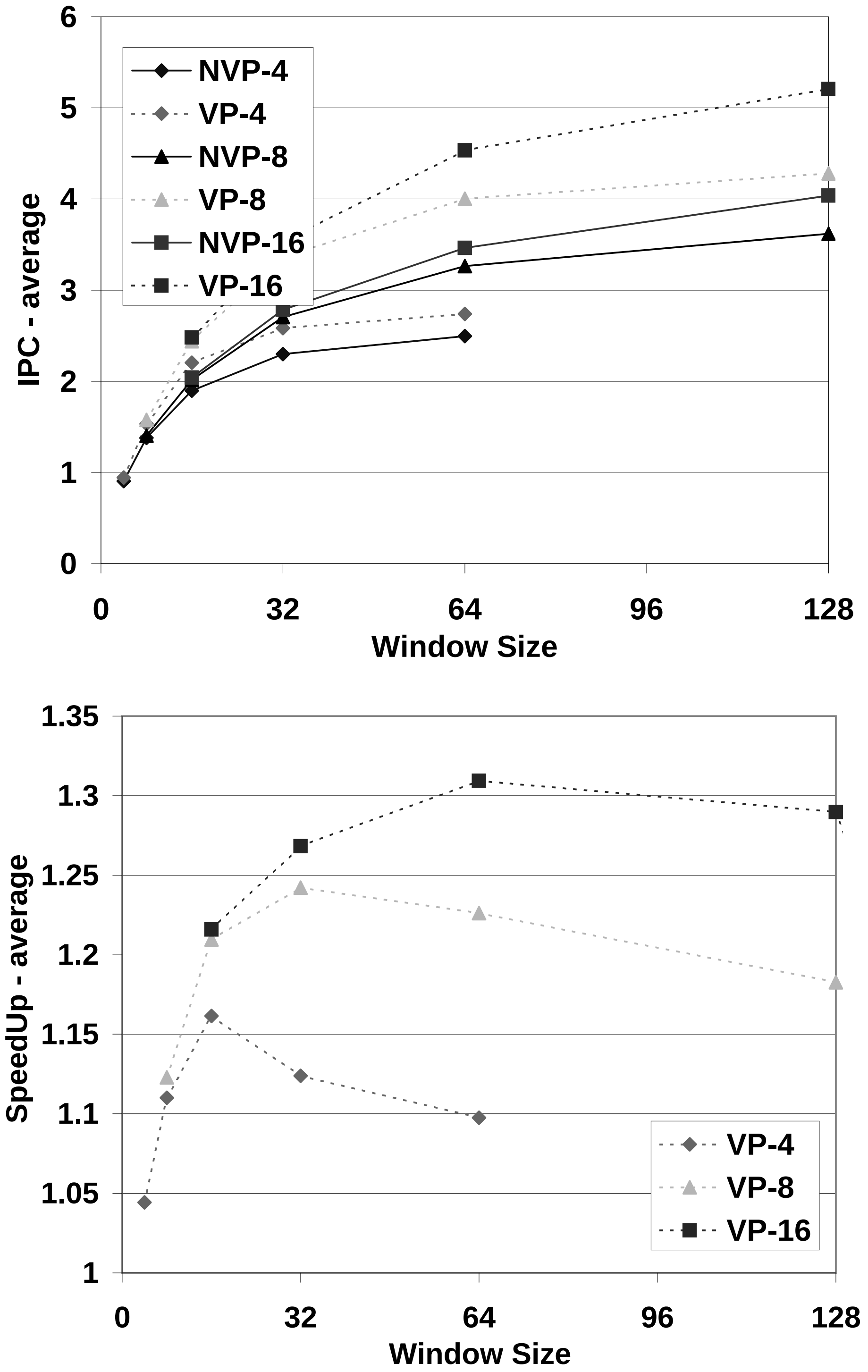

Figure 6.4: Performance for Integer Benchmarks

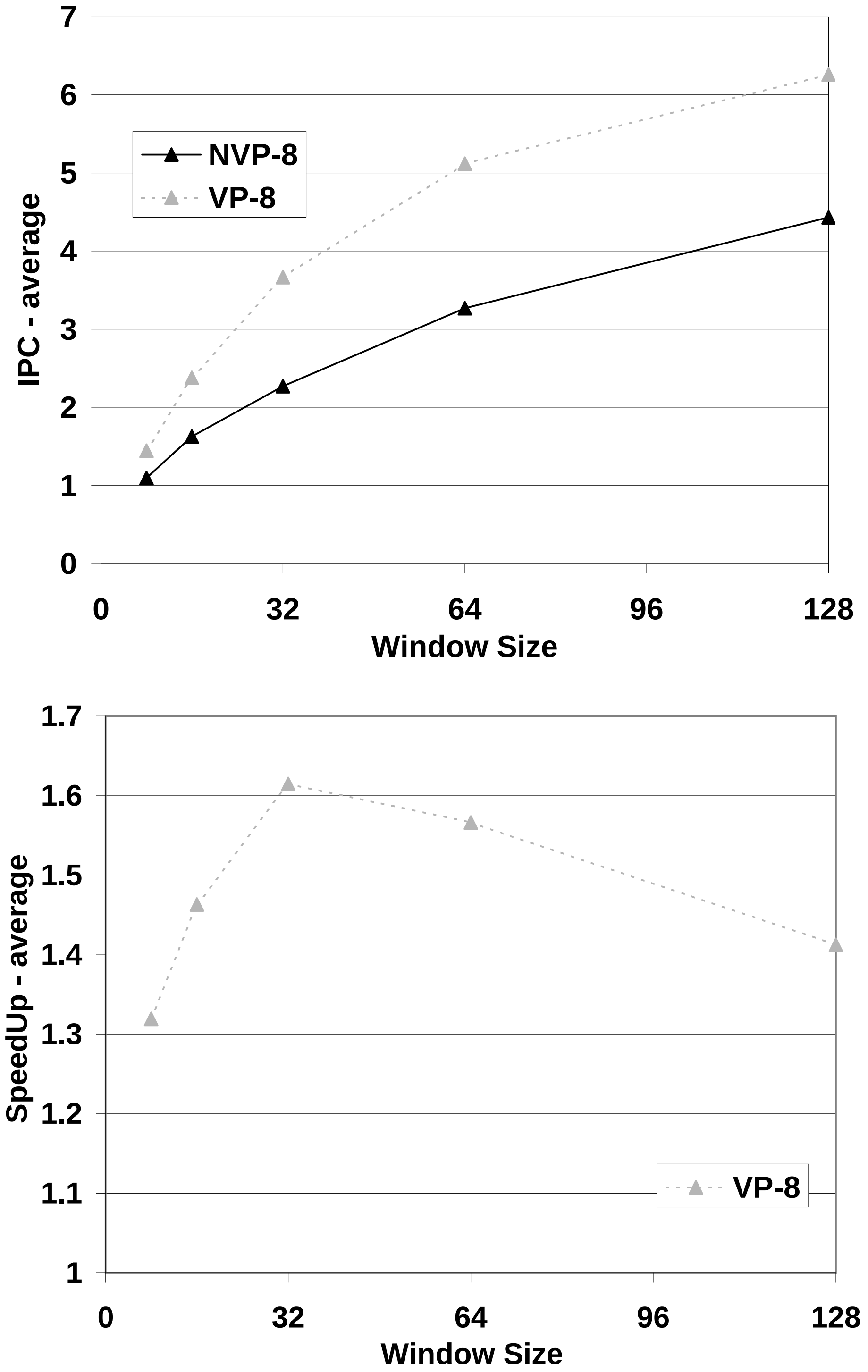

Figure 6.5: Performance for Floating Point Benchmarks

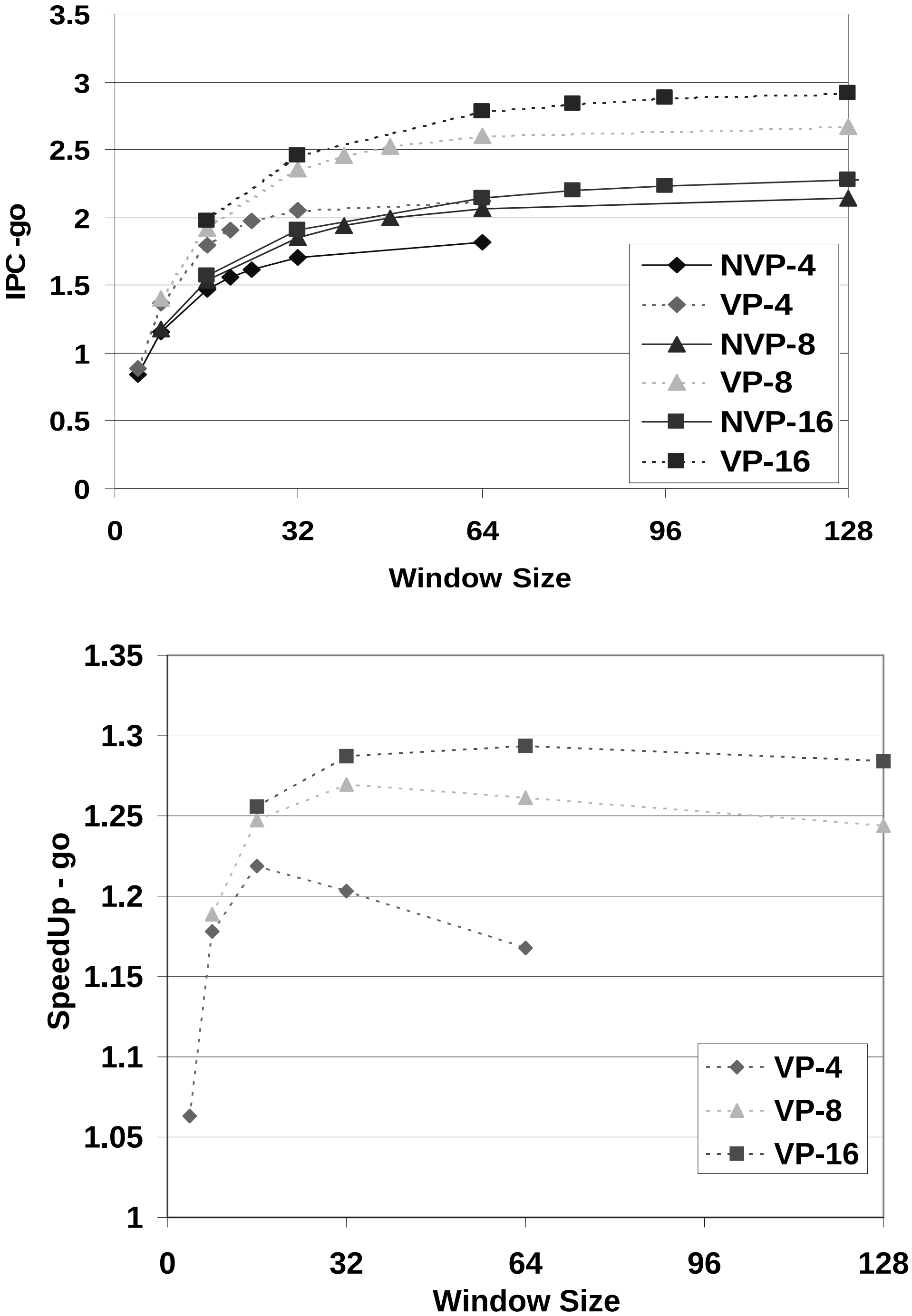

Figure 6.6: Performance for go

Benchmark	Mispred. Rate (%)	Instrs per Branch	Avg Instrs Bet Mispr
compress	9.7	7.0	71.5
gcc	8.5	6.5	77.3
go	16.7	8.3	49.3
ijpeg	7.3	13.1	169.5
m88ksim	1.7	7.3	431
perl	2.6	7.5	293
vortex	1.5	8.6	586
xlisp	3.3	6.4	192.5
applu	2.5	28.6	1121
fpppp	4.7	95.7	2044
mgrid	2.4	62.3	2540
swim	1.6	27.6	1711

Table 6.4: Benchmark Branch Misprediction and Instructions per Branch

significant misprediction rates and small instruction per branch ratios which translate to small effective window sizes (see Table 6.4). Note that the remaining benchmarks either have small misprediction rates and/or high ratios of instructions per branch and their trends are virtually identical to the perfect configuration. It is important to stress that the data do not indicate a relation between the effective window size and the amount of improvement from value prediction. As was mentioned in the previous section, the amount of improvement is influenced by other program properties such as dependence distance, instructions predicted, etc.

The results for the real configuration reenforce the key conclusions drawn for the perfect configuration: value prediction has performance potential and may be useful as a complexity effective alternative for achieving the same level of parallelism using smaller window and/or width.

Select Design Points: Based on the observations suggested by the data, we decided to focus our remaining analysis (Chapter 7) on three design points with the following width/

instruction window size: 4/24, 8/48 and 16/96. The three points were selected because their performance with value prediction was about the same or better – for most benchmarks – as compared to configurations without value prediction with the same width but bigger window sizes.

6.3.3 An Analytical Perfect Value Prediction Performance Model

The potential analysis so far has been limited in at least three ways: branches were not allowed to be resolved early based on speculative values, addresses were not predicted and load instructions were always forced to wait for all previous store addresses to be resolved and verified before they could issue to memory. Here we present an analytical model that enables the determination of the expected performance for a configuration similar to the perfect one (see Section 6.2) but without the above constraints. The analytical configuration, in addition to perfect value prediction and perfect fetch, assumes ideal speculative branch resolution, perfect memory disambiguation with loads allowing to issue before unresolved stores, single cycle execution for all instructions and instruction window size equal to the pipeline depth times width size.

The model computes performance based on five parameters: fetch/issue width or maximum IPC (w), instructions per branch (ipb), branch misprediction position (bmp), branch misprediction rate (m) and branch misprediction penalty (p). Branch penalty is the number of cycles following the detection of the misprediction before instructions from the correct path can issue – which is a pipeline specific parameter; we assume three cycles. The only parameter not commonly used is the branch misprediction position. This refers to the average position of a mispredicted branch in the list of instructions that issue simultaneously with the mispredicted branch, assuming these instructions are sorted according to dynamic program order. This number reflects the amount of useful work executed along with the mispredicted branch. This by definition must be $1 \leq bmp \leq w$. Note that according to the model, in all other cycles (except misprediction penalty cycles), the maximum possible number of instructions (*w*) is executed. We can determine the sensitivity of performance to misprediction position by calculating the performance using the minimum and maximum value of *bmp* for a given width.

Using this model, when there are no branch mispredictions a program with *N* instructions will require $\frac{N}{w}$ cycles to complete. The total number of cycles required to execute *N* instructions with branch mispredicts can be divided into:

$$\textit{Cycles with Maximum Parallelism} + \textit{Cycles with less than Max. Par.} + \textit{Misprediction Penalty Cycles}$$

The number of cycles with maximum (*w*) parallelism is given by:

$$\textit{Cycles with Maximum Par.} = \frac{(N - Instr.\ in\ Cycles\ with\ Small\ Par.)}{w}$$

Instructions are defined to have issued in cycles with small parallelism when they issue together with a mispredicted branch and the position of the mispredicted branch is less than *w*:

$$\textit{Instructions with Small Par.} = \textit{Number of Mispredicts} * (bmp\ \%\ w)$$

The number of cycles with small parallelism is a function of the mispredicts, given by:

$$\textit{Cycles with less than Max. Par.} = \textit{Number of Mispredicts} * \left\lceil \frac{(bmp\%w)}{w} \right\rceil$$

The number of mispredictions equals:

$$\textit{Number of Mispredicts} = \frac{N}{ipb} * m$$

Finally, cycles due to misprediction penalty is given by:

$$\textit{Misprediction Penalty Cycles} = \textit{Number of Mispredicts} * p$$

Combining all of the above gives the following expression for calculating the number of cycles required to execute a program of *N* instructions for the analytical configuration:

$$Cycles = \frac{N - \frac{N}{ipb}\ m\ (bmp\%w)}{w} + \frac{N}{ipb}\, m\, (\, p + \left\lceil \frac{(bmp\%w)}{w} \right\rceil)$$

The IPC can be obtained by substituting the derived equation into the following:

$$IPC = \frac{N}{Cycles}$$

In Fig. 6.7 we show the performance curves computed using the analytical configuration equation, for w = 4, 8 and 16, with ipb = 6.5, p = 3, bmp = w and bmp= 1, for misprediction rate between 0 to 0.25. The data reveal several interesting points. The data verify that branch misprediction is a major limitation to performance. More interestingly, the data reveal that branch misprediction becomes a more significant performance limitation with wider processors. The data also show that branch misprediction position becomes more important with wider processors. The analytical model can be used to study the sensitivity of performance to the other parameters such as instructions per branch, and misprediction penalty.

The curves in Fig. 6.7 can be used to bound the performance of a benchmark with 6.5 instructions per branch. Benchmark *gcc* has ipb = 6.5 and an 8.5% misprediction rate. The analytic data show that in the worst case the performance will be 3.3, 5.7 and 8.8 for width 4, 8 and 16 respectively. The corresponding performance obtained using the real configuration is 2.5, 3.6 and 4.2 (see Fig. 6.4). The above indicates that, for identical branch misprediction, the relative performance difference between the two configurations increases as processors get wider. Similar observations are consistent for the rest of the benchmarks. This as an indication that some of the assumptions we made for the real configuration are limiting performance and that under different assumptions the performance of the real configuration could be higher. We believe it is very important to explore the design space and find which parameters in the real configuration limit performance and consider ways of overcoming such limitations. We did not pursue this direction of research in this work.

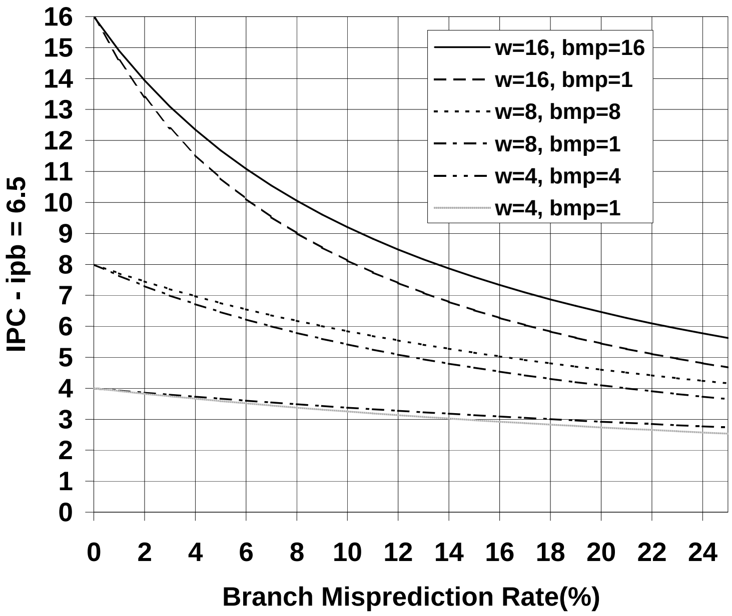

Figure 6.7: Performance using the Analytical Configuration

6.4 Summary

In this chapter, we demonstrated that perfect value prediction has significant potential for increasing performance. The experimentation for configurations with perfect and real branch prediction revealed similar but important trends. Value prediction can achieve high levels of parallelism faster with increasing window size. However, the relative improvements start to diminish when the window size becomes large enough that increasingly more data independent instructions become available. Another important trend is that value prediction offers more potential with increasing width; this is the case because wider processors expose more data dependences. Both of the above observations point to the use of value prediction as a a complexity effective approach that might enable similar or higher levels of parallelism to a configuration without value prediction while using smaller window and/or width.

An analytical configuration was introduced that under certain conditions enables the computation of the performance with perfect value prediction using an equation based on five parameters. The results based on the analytical configuration revealed that some of the assumptions for the real configuration seriously limit performance. This indicates that future research should be aimed at identifying and overcome the limitations of these assumptions.

Chapter 7

Value Predictors Implementations and Performance

7.1 Motivation and Outline

Accurate prediction is essential for the success of execution paradigms that rely on value prediction. In Chapter 3, it was shown that *context-based* prediction can predict values with high accuracy and perform better than *computational* predictors. However, the goal of the study in Chapter 3 was to determine the limits of value predictability using unbounded tables and did not consider either the implementation or the performance consequences of value prediction. In this chapter we investigate:

- the design space of value predictors,
- the ramifications of pipelining, branch misprediction, and multiple instruction issue on the design of value predictors, and,
- the performance and prediction accuracy for several value predictor configurations for the processor organizations outlined in Chapter 6.

The predictor design space exploration considers *last-value*, *stride*, *context-based* predictors as well as *hybrid predictors*, that combine stride and context-based prediction. Several parameters such as *tables size*, *hash functions* and *confidence* are examined.

Implementation and execution idiosyncracies of a processor may have a detrimental effect on the performance of a value predictor. In particular, we discuss the following consequences of placing a value predictor in a superscalar pipeline.

Delayed updates: in virtually all previous studies, predictors were immediately updated with the correct value following the prediction, unlike the case in practice where it may take many cycles after the prediction for the correct value to be known.

The effects of branch misprediction: instructions executed down the wrong path may update value predictor state speculatively. When the misprediction is detected, should these side effects be removed or be maintained?

Organizations for high bandwidth value predictors: multiple instruction fetch and issue may require parallel multiple predictions, possibly for multiple instances of the same instruction.

The performance of several value predictors and configurations are evaluated. The analysis of the results could prove useful in (a) determining the performance merits of value prediction, (b) establishing performance trends such as the relation between performance improvement and prediction accuracy, and (c) recognizing the relative importance of different parameters in the design space of value predictors.

7.2 Design Space of Value Prediction Models

In the next sections, we discuss a number of parameters in the design space of computational and context-based predictors. The discussion in this section does not consider implementation effects (these are examined in Section 7.3).

7.2.1 Computational Predictors

A computational prediction model produces a prediction by performing some computation on previous values. Computational predictors can quickly learn to predict correctly, and can even predict values not previously observed. However, they are only capable of predicting sequence of values that correspond to the computation of the predictor. Examples are *stride* prediction, that produces a prediction by adding a constant to the most recent value, and *last value* prediction, that predicts the next value to be the same as the previous one.

Example organizations of last value and stride predictors are shown in Fig. 7.1. The last value predictor consists of a single table that typically contains the last value produced by instruction(s) that map to an entry. The stride value predictor essentially consists of two tables: a last value table and a stride table. The last value table contains the last value produced by instructions that map to the entry whereas the stride will typically contain the difference between the two most recent values produced by instructions mapping to an entry.

A number of variants to the above basic schemes exist based on the replacement policy of table contents. For example, in a last value predictor, a value may not be replaced until it causes a number of mispredictions[12]. Similarly in a stride predictor, a stride may not be replaced with a new stride until it is found to be incorrect a specific number of times[13, 58, 59]. Another replacement policy can be based on tags that identify the instruction that uses a table entry[12]. A tag match is required for an instruction to update the predictor entry.

There are numerous combinations of processor state that can be combined to form an index to access the different tables, however, all proposals thus far use bits from the PC of the predicted instruction to form an index to last value and stride predictors. Also choice exists for how to organize prediction tables (direct mapped or set associative).

In addition to the above tables, it may be desirable to maintain tables for determining *prediction confidence*. The *prediction confidence* table can be used to assign a confidence

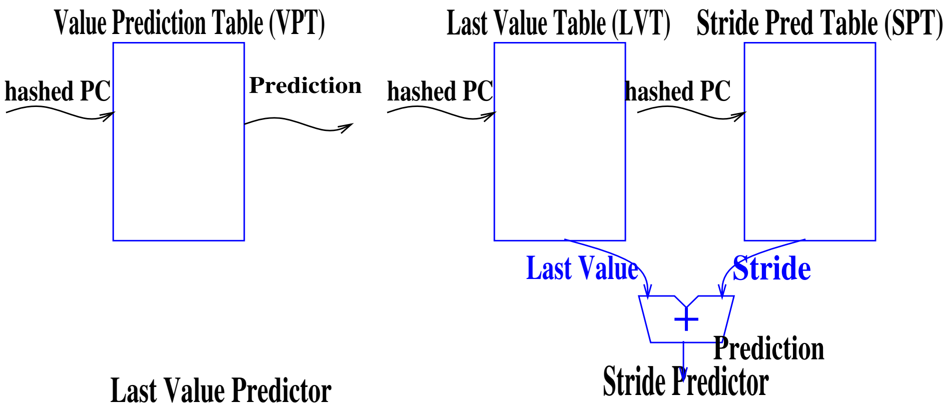

Figure 7.1: Generic Computational Value Predictors

to a value prediction and is useful in reducing misspeculation. Each entry of a confidence table can contain the state for a finite state machine such as a saturating or resetting counter that is incremented/decremented on success/failure of predictions from an entry[60]. An alternative to having a prediction confidence table is to assign static confidence information to instructions, based on profiling[56].

7.2.2 Context-Based Predictors

A context-based prediction model learns the values that follow a particular context and predicts one of the values when the same context repeats. A context-based predictor is said to be of *order-k* when the context includes information from *k* previous values. Context-based prediction enables the prediction of any repeated sequence of values. The main limitation of context-based prediction is that for a value to be predicted correctly it must have followed the same context at least once before.

We propose a two-level organization for implementing the context-based predictor introduced in Chapter 3. This organization consists of *value history table* (VHT) that contains the context - previous history values used to select the next prediction, and the *value prediction table*, that contains the predictions (Fig 7.2).

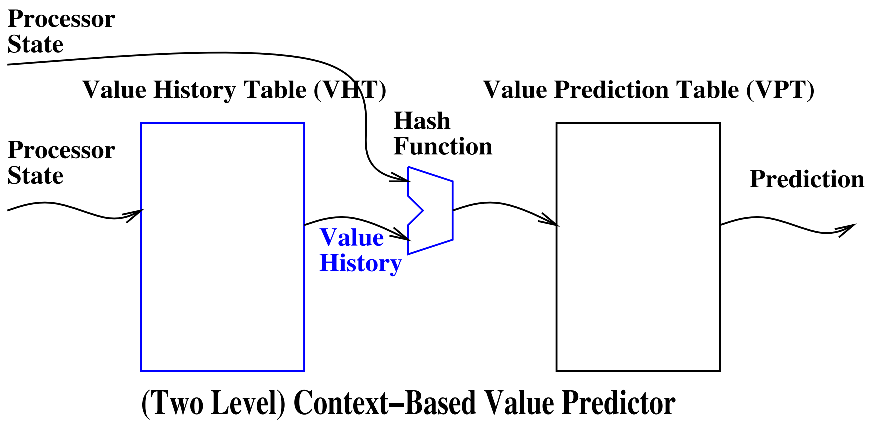

Figure 7.2: Generic Context-Based Value Predictor

To make a prediction for an instruction, information derived from processor state is used to form an index to access an entry in the value history table. The context selected from the value history table and information derived from processor state are reduced via a hash function to an index for accessing the value prediction table to get a prediction. An update is performed when the correct value for a predicted instruction becomes available. At that point the two entries accessed during prediction are updated with the correct value.

We note that the above processes for performing prediction and updates is typical of trace driven studies with immediate updates where only instructions on the correct path of execution are simulated. This does not take into consideration any implementation issues such as delayed updates due to pipelining and branch mispredictions that cause instructions to execute down the wrong path. These issues are discussed in Section 7.3.

One of the arguments for the choice of this particular organization is its resemblance to finite context predictors used in the ideal study in Chapter 3 to study the predictability of values, however, the tables here are bounded. Recall that finite context predictors are used for text compression and are known to have the best prediction (compression) performance[42, 41]. Another reason for choosing this organization is its similarity to the two-level branch predictors[5, 6]. Two-level branch predictors represented the evolution-

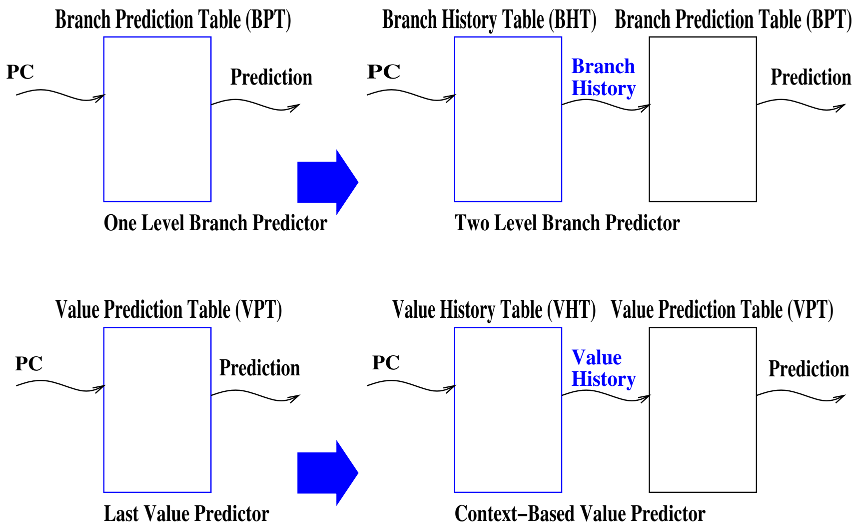

Figure 7.3: Analogy between Two Level Branch and Context-Based Value Predictors

ary step following single-level branch predictors[2, 4] that helped increase branch prediction accuracy. Similarly two-level (context-based) value prediction represents the next step following last value prediction[12](Fig. 7.3).

Almost limitless possibilities exist with regard to the information used to access predictor tables as well as the information stored in the predictor. Some of the possibilities are discussed next.

Value History Table Index: can be derived from processor state using information such as the address and the type of the instruction to be predicted, global branch history, register values, bits in the pipeline etc. A variety of schemes for forming value history table indices have been proposed for conditional branches in [25]. The choice of a particular index function could be dependent on the size of the value history table.

Contents of Value History Table: include k values (an order-k context) possibly in hashed form. The context is used to identify a point in a value sequence so that the next value can be predicted. The context is analogous to the history register or path information used for branch prediction and jump prediction [5, 6, 9, 29, 10]. In theory, the larger the order, the better for prediction accuracy, however, in a practical scheme this may result in increased aliasing.

A VHT table entry may also contain tag information, such as bits from the PC. This may reduce mispredictions due to aliasing by not using/updating a prediction from instruction(s) that do not match the tag. The tag replacement can be guided by a state machine such as a saturating counter.

Value History Table Size: for branches it is known that high accuracy can be attained even with a single value history table entry (global history registers) [5, 6, 7]. This is the case because branch values are single bits and a global history register, in combination with address information, can do relatively well in identifying a branch in a program. For value prediction, however, the granularity of the predicted values is larger and prohibits forming an index to the value prediction table using the full value history. The reduction of the context will result in value prediction table conflicts from different contexts (context aliasing). There are at least two approaches, and in some combination they can be used to alleviate this problem.

- Increase value history table size so that instruction context becomes more distinct; and,
- Incorporate additional information in the hash function to help identify the predictions in the value prediction table that are associated with a specific instruction.

Hash Function: the hash function forms an index into the value prediction table using context information from the value history table and information derived from processor state. The hash function can be better conceptualized as two distinct functions: a **context function** that reduces the k values of a context, and a **combine function** that merges the state from the processor and the output of the context function.

Context Function: Numerous context functions can be defined; in this work we consider the following three important classes of functions:

Select-Concatenate (C): select certain bits from each value in the context and concatenate them together. The bit positions and the number of bits selected from each context value can be different.

Select-Fold-Concatenate (FC): extends the previous function with the insertion of a folding step. The folding reduces the selected bits into a smaller number of bits. A folding operation that can be considered is the bitwise xor of the selected bits.

Select-Fold-Shift-Xor(SFS): uses a different operation for combining the folded context values. Each folded value is shifted by a number of bits and then all values are xored together. It should be noted that C and FC functions are special SFS cases.

A function for reducing an order-k context into an index with b bits can then be described in terms of the following operations: $\mathbf{R}(\mathbf{r}_{k-1} \,..\, \mathbf{r}_0)\mathbf{F}(\mathbf{f}_{k-1} .. \,\mathbf{f}_0)\mathbf{S}(\mathbf{s}_{k-1} \,..\, \mathbf{s}_0)$. The R operation selects r_j bits from the jth value in the context, the F operation folds the selected value into f_j bits which are left shifted by s_j bits and xored together with all values in the context. The b least significant bits of the resulting value is the index. In general r_j could be a bit vector but in this work we assume that r_j specifies the number of bits to be selected starting from the least significant position of a value and ending at bit position r_j-1. Figure 7.4 shows an example of the SFS function being used to form an index of 5 bits with a context of three values. The function used in Figure 7.4 can be defined as R(8,8,8)F(5,5,5)S(4,2,0). A good performing SFS function is one that folds the most recent value to be inserted in the history into the required number of bits, then shifts the value already in the context by some fixed amount, *p*, and then exclusive ors in the new value. This function does not require all the previous k values to be maintained in the context - a good property since that decreases cost. This may also require a simpler operation to form an index as compared to the general case. We will refer to this function as **Shift-Xor (FS)**. The number of bits in a value history table entry, divided by the shift amount, determine the order for this function. This function is defined as R(m,..,m)F(n,..,n)S(..,2*p,1*p,0*p) and in more concise form as R(m)F(n)S(p),

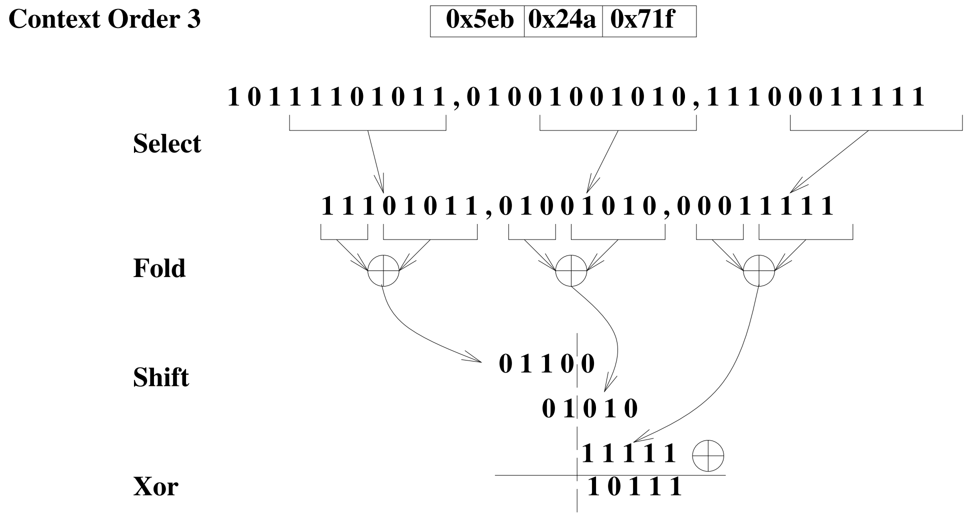

Figure 7.4: *Select–Fold–Shift–Xor* Function

the value of n should be the same as the length of the value history entry in bits. The example in Fig. 7.4 is an FS function R(8)F(5)P(2) of order 3.

The Combine Function: This function can be as simple as the identity of the output of the context function or can include processor state information. Processor state information can be effective in identifying the predictions associated with a particular instruction in the value prediction table. Information that can be used is a part of the PC address of the predicted instruction, branch history, etc. An effective operation for combining all the information is the exclusive or function[7].

Value Prediction Table Size: The table size is limited by cost and access latency, and directly determines the size of the index generated by the hash function, and possibly could influence the number of values used in a context.

Contents of Value Prediction Table: This can include one or more predictions per entry, a selection mechanism for choosing among many predictions, and a mechanism for prediction replacement. The replacement policy can be a finite state machine such as a saturating counter that is incremented/decremented on success/failure[2]. When the counter is below a threshold then an old prediction may be replaced with a new one. The same state machine, or a separate one, can be used to maintain the confidence for the correctness of a prediction.

An alternative replacement mechanism is to maintain two predictions *p1* and *p2*, known as *2-delta*[49]. The one, p1, is the prediction and the other, p2, is used to determine when p1 is replaced. p2 is always updated, however, p1 is only updated with the value in p2 when a new value matches the value in p2[49].

If more than one prediction is maintained per entry, then it is necessary to have a voting scheme to determine which prediction should be used. Examples of voting schemes include counters[42] (see Chapter 3).

Similar to last value and stride predictors, it may be desirable to have a prediction confidence table to reduce mispeculation. In addition to the prediction confidence table an *update confidence table* can be used. An *update confidence* table can be effective in eliminating *start-up* or *useless* updates that may pollute the prediction tables. Eliminating such updates can have at least two beneficial effects: reducing aliasing in the value prediction table, and reducing table size required for achieving a given level of prediction accuracy. Start-up updates are due to instructions that execute very few times or infrequently. Useless updates are due to instructions that are often mispredicted. An example policy to eliminate start-up updates would be to not update a value predictor table from a value history entry that has not been referenced more times than the number of values in the context. Useless updates appear to be more critical for context-based predictors because a single instruction can update any entry in the value prediction table (unlike the case of last value and stride predictors where an instruction can only update one entry). Useless updates can be eliminated by monitoring the prediction rate of all or few at a time value history table entries. The prediction rate of a value history table entry is the prediction rate of all instructions that use the entry. If a value history table entry following a certain number of predictions do not achieve a satisfactory threshold, they can be disabled from updating the value prediction table. The disabling can be permanent or be reset at specific intervals. The cost of adding an update confidence table

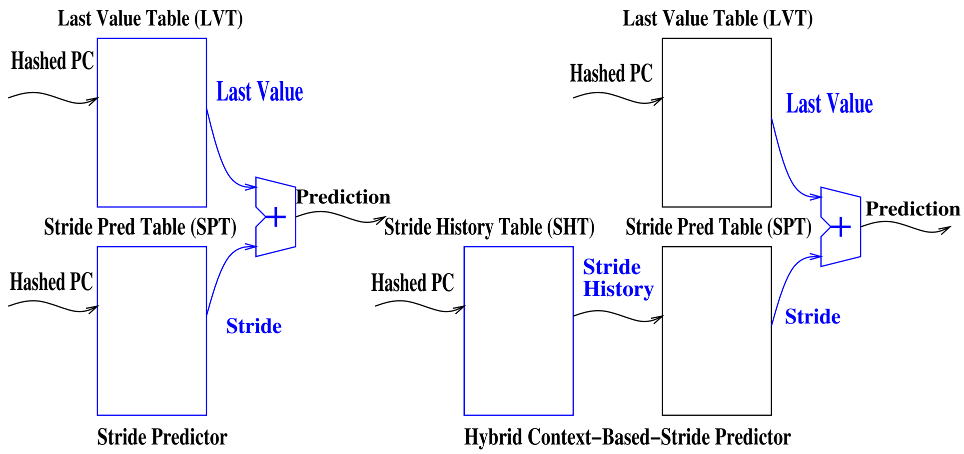

Figure 7.5: Hybrid Context-Based-Stride Value Predictor

should be weight against the gains from reducing the pressure and cost of the prediction table. Update confidence can also be implemented via profiling[56].

7.2.3 Hybrid Predictor: Context-Based Stride Predictors

In Chapter 3, it was suggested that a high performance implementation of a context-based predictor may be infeasible because instructions that produce several values contribute to predictability. However, it was shown that a hybrid configuration of a stride and context-based prediction may mitigate this problem since over 60% of the correct predictions by a context-based are captured by stride predictors.

To reduce the number of unique values that need to be learned by a context predictor we propose the use of a hybrid predictor that combines context-based and stride prediction[85]. This predictor is shown in Fig. 7.5. As the figure suggests this predictor is combining context (previous history information) to stride prediction. Its main components are the *last value table*, containing the last value produced by the instruction(s) mapping to an entry, *stride history table*, containing a finite number of the most recent strides between values of instruction(s) that map to an entry, and the stride prediction table that contains predicted strides.

The main difference, with a stride predictor, is the use of previous stride history to select the current stride for a prediction. This enables constant, stride and repeated stride sequences, (see Chapter 3), to be predicted using a constant number of entries in the value prediction table and repeated non-stride sequences to be predicted using their repeated differences.

This presents a difference from stride and context-based predictors. Stride predictors are not capable of predicting repeating non-stride sequences whereas context-based predictors are not capable of predicting stride sequences. Also, context-based predictors require as many prediction table entries as values in a repeated stride sequence, that may increase the likelihood of conflicts between instructions in the prediction table. Another property of this predictor is that, unlike previously proposed branch and value hybrid predictors[7, 28, 17], no selector is used to choose between its components.

A possible drawback of this approach is its property of repeating the same misprediction for some repeated sequences. For example, if the context consists of the last four strides, then a repeated stride sequence that counts from 1 to 10 will be mispredicted every time the loop starts over since the history is not long enough to capture the complete repeated behavior.

The stride history and prediction tables have similar design space to the context based prediction. However, strides are known to be in general smaller than values, 5 to 8 bits are sufficient to capture most strides[49, 85], and as a result the number of bits required in the history and prediction table entries can be smaller.

To perform a prediction for an instruction, information derived from processor state, such as bits from the PC, is used to form an index to access an entry in the last value table and stride history table. The context selected from the stride history table and information derived from processor state, are reduced via a hash function, not shown in Fig. 7.5, to an index for accessing the stride prediction table to get a stride. An update is performed when the correct(current) value for predicted instructions becomes available. The difference between the current and last values is used to update the stride history and stride prediction tables.

Finally, the last value table is updated with the current value.

7.2.4 Selected Predictor Configurations

For performance evaluation, the predictor configurations with the following parameters are considered:

Predictor	Last Value Table(LVT)	History Table(HT)	Prediction Table(PT)
Last Value -Index	–	–	PC
Last Value- Entries	–	–	1K, 8K, 64K
Last Value- Repl. State	–	–	1 bit sat. counter
Last Value- Entry Size (bits)	–	–	32 + 1
Stride -Index	PC	–	PC
Stride- Entries	1K, 8K, 64K	–	same as LVT
Stride- Repl. State	always	–	1 bit sat. counter
Stride- Entry Size (bits)	32	–	32 + 1
Context- Index	–	PC	history
Context- Entries	–	1K, 8K, 64K	1K, 8K, 64K
Context- Repl. State	–	always	1 bit sat. counter
Context- Entry Size (bits)	–	order*32 or $\log_2$(PT)	32 + 1
Hybrid -Index	PC	PC	history
Hybrid - Entries	1K, 8K, 64K	same as LVT	1K, 8K, 64K
Hybrid - Repl. State	always	always	1 bit sat. counter
Hybrid - Entry Size (bits)	32	order*32 or $\log_2$(PT)	32 + 1

An entry with a – indicates that the corresponding parameter is not relevant for a given predictor. All tables in this study are direct mapped and entries are not tagged and prediction tables contain only a single prediction. For the context and hybrid predictors, the size of an entry is either a function of the order or the number of entries in a prediction table depending on the context function used. When an FS function is used, the size of a history entry is $\log_2$(PT) bits; otherwise, is equal to the total number of bits of the values in the context. The

C and FC Functions for a 64K VPT		
Order	C	FC
1	R(16)	R(32)F(16)
2	R(8,8)	R(32,32)F(8,8)
3	R(6,5,5)	R(32,..,32)F(6,5,5)
5	R(4,3,3,3,3)	R(32,..,32)F(4,3,3,3,3)
7	R(4,2,2,2,2,2,2)	R(32,..,32)F(4,2,2,2,2,2,2)
SFS Context Functions for 2^t VPT		
Order	Shift Amount	Function
$\lceil t/p \rceil$	p	FS: R(32)F(t)P(p)
$\lceil t/p \rceil$	p	PFS: R(t)F(t)P(p)

Table 7.1: Context Function Descriptions

combining function used for context-based and hybrid predictors is *identity*. The different context functions used for the context-based predictors are shown in Table 7.1. When confidence is employed with any predictor, a table with 2^{16} entries is assumed. The confidence table is indexed using 16 bits of the predicted instruction's PC. Confidence is based on a 3 bit resetting counter that is incremented by 1 on correct predictions and reset to 0 on incorrect predictions. A prediction is considered confident when the confidence value is maximum. In the simulations we compare the performance of real confidence, using the confidence table, with oracle confidence. Note that in this study we do not consider update confidence mechanisms. The default predictor configurations used in the experimentation are:

a last value predictor with 64K entry prediction table,

a stride predictor with 64K entry last value table and prediction tables,

a context-based predictor with 64K entry history and prediction tables and FS context-function R(32)S(16)P(4), and

a hybrid predictor with 64K entry last value, history and stride tables with and FS context-function R(32)S(16)P(2).

7.3 Value Predictors Architecture with Pipelining Considerations

In this section we examine the ramifications of placing a value predictor in the superscalar processor presented in Chapter 5. A discussion of relevant issues for two-level branch predictors can be found in[71, 72, 73].

7.3.1 Delayed Updates

In the proposed microarchitecture, an instruction is predicted in the fetch stage, and updates the predictor deterministically with the correct information when it retires in program order. Consequently, is possible for an instance of an instruction to be predicted before a previous instance of it has updated the predictor (*delayed–update*). Provided that delayed–updates are frequent, then lower prediction rates should be expected from most predictors since multiple predictions will be performed using the same information (unlike the case with immediate updates).

The likelihood of the occurrence of delayed–updates is a influenced by the number of dynamic instructions between refetch of the same static instruction, processor parameters such as width and window size, branch misprediction, and program characteristics. Fig. 7.6 shows the dynamic instruction distance between refetch of the same instructions for different SPEC95 benchmarks. For example, the data show that 1–50% (average 19%) of instructions, depending on the benchmark, execute within 32 dynamic instructions. This suggests as processor configurations get wider and deeper, and windows grow larger, delayed–updates may increasingly limit prediction accuracy. However the above analysis (a) does not take into consideration branch misprediction which can increase the distance between refetch of instructions on the correct-path of execution, and (b) does not give insight about the performance implications of delayed updates.

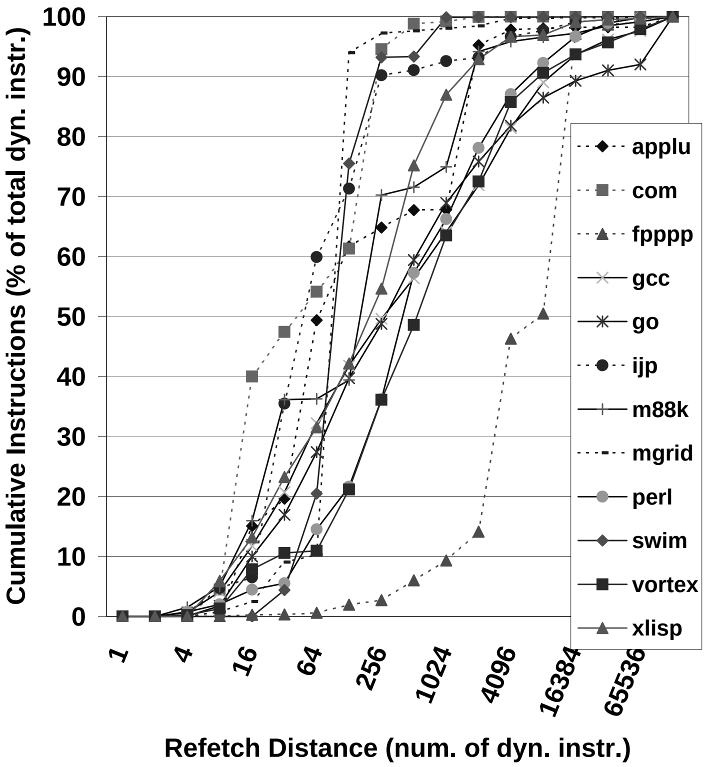

Figure 7.6: Refetch Distance

Provided that delayed–updates cause a performance degradation, one way to deal with them is to **not use predictions** from entries that have outstanding updates. However, this could decrease drastically the number of instructions that get predicted and the benefits of value prediction may be reduced. To alleviate this problem, predictor entries can be **speculatively updated** following an entry access. This is aimed at advancing the contents of the entry as if the updated value was available immediately after a prediction is made. Speculative updates may take place following the prediction of an instruction, using the predicted value, and after the execution of a predicted instruction, using the computed value. Updates at such times are speculative because the values used are not certain to be correct, for example a computed value can be incorrect because it was produced on the wrong path of execution or using speculative operands. The two types of speculative updates have different objectives:

speculative updating at time of prediction is aimed at helping with successive predictions of the same instruction, whereas the speculative updates following execution are aimed mainly at training predictors from instructions on the wrong execution path. Such instructions are often executed again on the correct path of execution and allowing them to speculative update a predictor can contribute in their correct prediction when they reexecute. Speculative updating after execution was applied in[57] to enable the reuse of instructions down the wrong execution path(Chapter 1). Although speculative updating following execution represents an interesting direction of research, in this work *we only consider speculative updates after prediction*.

The type of information that is speculatively updated and the information used to perform the update depends on the predictor used. For a stride predictor, when a prediction is made, the last value entry used to compute the prediction should be speculatively updated with the prediction. The stride need not be updated since the new stride is the same as the one used to compute the prediction. For a context-based predictor, the value history entry should be speculatively updated with the predicted value from the value prediction table. This essentially allows the history to point to the next value in a repeated sequence. For a hybrid predictor, the last value should be updated using the prediction, and the stride history should be updated with the predicted stride. Last value predictors do not need speculative updates since the predicted value and the value in the table are the same.

With speculative updating, an instruction can update a history entry twice, at prediction and retirement. Speculative update information could potentially be wrong. And mixing the two types of updates, in addition to being redundant, could lead to higher misprediction. To avoid this problem, two copies may need to be maintained for an entry that is speculative updated. The one to reflect the most recent speculative state whereas the other contain the most recent deterministic state. The two entries could be *synchronized* when a prediction is made from an entry with no outstanding predictions by replacing the contents of the speculative entry with the new speculatively updated history.

The above describes the functional requirements of a speculatively updated predictor. In the extreme case, an implementation of such a predictor will require two history tables (one for speculative and the other for deterministic updates). The two tables will be accessed in parallel and, based on whether or not there are outstanding updates, the appropriate contents will be used. Determining if there are outstanding predictions for an entry can be accomplished, for example, by performing an associative search of the outstanding instructions in the window that accessed the entry. We do not consider further the implementation of such a predictor, however, we assume its functionality in some of the simulations.

A subtle but possibly important option that arises due to delayed updates (with and without speculative update) concerns the information (**update information**) to use for performing deterministic updates: the **old** information at prediction time or the **current** contents of the entries. It is possible for an entry to have different contents at deterministic update time from prediction time. In this case, which entry should be updated in the value prediction table, the one where the prediction was read from (old) or the entry that is indexed using the current contents of the history table? Also when deciding whether to replace a value in the prediction table or updating the confidence, which value is compared against the correct value, the predicted value (old) or the current contents of the prediction entry to be updated(current)? Similar options exist for the state used for replacement and confidence.

Updating using the current contents of a predictor, requires a read–modify–write operation whereas using the old requires a write. If the latency of a read–modify–write operation is longer than a write, then possible advantages (if any) derived by using current update–information can be negated. In the experimentation, unless indicated otherwise, all deterministic updates are performed based on current update–information assuming the read–modify–write latency is equal to write latency. We will compare the performance of using old versus current update–information in Section 7.4.7. Note that when current update-information is used in combination with speculative updates, the current deterministic contents of the history table are used. Alternatively, the current speculative contents could have been used, however, they may have been updated with incorrect information and consequently update the table uselessly, which can lead to lower prediction accuracy.

7.3.2 Misprediction Recovery

Most current processors employ branch prediction to provide A smooth instruction supply. Mispredicted branches allow instructions down the wrong path to execute and update processor state. Most processors require on a branch misprediction to recover the processor state as if the instructions following the mispredicted branch were never fetched and executed. Recovery is necessary for maintaining correctness in registers and memory state. When using value predictors with speculative updates, however, recovery of value predictor state may be necessary for performance reasons. Speculatively updating a predictor is analogous to the speculative execution of instructions. And not removing the side -effects of speculative updates from instructions on the wrong path can lead to incorrect predictions.

To enable speculative updates and recovery on mispredicts, we propose the use of a buffer to hold speculative state (speculative history buffer) possibly for many instances of the same entry. This in principal similar to the approach used for dealing with speculative updates for transfer of control instructions[71, 72, 73]. To illustrate the required functionality of this buffer, we consider context-based value prediction with speculative updates. The value history table of the context-based predictor is indexed using the truncated PC of the predicted instruction. We assume a speculative history buffer where each entry contains the truncated PC of a predicted instruction and the speculatively updated history. Speculative updates affect the state in the speculative history buffer and the deterministic updates the state in the value history table. The buffer is maintained in a FIFO fashion according to program order - similar to a reorder buffer[77](Chapter 5). In fact the speculative history buffer may be unified with the retirement structure of a processor.

Fig. 7.7 shows different phases during prediction and update of two instances of the instruction with PC 100. When a prediction is needed, a query is made to both the value history table and the speculative history buffer. If there is no match in the buffer, the history from the table is used to obtain a prediction. The prediction is used to speculatively update the value history, however, this speculative value is saved in the buffer. Before the first instance of instruction 100 deterministically updates the table, one more instance requires prediction. This time the history from the buffer will be used to index the prediction table. As before, an entry will be allocated in the buffer for the new instance where the new speculatively update history will be stored. When the first instance of instruction 100 performs its update, the history in the value history table is used to determine which entry to update. Also the value history table contents are updated with the new value. The same process repeats when the next update is performed. Note that, at any given time, the contents of the value history table contain the state reflecting only the deterministic updates. Consequently, if a branch is mispredicted and precedes in program order both of the above instructions, the buffer can invalidate both entries and the correct state can be found in the value history table. If a branch between the two instructions is mispredicted, then the second instance will be invalidated and the speculative state of the first instruction will be preserved.

The above describes the functionality of a speculative history buffer – this does not suggests the best or even a good design for a speculative history buffer. We believe there are several alternatives to be considered in addition to the reorder buffer such as a history file, future file[77, 73] or possibly other structures. We will not explore the design of this structure in this work we will assume, however, its functionality when employing predictors with speculative updates. In Section 7.4.7 we compare the performance of configurations with and without speculative updates.

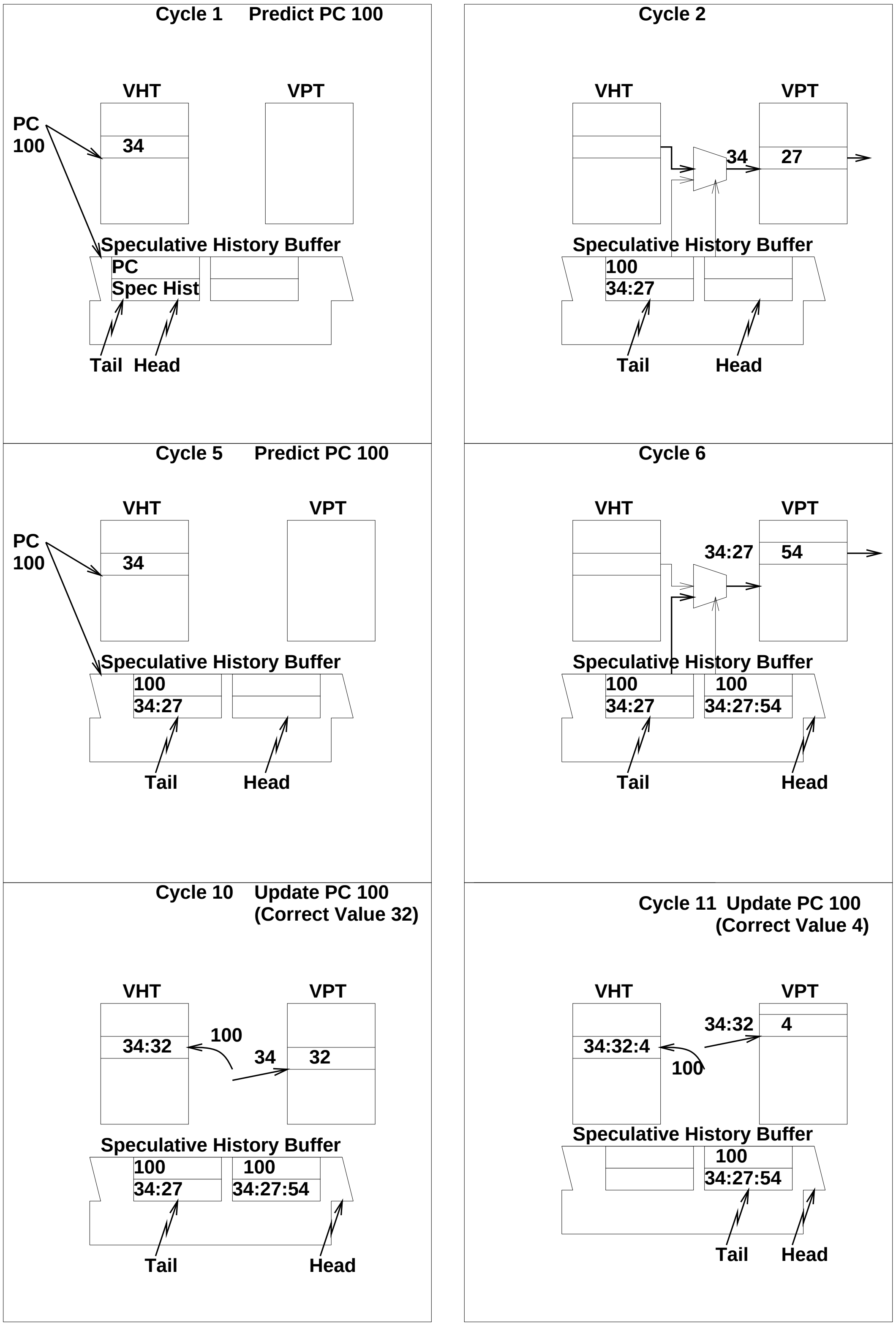

Figure 7.7: Example of Speculative Updates with a Context-Based Predictor

7.3.3 High Bandwidth Value Prediction

Current high performance processors have wide fetch and issue bandwidth. Consequently if value prediction is employed by a high performance processor, is likely that multiple predictions per cycle will be required. Supporting multiple predictions per cycle gives rise to two issues:

- Providing multiple predictions per cycle, and,
- Predicting multiple instances of the same instruction.

The above issues were also addressed recently for last value and stride predictors in[70].

We believe the value predictor organization should match the organization used to supply instructions. For example, many current processor employ an interleaved instruction cache capable of fetching any four consecutive instructions. To provide equal number of value predictions, a similar interleaved organization should be used for the value predictor. Fig. 7.8.a shows examples of how a last value and a context-based value cache predictors could be organized for a four-way processor.

Interleaved organizations can be effective for processors with widths greater than four, provided that long sequences of consecutive instructions are common (i.e. taken branches are infrequent). Long consecutive instructions can be achieved using code layout and loop unrolling techniques that minimize taken branches. Taken branches pose a problem for two reasons: their target PC cannot be determined in the same cycle they are fetched, and the PC of instructions from the target may "collide" with the PC of instructions before the taken branch when accessing the interleaved value predictor. The trace cache[32] is a technique proposed recently to enable high bandwidth instruction fetching even in the presence of taken branches. A trace cache can fetch multiple basic blocks simultaneously by storing dynamic sequences of instructions[32]. If a processor organization employs a trace cache then a trace-like value cache organization should be used to provide multiple value predictions. In Fig.

7.8.b we present the organization of a trace like value cache for a last value and context-based predictors capable of providing eight predictions. This cache can be indexed using the same index used to access the instruction trace cache. An instruction may have multiple entries in the value trace cache, unlike the case for the value cache, because instructions can belong to many consecutive sequences. When this happens it means that values produced by an instruction are divided into two or more sequences and may affect the predictability of the instruction.

The issue of multiple predictions for different instances of the same instruction in the same cycle does not appear to be important. This conclusion is based on the data in Fig. 7.6 that shows for most benchmarks a refetch distance of eight or less is rare, on the average close to 3%, and refetch distance of 16 is on the average 11%.

In this study we assume a bandwidth model where *shared* tables are accessed by as many instructions as need to be predicted in a given cycle. We believe this to be a good approximation of an interleaved predictor for issue widths of 4 and 8 instructions, however for issue width of 16 it is somewhat optimistic since multiple instances of an instruction will be predicted in the same cycle. Another consequence of sharing is that with context-based and hybrid prediction, instructions that would normally update one bank in an interleaved structure are able to update the entire shared table.

A possible directions for future research is to explore the effect of interleaving and trace caching on value prediction performance. Also in this study we predict all instructions — it is of interest to explore the necessity of doing so and determining the subset of instructions where prediction improve performance most.

7.3.4 Selected Configurations

For performance evaluation we consider the following **timing** models in combination with a predictor:

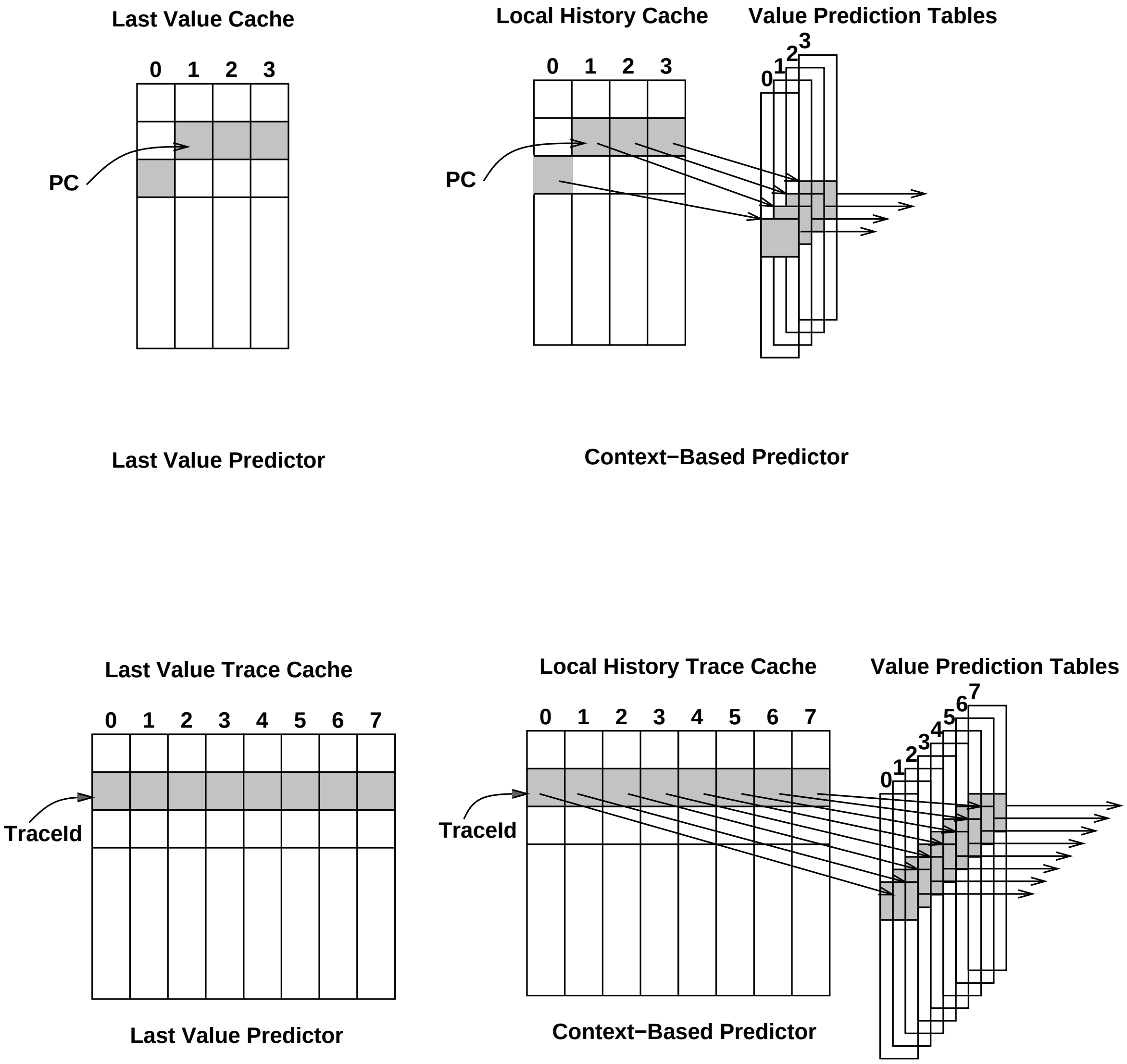

Figure 7.8: High Bandwidth Value Predictors

Model Name	Update Timing	Speculative Update	Update-Info
Immediate (IMM or I)	immediate	–	–
Spec-Current(SC) (default)	yes	yes	current
Spec-Old (SO)	delayed	yes	old
Det-Current (DC)	delayed	no	current
Det-Old (DO)	delayed	no	old

An entry with – indicates that a parameter is not relevant to a model. The model name or code for each model is used in presenting results. Note that the default model in simulations is the *Spec-Current*.

7.4 Results

7.4.1 Simulation Methodology

To evaluate the performance of the different value prediction models, we performed a simulation study for the microarchitecture presented in Chapter 5 using a modified out-of-order simplescalar simulator[74]. The simulations were conducted using the *real* configuration (described in Chapter 6). When value prediction is used the default speculative execution model was *great* and only instructions that produce 32 bit values were predicted. Simulations were performed for all integer SPEC95 benchmarks(Table 7.2). The benchmarks were compiled using the simplescalar *gcc* compiler with *-O3* optimization. Speedup was calculated as a ratio of the performance of a configuration with value prediction to an identical configuration without value prediction. When reporting average prediction rates we used arithmetic mean, so each benchmark effectively contributes the same number of total predictions. For average speedup calculation we used harmonic mean.

Benchmark	Input Flags	Dynamic Instr (mil)	Instructions Predicted (%)
compress	400000 e 2231	103	70.5
gcc	gcc.i	203	67.3
go	9 9	132	78.7
ijpeg	specmun.ppm	129	82.0
m88ksim	scrabbl.in	120	70.6
perl	modified train	40	63.9
vortex	modified train	101	61.9
xlisp	7 queens	202	61.7

Table 7.2: Benchmark Characteristics

7.4.2 Performance

Performance is reported for the following cases:

base–processor configuration without value prediction, (*BASE*),

base + last value prediction, (*LV*),

base + stride prediction, (*STR*),

base + context-based prediction, (*CXT*),

base + hybrid prediction, (*HYB*),

base + perfect value prediction, (*PERF*), and

base with bigger window, (*Issue Width/Window Size*)

The default predictor configurations are used, and the following four combinations of timing–models and prediction–confidence are examined: delayed update/real confidence (D/R), immediate update/real confidence (I/R), delayed update/oracle confidence (D/O), and immediate update/oracle confidence (I/O). The delayed update model uses the default timing model. Three base–processor configurations are considered with width/instruction window size of

4/24, 8/48 and 16/96.

Fig. 7.9 summarizes performance results by showing the average, maximum and minimum speedup for the different configurations, predictors, confidence and timing models.

The results show that value prediction can improve the base–processor performance on the average by about 3%–10% for the 4/24 configuration, 5%–15% for the 8/48, and 7%–20% for the 16/96 depending on the predictor, timing and confidence model used. The data indicate that the performance of last-value and stride predictors is almost identical across all configurations and benchmarks. The same observation holds for context-based and the hybrid predictors. This indicates that stride(hybrid) prediction provide no significant advantage over last value(context-based) prediction for the predictor configurations used. This may suggest that higher prediction accuracy - that stride(hybrid) predictors provide over last-value(context-based) - does not necessarily translate to higher performance. We examine the correlation of prediction accuracy to performance in Sections 7.4.3 and 7.4.4.

Another observation from the data is that across all configurations the performance of context-based and hybrid predictors is better than the performance of last-value and stride predictors. This should be expected because context-based and hybrid predictors can predict repeated non-stride sequences which are common in programs (Chapter 3), but which are not predictable by last value and stride predictors. One other conclusion from the data is that the speedup increases with larger issue width and window size because larger widths expose more dependences(Chapter 6)[70].

The data also suggest that last-value prediction is not sensitive to update timing, moving from delayed updates (D/X) to immediate updates (I/X) provides no improvement. Last-value prediction is not sensitive to update timing because an entry containing a correct prediction does not really need an updated value. And when a prediction is incorrect, immediately updating will only help if the updated value is useful in performing a prediction in the near future that would have been incorrect with delayed updating. The data suggest that such

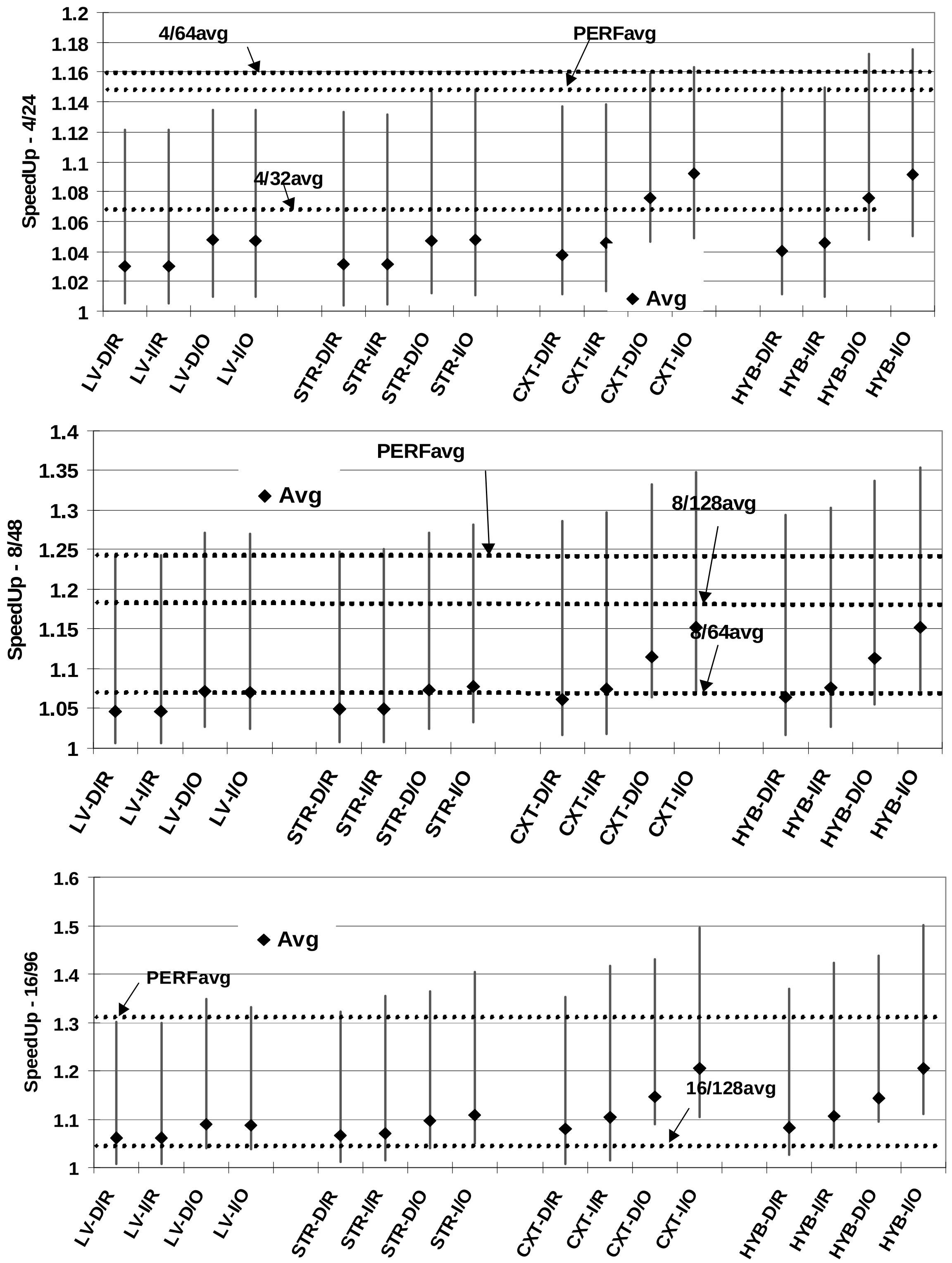

Figure 7.9: Average, Minimum and Maximum SpeedUp

events are rare and/or not on the critical path.

Similar observations about timing can be made for stride prediction, though stride prediction becomes a bit more sensitive to timing with bigger widths/windows. The increased sensitivity is due to an increase in repeated mispredictions. More specifically, for repeated stride sequences, a stride predictor always mispredicts the beginning of a loop. With delayed updates, any other predictions before the first misprediction is updated will also be mispredicted. The data show that the likelihood of performing such predictions increases with wider processors.

Contrary to last-value and stride prediction, context-based and hybrid predictors are sensitive to update timing. One of the main reasons for this is the sharing of the prediction table among multiple instructions. With immediate updates, when two or more instructions produce identical sequences, one of the instructions can be mispredicted but is able to train the predictor immediately and, as a result, the other instructions get predicted correctly. Another reason for the sensitivity to update timing is instructions that produce "almost" repeating sequences. As shown in Chapter 3, many of the correct predictions using context-based predictors are caused by instructions that are not 100% predictable. Immediately updating a predictor in the case of such instructions enables the context to point back to the correct sequence faster.

The data seem to support that confidence is a very significant performance parameter because moving from real confidence (X/R) to oracle confidence (X/O) provides a large performance increase (higher than the improvement achieved by immediate over delayed updating). This may indicate that either a lot of incorrect predictions are assigned high confidence and hence cause a lot of misspeculation, or many correct predictions are assigned low confidence and performance opportunity is lost. This indicates that the 3–bit resetting counter used to assign confidence is performing poorly. This also suggests that one of the keys for realizing the potential of value prediction is very accurate confidence predictors. This is examined further in Section7.4.3.

One other observation is that the performance achieved with realistic predictors is significantly lower than the available potential (PERF). This indicates that improving prediction accuracy can provide large improvements. The performance trends with increase prediction accuracy are examined in Section 7.4.4.

Finally, the data show that with increasing width/window size, value prediction may become a promising complexity effective technique. For example, the performance of configuration 16/96 with value prediction is, in many cases, close to or higher than with bigger window (16/128). This suggests that dependence elimination/tolerance techniques may enable same/higher performance with possibly less complex (narrower) processor organizations. This represents an interesting direction for future research.

7.4.3 Predictability

Fig. 7.10 shows the average prediction accuracy for the different predictors used in the previous section. Each graph contains the performance of four predictors for the different timing and processor configurations. Predictions are divided into four sets, the set of predictions that were correct with high confidence (YY), correct predictions with low confidence (YN), incorrect predictions with high confidence (NY), and incorrect predictions with low confidence (NN). The total number of correct predictions is the sum of sets YY and YN.

The results show that on the average 50% – 75% of the predictions are correct depending on the predictor, timing model and configuration used. As expected stride prediction is more accurate than last value, context-based is more accurate than stride and hybrid prediction is better than context. The improvement between last-value and stride and between context-based and hybrid is small as compared to the improvement when moving from a computational class predictor to a context-based class predictor, 5% and 15% respectively.

The data suggest that last-value predictor accuracy is not sensitive to timing and width/

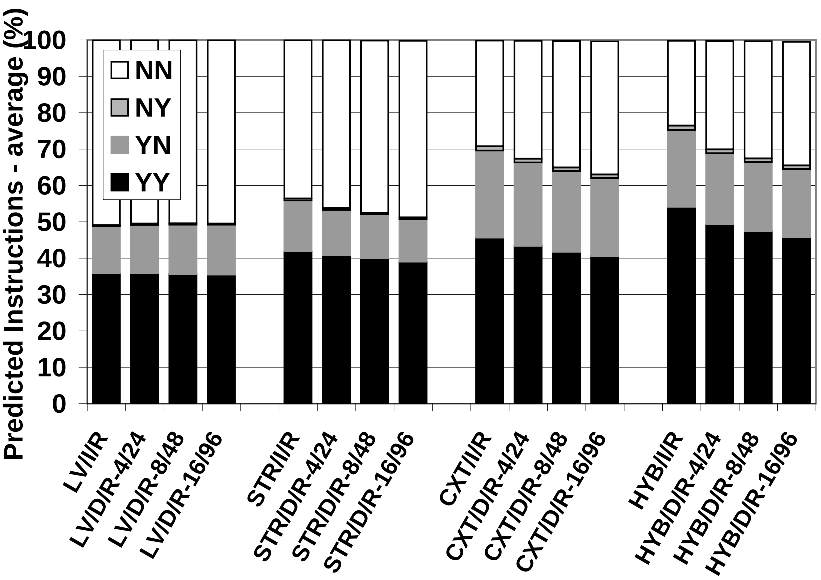

Figure 7.10: Average Prediction Accuracy

window size. This explains the invariance of last value prediction performance to the timing model in the previous section. On the other hand, the accuracy of stride, context-based and hybrid predictors is more sensitive to timing and width/window size. We observe that with delayed updates and increasing width/window size prediction rate decreases. This explains the performance decrease by these predictors with increasing width/window size in the previous section.

Another important observation is that the confidence method is successful in minimizing mispeculation (NY size is less than 1%), however this is done at the expense of a large set of correct predictions with low confidence (YN size is 10%–25% depending on the predictor, timing and configuration). This illuminates the large performance difference between real and oracle confidence that was observed in the previous section and reenforces the importance and need for accurate confidence estimation for value prediction.

The results reveal that prediction accuracy by itself is not always a good indication of the expected performance. For example, the hybrid predictor for benchmarks *compress* and *go* with immediate updates has about 8-9% higher prediction accuracy than the context-based predictor. However, the actual performance of context-based and hybrid prediction with oracle confidence (I/O) is the same for all configurations. This shows, as others have observed[12, 58, 59, 70], that performance is determined by the number of times a correctly predicted value enables the speculative execution of an instruction. With higher prediction rate, the likelihood of speculative execution is higher but higher prediction rate in general is not sufficient to guarantee it. Similar arguments can be made for misprediction. The correlation between prediction accuracy and performance is investigated in Section 7.4.4.

7.4.4 Performance and Prediction Accuracy Trends for different Predictors and Cost

This section aims at establishing prediction and performance trends among the different predictors for different costs. Fig. 7.11 shows the average prediction accuracy and speedup obtained for different predictors. The x-axis represents the number of entries of the prediction table used by a predictor. Each graph contains eight curves representing the behavior of the following predictors:

LV, last value predictor,

STR, stride predictor,

CXT-1024, context predictor with VHT of 1024 entries,

CXT-8192, context predictor with VHT of 8192 entries,

CXT-65536, context predictor with VHT of 65536 entries,

HYB-1024, hybrid predictor with VHT and LVT of 1024 entries,

HYB-8192, hybrid predictor with VHT and LVT of 8192 entries, and

HYB-65536, hybrid predictor with VHT and LVT of 65536 entries.

The results were obtained using the default timing model for an 8/48 processor configuration with oracle confidence.

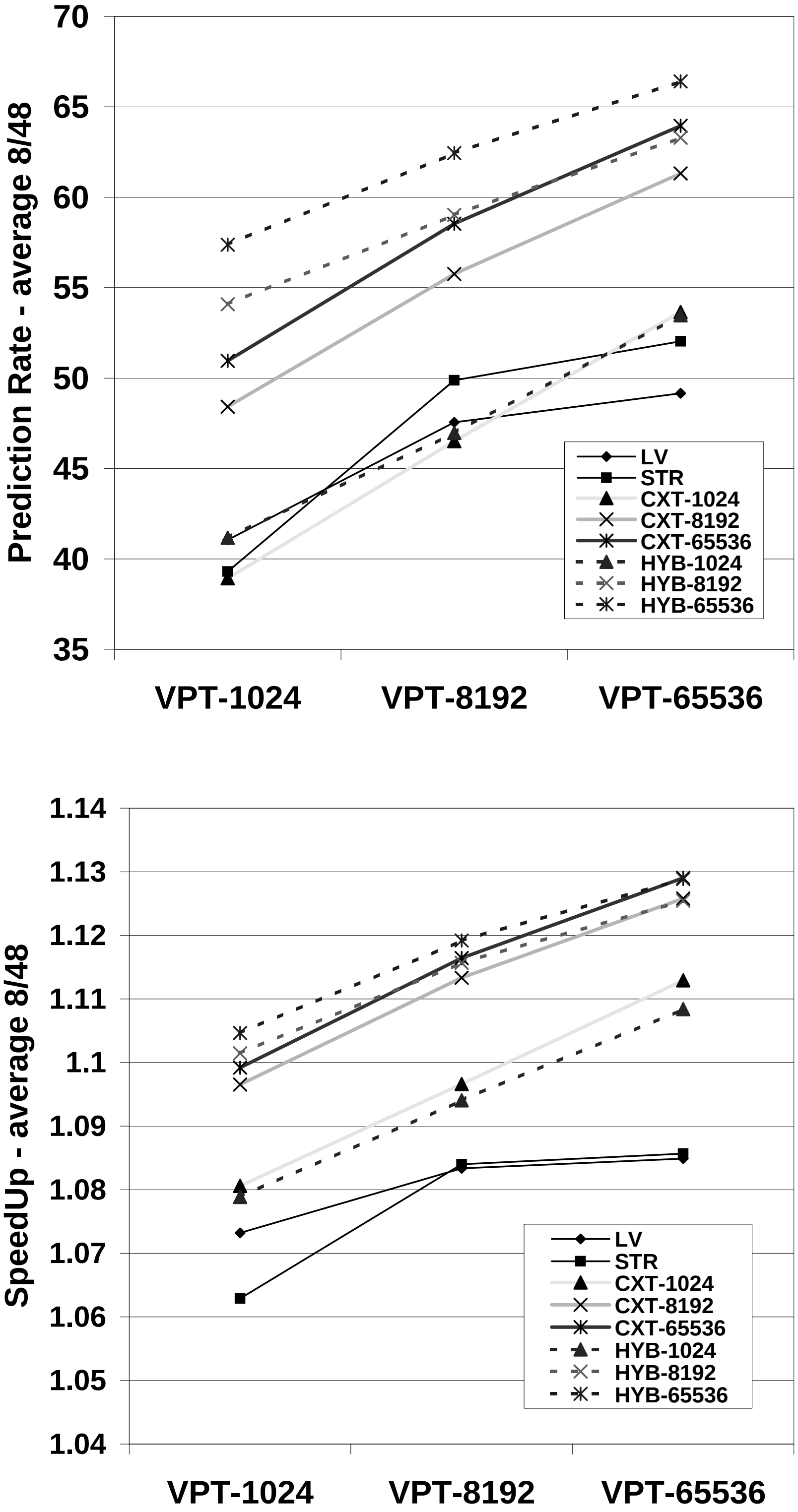

Figure 7.11: Average Prediction Accuracy and SpeedUp

The results show that, on the average, computational predictor configurations have better accuracy when compared to context and hybrid predictors with equal prediction table size and small history table (CXT-1024 and HYB-1024). Context-based and hybrid predictors accuracy increases with bigger history tables. This indicates that a 1024 entry history table suffers from significant aliasing between predicted instructions that impairs prediction accuracy. Increasing the history table size beyond 65536 entries may have small additional benefit since there is little benefit in going from an 8192 entry to a 65536 entry history table.

Increasing prediction table size helps last-value and stride predictors, but accuracy saturates for prediction tables with 8192 or more entries. Context-based and hybrid predictors appear to have significant potential with large tables (even bigger than 65536 entries). This is supported by the observation that prediction rate keeps increasing with bigger prediction table size, though the improvements come at increasingly higher cost. A possible alternative to increasing prediction table size (which may be undesirable due to longer latency and higher cost) is to employ better hash functions that utilize the table more effectively – we explore the performance of different hash functions and prediction table utilization in Section 7.4.5.

The above observations suggest that for context-based and hybrid predictors, it is important to have accurate value history per predicted instruction and big prediction tables capable of learning many predictions. Big prediction tables may be needed to reduce the likelihood from destructive interference with instructions that are not predictable and yet update the table. A possible direction of future research will be to establish the effects of such instructions and develop update methods that prevent such instructions from polluting prediction tables.

The accuracy achieved by last value and stride predictors of equal prediction table size is similar. Hybrid predictors appear to be more cost-effective compared with context-based prediction in achieving a given prediction accuracy, with the difference between the two predictors growing smaller with bigger prediction tables.

The data suggest that there is a correlation between prediction accuracy and performance for a given predictor. That is, increasing the accuracy of a predictor tends to increase performance. However, the results indicate a "weak" correlation between the prediction accuracy and performance of predictors from different classes. For example, in Fig. 7.11 on the average both last-value and stride prediction with 8192 entries achieve better accuracy than context and hybrid predictors with 8192 prediction entries and 1024 history entries. The relation is reversed in the speedup graph. This suggests that when comparing value predictors from different classes, it is more important to compare their performance than prediction accuracies.

7.4.5 Context Functions and Table Utilization

In this section, we attempt to determine good performing context-functions for context-based predictors. Prediction accuracy is studied for the context functions in Table 7.1. Figure 7.12 shows the average correct prediction rate for a context-based predictor with a 64K entry history and prediction tables using the C, FC, PFS and FS functions. The following PFS and FS functions were considered: PFS(16)(16)(p) and FS(32)(16)(p) with p= 16, 8, 5, 3 and 2 (corresponding to predictors with order = 1, 2, 4, 6, and 8). These results were obtained for a processor configuration with immediate timing model.

The data indicate that functions considering bits from the entire value outperform the functions that use a proper subset (FC and FS versus C and PFS). The data suggest that the FS functions have the best overall performance and are not very sensitive to the order of the predictor used. Accuracy increases with higher order but levels off, or even deteriorates, for large orders. This can be attributed to increase aliasing when hashing larger number of values. We examine the effect of order on prediction accuracy and performance for an FS function in the next section.

To study the efficiency of the FS(32)(16)(5) context function, we show in Fig. 7.12 the average cumulative total prediction and average cumulative normalized correct prediction in terms of prediction table utilization. The graph was obtained by sorting prediction table entries in decreasing order according to their utilization. This graph suggests that entries that are not accessed frequently do not contribute significantly to correct prediction. For example, 50% of the prediction table entries contribute to 5% of the correct predictions. This suggests that schemes with fewer value prediction table entries may be able to obtain the same accuracy (or conversely attain higher prediction with the smaller table size). Methods that can achieve this represent an interesting direction for future research.

7.4.6 Sensitivity of Performance to the Context-Function Shift Amount

In this section, we explore the sensitivity of prediction accuracy and performance to the shift amount of a context function for context-based and hybrid predictors. The results were obtained using an 8/48 processor configurations with immediate updating for a context-based predictor with 64K entry history and prediction tables and a hybrid predictor with 64K entry last value, history and prediction tables. The SFS context function FS(32)(16)(p) was used with p (*shift amount*) = 1, 2, 3, 4, 5, 6, 7 and 8. Note that the larger the shift amount, the smaller the number (order) of previous values that determine the context (corresponding to order 16, 8, 6, 4, 4, 3, 3, 2).

Fig. 7.13 shows the average results for the context-based predictor. The data suggest that both prediction rate and speedup are sensitive to the shift-amount. Typically speedup trends follows similar trends to prediction rate, however this is not always the case. For example, the speedup for benchmarks *compress* (Fig. 7.13) increases when prediction rate decreases. The hypothesis for this is that a change that causes a decrease in the prediction accuracy has two effects: enable new instructions to be predicted correctly and other correctly predicted instructions to become mispredicted. Though the second set is larger, the effect of the first set on performance is more important. The data indicate that, on the average, a shift amount of 4 is a good choice – however if an even better solution is desired, shift amount adaptivity

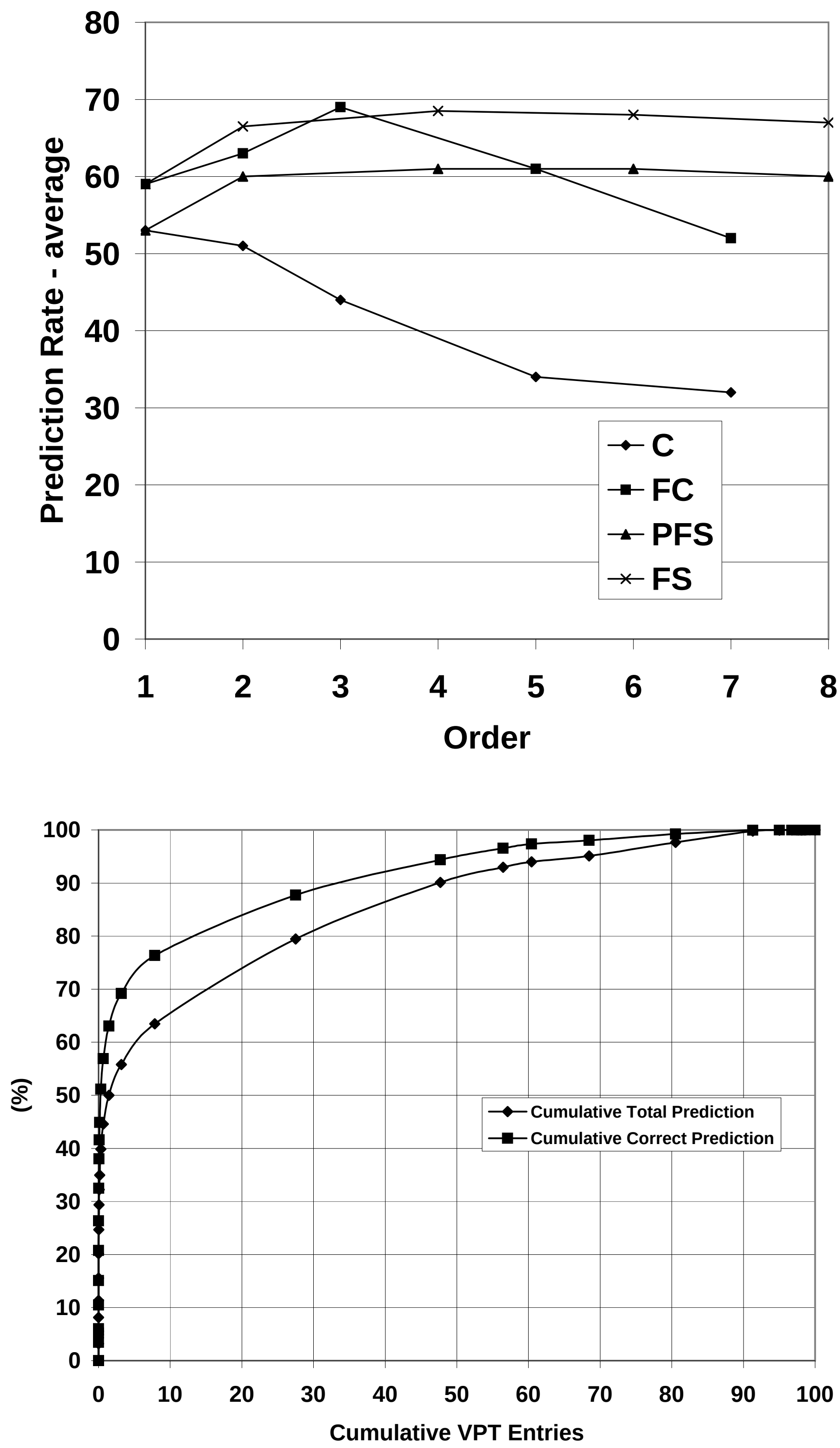

Figure 7.12: Prediction Accuracy for Different Context Functions and Table Utilization

should be explored[86].

Fig. 7.14 shows the average results for the hybrid predictor. The observations made for context-based prediction persist. The only significant difference is that small shift amounts have good average performance. The need for a smaller shift amount (and hence more values influencing the context) is due to the use of strides that tend to have smaller values, and having more of them may help reduce aliasing.

7.4.7 Timing Models

This section explores the effect of timing models on performance and prediction accuracy. The timing models SC, SO, DC and DO, discussed in section 7.3, are simulated for three processor configurations with width/window size 4/24, 8/48 and 16/96. Each configuration was simulated with real (R) and oracle (O) confidence. A context-based predictor with 64K entry history and prediction tables is used. Fig. 7.15 shows the average speedup and prediction accuracy for the different configurations.

The results show that speculative update timing models (SC and SO) have the best performance. The models have similar performance with oracle confidence, but have pronounced differences with real confidence. The performance difference between speculatively updating with current and old information is not significant, though using current information (SC) performs a little better than old (SO). The performance with no speculative updating and real confidence is better than using current information (DC), but with oracle confidence performance is better when using old information (DO).

The observed performance differences can be understood by the manifestation of the timing models on prediction accuracy. The data indicate that the total correct prediction accuracy (YY + YN) across the different models is about the same, with speculatively updated models having a slightly higher correct prediction accuracy. The main change in prediction behavior is that the use of old update information increases the number of incorrect predictions that are

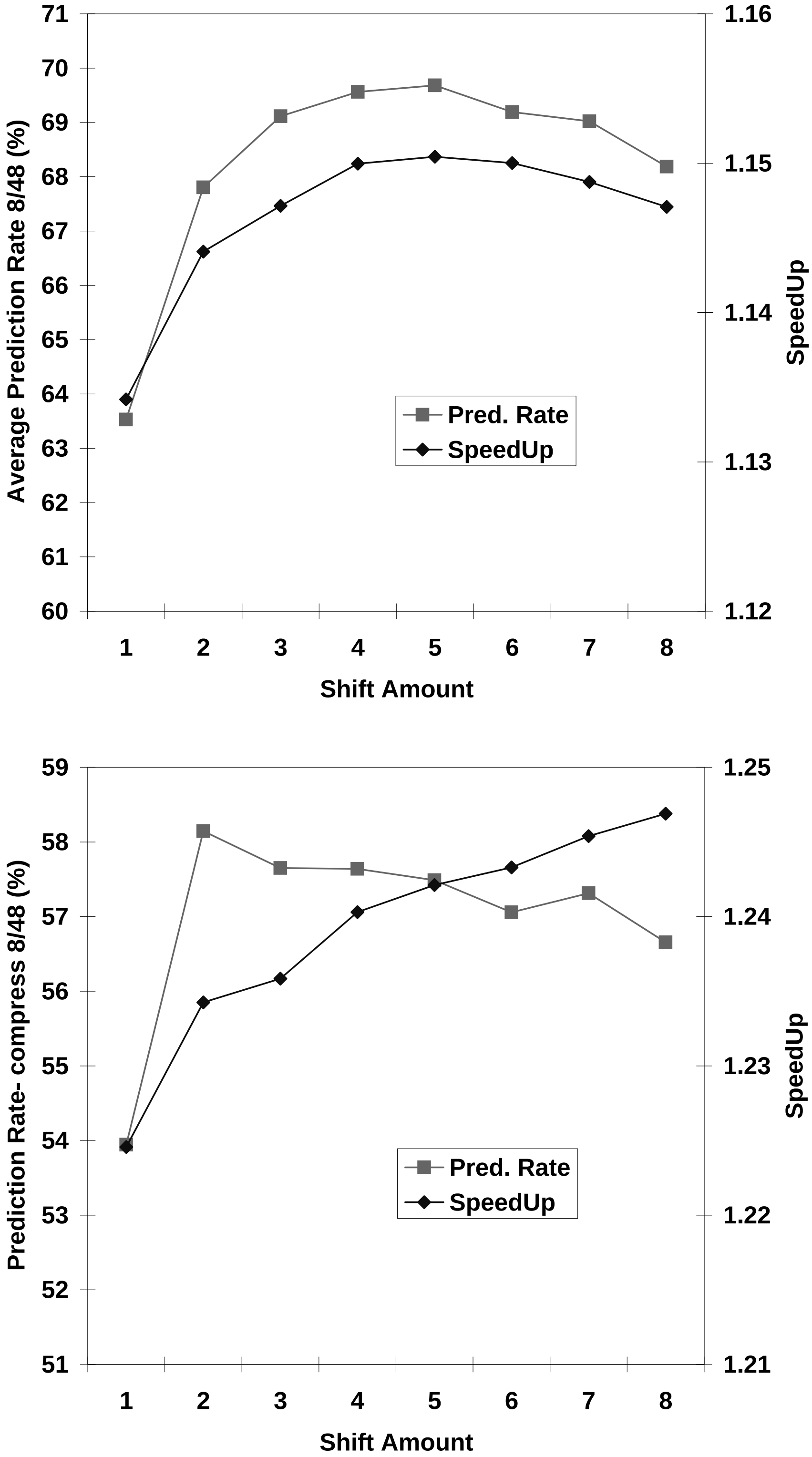

Figure 7.13: Prediction Accuracy and SpeedUp for Context-Based Predictor

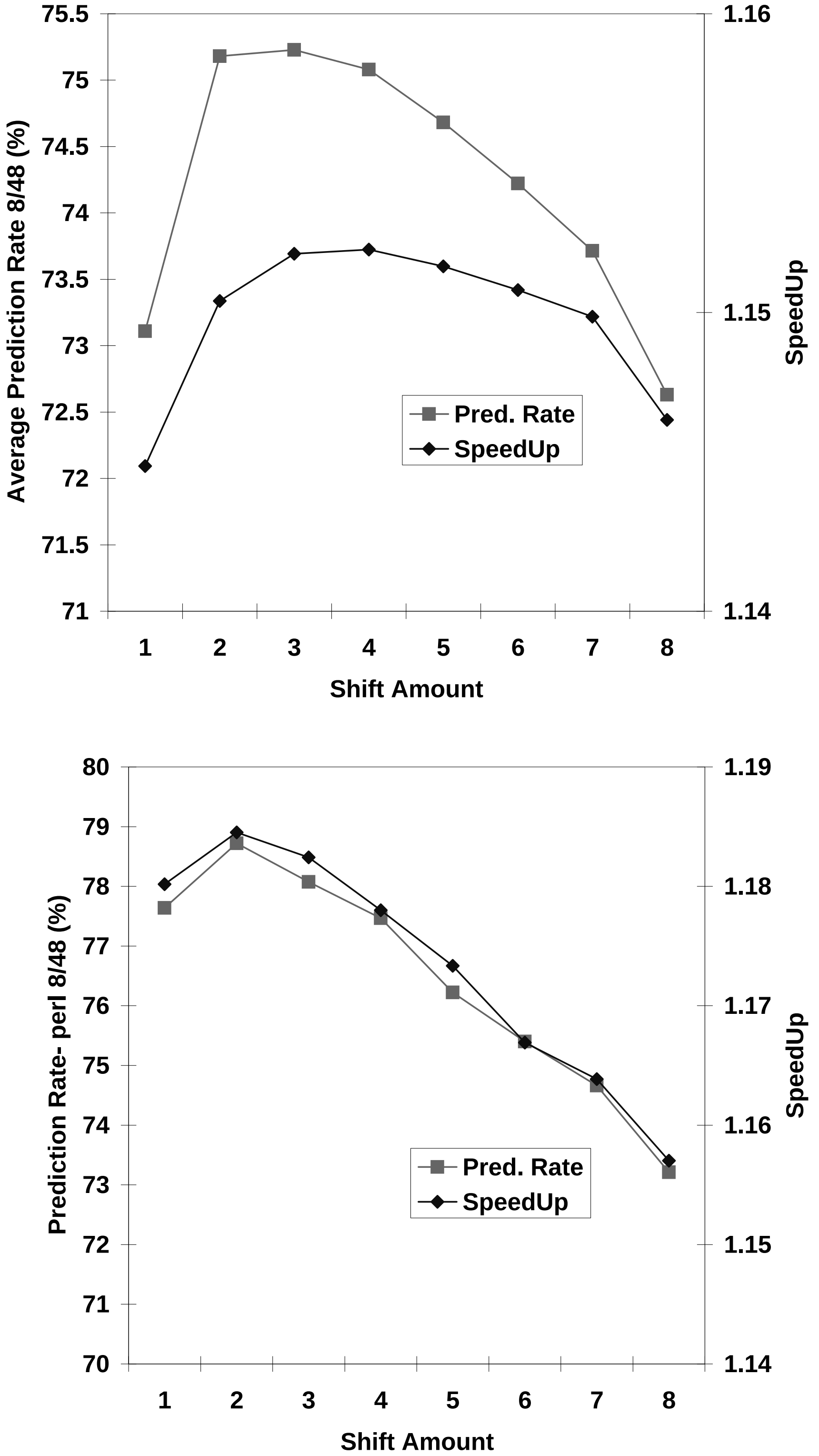

Figure 7.14: Prediction Accuracy and SpeedUp for Hybrid Predictor

classified as confident (NY) and the number of correct predictions that are classified as correct (YY). The model with no speculative updates and old update information (SO) has the most incorrect predictions that are classified as confident (NY). Consequently when SO model is used with real confidence, there is an increase in misspeculation and hence degradation in performance.

The data with real confidence appear to suggest that it may be preferable to have fewer correct and confident predictions (YY) than having more of them, if the number of incorrect predictions classified as confident (NY) increases.

Increasing width/window does not affect significantly the observations except a slight decrease in the total correct predictions by not speculatively updating models (DO and DC). This should be expected since with faster processing rates, due to bigger width, the likelihood of predicting an instruction before a previous instance of it increases.

7.4.8 Speculative Execution Models

This section investigates the performance of three speculative execution models: *good*, *great* and *super* discussed in Chapter 5. The performance of the models were obtained for three processor configurations 4/24, 8/48 and 16/96 using a context-based predictor with 64K entry history and prediction tables. Each configuration was studied for real (R) and oracle (O) confidence using the *SC* (D) and *IMM* (I) timing models (D/R, I/R, D/O and D/I). Fig. 7.16 shows the average speedup.

Two important observations can be made:(a) the *good* model behaves significantly worse as compared to the *great* and *super* models; in some cases having worse performance than the base configuration, and (b) there is no significant difference between the *great* and *super* models.

The first observation underlines the importance of fast verification latency. The verification latency for the *good* model was 1 cycle whereas for the *great* and *super* models it was 0. If 0

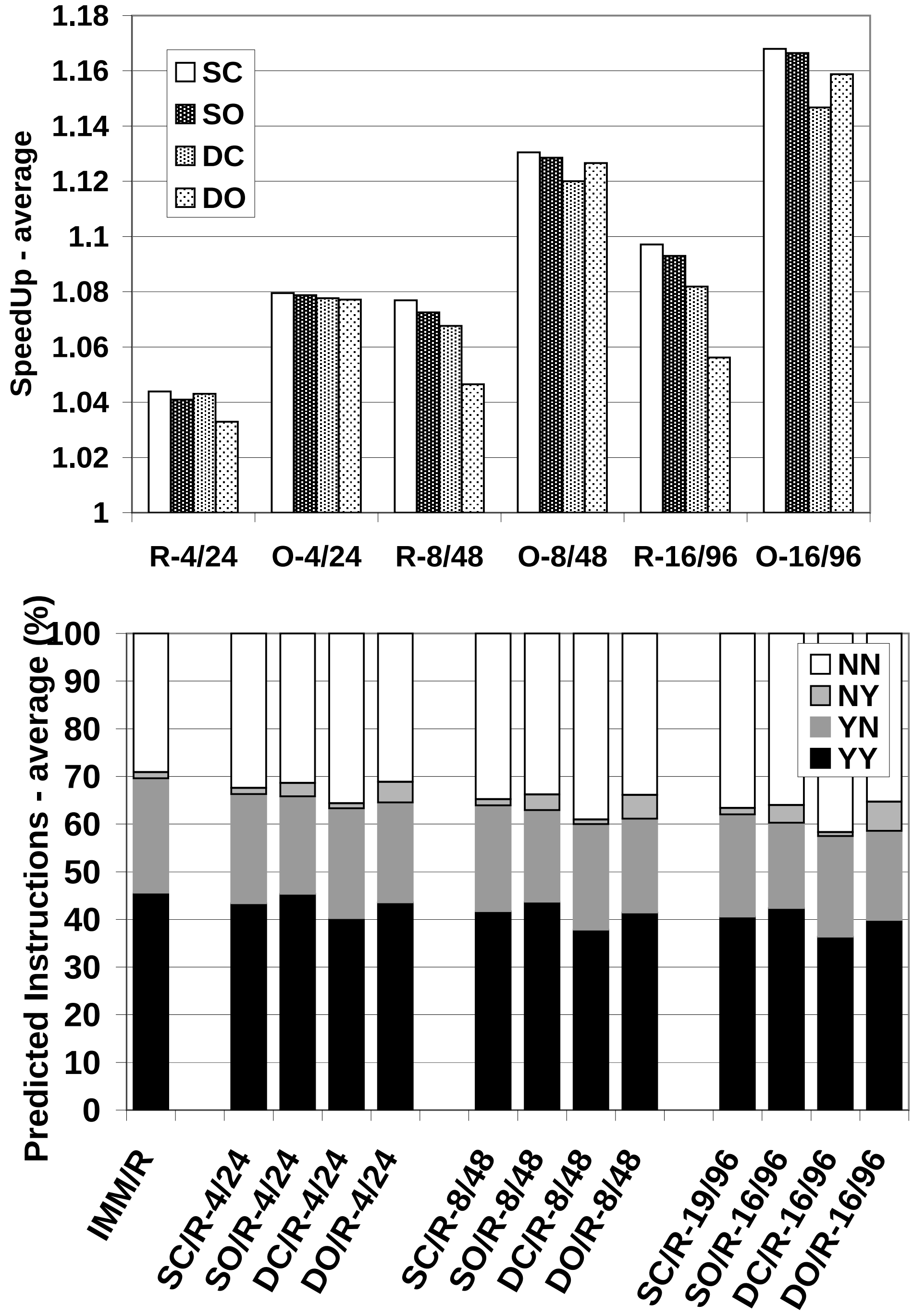

Figure 7.15: Timing Models Average SpeedUp and Prediction Accuracy

cycle verification is infeasible, it is imperative to explore speculative execution models where verification can be varied between 1 and 0, for example depending on whether an instruction has predicted inputs.

The small performance difference between the *super* and *great* models indicates that a cycle delay (a) in informing speculative branches and memory instructions that their inputs are (not)valid, and (b) in reissueing following mispeculation, are not critical to performance. Recall that branch and memory instructions are not allowed to be resolved speculatively. The reason that this cycle delayed is not so critical is because when these instructions are predicted correctly they enable the useful speculative execution of other instructions and hence their additional delayed is not usually exposed. Also the real confidence method used in this study, as we have observed earlier, allows very few mispeculations and thus quick reissue provided by the *great* model is not utilized. An interesting direction of future research will be to study the above models with different confidence mechanisms. In particular, we expect, with more frequent mispeculations, the benefits of the *great* model to be more substantial.

Another way to interpret the small performance difference between the *great* and *super* models is that accurate confidence can obviate the need for fast (and possibly complex) mechanisms for informing quickly speculative branch and memory instructions about the validity of their inputs. Recall that one of the reasons for using dedicated verification hardware (described in Chapter 5) was to communicate quickly to branch and memory instructions the speculative state of their inputs. The data indicate that this may not be important to performance. This, however, does not demonstrate that the dedicated verification hardware is not needed at all because is also used to perform selective invalidation. An interesting direction of future research will be to investigate the relative importance of different invalidation schemes (discussed in Chapter 5) and determine whether a dedicated hardware for invalidation is useful to performance.

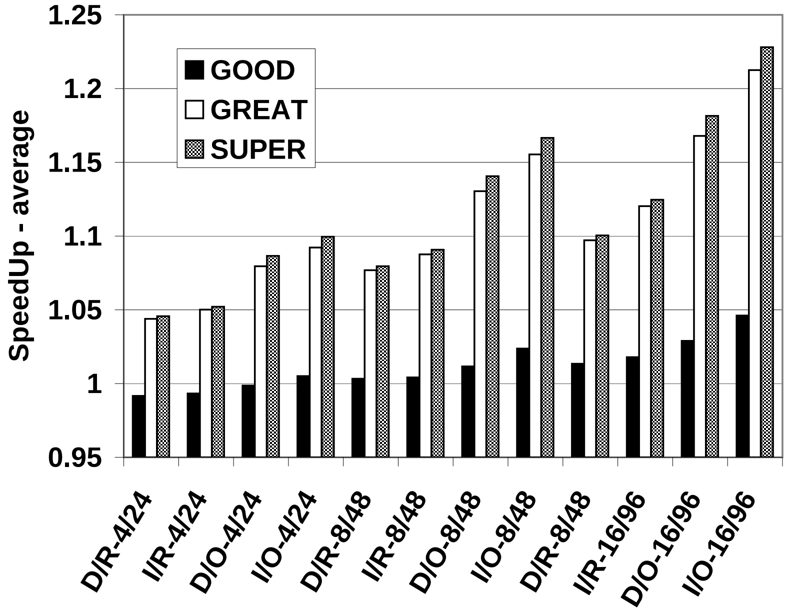

Figure 7.16: Speculative Execution Models Average Speedup

7.5 Summary

This chapter described the design space of value predictors. Various parameters related to the design of context-based and hybrid predictors, such as predictor table sizes and hash functions, were identified and discussed. We then considered the ramifications of pipelining, misprediction and multiple instruction fetch and issue on the design of several value predictors.

Selected predictor configurations and timing models were simulated for ideal and oracle confidence, various processor organizations and a few speculative execution models. The results reveal a number of interesting points.

Value prediction has potential to improve performance, for SPECINT95 benchmarks on the average 3%–20% depending on the predictor, timing and confidence model used. Also value prediction with increasing processor width value prediction provides larger improvements.

The best performing predictors were the context-based and hybrid. The performance of the predictors in the same class was similar, last-value with stride and context-based with hybrid. Context-based and hybrid predictors were found to be more sensitive to timing but all predictors were very sensitive to confidence. The data show that accurate confidence is critical for the success of value prediction. The data also indicated that value prediction can be a promising complexity-effective approach because, with smaller window size, it can achieve the same or higher performance of a bigger window without value prediction.

Prediction and cost trends suggest that hybrid predictors are the most cost effective predictors for achieving a given prediction accuracy. The data support the existence of a correlation between prediction accuracy and performance for a given predictor. That is, improved accuracy translates to better performance. However, the data indicate that when comparing different predictors, accuracy is not a good indication of their performance relation. For example, the results show the context-based predictor to achieve the same or better performance compared to a hybrid predictor for equivalent design points where context-based predictors have on the average lower prediction accuracy than hybrid.

We also presented a cost-effective hash function useful for forming an index to the prediction table for context-based predictors. This function has the best overall performance and does not require the complete previous values in the context. The best order for this function and SPECINT95 benchmarks was found to be four for context-based predictors and two for hybrid predictors. The best performing order, however, is different between benchmarks. Examination of the utilization of the prediction table entries when using this function, for a context-based predictor, showed that the prediction table was underutilized. This suggests that possibly better prediction can be achieved with the same table size or conversely a smaller table can be used to achieve the same accuracy.

Timing models do not affect significantly the prediction accuracy but they influence the number of incorrect predictions that are confident. The best performing model is one that employs speculative updates and deterministic updates with current information. No speculative updating and deterministic update with current information also has good performance. However, the model with no speculative updating and deterministic update with old information, has the largest number of incorrect predictions that are confident and hence the lower performance. With oracle confidence, the "simple" timing model (no speculative updating and update deterministically with old information) can perform as well as the speculative updating models. This indicates the importance of accurate confidence for value prediction and suggests that improved confidence can obviate the need for speculatively updating.

Finally, we found that fast verification latency is very critical to value prediction. A significant performance reduction was observed when going from a zero to one cycle verification latency. The data also showed that a single cycle delay, in the validation of instructions that cannot be speculative resolved but are predicted, is not very critical.

Chapter 8

Conclusions

8.1 Summary

In this thesis we performed an analysis of value predictability and consider its application to a superscalar processor. More specifically we discussed/presented:

- Two general prediction models for capturing the behavior of program values: computational and context-based models. The first is proposed for the prediction of value sequences that obey an arithmetic relation whereas the latter is applicable to repeated value sequences. It was demonstrated that if high value prediction accuracy is desired, is necessary to consider models that capture repeated behavior.

- Basic program predictability properties: *generation*, *propagation* and *termination*. These properties are used to demonstrate that: (a) program predictability is mainly due to program control structure and immediate values, but not input data, (b) predictability is often terminated when all incoming information is predictable, (c) predictability of an event is often influenced by only a few predecessors, and (d) long sequences of instructions with both predictable inputs and predictable outputs are common. The first observation is interpreted as an indication of the robustness of value prediction. The other observations can be used as guidelines for (a) improving predictors, and (b) developing software/hardware paradigms to exploit predictability.

- Ramifications of value prediction on the overall design of a processor. Issue, invalidation and verification policies were identified as being central to value prediction. We examined their relation and presented several options for their implementation. It was demonstrated that invalidation and verification of dependent chains of instructions can be performed in "parallel", contingent on technology constrains. A set of variables was proposed to describe concisely the parameters influenced by value prediction and speculative execution.

- The design of several value predictors. Emphasis was given on the implementation of two new predictors: a context-based predictor, and a hybrid predictor that combines context-based and stride prediction. The following consequences on the design of value predictors were also discussed: (a) delayed updates, speculative updates, and the use of old versus current information for deterministic updates, (b) removing the side-effects of speculative updates on branch mispredictions, and (c) high bandwidth value prediction.

- Performance trends when using perfect value prediction with perfect and real branch prediction. The data suggest that value prediction can reduce the instruction window size required to achieve a given level of parallelism.

- Performance trends with realistic value predictors that indicate: (a) value prediction has potential to improve performance, for SPECINT95 benchmarks, on the average 3%–20% depending on the predictor, timing and confidence model used, (b) hybrid predictors are the most cost effective predictors for achieving a given prediction accuracy, (c) there is a correlation between prediction accuracy and performance for a given predictor, i.e. improved accuracy translates to better performance, (d) when comparing different predictors, accuracy is not a good indication of their performance relation, (e) the predictors with the best performance are context-based and hybrid, (f) delayed-updating a context-based predictor does not affect significantly its correct prediction accuracy, it increases however the number of incorrect predictions with high confidence (i.e. mispeculations), and (g) fast verification latency and accurate confidence are very critical to value prediction, significant performance reductions are observed when going from a zero to one cycle verification latency and from oracle to realistic confidence.

8.2 Future Work

We recognize that this work does not represent a complete treatment of the subject of value predictability. It is our hope, however, that this work will lead to more comprehensive research of value predictability. As it was pointed out in the thesis there are several promising directions of future research, below we describe some of them:

- Improve context-based and hybrid predictors accuracy. This can be achieved with the use of (a) static or dynamic heuristics that *select* the instructions to predict[70], (b) additional structures that learn and eliminate repeated mispredictions, (c) better hash functions, and (d) adaptivity in the predictor order.

- Develop accurate mechanisms to assign confidence to value predictions.

- Use Side-Effect equality. Equality can be performed in at least two ways: by comparing the predicted value against the actual value (**value equality**) or by comparing the side-effects of the predicted and actual value (**side-effect equality**). Value equality requires strict equality whereas side-effect equality may not. Consequently, side-effect equality can have higher potential. Side-effect equality was also suggested in[15].

- Explore different speculative execution models and their interaction with timing and confidence models. One of the aims should be to establish the importance of selective invalidation as compared to complete squash, and identify organizations that are suited for value prediction/speculation.

- Investigate performance of value prediction in combination with speculative branch and/or speculative load resolution.

- Examine the effects of value prediction and other dependence elimination/tolerance techniques on window size and other processor resources. This may reveal that such techniques can reduce the importance of big windows.

- Study the effects of interleaving and trace caching on value prediction accuracy and performance.

- Determine the additional importance of value prediction with decentralized microarchitectures [31, 87, 11]. Technology trends indicate that communication latencies will grow and consequently value prediction may be more useful in tolerating long communication latencies.

- Fundamental studies of predictability and unpredictability.

- Compiler optimizations that exploit predictability.

- Applications to multiple processors.

The future quests aimed at eliminating program redundancy promise to be exciting.

Bibliography

[1] W. Buchholz, *Planning a Computer System*. McGraw-Hill Book Company, NY, 1962.

[2] J. E. Smith, "A Study of Branch Prediction Strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, pp. 135–148, May 1981.

[3] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction," *IEEE Transactions on Computers*, vol. 30, pp. 478–490, July 1981.

[4] J. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design," *IEEE Computer*, January 1984.

[5] T.-Y. Yeh and Y. N. Patt, "Two-Level Adaptive Branch Prediction," in *Proceedings of the 24th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 51–61, November 1991.

[6] S.-T. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation," in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 76–84, October 1992.

[7] S. McFarling, "Combining Branch Predictors," Tech. Rep. DEC WRL TN-36, Digital Western Research Laboratory, June 1993.

[8] C. Young and M. D. Smith, “Improving the Accuracy of Static Branch Prediction Using Branch Correlation,” in *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 232–241, October 1994.

[9] R. Nair, “Dynamic Path-Based Branch Correlation,” in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 15–23, December 1995.

[10] P.-Y. Chang, E. Hao, and Y. N. Patt, “Target Prediction for Indirect Jumps,” in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 274–283, June 1997.

[11] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, “Trace Processors,” in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 138–148, December 1997.

[12] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, “Value Locality and Data Speculation,” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, October 1996.

[13] F. Gabbay and A. Mendelson, “Speculative Execution Based on Value Prediction,” Tech. Rep. (Available from http://www-ee.technion.ac.il/fredg), Technion, November 1996.

[14] M. H. Lipasti and J. P. Shen, “Exceeding the Dataflow Limit via Value Prediction,” in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 226–237, December 1996.

[15] M. H. Lipasti, “Value Locality and Speculative Execution,” Tech. Rep. CMU-CSC-97-4, Carnegie Mellon University, May 1997.

[16] Y. Sazeides and J. E. Smith, “The Predictability of Data Values,” in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 248–258, December 1997.

[17] K. Wang and M. Franklin, “Highly Accurate Data Value Prediction using Hybrid Predictors,” in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 281–290, December 1997.

[18] A. Moshovos, S. E. Breach, T. J. Vijaykumar, and G. Sohi, “Dynamic Speculation and Synchronization of Data Dependences,” in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 181–193, June 1997.

[19] G. Z. Chrysos and J. S. Emer, “Memory Dependence Prediction using Store Sets,” in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 142–153, June 1998.

[20] M. J. C. Fu, S. Larin, and T. Conte, “Value Speculation Scheduling for High Performance Processors,” in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1998.

[21] Y. Sazeides and J. E. Smith, “Modeling Program Predictability,” in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 73–84, June 1998.

[22] Y. Sazeides and J. E. Smith, “Implementations of Context–Based Value Predictors,” Tech. Rep. ECE-TR-97-8, University of Wisconsin-Madison, Dec. 1997.

[23] D. W. Anderson, F. J. Sparacio, and R. M. Tomasulo, “The IBM 360 Model 91: Processor Philosophy and Instruction Handling,” *IBM Journal of Research and Development*, vol. 11, pp. 8–24, January 1967.

[24] B. T. Hailpern and B. L. Hitson, "S-1 Architecture Manual," Tech. Rep. STAN-CS-79-715, Stanford University, January 1979.

[25] T.-Y. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," in *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 257–266, May 1993.

[26] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The Agree Predictor: A Mechanism for Reducing Negative Branch History Interference," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 284–291, June 1997.

[27] C. Lefurgy, P. Bird, I.-C. K.Chen, and T. N. Mudge, "Improving Code Density Using Compression Techniques," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 194–203, December 1997.

[28] M. Evers, P.-Y. Chang, and Y. N. Patt, "Using Hybrid Branch Predictors to Improve Branch Prediciton in the Presence of Context Switches," in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 3–11, May 1996.

[29] Q. Jacobson, S. Bennet, N. Sharma, and J. E. Smith, "Control Flow Speculation in Multiscalar Processors," in *Procedings of the 3rd International Symposium on High-Performace Computer Architecture*, pp. 218–229, February 1997.

[30] Q. Jacobson, E. Rottenberg, and J. E. Smith, "Path-Based Next Trace Prediction," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 14–23, December 1997.

[31] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar Processors," in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 414–425, June 1995.

[32] E. Rotenberg, S. Bennet, and J. E. Smith, “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching,” in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 24–34, December 1996.

[33] K. Driesen and U. Holzle, “Accurate Indirect Branch Prediction,” in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 167–178, June 1998.

[34] S. McFarling and J. Hennessy, “Reducing the Cost of Branches,” in *Proceedings of the 13th International Symposium on Computer Architecture*, May 1986.

[35] J. A. Fisher and S. M. Freudenberger, “Predicting Conditional Branch Directions From Previous Runs of a Program,” in *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*, October 1992.

[36] J. R. Larus, “Efficient Program Tracing,” *IEEE Computer*, vol. 26, pp. 52–61, May 1993.

[37] S. Mahlke and B. Natarajan, “Compiler Synthesized Dynamic Branch Prediction,” in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 153–164, December 1996.

[38] N. Gloy, C. Young, J. B. Chen, and M. D. Smith, “An Analysis of Dynamic Branch Prediction Schemes on System Workloads,” in *Proceedings of the 23rd International Symposium on Computer Architecture*, pp. 12–21, May 1996.

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

[40] M. Evers, S. J. Patel, , R. S. Chappel, and Y. N. Patt, "An Analysis of Correlation and Predictability: What Makes Two-Level Branch Predictors Work," in *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.

[41] I.-C. K. Cheng, J. T. Coffey, and T. N. Mudge, "Analysis of Branch Prediction via Data Compression," in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 128–137, October 1996.

[42] T. C. Bell, J. G. Cleary, and I. H. Witten, *Text Compression*. Prentice-Hall Inc., New Jersey, 1990.

[43] A. J. Smith, "Cache Memories," *ACM Computing Surveys*, vol. 14, pp. 473–530, September 1982.

[44] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers," in *Proceedings of the 17th International Symposium on Computer Architecture*, pp. 364–373, May 1990.

[45] S. Palacharla and R. E. Kessler, "Evaluating Stream Buffers as a Secondary Cache Replacement," in *Proceedings of the 21st International Symposium on Computer Architecture*, pp. 24–33, April 1994.

[46] J. L. Baer and T. F. Chen, "An Effective on-chip Preloading Scheme to Reduce Data Access Penalty," in *Proceedings of Supercomputing*, pp. 176–186, November 1991.

[47] T. F. Chen, "Data Prefetching for High Performance Processors," Tech. Rep. TR-93-07-01, University of Washington, July 1993.

[48] T. F. Chen and J. L. Baer, “Effective Hardware-Based Data Prefetching for High Performance Processors,” *IEEE Transactions on Computers*, vol. 44, pp. 609–623, May 1995.

[49] R. J. Eickemeyer and S. Vassiliadis, “A Load Instruction Unit for Pipelined Processors,” *IBM Journal of Research and Development*, vol. 37, pp. 547–564, July 1993.

[50] T. M. Austin, D. N. atikatos, and G. S. Sohi, “Streamlining Data Access with Fast Address Calculation,” in *Proceedings of the 22nd International Symposium on Computer Architecture*, pp. 369–380, June 1995.

[51] S. Mehrotra and L. Harrison, “Quantifying the Performance Potential of a Data Prefetch Mechanism for Pointer-Intensive and Numeric Programs,” in *CSRD TR 1458, University of Illinois, Urbana-Champaign*, November 1995.

[52] S. Mehrotra and L. Harrison, “Examination of a Memory Access Classification Scheme for Pointer Intensive and Numeric Programs,” in *Proceedings of the 10th International Conference on Supercomputing*, pp. 133–140, May 1996.

[53] D. Joseph and D. Grunwald, “Prefetching using Markov Predictors,” in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 252–263, June 1997.

[54] W. S. M. Lipasti, S. Kunkel, and R. Roediger, “Software Prefetching in Pointer and Call Intensive Environments,” in *Proceedings of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 231–236, November 1995.

[55] C. Luk and T. Mowry, “Compiler based Prefetching for Recursive Data Structures,” in *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 222–223, October 1996.

[56] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?," in *Proceedings of the 30th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 270–280, December 1997.

[57] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse," in *Proceedings of the 24th International Symposium on Computer Architecture*, pp. 194–205, June 1997.

[58] Y. Sazeides, S. Vassiliadis, and J. E. Smith, "The Performance Potential of Data Dependence Speculation & Collapsing," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 238–247, December 1996.

[59] J. Gonzalez and A. Gonzalez, "Speculative Execution Via Address Prediction and Data Prefetching," in *Proceedings of the 11th International Conference on Supercomputing*, pp. 196–203, July 1997.

[60] E. Jacobsen, E. Rotenberg, and J. E. Smith, "Assigning Confidence to Conditional Branch Predictions," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 142–152, December 1996.

[61] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun, "Confidence Estimation for Speculation Control," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 122–131, June 1998.

[62] S. S. Mukherjee and M. D. Hill, "Using Prediction to Accelerate Coherence Protocols," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 179–190, June 1998.

[63] K. So and R. N. Rechtschaffen, "Cache Operations by MRU Change," *IEEE Transactions on Computers*, vol. 37, pp. 700–709, June 1988.

[64] A. Agarwal and S. D. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches," in *Proceedings of the 20th International Symposium on Computer Architecture*, pp. 179–190, May 1993.

[65] A. Sodani and G. S. Sohi, "An Empirical Analysis of Instruction Repetition," in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 35–45, October 1998.

[66] D. Hammerstrom and E. Davidson, "Information Content of CPU Memory Referencing Behavior," in *Proceedings of the 4th International Symposium on Computer Architecture*, pp. 184–192, March 1977.

[67] D. Citron and L. Rudolph, "Creating a Wider Bus Using Caching Techniques," in *Proceedings of the 1st International Symposium on High-Performace Computer Architecture*, pp. 90–99, February 1995.

[68] M. H. Lipasti and J. P. Shen, "Superscalative Microarchitecture for Beyond AD 2000," *Computer*, September 1997.

[69] E. Rotenberg *personal communication*, 1999.

[70] F. Gabbay and A. Mendelson, "The Effect of Instruction Fetch Bandwidth on Value Prediction," in *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pp. 272–281, June 1998.

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

[72] E. Hao, P.-Y. Chang, and Y. N. Patt, "The Effect of Speculatively Updating Branch History on Branch Prediction Accuracy, Revisited," in *Proceedings of the 27th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1994.

[73] S. Jourdan, T.-H. Hsing, J. Stark, and Y. N. Patt, "The Effects of Mispredicted-Path Execution on Branch Prediction Structures," in *PACT'96*, October 1996.

[74] D. Burger, T. M. Austin, and S. Bennett, "Evaluating Future Microprocessors: The SimpleScalar Tool Set," Tech. Rep. CS-TR-96-1308, University of Wisconsin-Madison, July 1996.

[75] G. Sohi, "Instruction Issue Logic for High Performance, Interruptible, Multiple Functional Unit, Pipelined Computers," *IEEE Transactions on Computers*, vol. 39, pp. 349–359, March 1990.

[76] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM Journal of Research and Development*, vol. 11, pp. 25–33, January 1967.

[77] J. E. Smith and A. R. Pleszkun, "Implementing Precise Interrupts in Pipelined Processors," *IEEE Transactions on Computers*, vol. 37, pp. 562–573, May 1988.

[78] S. Palacharla, N. P. Jouppi, and J. E. Smith, "Complexity-effective superscalar processors," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206–218, June 1997.

[79] A. Sodani and G. S. Sohi, "Understanding the Differences between Value Prediction and Instruction Reuse," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 205–215, December 1998.

[80] E. Rotenberg, Q. Jacobson, and J. E. Smith, "A Study of Control Independence in Superscalar Processors," Tech. Rep. CS-TR-98-1389, University of Wisconsin-Madison, December 1998.

[81] M. Lipasti *personal communication*, 1999.

[82] Y. Sazeides *personal communication*, 1996.

[83] G. Reinman and B. Calder, "Predictive Techniques for Aggressive Load Speculation," in *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture*, pp. 127–137, December 1998.

[84] M. S. Lam and R. P. Wilson, "Limits of Control Flow on Parallelism," in *Proceedings of the 19th International Symposium on Computer Architecture*, pp. 46–57, May 1992.

[85] Z. Li, Y. Sazeides, and J. E. Smith, "Delta Predictors," *unpublished report*, 1998.

[86] T. Juan, S. Sanjeevan, and J. J. Navarro, "Dynamic History-Length Fitting: A Third Level of Adaptivity for Branch Prediction," in *Proceedings of the 25th International Symposium on Computer Architecture*, pp. 155–166, June 1998.

[87] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 1–12, June 1997.