

ABSTRACT

Title of dissertation: SPECULATIVE DATA
DISTRIBUTION IN SHARED
MEMORY MULTIPROCESSORS

Sean Leventhal

Dissertation directed by: Professor Manoj Franklin
Department of Electrical Engineering

This work explores the possibility of using speculation at the directories in a cache coherent non-uniform memory access multiprocessor architecture to improve performance by forwarding data to their destinations before requests are sent. It improves on previous consumer prediction techniques, showing how to construct a predictor that can handle a tradeoff of accuracy and coverage. This dissertation then explores the correct time to perform consumer prediction, and show how a directory protocol can incorporate such a scheme. The consumer prediction enhanced protocol that is developed is able to reduce the runtime of a set of scientific benchmarks by 10%-20%, without substantially reducing the runtime of other benchmarks; specifically, those benchmarks feature simple phased behavior and regularly distribute data to more than two processors.

This work then explores the interaction of consumer prediction with two other forms of prediction, migratory prediction and last touch prediction. It demonstrates a mechanism by which migratory prediction can be implemented using only the stor-

age elements already present in a consumer predictor. By combining this migratory predictor with a consumer predictor, it is possible to produce greater speedups than did either individually. Finally, the signatures of the last touch predictor can be applied to improve the performance of consumer prediction.

SPECULATIVE DATA DISTRIBUTION
IN SHARED MEMORY MULTIPROCESSORS

by

Sean Leventhal

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Manoj Franklin, Advisor
Professor Donald Yeung
Professor Bruce Jacob
Professor Charles Silio
Professor Alan Sussman

© Copyright by
Sean Leventhal
2008

Dedication

As with most graduate students, this dissertation has consumed days or weeks of my time without mercy. I doubt I would have ever come this far if it were not for my wife, Patty, watching over my shoulder, putting up with me being distracted from the rest of the world, and occasionally drawing me back to it. For every late night of typing and programming she had to put up with, every time she had to deal with all the things I did not have time for, and every time she acted as a sounding board for something very dry, I dedicate this dissertation to my wife, Patty.

Acknowledgments

I would like to acknowledge a number of people for helping me reach this point:

- My parents — for their patient proofreading of something full of words that they could not identify.
- Dr. Franklin — for all the time he put in getting me to this point.

Table of Contents

List of Figures	viii
List of Abbreviations	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Summary of Work	3
1.2.1 Consumer Prediction	4
1.2.2 Migratory Prediction	5
1.2.3 Timing Prediction	6
1.2.4 Combined Coherence Prediction	6
1.3 Research Contributions	7
2 Background	10
2.1 Multiprocessor Design	10
2.2 Memory Sharing Patterns	12
2.2.1 Migratory Data	12
2.2.2 Widely Shared Data	13
2.3 Multiprocessor Memory Coherence	16
2.4 Coherence Prediction	19
2.4.1 Inward Coherence Prediction	19
2.4.2 Outward Coherence Prediction	21
2.4.3 Administrative Coherence Prediction	23
2.5 Coverage of Previous Work	25
2.5.1 Division of Sharing Patterns	25
2.5.2 Multi-Read To One Writer Sharing	29
2.5.3 Migratory Data	33
3 Experimental Methodology and Terminology	34
3.1 Measuring the Performance of Coherence Prediction	34
3.1.1 Measuring Prediction Quality	34
3.1.2 Measuring System Behavior	35
3.1.3 Predictor Naming Conventions	37
3.2 Experimental Methodology	38
3.3 Directory Protocol Implementation	43
4 Consumer Prediction	54
4.1 Theoretical Consumer Prediction Behavior	55
4.1.1 Consumer Predictor Architecture	55
4.1.1.1 Perceptron Consumer Prediction Design	55
4.1.1.2 Consumer Predictor Training	57
4.1.2 Sensitivity and PVP of Consumer Prediction Schemes	59
4.1.3 Performance of Predictors Acting on Restricted Information	62

4.2	Structure of A Consumer Predictor	66
4.2.1	Supporting A Consumer Predictor in A Directory Protocol . .	66
4.2.1.1	Coherence and Training Issues	66
4.2.1.2	Implementation Requirements	69
4.2.2	Supporting Program Counter Based Techniques	71
4.2.3	Directory and Cache Changes to Support Consumer Prediction	72
4.3	Behavior of Consumer Prediction	78
4.3.1	Measuring the Behavior of a Consumer Predictor	78
4.3.2	Varying the Predictor Design	80
4.3.2.1	Runtime Results Across Varying Predictor Designs .	81
4.3.2.2	Message Transmission Results Across Varying Predictor Designs	82
4.3.2.3	Bandwidth Usage Results Across Varying Predictor Designs	87
4.3.2.4	Miss Rates Across Varying Predictor Designs	88
4.3.2.5	Response Time Results Across Varying Predictor Designs	90
4.3.3	Tying Processor Number Into the Indexing Scheme	93
4.3.3.1	Writer-Mixing	93
4.3.3.2	Address-Mixing	97
4.3.4	Global Consumer Prediction	98
4.3.5	Variance of Network Latency	100
4.3.5.1	Runtime Results	101
4.3.5.2	Effect on Message Transmission	104
4.3.5.3	Bandwidth Usage	108
4.3.5.4	Directory Response Time	108
4.3.6	Relationship Between Benchmark and Predictor Performance .	113
4.4	Tuning the Perceptron	114
4.4.1	Varying the Perceptron Threshold	114
4.4.2	Training a Perceptron With Non-Existent Data	116
4.5	Applying Confidence Estimation to Consumer Prediction	117
4.5.1	Supporting A Confidence Estimator	117
4.5.2	Performance of Confidence Estimated Address-Indexed Consumer Prediction	119
4.6	Summary	119
5	Migratory Prediction	121
5.1	Migratory Consumer Predictor Architecture	122
5.1.1	Predictor Architecture	122
5.1.2	Protocol Modification	125
5.2	Address Based Migratory Consumer Predictors	126
5.2.1	Effects of History Depth on Migratory Prediction	126
5.2.2	Preventing Lock-In in Migratory Prediction	130
5.2.2.1	State Additions	131
5.2.2.2	History Table Modification	133

5.2.2.3	History Table Corruption	134
5.2.2.4	Not-Migratory Counters	134
5.2.3	Results of Lock-In Prevention Mechanisms	135
5.3	Instruction Based Migratory Consumer Predictors	139
5.3.1	Naive Instruction-Based Migratory Prediction	140
5.3.2	Second-Look Instruction-Based Migratory Prediction	141
5.3.3	Forward-Ahead Second-Look Instruction-Based Migratory Prediction	143
5.3.4	Second-Look Instruction-Based Migratory Prediction	145
5.3.4.1	Interprocessor Message Effects of Instruction Based Prediction	146
5.3.4.2	Bandwidth Effects of Instruction Based Migratory Prediction	148
5.3.4.3	Directory Latency of Instruction Based Migratory Prediction	149
5.3.4.4	Miss Rate of Stores With Migratory Prediction in Use	150
5.3.4.5	Miss Rate of Instruction Based Migratory Prediction	151
5.4	Indexing a Migratory Predictor With Mixed Information	151
5.4.1	Address-Mixed Migratory Prediction	151
5.4.2	CPU-Mixed Migratory Prediction	152
5.5	Summary	153
6	Timing Prediction	156
6.1	Self Invalidation	156
6.1.1	Changes to the Directory And Cache to Support Self Invalidation	159
6.1.2	Training The System	161
6.1.2.1	Confidence Estimator Training	161
6.1.2.2	External Training Information	166
6.1.3	Summary of Training Variation	170
6.2	Indexing Other Predictors with Last Touch Signatures	171
6.2.1	Consumer Prediction	172
6.2.2	Migratory Prediction	175
6.3	Summary	176
7	Combining Coherence Prediction Mechanisms	178
7.1	Potential Benefits of Combining Migratory Prediction With Consumer Prediction	179
7.2	Implementing a Joint Consumer/Migratory Predictor	179
7.3	Address-Indexed Joint Consumer/Migratory Prediction	182
7.3.1	Instruction-Indexed Joint Consumer/Migratory Prediction	184
7.4	Summary	185

8	Conclusions	190
8.1	Consumer Prediction	190
8.2	Migratory Prediction	191
8.3	Timing Prediction	192
8.4	Combined Prediction	192
8.5	Future Work	193
	Bibliography	195

List of Figures

2.1	Migratory Sharing Example	14
2.2	Widely Shared Data Example	15
2.3	Five Different Categories of Coherence Events.	27
2.4	Many Readers To One Writer Ideal Speculative Behavior	30
2.5	The ideal behavior of last touch prediction on an MROW event	31
2.6	The ideal behavior of an ILT on a MROW event	32
3.1	Measurements of Prediction Quality	35
3.2	Simulation Parameters Used	40
3.3	State Transition Diagram of an MSI Protocol.	43
3.4	States of the Cache	46
3.5	States of the Directory	47
3.6	Cache Events	48
3.7	Directory Events	49
3.8	Directory State Machine	50
3.9	Cache State Machine	51
4.1	General Consumer Predictor Architecture	54
4.2	Perceptron Consumer Predictor Architecture	56
4.3	Structure of a Perceptron Predictor	57
4.4	Co-optimal Consumer Predictors by Trace Performance	61
4.5	Behavior of several perceptrons across variations in threshold	63
4.6	Trace Based Consumer Predictor Behavior by Benchmark	64
4.7	Cache-Consumer Predictor Race Conditions	68

4.8	Three options for implementing speculative data forwarding.	70
4.9	Consumer Prediction Cache States	74
4.10	Consumer Prediction Cache Events	74
4.11	Consumer Prediction Cache State Machine	75
4.12	Example of Consumer Prediction Behavior	76
4.13	Predictor Designs Considered Here	81
4.14	Consumer Prediction Relative Execution Time	82
4.15	Consumer Prediction Relative Invalidations	83
4.16	Consumer Prediction Relative Requests	86
4.17	Consumer Prediction Relative Bandwidth	88
4.18	Consumer Prediction Relative Read Misses	89
4.19	Consumer Prediction Relative Write Misses	90
4.20	Consumer Prediction Read Latency	92
4.21	Consumer Prediction Write Latency	92
4.22	Runtime of Mixed Writer-Instruction Indexing	95
4.23	Invalidation Count of Mixed Writer-Instruction Indexing	95
4.24	Request Count of Mixed Writer-Instruction Indexing	96
4.25	Read Miss Rate of Mixed Writer-Instruction Indexing	97
4.26	Runtime of Mixed Address-Instruction Indexing	98
4.27	Runtime Benefits of Global Prediction	99
4.28	Global Prediction Invalidation Count Changes	100
4.29	Consumer Prediction Runtime Across Network Latency	102
4.30	Consumer Prediction Invalidations Across Network Latency	106
4.31	Consumer Prediction Requests Across Network Latency	107
4.32	Consumer Prediction Bandwidth Across Network Latency	109

4.33	Consumer Prediction Read Latency Across Network Latency	111
4.34	Consumer Prediction Write Latency Across Network Latency	112
4.35	Effects of Varying Perceptron Threshold on Runtime	115
4.36	Effects of Varying Perceptron Threshold on Request Count	115
4.37	Effects of Varying Perceptron Threshold Invalidation Count	116
4.38	Benefits of Perceptron Recognizing "Non-Data"	117
4.39	Confidence Estimated Consumer Prediction Performance	120
5.1	Example of Migratory Prediction	124
5.2	Migratory Prediction Directory Events	126
5.3	Migratory Prediction Cache Events	126
5.4	Migratory Prediction Cache State Machine	127
5.5	Migratory Prediction Directory State Machine	128
5.6	Address-Indexed Migratory Prediction Performance	129
5.7	Address-Indexed Migratory Prediction Request Count	130
5.8	Directory Protocol of State Additions Scheme	132
5.9	"Lock-in" Preventing Migratory Prediction Runtime	136
5.10	"Lock-in" Preventing Migratory Prediction Invalidation Count	136
5.11	"Lock-in" Preventing Migratory Prediction Request Count	137
5.12	"Lock-in" Preventing Migratory Prediction Bandwidth	138
5.13	"Lock-in" Preventing Migratory Prediction Write Latency	138
5.14	"Lock-in" Preventing Migratory Prediction Write Miss Count	139
5.15	Naive Instruction-Indexed Migratory Prediction Performance	142
5.16	Migratory Prediction Directory Events with PC information	143
5.17	Migratory Prediction Directory State Machine with PC information	144

5.18	Instruction-Indexed Migratory Prediction Runtime	146
5.19	Instruction-Indexed Migratory Prediction Invalidation Count	147
5.20	Instruction-Indexed Migratory Prediction Request Count	147
5.21	Instruction-Indexed Migratory Prediction Bandwidth	148
5.22	Instruction-Indexed Migratory Prediction Read Latency	149
5.23	Instruction-Indexed Migratory Prediction Write Latency	150
5.24	Instruction-Indexed Migratory Prediction Write Miss Count	151
5.25	Address-Instruction Mixed Migratory Prediction	152
5.26	CPU-Instruction Mixed Migratory Prediction	153
6.1	Speculative Self Invalidation Example	157
6.2	The structure of Lai and Falsafi’s Last Touch Predictor	158
6.3	Self Invalidation Cache States	160
6.4	Self Invalidation Cache Events	160
6.5	Self Invalidation Cache State Machine	161
6.6	Self Invalidation Confidence Estimator Design Tradeoff	165
6.7	Self Invalidation Training Logic Size Tradeoff	166
6.8	Externally Trained Self Invalidation Performance	168
6.9	Externally Trained Self Invalidation Message Counts	169
6.10	LastTouch-Indexed Consumer Prediction Runtime	173
6.11	LastTouch-Indexed Consumer Prediction Invalidation Count	173
6.12	LastTouch-Indexed Consumer Prediction Request Count	174
6.13	LastTouch-Indexed Migratory Prediction Runtime	176
7.1	Example of Poor Consumer Predictor Performance	180
7.2	Address-Indexed Combined Migratory/Consumer Runtime	183

7.3	Instruction-Indexed Combined Migratory/Consumer Runtime	186
7.4	Instruction-Indexed Combined Migratory/Consumer Invalidations . .	187
7.5	Instruction-Indexed Combined Migratory/Consumer Requests	188

List of Abbreviations

CMP	Chip Multiprocessor
CC-NUMA	Cache Coherent Non-Uniform Memory Access
FN	False Negative
FP	False Positive
ILT	Invalidation Lines Table
MPM	Message Passing Multiprocessor
MROW	Many Readers To One Writer
MSI	Modified-Shared-Invalid
NUMA	Non-Uniform Memory Access
OROW	One Reader To One Writer
OWMR	One Writer To Many Readers
OWOR	One Writer To One Reader
OWOW	One Writer To One Writer
PC	Program Counter
PVP	Predictive Value of a Positive Test
S-COMA	Simple-Cache Only Memory Architecture
SMMP	Shared Memory Multiprocessor
SMP	Symmetric Multiprocessor
TN	True Negative
TP	True Positive

Chapter 1

Introduction

1.1 Motivation

In the last three years, the first single chip multiprocessors (CMPs) have entered the market of general purpose computers. These include the Pentium Extreme Edition [51] from Intel and the Athlon 64 X2 [53] from AMD. In the embedded systems market, the Cell processor [17], designed by IBM as a high-end embedded processor, contains 9 distinct processing units, each with its own instruction stream. It is effectively a CMP and was designed for use in the Playstation 3, Sony's newest video game console. Its main competitor, the XBox 360 [54] also employs a CMP. Multiprocessors are now entering the world of general purpose computing and consumer electronics, but they are not new. As of November 2007 over ninety percent of the 500 fastest supercomputers in the world had more than 1024 processors [52].

Multiprocessors are valuable for a number of reasons. Modern supercomputer designers choose to use multiprocessors because single processors cannot provide performance in the range they desire. Many servers use multiprocessors because the tasks they perform, such as database lookups, web servers, etc., are naturally parallel [10, 46]. Desktops and workstations are turning to multiprocessing because additional optimizations to single processor architectures are providing diminishing returns in terms of speedup, while at the same time dramatically increasing verifi-

cation complexity [17] and power consumption [44]. Verification time can be greatly reduced by relying on previously tested processors and then linking them together into a CMP. However, additional verification of the coherence protocol that links these processors may be necessary.

It is a necessity that the individual processors in a multiprocessor can communicate with one another. One class of common general purpose multiprocessors — shared memory multiprocessors — do all communication through memory. Additionally, with the emergence of CMP processors, it is increasingly likely that the individual nodes of message passing multi-processors will themselves be shared memory multiprocessors. The memory must provide performance with caching, and at the same time maintain coherence among the caches. The speed with which coherence events take place limits the ability of the processors to communicate and share data. As individual cores have become more powerful, the percentage of execution time devoted to coherence action has increased [41].

Several techniques have been proposed to enhance the speed of coherence protocols. Originally, these took the form of modifications to the state machine that made up the protocol [7, 11, 45]. Later, coherence predictors were developed that preemptively identified events, and initiated them early to provide speedups [19, 22, 27, 28, 38, 42, 48] as were prefetching schemes [25, 37, 41, 50]. However, only rarely have these later schemes been evaluated in terms of their performance benefits.

These past works have always addressed only a single class of events at a time, and frequently focus on prediction accuracy, rather than speedup or the effect

of speculation on the system behavior. To cover all possible events speculatively requires a large number of different predictors and modifications to the coherence protocol. From a hardware designer's point of view, adding numerous different schemes to a cache may be unacceptable. Each of these schemes interacts with the others in an unknown way, requires access to all requests made to the cache, and needs additional space. Given these three problems, it seems unlikely that a system designer will include more than one or two of them. Further, none of these predictors were studied in the context of varying system designs, and few were studied in the context of a single full system design. Whether coherence prediction is more useful in systems with low or high network latency is not a question that has been previously addressed, nor is how these distinct prediction mechanisms can be merged into a single predictor. This dissertation addresses some of these concerns with a unified coherence predictor.

1.2 Summary of Work

The work presented here focuses on the interactions between different forms of coherence prediction and the behavior of a multiprocessor. The end goal in approaching this problem was to develop a single architecture of predictors that would operate cooperatively to move data to processors before those processors requested access to the data. This study attempts to accomplish this goal by focusing on three types of prediction: Consumer Prediction, Migratory Prediction, and Timing Prediction.

Together, consumer prediction and migratory prediction can provide greater speedups than either can alone. However, in general the additional benefits of using all at once are minimal.

1.2.1 Consumer Prediction

Consumer Prediction has been previously studied by Kaxiras and Young [19]. In consumer prediction, the readers of a written line are identified before they send requests for the data. Kaxiras and Young’s work focused specifically on the accuracy of a few predictor schemes, and addressing the extremes of sensitivity and PVP, rather than a tradeoff. This dissertation expands on their work, showing new ways to predict, and exploring the actual mechanics of prediction; specifically when to predict, where to predict and how to train the predictor. It then continues to analyze the behavior of a multiprocessor with consumer prediction.

The results presented here show that there is a class of benchmarks that benefit from consumer prediction, and a class that does not. Specifically, consumer prediction best targets applications with strictly controlled phase behavior and memory ownership. Consumer prediction is weakest in aiding benchmarks that use pointer based data structures, such as trees.

These results also show the importance of simulating the complete system, rather than working with traces, as the actual sharing patterns exhibited by the benchmarks can change when consumer prediction is in place. For example, most of the Volrend benchmark’s (from the SPLASH-2 benchmark suite [49]) sharing ex-

ists to distribute data from slower processors to faster processors. When consumer prediction increases the speed of these slower processors, the amount of sharing changes as different processors are involved in these exchanges, and fewer are necessary. These kinds of feedback effects of prediction have not been properly modeled by previous work in this area, by virtue of their focus on trace-based simulation.

Consumer prediction can increase the speed of requests for Modify copies of lines, despite the fact that it does not transmit them speculatively. Because consumer prediction distributes the data earlier, it is less likely that requests for Modify permission will arrive during a transient state. That means that it is more likely that requests to the directory for Modify permission will be serviced immediately.

1.2.2 Migratory Prediction

Migratory Prediction was first identified by Cox et. al [7]. It is a form of sharing in which a single processor reads and writes a line, and then passes that line on to another processor that will read and write. This work focusses on implementing migratory prediction in such a way that it can be combined with consumer prediction and timing prediction, and requires little additional hardware.

It demonstrates a migratory predictor that uses the same training information and stored history information that is already used by consumer prediction. Several options for training this predictor are also explored. This dissertation also discusses the complexities of implementing this predictor, and how the directory can be modified to take advantage of it.

In general the performance benefits of migratory prediction are lower than the benefits of consumer prediction.

1.2.3 Timing Prediction

This dissertation explores a number of different predictors that are intended to move data away from the processors and back to the directory so that Consumer Prediction and Migratory Prediction can be performed. Previous work has focused on specifically identifying the final touch to a line before an invalidate message can arrive [28]. The results presented here show that last touch prediction is less effective than previously reported when in a system with an operating system present.

1.2.4 Combined Coherence Prediction

Finally, this dissertation demonstrates how each of these predictors can be placed together, and work in concert to speculatively move data to processors before they request access. It explores two distinct options, one in which additional information about the instructions that have touched lines can be forwarded to the directory, and one in which no additional data beyond acknowledgments can move between the components of a system. The results of this study show that the combination of migratory and consumer prediction can produce slight benefits over either alone.

1.3 Research Contributions

This dissertation explores options for combining multiple coherence prediction techniques. It shows how it is possible to combine them without producing many, fully independent predictors, and how they can be combined so that they do not interfere with each other. In the process, several individual predictive techniques and interactions between predictors are discussed. These smaller contributions are listed below.

1. *Application of Perceptron Techniques to Consumer Prediction:* This work describes a novel consumer predictor that uses a global perceptron to identify future sharers. This predictor is able to operate in an accuracy vs. coverage tradeoff not available to previous predictors.
2. *Complete Implementation of Consumer Prediction:* This work is the first to produce a working system using consumer prediction at the directories to achieve processing speedups. Previous work has shown only the accuracy of these predictors. In addition to evaluating their accuracies, this dissertation demonstrates how the schemes proposed in previous work can produce speedups. This work then proposes alternate predictor architectures, and demonstrate the speedups that can be achieved.
3. *Inclusion of Migratory Pattern Identification in Consumer Prediction:* This dissertation demonstrates a method for constructing a migratory predictor that uses the same information available in the consumer predictor. It shows

that this migratory predictor can outperform other migratory schemes, and that it can be further improved by combining it with consumer prediction, at negligible hardware cost.

4. *Extension of Training Mechanisms in Multiprocessor Systems:* Previous work has demonstrated the potential of speculatively self-invalidating cache lines to provide substantial speed increases in directory-based multiprocessors. The most effective prediction scheme, last touch prediction, places a trace based predictor at each cache to identify instruction patterns that correspond to incoming invalidation messages. This study proposes and investigates the idea of sharing training information across caches. This sharing of information enables speedups of as much as 22% over a standard directory protocol and 12% over a directory protocol implementing previous last touch schemes. This work then explores the details of training such a system, and the performance tradeoffs of the decisions made in this stage of the design.
5. *Coherence Predictor Tuning for Full System Design:* Unlike the behavior of branch predictors, coherence predictors are not properly described by a single accuracy number. Two coherence predictors are needed: sensitivity and Predictive Value of a Positive Test (PVP). In isolation it is not always possible to identify the better of two predictors. This study evaluates how this tradeoff in consumer prediction behavior translates to performance benefits, and which systems benefit the most from a high or low network latency.
6. *Analysis of the Influence of Realistic Training on Coherence Prediction:* This

work analyzes the behavior of consumer prediction, and last touch prediction under a more realistic training model. Because training information is not immediately available, the theoretical results previously reported are not necessarily valid. For many cases the predictors will not have been updated from previous predictions when the next prediction is made. Additionally training information can arrive out of order. This dissertation demonstrates the result of these effects.

Chapter 2

Background

2.1 Multiprocessor Design

Multiprocessors can be divided into two groups: (i) Message Passing Multiprocessors (MPMs) which give each processor a separate memory space and use explicit *send* and *receive* commands to move data and, (ii) Shared-Memory Multiprocessors (SMMPs) which move data between processors via stores and loads to a shared memory space. SMMP systems are considered easier for programmers, allowing them largely to ignore hardware implementation details [46]. In many ways this is similar to programming in a high level language as compared to programming in assembler. However, SMMPs provide challenges to the hardware designer that MPMs do not.

- Coherence — How can we guarantee that the data seen by each processor are in some way coherent, so that we can be sure we are not looking at out-of-date data? This is not an issue if we do not implement caching, but such an omission will lead to poor performance.
- Physical Location of Memory — Where should we locate the actual memory and how should the processors access the memory?

Both of these problems have been researched in depth. Coherence can be maintained with either a snooping protocol [15] or with a directory protocol [6]. In a snooping protocol, memory accesses are transmitted across a bus or other strictly ordered substrate that is shared by all processors [15]. This allows the accesses to be easily serialized across processors. In a directory protocol, a centralized location called a directory tracks the current state of each line of memory [6]. When a processor writes a line to the directory, it first ensures that any other processor with a copy knows that it is no longer valid. It is possible to implement a coherence protocol using neither a directory nor a snoopy bus [32], or a hybrid of a directory and a snoopy bus [34]. However, such protocols are relatively new and often more difficult to verify and purge of race conditions.

Memory can be located at an equal distance from all processors, or distributed such that some parts of the memory are closer to some processors than others. Systems in which memory is located at an equal distance from all processors are called Symmetric Multiprocessors (SMPs). Systems in which memory is spread, some closer to some processors than others, are called Non-Uniform Memory Access (NUMA) systems. However, SMP systems are limited by memory bandwidth and can produce long latencies. NUMA systems address this by spreading memory around the network [16, 46]. The advantage of SMPs is that placement of data in memory is not as important, whereas in a NUMA it is important to locate the most frequently accessed data as close as possible to the processor that uses it.

2.2 Memory Sharing Patterns

In 1989, Weber and Gupta published their work on invalidation patterns [47]. They analyzed several programs and categorized the memory sharing behavior by cache line, describing five categories. These categories are Code/Read Only, Migratory, Synchronization, Mostly-Read, and Frequently Read/Written. Code/Read Only data never suffer any invalidations. Migratory data are never read between multiple processors between writes. This typically corresponds to locked data. Synchronization data corresponds to locks and other such structures which are often heavily requested. Weber and Gupta showed that the difficulties associated with synchronization data can be reduced by more carefully written programs. Mostly-Read data is read by a large number of processors, and occasionally written. Frequently Read/Written data shows more complex behavior. This categorization led to a number of different works intended to target specific patterns of behavior [4, 7, 23, 24, 45].

2.2.1 Migratory Data

Two techniques were independently developed to handle the case of migratory data [7, 45]. Both suggest modifications to the state machine that tracks read and modify permission. The state machine was modified to detect cases in which a processor received migratory data. This data was then flagged and when another processor requested read permission to the data, it was assumed that the requester would eventually want to write the data. Modify permission was sent along with

the data.

Later work by Kaxiras and Goodman showed that it was possible to identify migratory data by the code operating on it [22]. They demonstrated a system in which loads associated with migratory data are marked. The processor then requests modify permission rather than read permission. The difference between this and the previous attempts is similar to the difference between inward and outward coherence prediction, one focuses on moving data from a distant position to the predictor, and the other focuses on moving data from the predictor to a distant position.

Rajwar, Kagi and Goodman proposed that the locks themselves could be detected by the hardware [40]. They identified lines of memory containing locks and the associated lines containing the protected data. In this way the data could be sent to a distant processor as soon as a lock was released, before any request had been made.

2.2.2 Widely Shared Data

Bianchini and LeBlanc recognized another sharing pattern that they consider important: widely shared data [4]. This is related to, but not the same as, Mostly Read data [47]. They pointed out that bandwidth becomes more important as the processor count increases, and that a small number of cache lines uses up an increasing amount of bandwidth as the processor number grows. They called this data “hot”. They then developed a system called *eager combining* in which this data would be sent to a number of servers spread throughout the system. Rather

```

...
while (Global->Queue[num_nodes][0] > 0) {
    ...
    ALOCK(Global->QLock,local_node);
    work = Global->Queue[local_node][0]++;
    AULOCK(Global->QLock,local_node);
    while (work < lnum_blocks) {
        xindex = xstart + (work%lnum_xblocks)*block_xlen;
        yindex = ystart + (work/lnum_xblocks)*block_ylen;
        for (outy=yindex; outy<yindex+block_ylen && outy<ystop; outy++) {
            for (outx=xindex; outx<xindex+block_xlen && outx<xstop; outx++) {

                /* Trace ray from specified image space location into map. */
                /* Stochastic sampling is as described in adaptive code. */
                foutx = (float)(outx);
                fouty = (float)(outy);
                pixel_address = IMAGE_ADDRESS(outy,outx);
                Trace_Ray(outx,outy,foutx,fouty,pixel_address);
            }
        }
        ALOCK(Global->QLock,local_node);
        work = Global->Queue[local_node][0]++;
        AULOCK(Global->QLock,local_node);
    }
    ...
}
...

```

Figure 2.1: An example of migratory data. This is the main processing loop of Volrend. The memory location that is used to determine the next block to process (stored in work) is migratory. Once a lock is obtained it is read and written in succession. No other processor can access it during this time because of the locks.

```

...
for (step=0; step<ROTATE_STEPS; step++) {
    ...
    BARRIER(Global->SlaveBarrier,num_nodes);
    ...
    if (my_node == ROOT) {
        Select_View((float)STEP_SIZE, Y);
    }
    BARRIER(Global->SlaveBarrier,num_nodes);

    Global->Counter = num_nodes;
    Global->Queue[num_nodes][0] = num_nodes;
    Global->Queue[my_node][0] = 0;

    Render(my_node);
    ...
}
...

```

Figure 2.2: An example of widely shared data. This is the outermost loop of Volrend. The `Select_View` function is called by only a single processor (the ROOT processor). It sets a number of data items that are used to determine the position of the “camera” in this phase of rendering. Every Processor accesses `Render`, which reads all of this data.

than contacting the directory, a processor in the system could contact the nearest server to receive the latest version of the data. There are two main advantages to this approach. First, the traffic of many processors trying to read at the same time is spread throughout the network, rather than all being routed to a single point. Second, the servers can be spread throughout the system so that individual processors need not send messages very far to receive the data.

Kaxiras, and later Kaxiras and Goodman proposed the GLOW extensions to a directory based multiprocessor [23, 24]. These extensions were intended to handle the same “hot” data that Bianchini et al. aimed to handle with eager combining. A set of GLOW agents are placed throughout the network, forming a tree such that

the processors are spread out across them. These agents act as directories to the processors below them, and request data from the agents above them in the tree as a processor would. In this way contention for network resources is spread throughout the system.

Later, Kaxiras et al. proposed extensions to the GLOW agents [20, 21]. The GLOW agents are extended to dynamically identify widely shared data and act only on those data, allowing other requests to query the directory normally. The GLOW agents watch for requests to the same line by multiple processors in quick succession. When they see this occurring, the line is marked as widely shared and the agent acts to combine requests as originally proposed.

2.3 Multiprocessor Memory Coherence

Memory coherence in a multiprocessor, refers to the way those data that are potentially stored multiple caches are handled. The coherence protocol is the system that forces the data in the caches to remain coherent, such that the cache remains invisible, and programs can be written without concern for moving data between processors.

A number of different forms of speculation have been applied to memory coherence that do not resemble traditional prediction. For example, the migratory sharing schemes described earlier use speculation based on simple static patterns to recognize a common behavior [7, 45]. Dubois et al. propose adding a Stale state to the cache [11]. This Stale state allows the system to take advantage of additional

slack provided by a weak consistency processing model.

Others have suggested switching between multiple protocols, depending on the situation. For instance, Falsafi and Wood proposed Reactive NUMA [13]. Reactive NUMA combines the benefits of Simple Cache Only Memory Architecture (S-COMA), in which pages do not belong to a specific memory in the system but can be moved between locations, and Cache Coherent Non-Uniform Memory Access (CC-NUMA) in which each page has a specified location. They do this by dynamically detecting the pages that will benefit from S-COMA and applying it only to those pages. Bandwidth adaptive snooping, proposed by Martin et al., switches between a broadcast protocol and a snooping protocol based on the current network utilization [34]. When the network is not busy, they apply a broadcast snooping protocol to reduce overall latency. When the network is heavily loaded, the system switches to a directory protocol to reduce the network utilization.

Sorin et al. proposed that a protocol could be designed that detects rare race conditions, but does not handle them correctly [43]. When the race conditions are detected, the system rolls back to a previous checkpoint and applies a slower protocol that handles these conditions. With this technique a processor can use a fast but not wholly correct protocol the majority of the time.

Token coherence takes this approach one step further [32, 36] by taking away the need for a checkpoint. Martin et al. propose that tokens can be used to decide on permissions. Each line of memory is assigned a number of tokens, typically larger than the number of processors. Any processor with a token is allowed to read the line. Only a processor having all the tokens is allowed to write the line. Within this

context any speculative mechanism can be employed, so long as the three conditions above are respected and tokens are never created or lost.

In order to guarantee forward progress, a timeout condition is applied. Whenever a request spends too long waiting, the system uses a slower broadcast-based protocol. This protocol guarantees correct execution and overrides any and all speculative mechanisms in the system.

Another option proposed is coherence decoupling. Huh et al. propose that the processing unit itself be allowed to speculatively execute on invalid data [18]. If the speculation turns out to be incorrect, the processor rolls back. If it is correct, the processor commits the changes.

Coherence decoupling allows processors to ignore false sharing. If a processor is currently writing a word, another processor can read a different word on the same cache line speculatively. When the reader discovers that that word did not change, it can commit any data that it has already produced. Coherence decoupling can also be used in conjunction with data forwarding. A writer can send out copies of data to potential readers before the coherence protocol has officially given them read permission. Thus the readers can begin working with some data before the coherence protocol would otherwise allow. The main disadvantage of coherence decoupling is that the coherence protocol must be visible to the processing core. The commit and issue logic needs to be redesigned to take this into account.

2.4 Coherence Prediction

A number of different techniques have been proposed for using explicit predictor structures to accelerate coherence protocols. For convenience these techniques can be organized into *inward*, *outward*, and *administrative* techniques. In inward techniques a processor speculatively requests that its own permissions be increased. The classic example of this would be prefetching. Using prefetching, members of the system — typically either directories or processors — seek to move data from distant locations closer to themselves. In outward prediction, members of the system — typically either directories or processors — seek to move data from themselves closer to where those data will eventually be needed. In administrative prediction, a processor reacts to a real request for additional privilege by using speculation to take a shortcut around the strictly necessary messages that would normally be used. A good example of this would be owner prediction, proposed by Acacio et al. [2].

2.4.1 Inward Coherence Prediction

Inward predictor techniques are those in which a processor identifies data it predicts it will want access to and seeks to bring any such data closer to itself. These techniques are effectively the same as prefetching, although modified to perform in a multiprocessor system with cache coherence.

Mowry and Gupta proposed that software prefetching could be used to improve multiprocessor speeds [37]. They modified several benchmarks by hand to add prefetching. Both read and modify permission were prefetched as appropriate. This

showed large performance benefits, but required the prefetches to be inserted by hand.

Later, Dubois et al. described a hardware scheme for prefetching data in multiprocessor systems [12]. They implemented both stride and sequential read prefetching in a multiprocessor simulation. This yielded a dramatic reduction in read misses, a substantial reduction in execution time for some benchmarks, and an increase in network traffic.

Zhang and Torrellas proposed to create links between objects that may be distant in memory but are likely to be accessed near each other in time [50]. The compiler and programmer are responsible for identifying objects that are linked together. This information is passed to the processor. Whenever an object is requested, the pointers that it contains are also requested. Zhang and Torrellas show that this is beneficial to programs with irregular data structures.

Ranganathan et al. studied the interactions between prefetching and processor complexity [41]. They showed that as the complexity and processing power of the CPU grew, prefetching was less able to reduce memory latencies. While prefetching yielded a benefit, the amount of time devoted to memory events grew as the processor became more powerful.

Later work by Koppelman described neighborhood prefetching which combined the benefits of the previously proposed ones [25]. Neighborhood prefetching kept a record by PC of those cache misses that regularly occurred together. This provided the benefits of stride and sequential prefetching by recording that adjacent or equally spaced lines miss in sequence. It also linked objects in arbitrary positions

in memory, with no regards to strides, or sequential access.

2.4.2 Outward Coherence Prediction

Outward coherence prediction techniques are those in which processors and directories choose to move data and permission away from themselves. Research work in this area falls into three main categories: consumer prediction and data forwarding, timing prediction, and request memory prediction.

Several general predictor structures have been proposed for coherence prediction [27, 38]. The first of these techniques, proposed by Mukherjee and Hill, was the Cosmos Coherence Message Predictor [38]. The Cosmos predictor keeps track of strings of coherence traffic associated with each address. It also keeps a global table of strings of coherence traffic it has previously seen. Each message is added to its respective string. A check is made in the global table, and if the next predicted message is something the system can act on, it does so. Mukherjee and Hill show that coherence events are predictable, but do not show the performance results of acting on them.

Lai and Falsafi highlight a number of problems with Cosmos in their work [27]. They point out that there are a number of cases in which the ordering of messages is unimportant. Their Memory Sharing Predictor removes this ordering information to simplify the look-up process. It is interesting to note that once this is done, their predictor becomes a consumer predictor. When a processor has write permission and sees a read request from a distant processor, the processor attempts

to predict which other processors will request read permission. The processor also uses a simple heuristic to predict when it has finished writing a line.

Kaxiras and Young summarized previous consumer prediction schemes [19]. They demonstrated that all of the previous techniques are strongly related to one another. They proposed a taxonomy that encapsulated all of the predictors, plus additional terminology to describe them: prevalence, sensitivity, and predictive value of a positive test (PVP). This terminology allowed Kaxiras and Young to describe the performance of a consumer predictor with only two numbers. They then performed a search across all of the consumer predictors which their terminology could describe and listed the best in terms of sensitivity and PVP. This dissertation uses the terminology of Kaxiras and Young.

Rajwar et al. proposed that data can be speculatively linked to the lock to which they logically belong [40]. When a processor acquires ownership of the line containing a lock, any data associated with the lock can be sent along. Wenisch et al. propose that data is typically accessed in a specific order which they call a stream [48]. When a processor detects another processor requesting a sequence of addresses, it also transmits the additional data in the sequence. In this way, read permission can be moved between processors without a need for read requests when those two processors follow similar paths through memory.

There are two principal works in timing prediction. Last touch prediction proposed by Lai and Falsafi, seeks to identify times at which a processor is “done” with a line [28]. At these times the processor will receive an invalidation message before accessing the line again. The authors base their work on the insight that the

sequence of accesses a processor makes to a line during its lifetime is predictable and repetitive. The system stores the strings of program counter values (PCs) that have touched each line (compressed via truncated addition) and a hash table of strings of PCs that previously corresponded to last touches. When a match is found, the line is invalidated and the directory is notified.

Downgrade prediction is a similar technique [42]. Rather than looking for times at which to invalidate data, downgrade prediction looks for times when modify permission should be released. Functionally, this is identical to last touch prediction; however, the table is updated when modify permission is released to give another processor read permission, rather than when it gives up access entirely. When a match is found, the downgrade predictor releases modify permission and retains read permission.

2.4.3 Administrative Coherence Prediction

Administrative Coherence Prediction focuses on circumventing parts of the coherence protocol. These techniques are speculative, but do not seek to change the permissions of any cache lines before a request has arrived at the processor. Instead, they seek to speed coherence transitions that have been requested by the processor.

In owner prediction, proposed by Acacio et al., a processor that wants read permission tries to predict the owner of a line [2]. The processor then sends its request to the directory and to the predicted owner. If the prediction is correct, the owner responds with data, which is available after just a two-message latency. If

the prediction is incorrect, the directory forwards the request to the correct owner and data is available after a latency of three messages.

Acacio et al. later proposed a more aggressive predictor [1]. They proposed an Invalidation Lines Table (ILT) to speed the process of requesting modify permission. When a processor requests modify permission, it predicts the current readers of a line. The processor then sends requests to all of the predicted processors as well as the directory. If the prediction is correct, the directory tells it so and the other processors release the read permission they had. The entire process requires a two-messages latency. If it is incorrect, the directory notifies the processors that were not predicted. The entire process requires a three-messages latency.

Martin et al. describe how coherence prediction techniques can be used to reduce bandwidth in addition to latency [33]. They show that these techniques can dramatically decrease the bandwidth consumed when using a multicast snooping protocol. This is because these techniques decrease the number of messages that need to be multicast.

Finally, Lenosky et al. show that it is possible to predict the lines of the cache that need to be included in coherence prediction [30]. Previous techniques have assumed that some data are kept with each line in the cache, for instance, the trace of PCs to touch it in last touch prediction [28]. Their work shows that such a large quantity of storage is unnecessary. The number of lines involved in coherence activity is typically small. They propose an architecture that stores information only for lines known to be shared among processors. This dramatically decreases the size of coherence predictors with little affect on the accuracy of the predictors.

2.5 Coverage of Previous Work

The purpose of this dissertation is to create a predictor architecture that covers a wider range of sharing cases than have been previously suggested. To understand how to accomplish this, it is necessary to understand the limitations of the previous work. In this section one useful way to categorize sharing patterns is addressed. Each of the previous techniques that address each of these patterns and their limitations are discussed.

2.5.1 Division of Sharing Patterns

There are a variety of ways to categorize sharing. The most commonly cited paper on the subject analyzed cache lines and divided them into five categories, several of which have been discussed earlier [47]:

- *Code/Read Only Data* is never written once loaded from storage,
- *Migratory Data* only belongs to one processor at a time,
- *Synchronization Data* contains locks and other such structures accessed by many processors,
- *Mostly-Read Data* is rarely written and read by a number of different processors between reads,
- *Frequently Read/Written Data* is read and written regularly.

A number of papers have proposed ways of altering coherence protocols to exploit migratory data [7, 9, 45].

The categorization above considers sharing on the basis of cache lines. There is a strong intuitive reason for this. Each cache line corresponds to a set of data or a single variable in the case of single word lines. The line's behavior can be related directly to the data contained within. Others have suggested that sharing patterns may be better analyzed by Program Counter (PC) [22].

To better understand what cases are covered by previous predictors, it is possible to categorize coherence events. These coherence events are shown in Figure 2.3, and summarized below.

- One Writer To Many Readers (OWMR) — In this case, a processor has modify permission and many processors will be requesting read permission before another receives modify permission.
- One Writer To One Reader (OWOR) — In this case, a processor has modify permission and only one other processor will request read permission before the line is written again. The line here must be written by a cache other than the reader. This case is similar to OWMR.
- One Writer To One Writer (OWOW) — In this case, a processor has modify permission and another will be requesting modify permission. This may happen after either of the two processors read. When there is an intermediate read by the second processor, it is identical to migratory sharing. When there is no read, it is a silent store.
- Many Readers To One Writer (MROW) — In this case, a number of other processors have read permission and a request arrives for write permission.

This category includes widely shared data in addition to data only read by a small subset of processors between writes.

- One Reader To One Writer (OROW) — In this case, only a single processor has read permission and another processor requests write permission.

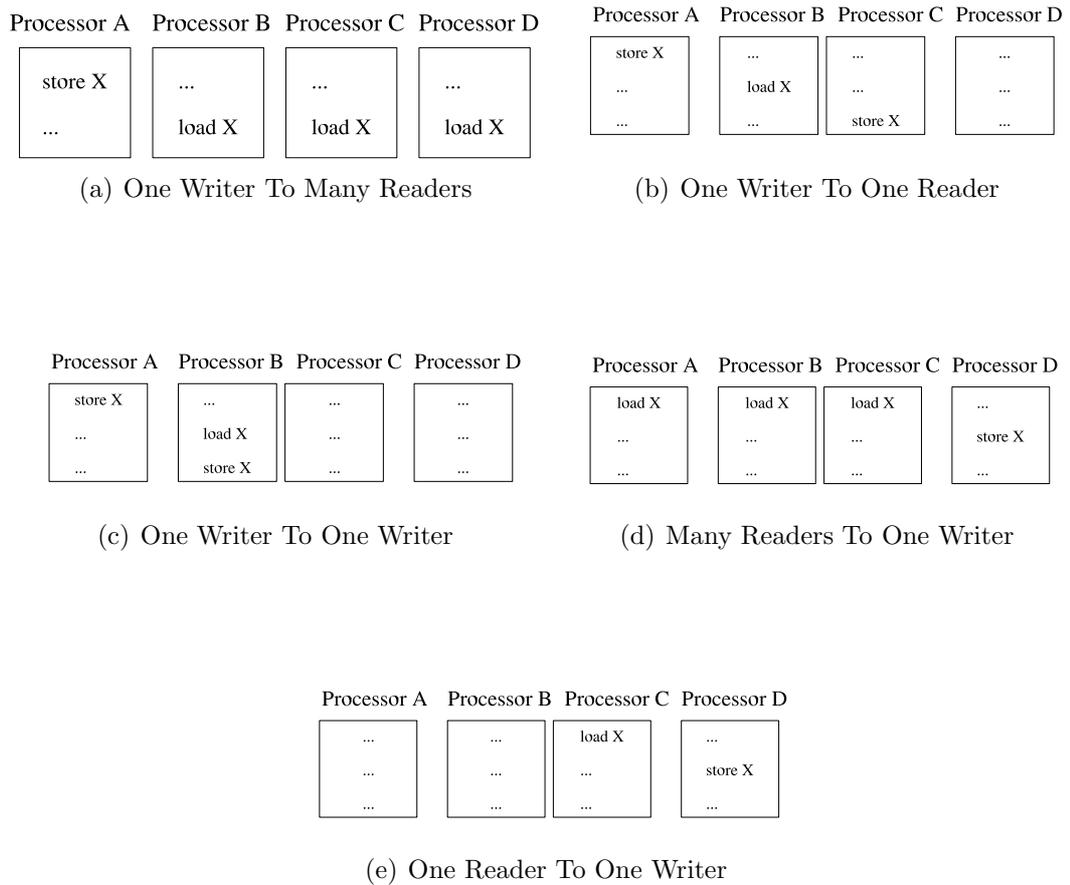


Figure 2.3: Five Different Categories of Coherence Events.

Each of these categories represents a subtly different prediction task. OWMR is related to producer-consumer prediction. Ideally, a cache could predict the processors that will request read permission and forward data to them before they actually request it. Predicting only a subset of these processors may still produce

speedups, as those processors which were correctly identified can receive the benefits. Predicting additional processors can cause slowdowns by using extra bandwidth or increasing the time needed for another later operation by requiring extra invalidations when it seeks a Modify copy. Several systems have been proposed to handle this [19, 22, 27, 42] and will be explored in more depth in Chapter 4. The OWOR case is similar enough to be handled by the same mechanisms.

OWOW contains silent stores and migratory data. Migratory data results from data structures contained behind locks, which are logically owned by a single thread at a time. Several modifications to coherence protocols have been proposed to handle migratory data [7, 9, 45]. All of them work by altering the coherence protocol. The disadvantage of modifying the coherence protocol is that more complex protocols are more difficult to verify. One advantage of coherence prediction is that it can be implemented with minimal changes to the coherence protocol, thus simplifying the task of protocol design [27]. The previous predictor based approach to migratory data was proposed by Kaxiras and Goodman in [22]. This was an inward method in which the PC was used to identify reads associated with the first touch in migratory sharing. But it did nothing to act before an instruction arrived at the processing unit requesting data.

MROW is a case where a processor requests write permission and several other processors must be invalidated. Two techniques can hide some of the latency of this event. Last touch prediction can remove a level of indirection by invalidating data early at the readers [28]. An Invalidation Lines Table (ILT), as proposed by Acacio et al. [1] and several other related techniques [2, 33] can remove a level of indirection

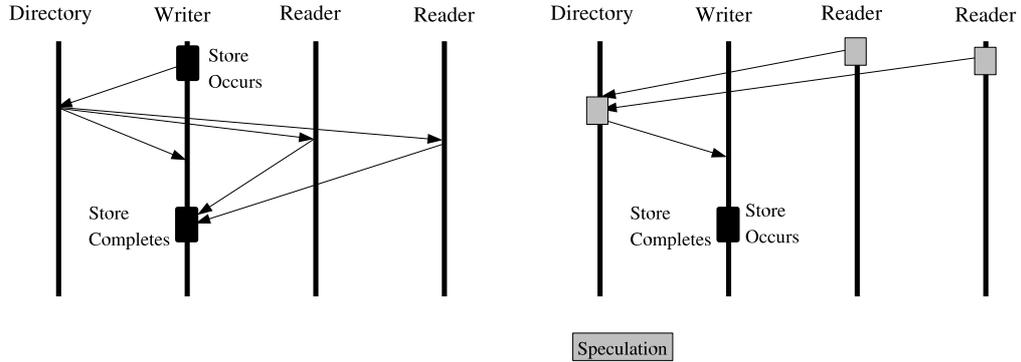
by predicting the current readers at the future writer, and skipping the directory. These two techniques have contradictory goals — one seeking to move information to the directory and the other seeking to bypass the directory. If a reader has invalidated itself using last touch prediction, the ILT will generate extra messages contacting that reader. Also, neither of these two forms of prediction moves the writer into modify permission before it has reached the store in its instruction flow, which would be ideal. OROW is a subset of MROW, and is slightly easier to predict in that only a single reader must successfully give up permission.

In addition to the issues described above, it is necessary to predict the timing of these events. Last touch prediction [28] and downgrade prediction [42] both already attempt to address this issue.

2.5.2 Multi-Read To One Writer Sharing

Possibly the most frequent example of a coherence event is the MROW transition. This event happens every time a processor requests write permission that is not covered by OWOW (Migratory Sharing) and where more than a single processor has read the data. Although several schemes have been proposed to aid this transition, none have attempted to completely remove all of the latencies through speculation. One of the more recent works in coherence prediction specifically mentions this limitation [27]. Figure 2.4 shows the messages that must be sent in order to properly account for an MROW event in a non-speculative directory protocol. It also shows the ideal case in which the owners speculatively release ownership and

forward it to the writer ahead of time.



(a) Coherence protocol behavior in MROW

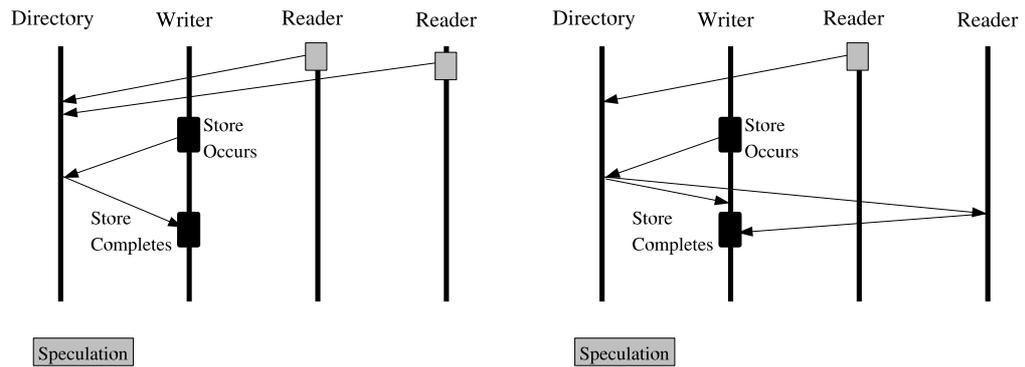
(b) Ideal speculative behavior

Figure 2.4: Normal behavior of a coherence protocol to an MROW event and the ideal speculative behavior that removes all latencies.

Two different predictors have been suggested to speed up this transition. The first is last touch prediction [28]. Last touch prediction identifies times when a processor is “done” with a line. It then prompts the cache to release ownership of the line. Figure 2.5 shows how Last Touch Prediction can speed an MROW event, as well as an example of how it can fail.

Using an ILT as proposed by Acacio et al. [1], the processor requesting write permission attempts to guess the current set of readers and bypass the directory. This process, as well as a case in which it fails, are shown in Figure 2.6.

First, notice that both an ILT and last touch prediction fail to hide all of the latency of the write even when working correctly. Last touch prediction requires the writer to contact the directory and then wait for a response. The ILT requires the requester to wait for the directory and readers to respond. Both reduce latency, but neither fully hide it.



(a) Ideal behavior of last touch prediction.

(b) Last touch prediction failure.

Figure 2.5: The ideal behavior of last touch prediction on an MROW event

Figures 2.5 and 2.6 also illustrate one of the inherent difficulties of handling MROW cases. In order for the full benefits of speculation to be achieved, many predictions must be made correctly. Last touch prediction requires that all the readers successfully invalidate their cache lines. An ILT requires that all of the readers are successfully identified by the writer. While a partial prediction can be useful, even a single mistake changes the longest path in messages from two to three. For this reason MROW events are the most difficult to handle speculatively.

In addition to the difficulty of coordinating the readers, the ideal predictor shown in Figure 2.4 can mispredict by sending write permission to the wrong processor, or more than one processor. If the wrong processor has write permission, the other processor will have to request it. This operation may be faster than with no prediction, as the directory will only have to contact a single processor. By placing the structure that forwards write permission at the directory, the second issue, that of multiple processors being selected as writers, does not exist. As with last touch

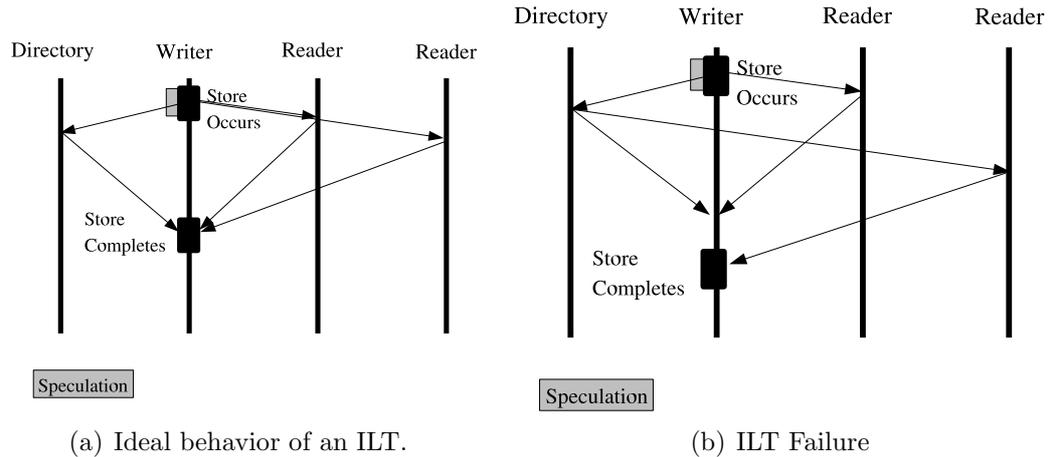


Figure 2.6: The ideal behavior of an ILT on a MROW event

and the ILT, there is a potential small benefit when not all processors give up read permission. Unlike last touch and the ILT, this ideal prediction has the potential to negate all of the delay incurred by a request for modify permission.

A key difficulty with speculatively moving modify permission to a future writer is that of authority. In the case of migratory sharing, one processor with modify permission gives up all access permission and gives another processor modify permission. Because the first processor had modify permission, we know that no other processor does. No other processor can receive read or modify permission from the directory until the first is invalidated. Thus we know that coherence is maintained. Similarly, with consumer prediction, the writer has modify permission and thus can downgrade itself and distribute read permission.

In the case of MROW, no individual processor has the authority to grant the future writer modify permission. This is because no processor can be aware that it is the only sharer at any time. Thus the directory must be involved in order to

forward modify permission, whereas the directory was largely uninvolved in previous schemes.

2.5.3 Migratory Data

Migratory data is data that is only owned by a single processor at a time. These data are typically read and written by a single processor, then read and written by another. Several techniques have been applied to speed up the transitions of Migratory Data. These either pass write permission when only read permission is requested [7, 45] or identify loads that will be shortly followed by stores and request modify permission speculatively [22]. None of these schemes seek write or read permission before the processor reaches an instruction that requests the data. Further, all of these techniques require additional decision-making capability to be added to the cache. This has a tendency to make verification more difficult and to interact poorly with other optimizations.

Chapter 3

Experimental Methodology and Terminology

This chapter describes how the results presented in the rest of the dissertation were created. It describes each of the benchmarks used, as well as each of the terms used to measure performance and system behavior. Finally, it discusses the coherence protocol implementation that was used for all of the simulations.

3.1 Measuring the Performance of Coherence Prediction

3.1.1 Measuring Prediction Quality

Predictions can be sorted into (i) false positives (FP), (ii) false negatives (FN), (iii) true positives (TP), and (iv) true negatives (TN), depending on the prediction made (P/N), and its correctness (T/F). It is important to note that in the case of many coherence prediction tasks the two false cases have different results. A false positive incurs a penalty over normal execution as a speculative action is taken incorrectly and may need to be corrected. A false negative results in normal execution, as though no predictor is present. When the event that should have been predicted occurs, the system continues normally. Both false positives and false negatives are mispredictions, but only one of the two will cause a performance penalty.

Similarly, true negatives do not result in any benefit, while true positives should yield an improvement in performance. Both of these are correct predictions,

Prevalence	$\frac{TP+FN}{TP+TN+FP+FN}$
Sensitivity	$\frac{TP}{TP+FN}$
Predictive Value of a Positive Test (PVP)	$\frac{TP}{TP+FP}$

Figure 3.1: The three terms used in this dissertation to quantify the behavior of coherence predictors.

but only one of the two is of value. Ideally, a predictor would maximize the number of true positives and minimize the number of false positives. To quantify this behavior three terms are defined in Figure 3.1. These terms were originally proposed by Kaxiras and Young [19] to describe the behavior of a consumer predictor.

The prevalence, or frequency of positive cases, is a property of the values being predicted and not the predictor itself. Thus, we can reduce comparisons of coherence predictors to two terms: *sensitivity* (the number of potential positives that were correctly predicted), and *PVP* (the reliability of a positive prediction). Notice that when the number of true positives is maximized the *sensitivity* will be one, and when the number of false positives is zero the PVP will be one.

3.1.2 Measuring System Behavior

In this dissertation, a number of aspects of system performance in order to quantify and explain the behavior of the proposed prediction schemes are measured. Each of these terms is defined here, as well as an indication of why it is meaningful.

1. *Relative Execution Time*: This is the number of cycles it took to complete the benchmark, relative to the total number of cycles it took to complete the benchmark with no speculation in place. Numbers below one correspond to

speedups, and numbers above one correspond to slowdowns.

2. *Relative Invalidations*: This is the number of invalidation messages sent by the directory divided by the number of invalidations sent by a control experiment with no speculation present. Invalidation messages are sent whenever a processor seeks a Modify copy of a line and other processors possess copies of that line. Sending extra copies of data will cause this number to rise as those new copies need to be invalidated.
3. *Relative Requests*: This is the number of request messages sent from the caches to the directory, divided by the number of requests sent by a control experiment with no speculation present. Sending extra copies of data will cause this number to fall, as request messages are not necessary. Self invalidation can cause this number to rise, as incorrectly removed lines need to be recovered.
4. *Relative Bandwidth Usage*: This is the percentage of the interconnect bandwidth that is consumed relative to a control experiment. Bandwidth is used by two distinct types of messages in this system: control messages and data messages. Messages that carry data change in size depending on the size of the line, but by default are approximately eighty bytes. Control messages are approximately sixteen bytes, depending on the exact data being passed, and do not change in size with the line size. In general, Data carrying messages correspond to the majority of the consumed bandwidth in a system.
5. *Relative Load Miss Rate*: This is the miss rate of loads in the cache, relative

to a control experiment with no speculation present. Speculatively sending data to distant processors can cause this number to fall, as the data arrives before the processor needs it. Self invalidation can cause this number to fall, as processors give up data they do not need.

6. *Relative Store Miss Rate*: This is the miss rate of stores in the cache, relative to a control experiment with no speculation present. Notice that a store miss includes a case where a Shared copy of the line is present in the cache. These cases would have been hits if the instruction had been a load, but because the permissions were not present for a store they are treated as misses. The behavior is similar to that of loads, but it should not be directly affected by speculatively sending Shared copies. Migratory prediction may cause this number to fall.
7. *Relative Load Miss Latency*: This is the average number of stall cycles caused by a load that misses in the L1 cache, divided by the average for a control experiment with no speculation.
8. *Relative Store Miss Latency*: This is the average number of stall cycles caused by a store that misses in the L1 cache, divided by the average for a control experiment with no speculation.

3.1.3 Predictor Naming Conventions

In order to make the large number of predictors evaluated easier to differentiate, this dissertation uses a standardized naming convention for those predictors

that are designed around the most common architecture. This naming convention is based on the conventions used in [19]. Each consumer predictor’s name specifies the function it applies, the index of the history table, and its depth as follows: $Function(index)^{Depth}$. For instance, $Union(addr_{16})^4$ is a depth four union predictor indexed by the low sixteen bits of the address.

When migratory prediction is used, it will be specified as $Mig(index)^{Depth}$. The use of the word *Mig* indicates that this is a migratory, rather than consumer predictor. Finally, when both are implemented they will be referred to as $Function(index)^{Depth} + Mig(index)^{Depth}$ when both the index and depth are different, or $[Function^{Depth} + Mig^{Depth}](index)$ when only the depth differs.

3.2 Experimental Methodology

All studies in this dissertation were performed either by analyzing traces captured from the GEMS multiprocessor simulator [35], or are the result of live simulations using GEMS. New protocols were developed within GEMS to support the forms of speculation desired. GEMS requires full system simulation, with Solaris 10 as the underlying operating system in these results.

The GEMS simulator is a set of modules that can be used with the Simics simulator. The Simics simulator emulates a full system — in this case running Solaris 10 — clocked by the GEMS modules. Whenever it executes a memory instruction it halts and notifies the GEMS modules. These modules then notify Simics of when it can continue execution on that processor and when that processor

should stall.

Using the Ruby module of GEMS causes Simics to stall a processor whenever it has an outstanding memory request. This effectively emulates a CPU that cannot take advantage of out-of-order execution.

Within Ruby, new memory protocols are defined as state machines in a scripting language. Each protocol is made up of messages, cache events, directory events, cache states, directory states, and state transitions. The events act as inputs to the state machine, and the transitions define the actions taken by the system for any event/state pair, as well as the final state of the system when that transition completes.

One of the difficulties inherent in multiprocessor simulation is the fact that different code paths can be followed, depending on the behavior of the memory system. As an example, consider the situation where threads are assigned IDs. A simple way to implement this is to use a lock, where each thread assigns itself an ID and then increments the locked variable. These IDs could then be used to determine which processor performs which task. For example the processor with the lowest ID could act as a server, telling the other processors what to do.

In this scenario, the processor that acts as the server is the one that receives permission to the locked ID counter first. However, the memory model does not define the processor that should first receive this permission; it will likely be decided by the processor that is physically closest to the directory, as that request is most likely to arrive first.

Because of the reality that the code path can change, and because of the fact

Parameter	Value
Processor Count	8
L1 Cache Latency	2 Cycles
L2 Cache Latency	10 Cycles
Network Link Latency	10 Cycles
Directory Latency	20 Cycles
Network Topology	2D Torus

Figure 3.2: Simulation Parameters Used

that GEMS simulates an operating system — complete with internal and external interrupts — the results of these simulations are noisier than the typical results from a single processor simulation, in the sense that had the benchmark been started when the operating system was in a different state, the results may have changed. That said, the results presented are cycle accurate, but they should be considered in the context of the surrounding results, rather than singly.

Table 3.2 shows the default simulation parameters used. On some occasions results are provided showing what happens when one of these parameters is varied. When not specified in the discussion of an experiment, these default parameters were used.

The benchmarks are taken from the SPLASH-2 multiprocessor suite [49] and ported from the Split-C benchmark suite [8], and are intended to simulate scientific and other processor intensive applications. These all use the default inputs for each benchmark.

- *BARNES*: A divide and conquer approach to the N-body problem, of calculating the forces exerted by multiple “bodies” in space. The region is split, depending on particle distribution and then distributed among the processors.

The process is composed of a loop, waiting for a barrier at each iteration and then performing a series of barriered operations, including calculations across a tree structure, and rebuilding the tree as particles move. Input size is 16348 Particles.

- *CHOLESKY*: Cholesky Factorization on a sparse matrix. This program is comprised of a number of initialization phases, separated by barriers, followed by a calculation phase. During the calculation phase, each processor requests a block, processes it, and proceeds to request another block until all are complete. The input matrix is 13992×13992 .
- *EM3D*: 3-Dimensional Electromagnetic Simulation. This algorithm comes from the Split-C benchmark suite. It follows a sequence of phases in which alternating sets of nodes are updated based upon the other set. A graph of two colored nodes is maintained, where each node is connected only to nodes of the other color. These edges determine which nodes are needed to calculate the result at the next step. Each node connects to three others, of which there is a 25% chance that each will belong to a different processor than the current node.
- *FFT*: Fast Fourier Transform. This algorithm divides the input into chunks that can be operated on by each processor. After each phase of processing, the data is permuted among the processors in the system. The input size is 64K entries.

- *LU*: LU factorization of a dense matrix. A matrix of values is divided into blocks and distributed to the processors as each completes a task. Completing any block potentially frees other blocks to be worked on until the task is complete. The input matrix is 512×512 .
- *OCEAN*: Simulation of water flow in a large body. This breaks the simulation down into blocks where communication occurs along the boundaries. Individual phases of calculation are separated by barriers. The calculation is performed on a 258×258 grid.
- *RADIX*: Performs a Radix sort on a set of random data. In this sort each processor performs a sort on a subset of the data. The data is then collated and redistributed for more passes. The sort is performed on 256K entries.
- *WATER*: Models the interaction of water molecules. Rather than calculate all possible interactions, this program groups particles by location so that less comparisons are needed. The input size is 512 molecules.
- *VOLREND*: Takes as input a three dimensional spatial grid and raytraces to produce an image. This process is performed over multiple iterations as the camera is rotated around the object.

Each benchmark was modified so that threads were pinned to individual processors in order to prevent thread migration. Only the parallel portion of the benchmarks is simulated, starting from a checkpoint of the system after the initialization was complete, and the data was either loaded or generated.

3.3 Directory Protocol Implementation

The protocols used in this paper are all variants of the Modified-Shared-Invalid (MSI) protocol [6]. The MSI protocol is typically represented as shown in Figure 3.3. At any given time either a single cache is in the Modify state and all the rest are in the Invalid state, or many caches are in the Shared or Invalid State. When a cache requests modify permission, any others with share permission are forced to the Invalid state. In the state machine shown the cache requesting modify permission sees an internal store, and the other state machines receive an external store. When a cache requests sharing permission, any that possess modify permission are forced to the Shared state. In the state machine shown the cache requesting modify permission sees an internal load, and the other state machines receive an external load.

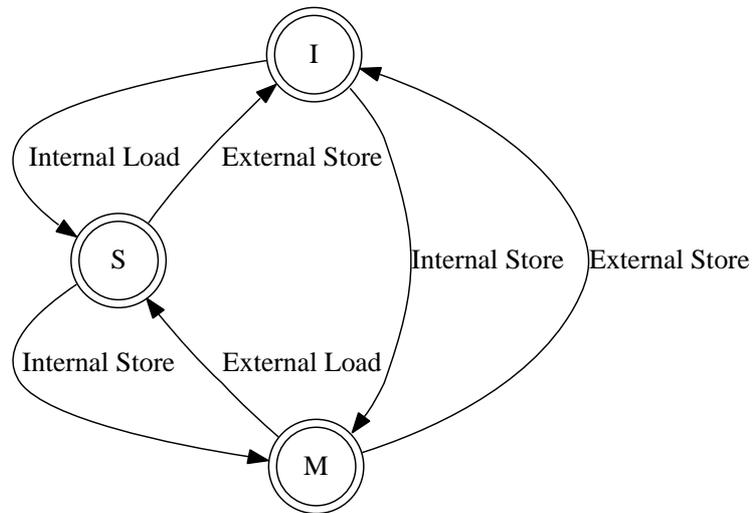


Figure 3.3: State Transition Diagram of an MSI Protocol.

While Figure 3.3 helps to explain the concept, the reality is substantially more complex in a directory based multiprocessor. A more complete design features many state machines for each line of memory, one housed at each cache and one

at the directory. These state machines interact with each other through a network, and this time delay forces the creation of intermediate states, as well as additional inputs and outputs to handle communication. In addition, caches have to drop lines occasionally in order to fit requested lines, either due to capacity or conflict misses. This means that the possibility must exist that the state of the directory and caches will not be synchronized with each other when events occur on a line.

A number of optimizations exist that do not properly handle the rare case of frequent cache line returns due to conflict misses. For instance, there is a common optimization present in the GEMS toolset in which a directory servicing a sharer that receives another request for sharing permission will add that request. Intuitively, this seems correct, however, it is possible to construct a program in which a single processor may be forever denied Modify permission due to a number of other processors constantly requesting and then returning that cache line due to thrashing. Other such cases occur due to the fact that a directory must spin through the messages it has received to find the correct ones to execute.

For the purposes of this dissertation, an MSI protocol was constructed based on those in the GEMS toolset. This protocol was constructed to avoid other cache optimizations, as many of the techniques proposed make use of rare events that interact poorly with these simple assumptions. The specific deadlock case above was left in the protocol for much of the analysis due to its rarity as well as fact that a variety of corrections could be envisioned. It could be fixed, but it did not occur in any of the benchmarks or predictors listed once they were functioning correctly.

Another major difference between the protocol implemented in this disser-

tation and the protocols included with GEMS is that the one used features no responses to “extra messages.” These are messages that arrive at a cache or directory and are acted upon despite not matching the current state. Responses to “extra messages” are a product of optimizations that take the directory out of the loop, effectively not updating it during transitions. The most common would be a cache that responds with an acknowledgment of invalidation whenever it receives an invalidation, regardless of whether it has a line. These optimizations make it effectively impossible to implement many other optimizations correctly. They also cause subtle and rare race conditions where extra acknowledgments arrive during a later transition

Figures 3.4, 3.5, 3.6, 3.7, 3.8, and 3.9 show the baseline protocol used as a control in the experiments that follow. In these diagrams states related to movement between the L1D, L1I, and L2 caches of these machines have been omitted. The naming conventions have been chosen to match the other protocols provided with GEMS.

Figure 3.4 is a list of the states that the lines of a cache can occupy. Specifically, there is one state for a line that is not in the cache, one state for each of the “stable” positions the cache can be in (Invalid copy, Shared copy, Modify copy), and a transient state corresponding to each possible step up in permissions from one of the “stable” states. These transient states denote that the system is waiting on information from the directory.

Figure 3.5 is a similar list for the directory. All lines must always be present in the directory, so there is no “not present” state. The directory has the same

<i>State Name</i>	<i>Description</i>
NP	Line is not present in the cache
I	An Invalid copy of the line is present in the cache
S	A Shared copy of the line is present in the cache
M	A Modify copy of the line is present in the cache
IS	An Invalid copy is present, A Shared copy has been requested
IM	An Invalid copy is present, A Modify copy has been requested
SM	A Shared copy is present, A Modify copy has been requested

Figure 3.4: States of the Cache

“stable” states as the cache, though the Shared state indicates that some caches have a Shared copy, and the Modify state indicates that a cache has a Modify copy, rather than the directory has a Modify copy. Thus the state of a directory line corresponds to the state of some cache, rather than the state of the line in the directory.

The directory has more transient states than the Cache, one for each combination of permission that could have been requested and stable state that the system could have started in. Also, the directory has additional state information that is not considered here, such as the processors that are seeking permission. These are not treated as separate states in the state machine, but are occasionally relevant to the events that can occur in a given state.

Figure 3.6 is a list of the events that can occur at the cache causing the cache to transition from one state to another. These represent messages from the processor and directory, such as a load instruction being executed causing an event at the cache, or an invalidation from the directory. They also include an L2 Replacement,

<i>State Name</i>	<i>Description</i>
I	No cache has a valid copy of this line
S	One or more caches possess Shared Copies of this line
M	One cache possesses a Modify copy of this line
IS	No cache has a valid copy, one has been sent a Shared copy
IM	No cache has a valid copy, one has been sent a Modify copy
SS	One or more caches possess a valid Shared copy, one or more has been sent a Shared copy
SM	One or more caches possess a valid Shared copy, a request for a Modify Copy has arrived. The sharers have not yet invalidated.
MS	One cache has a Modify copy, another has requested a Shared copy. Waiting on the Correct Version from the Modifier.
MM	One cache has a Modify copy, another has requested a Modify copy. Waiting on the Correct Version from the Modifier.

Figure 3.5: States of the Directory

which corresponds to this line being pushed out of the cache due to either a conflict or capacity miss.

<i>Event Name</i>	<i>Description</i>
Load/Ifetch	The Processor has performed a load operation
Store	The Processor has performed a store operation
L2 Replacement	Cache line must be sent back to memory
Inv	Invalidation arrived from the directory
Reduction	Message arrived from the directory indicating reduction from M to S
Data	Line arrived from the directory
Data Exclusive	An exclusive copy of the data arrived from the directory

Figure 3.6: Cache Events (All events pertaining to swapping between L2 and L1 caches omitted.)

The corresponding table for the directory, Figure 3.7, is slightly more complex. It includes a number of events that have been split to account for additional hidden state information. For instance, the given state diagram does not specify the exact processors that have a Shared copy when the directory is in a Shared state. Thus the event corresponding to a request for exclusive permission has been split into a request for exclusive permission when a cache is the only one with a shared copy of the line, and one when multiple caches have a Shared copy of the line. This allows the state machine representation to be drawn without consideration to the total number of processors in the system.

Figures 3.8 and 3.9 are the actual state machines for the directory and cache respectively. “Stable” states have been represented by a double circle, while tran-

<i>Event Name</i>	<i>Description</i>
GETX	A cache has requested a Modify copy
GETX Alone	A cache has requested a Modify copy and is the only cache with a Shared copy
GETUP	A cache with a Shared copy has requested a Modify copy
GETUP_Alone	A processor with the only outstanding Shared copy has requested a Modify copy
GETS	A processor has requested a Shared copy
Inv_Ack	Acknowledgment of an invalidation
Inv_Ack_Last	The last outstanding acknowledgment of invalidation for this line has arrived
Data	A line has arrived from a modifier that retained a Shared copy
Data_Exclusive	A line arrived from a modifier that no longer holds a valid copy
Data_Ack	A new owner of a line acknowledges receipt
Data_Ack_Last	The last outstanding new owner acknowledges receipt of a line
Reduce_Ack	The last Modify owner now possesses only a Shared copy

Figure 3.7: Directory Events

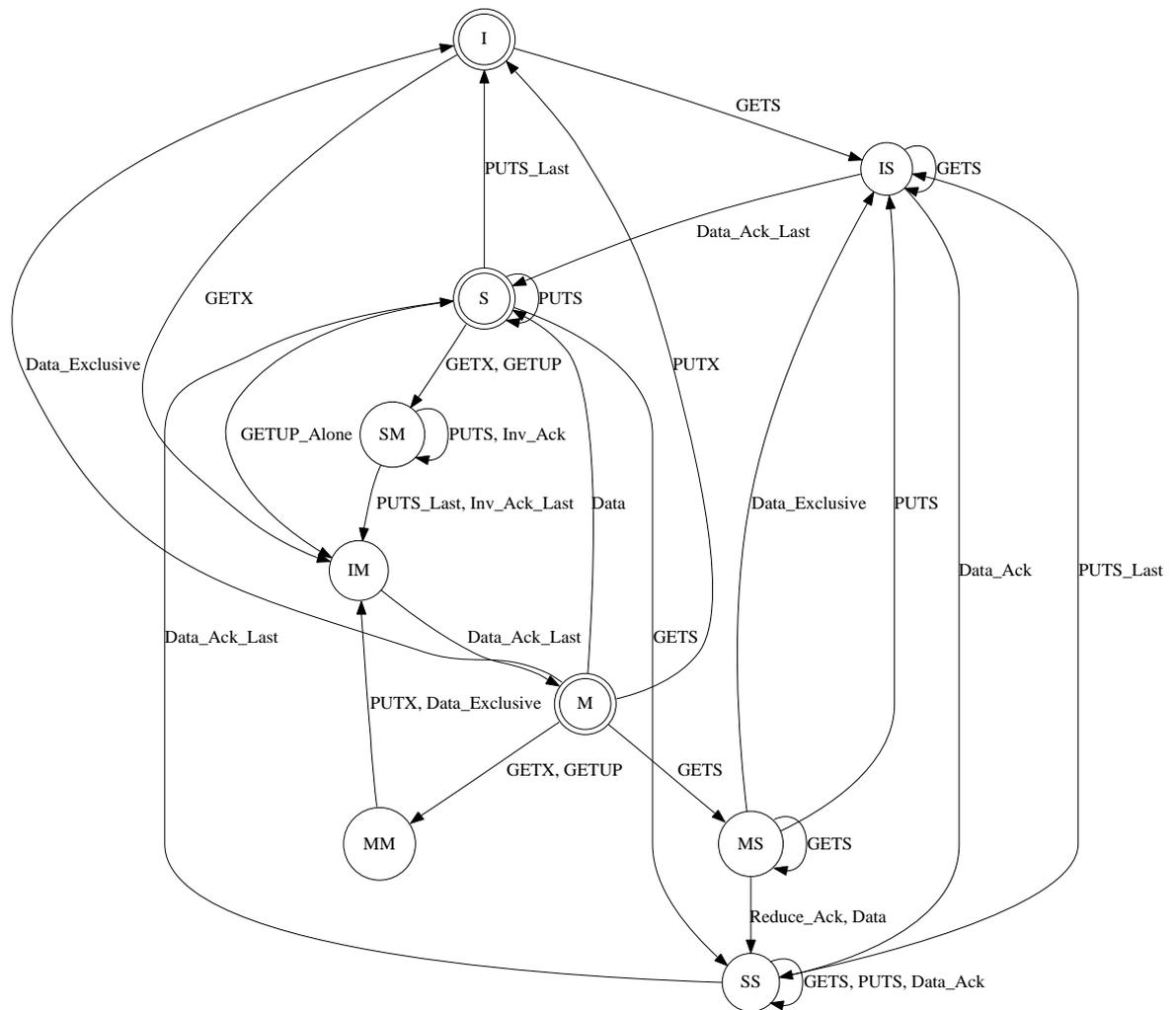


Figure 3.8: State Machine Representation of the Basic Directory. Inputs with no defined transition in this state machine wait at the directory to be processed when a transition is defined.

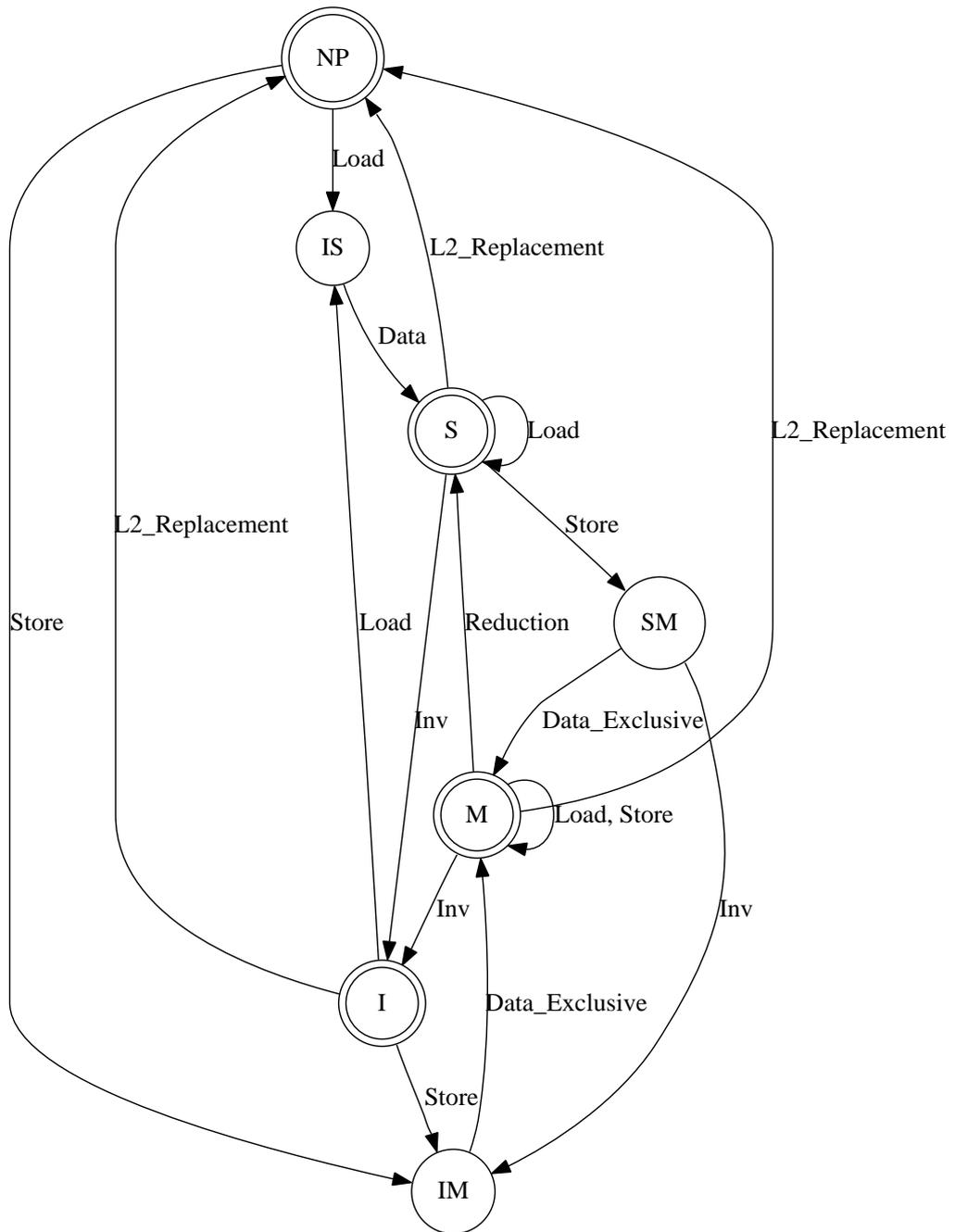


Figure 3.9: State Machine Representation of the Basic Cache. Inputs from the directory with no defined transition are discarded by the cache. Inputs from the processor with no defined transitions stall.

sients are represented by a single circle. Events that have no corresponding edge on the state machines presented are ignored by the cache. They either should never occur (such as data arriving when it has not been requested) or are the result of outdated information (such as an invalidation for a line that was removed from the cache due to an L2 Replacement event). At the directory, whenever a message arrives that does not have a corresponding edge in Figure 3.8, it is held and checked again later. For instance, a request for exclusive permission that arrives when the directory is already passing out sharing permission for that line is held until the current transient is resolved. These messages will cycle, so they will not necessarily be processed in the order they arrived. This is not unreasonable, given that they are not guaranteed to arrive in the order they were sent.

No matter the level of speculation used, the protocol implemented follows the same design strategy: All permissions are coordinated by the directory. The cache discards all messages from the directory that are undefined for its current state. The cache stalls on any request from its processor that is undefined for the current state. The directory responds to every message it receives, cycling through other messages whenever a non-allowed message arrives. For some forms of speculation, this requires the cache to send extra messages to the directory; however, it greatly simplifies design, debugging, and verification.

For example, you will notice that no acknowledgment messages arrive at the cache. The cache itself remembers nothing about what is happening in the system that is not related to a line that it has requested. It acknowledges everything with the directory in order to guarantee that the directory is kept up to date, and the

directory concerns itself with confirming that transitions are complete.

These protocols have not been formally verified. Random testing has been performed for all of them. In addition, regardless of the benchmarks presented in any given graph or chapter, all of the protocols used successfully completed every benchmark used in this dissertation.

Chapter 4

Consumer Prediction

The concept of a consumer predictor is simple, predict the set of sharers that will use a line and forward the data to them speculatively. The implementation, however, is complex. Previous work in this area has been content to discuss the predictive power of such a predictor, without discussing the effects on the full system [19]. The general structure is shown in Figure 4.1. A history of information about previous sharing is kept in some place in the system. When a prediction is needed, this history is queried, and for each processor to be predicted, a function of some kind is computed of the history of *that processor*.

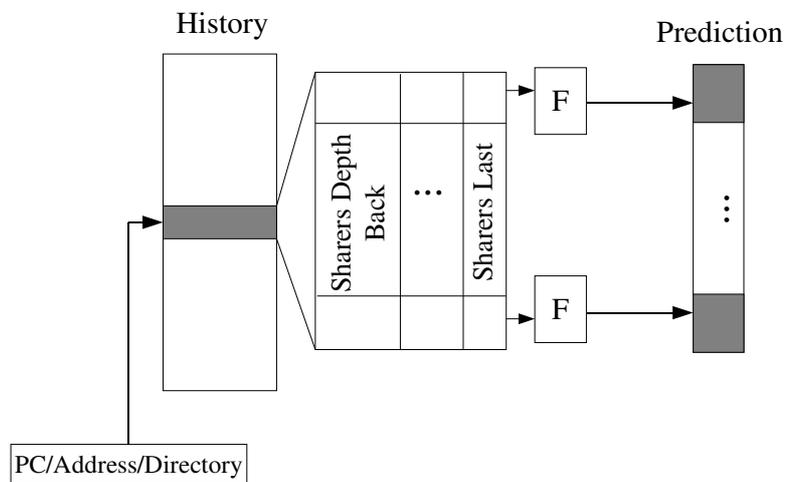


Figure 4.1: General Consumer Predictor Architecture

This chapter discusses a new form of consumer predictor that uses a perceptron to predict consumers. It explains the changes that must be made to a simple protocol in order to support consumer prediction. Finally, the results of implementing such

a protocol are demonstrated, and its effects on the memory system as well as the performance benefits that can be achieved are analyzed.

4.1 Theoretical Consumer Prediction Behavior

In this section the theoretical performance of consumer predictors is analyzed in terms of sensitivity and PVP. In previous work these terms were used to quantify the performance of consumer predictors and can give some hint about their final performance.

A perceptron is applied to the task of predicting the consumers of data [31]. While this predictor does not strictly outperform all those proposed before, it does provide a different tradeoff between sensitivity and PVP [19]. This predictor applies to the OWMR and OWOR cases described above. This section discusses, contrasts, and compares the predictive accuracy of previous consumer predictors, as well as the perceptron consumer predictor proposed here.

4.1.1 Consumer Predictor Architecture

4.1.1.1 Perceptron Consumer Prediction Design

The overall structure of the predictor used is similar to those used in previous work [19] and is shown in Figure 4.2. The history table contains data records that record the set of sharers between two processors receiving a Modify copy. This record is stored as a bitmap with a single bit for each processor. Whenever a new bitmap is created it is shifted in to an entry in the history table, shifting other bitmaps back,

or potentially out of the table entirely. The number of bitmaps stored for each entry in the history table is referred to as the depth.

The table is indexed by some combination of bits from the address, the PC currently writing the line, the directory, and the processor. The exact bits present also define the placement of the tables. For example, including bits identifying the processor in the hash is equivalent to placing a separate table at each processor. The value at the location indexed is sent to a set of perceptrons that decides if another processor will consume the given value. Each history table has a separate perceptron for each potential consumer. It differs from the previously proposed consumer predictors in that each perceptron operates on all of the possible inputs, rather than only on those that correspond to its processor.

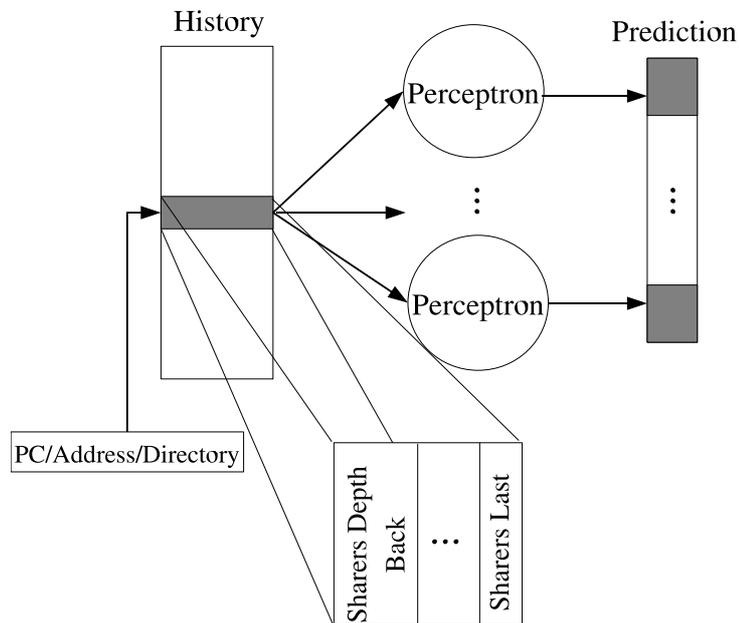


Figure 4.2: Perceptron Consumer Predictor Architecture

Figure 4.3 shows the design of the perceptron. The history for a given index is

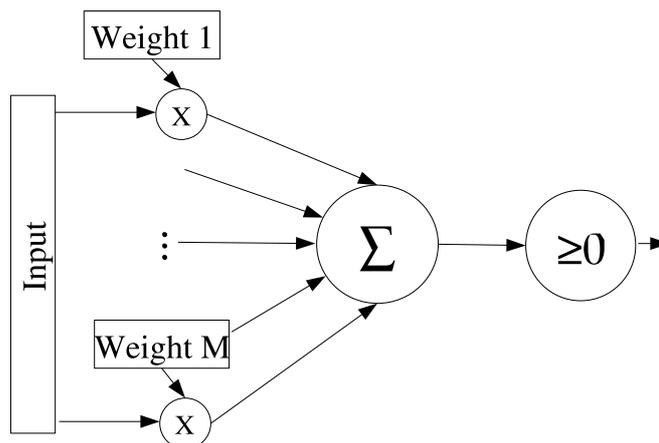


Figure 4.3: Structure of a Perceptron Predictor

used as input to the perceptron, which has a corresponding weight for each bit. The weights are either added or subtracted from a sum depending on the corresponding bit in the input. If the sum is greater than zero, the perceptron predicts positive, otherwise it predicts negative. The perceptron treats the presence of a processor in a group of sharers as a 1, and its absence as a -1 .

4.1.1.2 Consumer Predictor Training

There is one structure that needs to be maintained for a consumer predictor to work — the history table — and one more for a perceptron consumer predictor to work — the perceptron weight table. The history table keeps *depth* bitmaps for each index. Each bitmap contains the last set of sharers corresponding to this index, which may include both the PC and the address information. If the history tables are located at the directory, it is relatively easy to track all sharers, as the directory is responsible for tracking that information in order to maintain coherence. If the history tables are located at each processor, updating them is slightly more

complicated. In this case information about sharers can be piggybacked onto an existing response message from the directory whenever a processor requests exclusive access.

Both the history table and the perceptron weights are updated only when another processor requests exclusive access. However, if they are located at the cache, the data may not arrive until the perceptron itself has changed. The history table may also receive information out of order if it was indexed in terms of instructions. These two factors are not apparent in the trace based analysis of the predictors, but will be made clearer in the runtime performance. In either case, for techniques where it matters, the perceptron performs updates based on the prediction it would make *when it has received all the information needed to update*. Doing this prevents some, but not all, of these hysteresis effects, and reduces storage requirements. It would also be possible to store the information needed to update the perceptron when the original prediction is made, but this would fail to account for more recent changes to the perceptron, and would require additional information.

Once data arrives, each perceptron is evaluated to see if it needs to be updated based upon the prediction it would have made. A perceptron is updated when its output disagrees with the actual behavior of the system, or if the magnitude of the sum was less than some threshold. Each weight in the perceptron is incremented if the corresponding input agreed with the output, and decremented if the input disagreed with the output. Thus, the threshold decides when the perceptron stops training. A low threshold means that the resulting weights are able to adapt more quickly if the behavior of the program changes. A high threshold means that the

perceptron itself will be slower to change, and thus be less influenced by brief changes in program behavior. This predictor is referred to as $Perceptron_{threshold}(index)^{depth}$.

4.1.2 Sensitivity and PVP of Consumer Prediction Schemes

This section evaluates a large number of predictors, searching the design space across depth, index, and function. This is performed using trace-based simulation. When the trace-based simulator parses the lines it assumes that the L2 cache of each processor is infinite. These simulations are performed for a number of different history table configurations: a single global history table, a history table at each directory, and a history table at each processor. In all cases the results show that it is best to place a history table at each processor.

The traces were generated by the SPLASH-2 [49] benchmark suite using only the Ruby module of GEMS [35], simulating a 16 processor system with in-order execution, a 64KB L1 cache, and a 16MB L2 cache. The traces are gathered from a system with a finite cache size, but they are then simulated by a system with an infinite line size. The default input set was used for each benchmark.

This study explored the space of predictors that use as many as 1M entries per history table, depths as high as 4, and the prediction functions Union, Intersection, Two-Level Up/Down Counters, and Perceptron. Union, Intersection and Two-Level Up/Down Counter predictors were all proposed previously [19]. Each represents a different function which can be applied to the information stored in the history table. The threshold of the perceptron was varied from 10 to 500. All of the dynamic

predictors are evaluated based upon predictions made as the results become known.

Figure 4.4 shows the set of co-optimal predictors generated by each function in a 16 processor system. The individual predictors are displayed in terms of their PVP — how likely a positive prediction is to be correct — and their Sensitivity — how likely they are to predict positive when they should. Displayed are those predictors for each function that were co-optimal across these two parameters.

As you can see, the perceptron completely dominates the Two-Level predictor, as well as the more sensitive intersection predictors and higher PVP union predictors. The perceptron has many more points because threshold was varied. With threshold held constant, only a few predictors are co-optimal. In order to choose a predictor for a full system design, it is necessary to understand the relative worth of PVP and sensitivity. This is discussed at length later in this chapter.

Table 4.1 displays each of the predictors in the co-optimal set. The results indicate that the perceptron predictor does best with a single indexing scheme, and can be tuned across a wide range by varying only the threshold. The perceptron achieves performance between Intersection and Union in both PVP and sensitivity at the same time. It is possible to increase the sensitivity of the intersection predictors by decreasing their depth, but they never achieve the sensitivity of any of the other predictors. This is because a shorter depth means that fewer bitmaps must be consistent with one another for a prediction to be made. None of the other predictors can offer as high a PVP as the intersection predictor. Similarly, the Union predictors dominated in terms of sensitivity, but had relatively low PVPs.

While intersection based predictors can be tuned across a range of PVP and

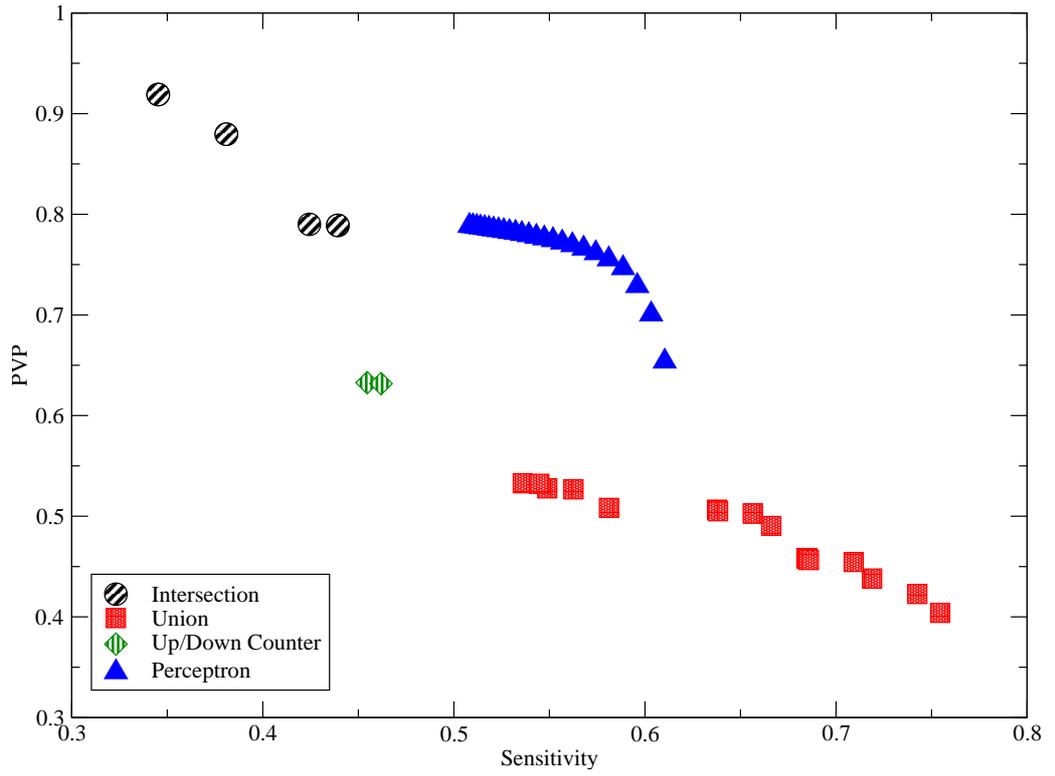


Figure 4.4: The set of co-optimal predictors found for each prediction function. Note the offsets on both axes.

<i>Predictor</i>	<i>Sens.</i>	<i>PVP</i>	<i>Predictor</i>	<i>Sens.</i>	<i>PVP</i>
$Inter.(pid + pc_{16} + addr_2)^2$	0.434	0.789	$2Level(pid + pc_{10} + addr_2)^4$	0.461	0.632
$Inter.(pid + pc_{16})^2$	0.439	0.789	$2Level(pid + pc_{14})^4$	0.455	0.633
$Inter.(pid + pc_{16})^3$	0.381	0.880	$Per_{.10}(pid + pc_6 + addr_{12})^4$	0.610	0.654
$Inter.(pid + pc_{16})^4$	0.345	0.919	\vdots	\vdots	\vdots
$Union(pid + pc_{16} + addr_2)^4$	0.743	0.423	$Per_{.490}(pid + pc_6 + addr_{12})^4$	0.508	0.789
$Union(pid + pc_{18})^3$	0.719	0.438	$Union(pid + pc_{10} + addr_8)^2$	0.581	0.508
$Union(pid + pc_{16} + addr_2)^3$	0.709	0.454	$Union(pid + pc_8 + addr_{10})^2$	0.563	0.527
$Union(pid + pc_{14} + dir)^3$	0.685	0.458	$Union(pid + pc_{18})^4$	0.755	0.404
$Union(pid + pc_{16} + addr_2)^2$	0.656	0.503	$Union(pid + pc_6 + addr_{12})^2$	0.545	0.532

Table 4.1: Co-optimal Consumer Set predictors for each of the predictor functions. Where performance matched within three significant figures only the smallest predictor is shown.

sensitivity, they offer a very discrete set of performances based upon the depth of the history table. Also, tuning both intersection and union predictors requires changing the indexing scheme, which means that any data within the table is effectively lost. The perceptron performs best when located at the processors, and when the history table is indexed with twice as many address bits as program counter bits.

The behavior of the perceptron predictor can be adjusted using its threshold, without changing the indexing scheme. Intuitively, the higher the threshold the slower the perceptron will be to adapt, and thus it will miss opportunities where a more flexible predictor could have reacted. At the same time, a higher threshold should result in a perceptron that is less affected by a few occurrences that do not exactly match more common patterns. As can be seen from Figure 4.5 the results match intuition. Here, the sensitivity and PVP of the four best perceptron predictors are shown as the threshold is varied. Raising the threshold means a higher PVP, but a lower sensitivity.

Figure 4.6 shows the behavior of one co-optimal perceptron, intersection, and union predictor on each of the benchmarks tested. The overall trends of the various functions are constant across all of the benchmarks tested. In general, PVP is more consistent across benchmarks than is sensitivity.

4.1.3 Performance of Predictors Acting on Restricted Information

For various reasons that will be discussed in the next section it is desirable not to implement prediction at the cache, as every predictor in the co-optimal set

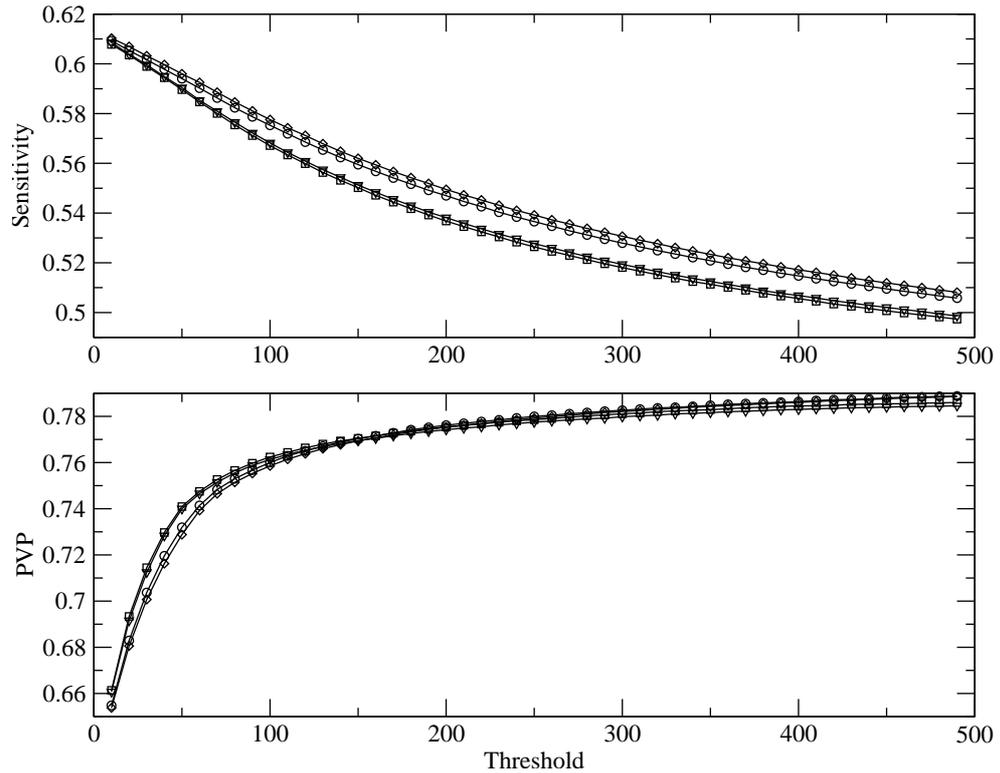


Figure 4.5: Behavior of several perceptrons across variations in threshold

would require. Further, there are reasons that the instruction information may be difficult to acquire. From the data gathered earlier it is possible to describe the performance of such predictors.

Figure 4.2 shows the change in performance of several predictors, first when it is moved to the directory, and then when it is changed to an address-indexed mode. Each of the three functions behaves differently when the indexing mechanism changes.

In the case of intersection the placement of the tables at the directory, with the cpu number still included causes a dip in both sensitivity and PVP. A larger dip in both occurs when the cpu number is removed. Interestingly, there is little difference between $Intersection(dir + pc_{16})^4$ and $Intersection(dir + addr_{16}^4)$. This emphasizes

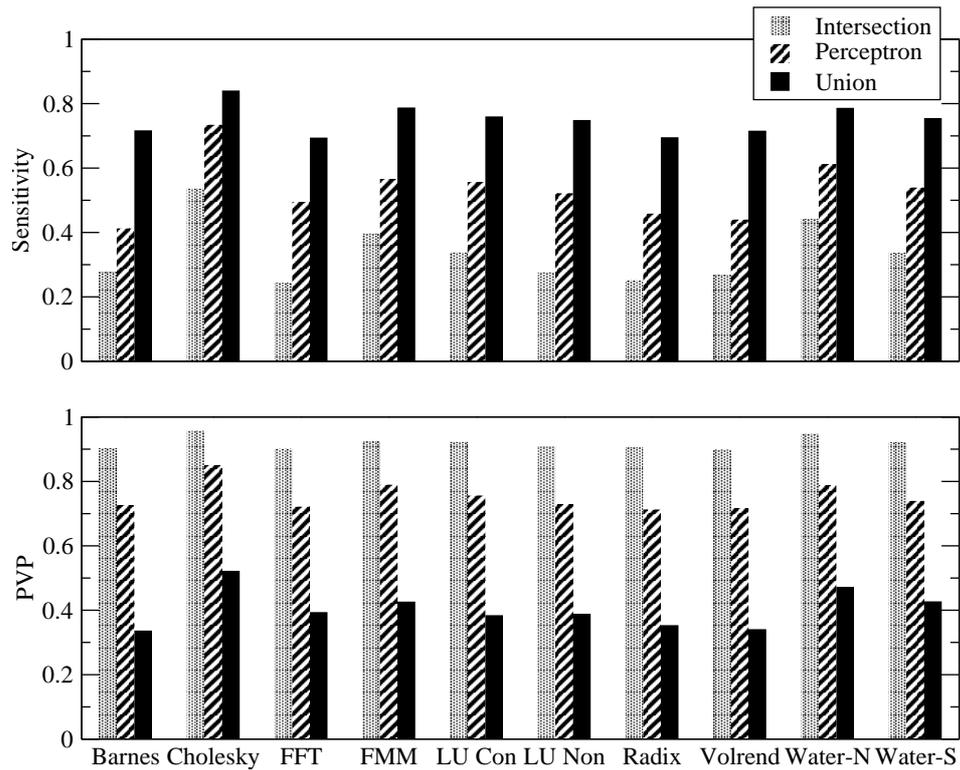


Figure 4.6: Behavior of three consumer-set predictors on the SPLASH benchmarks used: $Intersection(pid+pc_{18})^4$, $Perceptron_{120}(pid+pc_{10}+addr_2)^4$, and $Union(pid+pc_{18})^4$.

Predictor	Sens.	PVP
<i>Inter.(pid + pc₁₆⁴)</i>	0.345	0.919
<i>Inter.(dir + pid + pc₁₆⁴)</i>	0.278	0.908
<i>Inter.(dir + pc₁₆⁴)</i>	0.191	0.823
<i>Inter.(dir + addr₁₆⁴)</i>	0.199	0.834
<i>Union(pid + pc₁₈⁴)</i>	0.755	0.404
<i>Union(dir + pid + pc₁₈⁴)</i>	0.707	0.427
<i>Union(dir + pc₁₈⁴)</i>	0.748	0.403
<i>Union(dir + addr₁₈⁴)</i>	0.659	0.420
<i>Per._{.50}(pid + pc₆ + addr₁₂)⁴</i>	0.571	0.764
<i>Per._{.50}(pid + pc₁₆)⁴</i>	0.556	0.755
<i>Per._{.50}(dir + pid + pc₁₆)⁴</i>	0.468	0.706
<i>Per._{.50}(dir + pc₁₆)⁴</i>	0.383	0.638
<i>Per._{.50}(pid + addr₁₆)⁴</i>	0.534	0.726
<i>Per._{.50}(pid + dir + addr₁₆)⁴</i>	0.431	0.660
<i>Per._{.50}(dir + addr₁₆)⁴</i>	0.441	0.641

Table 4.2: Affects of Changing the Indexing Scheme.

that from the perspective of the intersection consumer predictor the processor that wrote the line contains a substantial amount of information.

On the other hand, Union changes little in the transition from the processor to the directory. However, the sensitivity dips substantially in the change to address indexing, and the PVP rises slightly.

The original perceptron predictor identified used a mix of pc and address information. Figure 4.2 shows that there is not a substantial difference between the perceptron addressed by just one or the other of these. On the other hand, the performance of the perceptron predictor falls in the transition to the directory.

Notice that the address based scheme handles this transition better than the pc scheme. This emphasizes the fact that pc information is associated with the

processor, where address information is associated with the directory.

4.2 Structure of A Consumer Predictor

4.2.1 Supporting A Consumer Predictor in A Directory Protocol

4.2.1.1 Coherence and Training Issues

The previous section discussed the general form of a consumer predictor. However, the previously proposed designs were general, with no attention to how data distribution could be accomplished, or its effects. They also ignored the possibility that an implementable training mechanism could result in out-of-order training. One of the main assumptions made was that the predictor could be placed at any point in the system. The result of this is that the final predictors are located at the caches. There are two major challenges to placing the predictors at the caches: training the predictor and maintaining coherence.

In order to train a predictor, data about where in the system a line has resided is necessary. If the predictor is located at a cache, these data must be forwarded to the cache before predictions are made. Doing so will either require additional messages being sent throughout the system constantly, or potentially substantial latencies in training as data are “piggybacked” around the system. While this forwarding is possible, it represents an additional use of bandwidth, and unnecessary delays in training.

Once a consumer prediction has been made, data must be forwarded to obtain

any benefit. In order to maintain coherence, the directory must be updated promptly whenever such a prediction is made. However, if the directory attempts to resolve another request before it is updated, a variety of race conditions can occur that cause coherence errors.

In general, attempts to implement data forwarding at the caches lead to persistent race conditions. Figure 4.7 shows one example of the resulting race conditions. It is a frequent occurrence that requests occur at or near the same time as data forwarding. When the directory sees such requests before it is aware that data has been forwarded, it incorrectly determines the caches that must receive invalidations. The directory can correct by sending more invalidations when it receives such a message. However, without guarantees about message timing, these invalidations may arrive *before* the forwarded data. Additionally, the information about the data forwarding may arrive *after* the directory has granted another processor a Modify copy.

Because of the coherence issues, all forwarded data in a directory system must pass through the directory before being sent to consumers, or additional complexity must be added to the caches to handle race conditions. Of the options shown in Figure 4.8, this work implements prediction at the directory. Placing the predictor at the cache will make training difficult and gives no performance benefit in terms of the number of messages needed to move data to the consumers. Interactions between transient states are difficult to verify and debug. On the other hand, predictors implemented at the directory are relatively easy to verify, can take advantage of the fact that many protocols already support sending multiple Shared Copies at once, and are relatively easy to train. While the earlier data are valid, it will be more

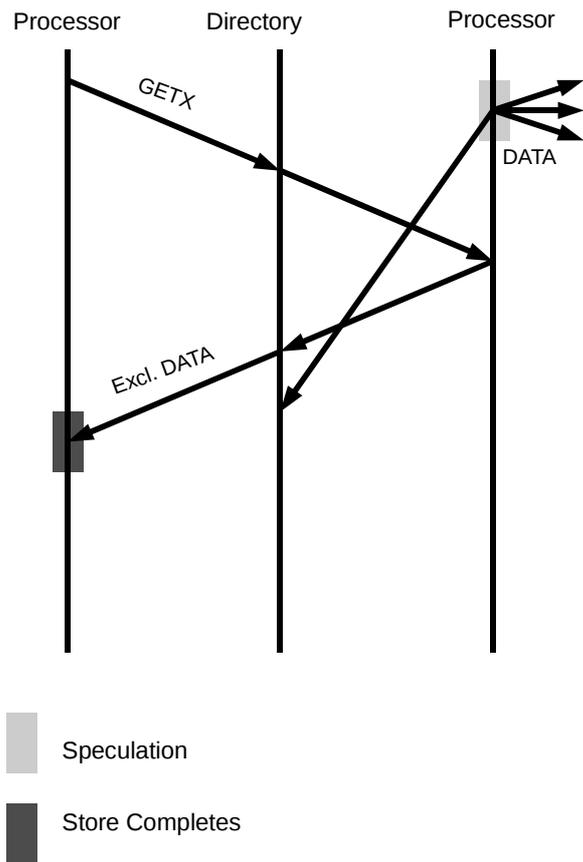


Figure 4.7: Potential Race Conditions Occurring due to Cache Based Data Forwarding

difficult to implement correctly.

4.2.1.2 Implementation Requirements

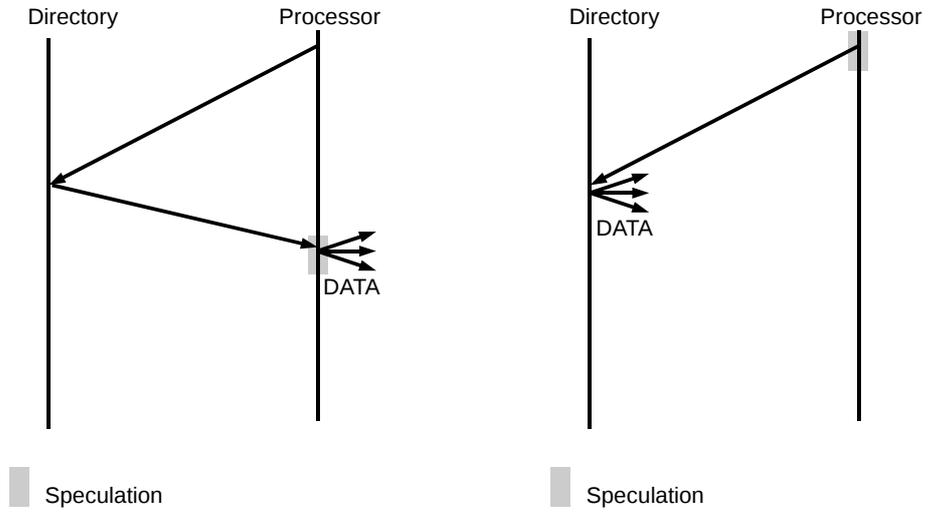
The consumer predictors discussed in this chapter all follow the same general format: a table of histories is maintained and updated each time a processor requests modify permission. This table is indexed by some combination of address information, PC information corresponding to the last instruction to write the line, and processor information.

Maintaining the history is relatively simple as the directory is aware of every sharer on every line. The only complex part is handling a case where the speculation causes this information to be incorrect. In this dissertation, this phenomenon is referred to as “lock-in.” In lock-in processors are speculatively identified as sharers and send a Shared copy. Even though the processors may never access that copy, the directory must remember that they possess it. This can result in a sharing pattern locking into the history table, and never changing.

The problem of lock-in is resolved using a variation on the dirty bit. When a cache has a Shared copy, it can use its dirty bit to remember whether the line was touched at all, rather than whether it has been written. This bit can then be included in the message sent when the processor acknowledges that it has been invalidated to the cache.

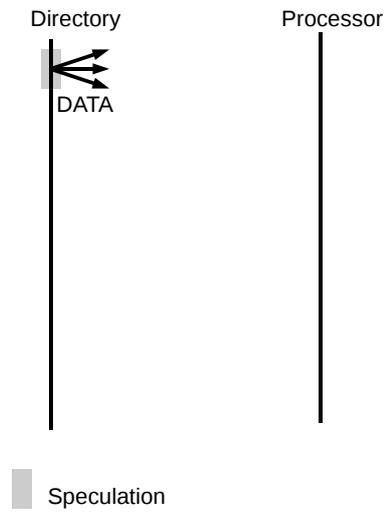
Data is forwarded on two occasions:

- Whenever a processor requests read permission and a Modify copy is present



(a) Speculation After Acknowledgment

(b) Speculation Through Directory



(c) Speculation At Directory

Figure 4.8: Three options for implementing speculative data forwarding.

in another processor's cache.

- Whenever a processor with modify permission releases a line on its own.

This means that whenever the first request for a Shared Copy arrives after a change to a line, the system will attempt to predict the other consumers. The first consumer cannot receive any benefit. Additionally, whenever a line that has been modified is released from a cache a prediction is made. In the case of contention in the cache, this is likely to be of little value as there is no reason to believe that line should be sent to any other processors. The purpose of caches is to avoid the loss of such important data, and in fact, if functioning poorly, consumer prediction could aggravate thrashing in the cache. In the case of speculative self invalidation, this may provide large benefits. The issue of combined speculative self invalidation and consumer prediction is discussed in Chapter 7.

4.2.2 Supporting Program Counter Based Techniques

In order to support the PC based techniques identified in previous chapters, information about the PC of the last store to access a line must be forwarded to the directory. In addition, information about the last writer of a line must be stored at the directory for training purposes. This is accomplished by modifying the caches and the messages sent by the cache to the directory.

The cache is modified so that on any store operation the PC of the instruction is written into a new field associated with its corresponding cache line. The messages are modified so that any packet containing data sent from a cache to the directory

will contain this value.

This technique requires an increase in cache size, an increase in directory size, and an increase in message size. Because of this, this study was performed using both the higher sensitivity/PVP PC based techniques, and less accurate but also less expensive address based techniques in this chapter.

4.2.3 Directory and Cache Changes to Support Consumer Prediction

In order to provide data speculatively to the caches, changes must be made at both the directory and the cache levels. The same overall implementation strategy discussed in Chapter 3 has been employed. All of the decision making and synchronization occurs at the directory.

However, an additional race condition occurs in which a processor that has requested data receives a speculative copy of those data. In this case the directory is waiting on an acknowledgment from the processor, but first receives a request for data. Because this request could have been sent after an acknowledgment, it cannot be discarded. This request will remain waiting to be processed until the requester loses access permission. At that point an unexpected data message will be transmitted.

To resolve this, the cache must send a response to the directory notifying it of the issue. Once the cache's request has been discarded by the directory, the cache can acknowledge that the transaction is complete. The cache effectively locks the directory in its current transient state, where it cannot respond to other requests,

until the issue is resolved. Figures 4.9, 4.10, and 4.11 show these changes. The additional transient state, SU, at the cache represents the cache waiting to respond to the directory until the directory finishes removing the request. Even the lowest PVP predictors in the system do not create this race condition on more than a single percent of the speculative data lines they transmit. One of the new events at the cache is also related to this race condition. RemReqAck is the acknowledgement that the directory sends the cache to tell it that its request has been found and removed.

The directory also has a new event to handle this, corresponding to the warning message from the cache identifying a request to be removed. However, the directory does not change state based on this message; instead the message remains in the directory's queue until a request arrives stating that it can be removed. This guarantees that the directory never leaves the transient state corresponding to giving a Shared copy to the predicted cache until after the request is gone.

The other two new Events that can occur at the cache correspond to the arrival of speculative data. In particular, if speculative data arrives and there is no space in the cache, it does not evict any other lines. Instead, it is immediately sent back to the directory as though there had been an L2 Replacement. It might be possible to implement a victim-cache like structure that temporarily holds the data; however for the benchmarks in question there is space for the line in the L2 or L1 cache most of the time. This is because frequently shared data leaves invalid copies behind in the cache. Thus, most speculatively transmitted datum arrives to find the same place in the cache that it had previously occupied.

<i>State Name</i>	<i>Description</i>
SU	Received speculative data after requesting a Shared copy

Figure 4.9: Additional Cache States Necessary To Support Consumer Prediction

<i>Event Name</i>	<i>Description</i>
DataSpec	Speculative data arrived from the directory
Data no space	Data arrived from the directory, but does not fit in the cache
RemReqAck	Directory Acknowledges that the request has been handled.

Figure 4.10: Additional Cache Events Necessary To Support Consumer Prediction

The only changes necessary at the directory are the capability to acknowledge these messages while deleting the request, and the capability to predict sharers and multi-cast data. Neither of these corresponds to new states or transitions, though the actions taken on those transitions may change. Specifically, when first entering a transient state leading to the Shared state, more target caches are likely to be present.

An example of the behavior of this system is shown in Figure 4.12. In this example, all of the messages, state transitions, and memory instructions for a single line of memory are shown. The sequence of instructions is similar to that encountered by any global variable that is owned by a single processor. For example, the “normal” information that is held in Volrend to indicate the viewing angle will follow this pattern, with the exception that there will be more loads between stores. This pattern also appears in false sharing, where one processor is updating a line, and other processors need to request access.

Figure 4.12 shows the accesses to this line. The store instructions executed by

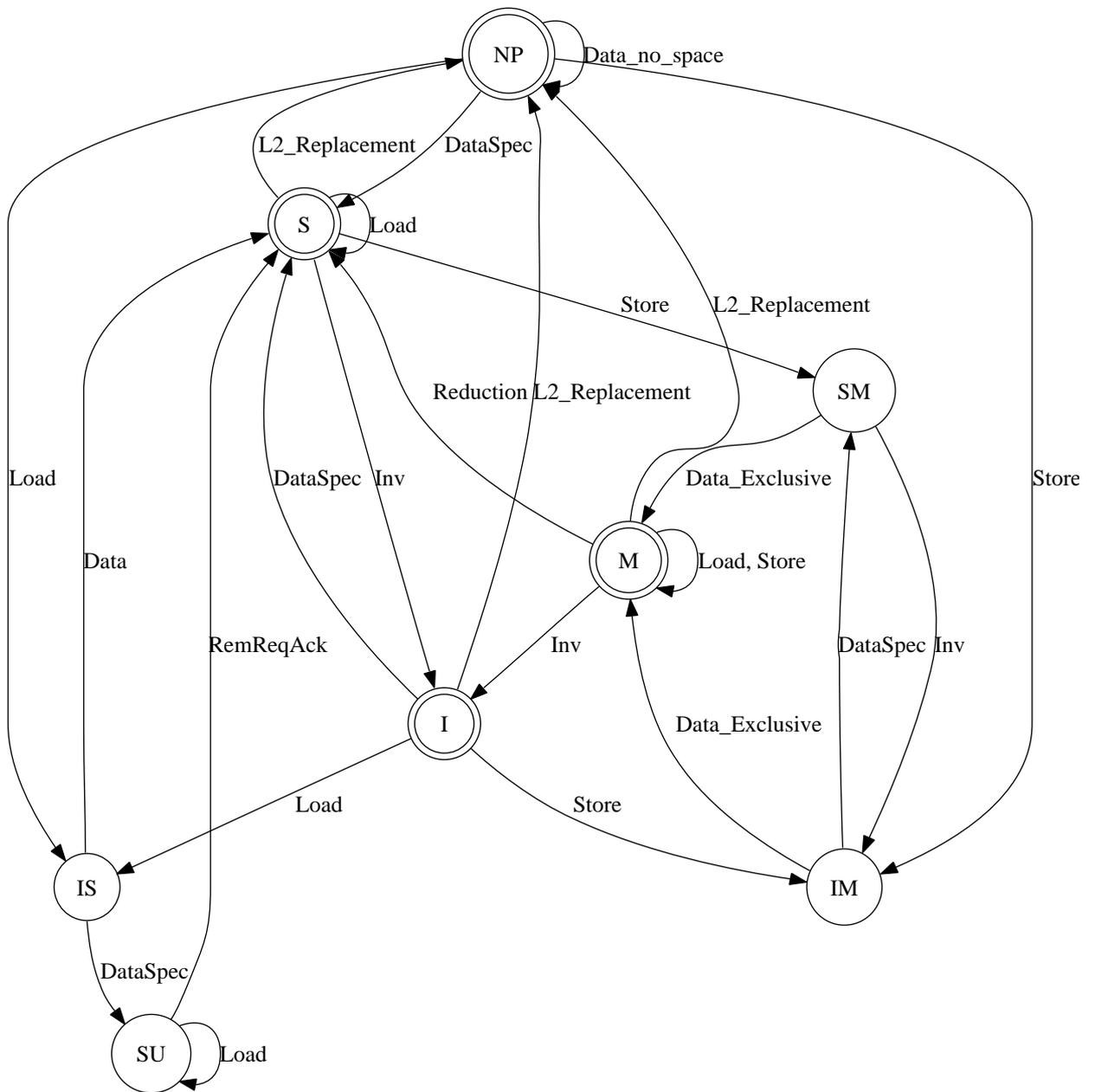


Figure 4.11: State Machine Representation of a Cache Supporting Timing Prediction and Consumer Prediction.

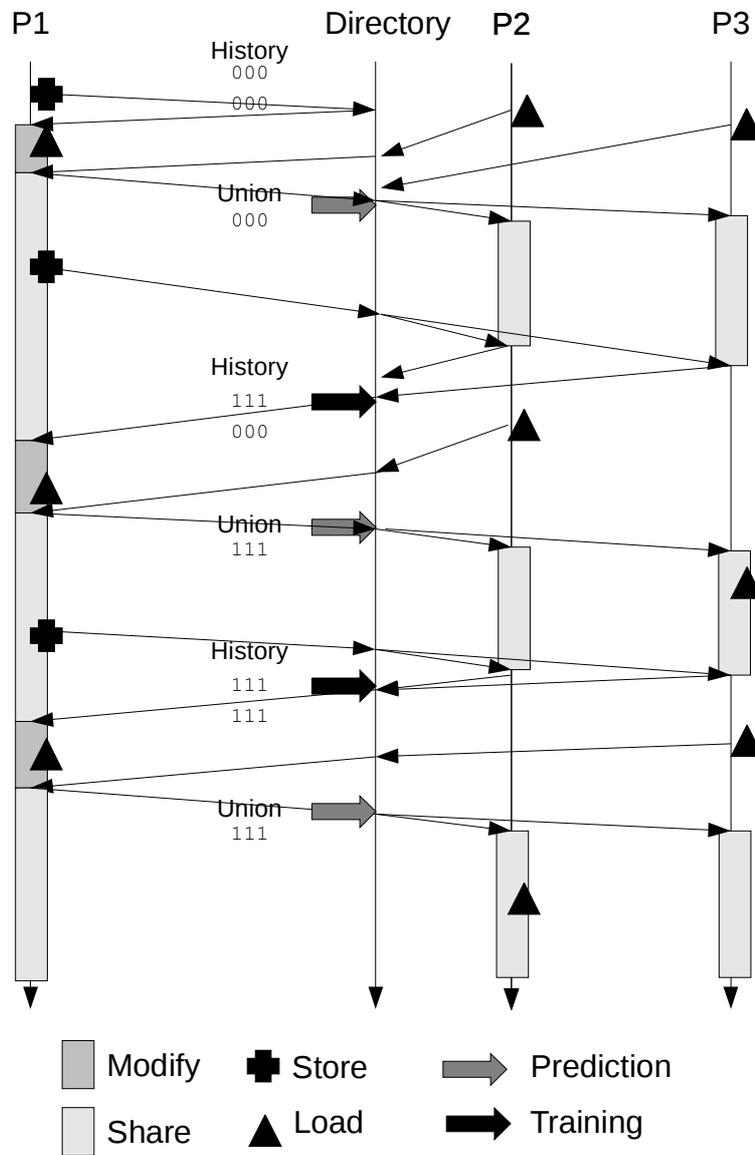


Figure 4.12: Example of the Behavior of Union Prediction on the distribution of a Global Variable.

the first processor each trigger requests to the directory for modify permission. Once the value is written, the other processors access it. The first processor is invalidated and the data are sent to the requesters.

The next time a processor requests a Modify copy invalidations are sent. Only once all are returned is the history table updated. This ensures that the sharing pattern will be accurate, in case one of the Sharers never reads their copy.

When a request from the second processor arrives the directory is contacted as normal. When the data reaches the directory a speculation is made. Because the history table remembers all of the processors having shared permission in the past it transmits the line not only to the requesting processor, but also any other processors — in this case the third processor. This means that the third processor does not have to wait on the directory to perform its load.

Similarly, in the next phase the third processor requests a Shared copy first. Because the second processor had accessed the line before it is sent a copy before it requests it, reducing the latency of that memory access.

Had an intersection predictor been used instead it would have not helped on the second set of requests, as no processors had *always* wanted access for as long as it could remember. On the third set of requests the behavior would have been identical.

The behavior also would not change if the writer changed from phase to phase. For instance, had processor two performed the second write, Union predictor would have forwarded access to whichever of processor one and three had not requested access.

4.3 Behavior of Consumer Prediction

4.3.1 Measuring the Behavior of a Consumer Predictor

In a previous section, the PVP and sensitivity of a variety of consumer predictors were analyzed, and it was shown that it is possible to tune predictor performance, by controlling the indexing method or the threshold of the perceptron predictor. This section analyzes the actual system impact of implementing a consumer predictor. This analysis is performed in terms of the eight factors identified in Section 3.1.2. If consumer prediction works as expected, the following should occur:

1. *Relative Execution Time*: Ideally, execution time will fall as the amount of time spent waiting for the memory system falls. In the worst case, execution time will rise as the amount of time necessary for the directory to respond with a Modify copy rises due to an increased number of sharers.
2. *Relative Invalidations*: It is expected that False Positives in a consumer predictor will increase the number of invalidations. True Positives, False Negatives, and True Negatives should have no effect. Thus in general the number of invalidations will rise when consumer prediction is used. The increase in invalidations should be inversely proportional to the PVP of the predictor used.
3. *Relative Requests*: It is expected that True Positives will decrease the number of requests that caches need to send to the directory. False Positives, False Negatives, and True Negatives should have no effect. In general it is

expected that the number of request messages will decrease when consumer prediction is used. The decrease in request messages should be proportional to the sensitivity of the predictor used.

4. *Relative Bandwidth Usage*: It is expected that False Positives will cause the bandwidth consumed to rise as additional data are transmitted, and True Positives will cause it to fall as fewer requests are necessary. False Negatives will result in the same messages being transmitted as before, and True Negatives will correspond to no messages being sent or needed. Because transmitting data consumes substantially more bandwidth than a control message, the expectation is that total bandwidth used will rise. As the PVP of the predictor rises this increase in bandwidth usage will decline.
5. *Relative Load Miss Rate*: True positives should result in a load miss being removed, as the data arrives at the cache before they are requested. False positives should have no affect on this parameter. In general, correct consumer prediction should cause the Load Miss Rate to fall.
6. *Relative Store Miss Rate*: Consumer prediction should have no affect on this parameter, as Modify Copies are not being speculatively distributed.
7. *Relative Load Miss Latency*: This is the number of stall cycles caused by an average L1 cache miss that results in a request for a Shared copy. When consumer prediction works perfectly, this number should fall when the invalid lines being filled in by consumer prediction are in the L2 cache. When these

lines are located in the L1 cache, there should be no change in this number. However, if speculative messages are already in transit when requests are sent, this number should fall.

8. *Relative Store Miss Latency*: This is the amount of time the average L1 Store Miss takes. Consumer prediction should increase this slightly as more invalidations may need to be sent and acknowledged before the directory can transmit a Modify Copy.

4.3.2 Varying the Predictor Design

This section explores the effects of a variety of consumer predictors on performance. The objective is to determine the tradeoffs in performance among the predictors, and to identify key qualities of execution that lead to the superiority of any given scheme.

Each of the predictors shown in Figure 4.13 is considered. These have been chosen to cover the simplest implementations, using only the address information, or only the instruction information, and to include the predictors as similar as possible to those identified previously. These are not the co-optimal predictors, as those are located at the caches. Instead these are a set of predictors intended to span the range of performance at the directory. They all have a depth of four, and the results from Section 4.1.2 indicate that intersection will be at the limit of PVP for the given indexing scheme, Union will be at the limit of sensitivity, and Perceptron will be somewhere in between the two.

<i>Predictor Name</i>	<i>Description</i>
$Intersection(dir + Addr_{15})^4$	Intersection Predictor based on address information
$Union(dir + Addr_{15})^4$	Union Predictor based on address information
$Perceptron_{50}(dir + Addr_{15})^4$	Perceptron Predictor with threshold 50, based on address information
$Intersection(dir + pc_{15})^4$	Intersection Predictor based on PC information
$Union(dir + pc_{15})^4$	Union Predictor based on PC information
$Perceptron_{50}(dir + pc_{15})^4$	Perceptron Predictor with threshold 50, based on PC information

Figure 4.13: Predictor Designs Considered Here

4.3.2.1 Runtime Results Across Varying Predictor Designs

Figure 4.14 shows the relative runtime obtained with each of the consumer predictors listed in Figure 4.13. From this graph several things are clear. Most importantly, only some benchmarks benefit substantially from consumer prediction. The three benchmarks that benefit substantially from consumer prediction are FFT, LU, and Volrend. The reasons for this difference are discussed in detail in the following sections, but are centered on the number of processors that typically share a line, and the nature of the processing these benchmarks perform.

For the benchmarks where some form of consumer prediction is effective with an address based history table, the Perceptron predictor outperforms the other two predictors, or is within one percent. In those cases where consumer prediction based on an address table is not effective, the “best” predictor is harder to determine, with both union and perceptron resulting in the lowest run-time an equal amount of the time.

The pattern is different for instruction based predictors. For these, there

are cases where each of the three predictors performs best. Additionally, Barnes and Cholesky, both of which suffered performance penalties with address based prediction, benefit slightly from instruction based prediction. An instruction indexed table decreases all of the remaining performance penalties to at most one percent.

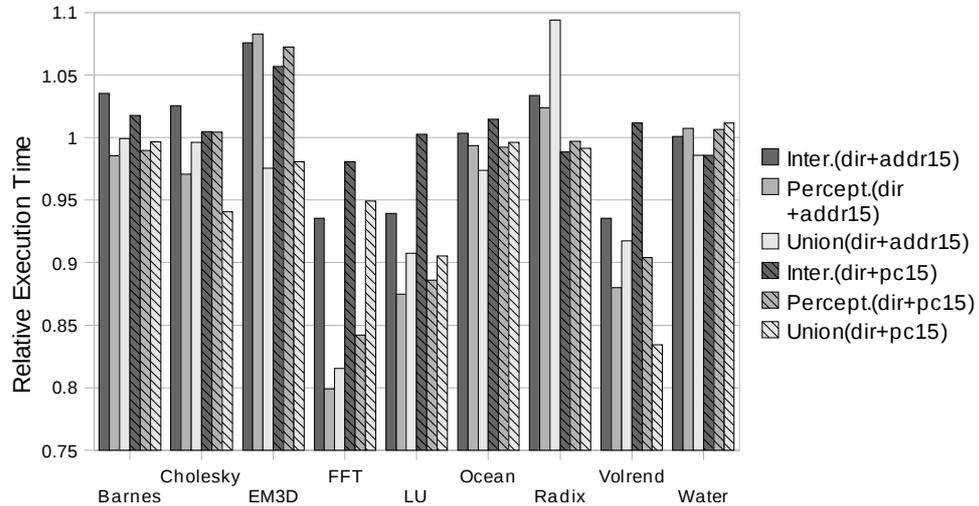


Figure 4.14: Runtime Benefits of Consumer Prediction for Address and PC schemes

4.3.2.2 Message Transmission Results Across Varying Predictor Designs

Figure 4.15 shows the relative number of invalidations sent with speculation in place. As expected, the number of invalidations grows as the PVP of the predictor used falls. The exact rate of growth is relatively consistent for intersection and perceptron predictors. The exception to this rule is Volrend.

When comparing the instruction-indexed table and the address-indexed table, the main feature to notice is that the behavior of the Union predictor changes the most. For both the Intersection and Perceptron predictors, the total number

of additional invalidations remains relatively constant. On the other hand, the instruction based Union predictor results in increases in the total number of extra invalidations as opposed to the address based union predictor across the board. As Figure 4.16 shows, in many cases this increase happens despite the fact that the number of requests does not fall. The instruction information here is not a good indexing scheme in many cases as the processor accessing the data was extremely relevant in the trace based results. Further, instruction based prediction suffers from additional hysteresis effects that address-indexed prediction does not due to the fact that training information will not necessarily arrive in order. The intersection predictor is best able to filter out the irrelevant data, and the union predictor is least able to do so.

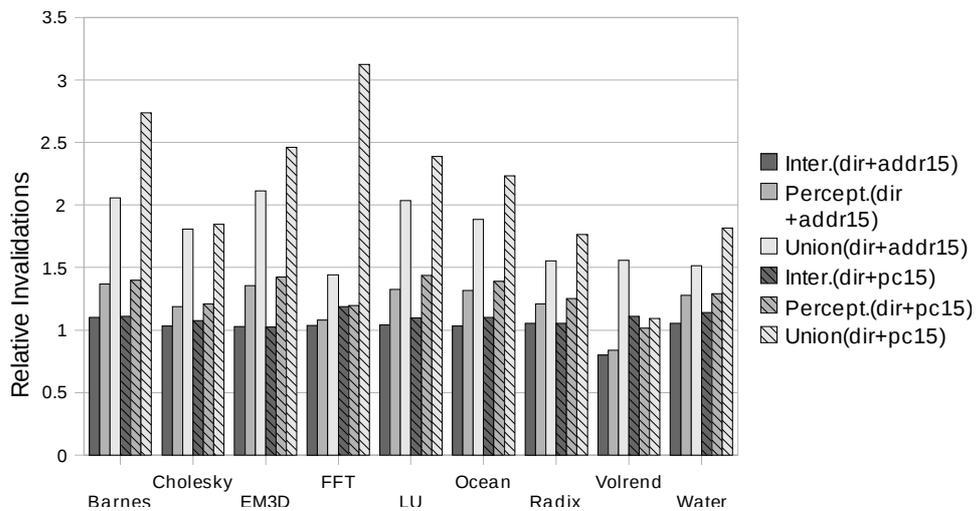


Figure 4.15: Change in Invalidation Messages Transmitted for Various Consumer Predictor Schemes

In addition, for FFT and Volrend there are several predictors that cause the total number of invalidations to fall. This occurs in spite of the fact that the total number of instructions executed stays relatively constant or increases with the

transition to these predictors.

There are several reasons why this could occur. First, because the exact sequence of instructions executed can vary while still resulting in correct execution in a shared memory multi-processor, it could be that the total number of instructions executed has fallen. This is not the case. Though the total number of instructions executed does change, the magnitude of that change is within one to two percent of the total, while the number of invalidations in Volrend drops by as much as 30%. This is substantially smaller than the change in the number of invalidations.

Another possibility is that the order in which instructions occur on lines relative to the directory has changed. Because a cache receives a line earlier, more operations can be performed on it. This reduces the total number of times the line needs to be invalidated.

This behavior should be unusual with an individual variable, as such behavior could cause the final result of the program to be dependent on the speed of the directory. However, the behavior is a reality of locks, barriers, and other synchronization variables. It is also a possibility that this behavior can be caused by false sharing. When false sharing occurs, many valid orders of access may exist to a line that not only satisfy the memory model, but also the program flow. This effect is consistent with the fact that the total number of instructions executed has not changed significantly.

However, Volrend does not spend substantial amounts of time in synchronization, while LU does [49]. When the total number of invalidations is not normalized, it is small with respect to other benchmarks. Volrend shares data in three ways:

First, it has a small set of data that is written at the beginning of each phase and then distributed to all of the processors. Second, it has a set of queues, which it uses to re-distribute work when one processor finishes its assigned tasks ahead of another. Third, it has some amount of false sharing due to data layout.

The majority of the invalidations in Volrend come from the second kind of sharing. Consumer prediction accelerates the first and third kinds of sharing. The important global data are identified early in execution, and all processors receive it more quickly. False sharing is recovered from faster, as data are sent back to all the owners when the first processor requests access. As a result, some processors do not fall as far behind the others, and there is less activity in re-distributing work. Thus the code that produces the most invalidations is not exercised as much, and the total number of invalidations falls.

Figure 4.16 shows the relative number of requests needed with speculation in place. For those benchmarks where performance benefits are reduced by the switch to instruction based prediction, the number of requests increased, despite the increase in invalidations. For those benchmarks where performance benefits appeared due to consumer prediction the total number of requests decreased.

Another interesting thing to note is the performance of the address based Union predictor on the Water benchmark. The total number of requests sent was reduced by nearly 20%. This reduction is greater than the reduction for FFT and LU, two benchmarks with excellent runtime behavior. In addition, the total number of extra invalidations transmitted by this scheme was comparable to that for FFT and Volrend, and lower than that for LU. Despite this, there was no performance

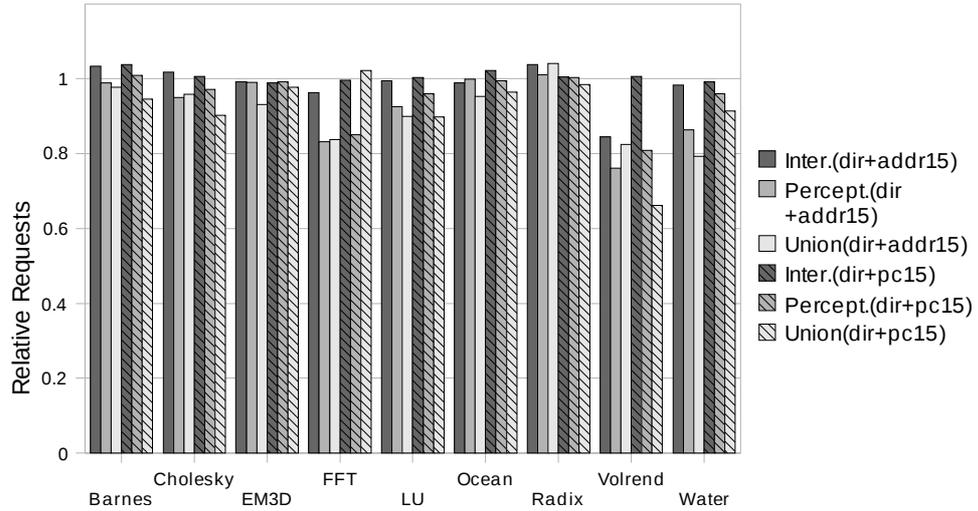


Figure 4.16: Change in Request Messages Transmitted for Various Consumer Predictor Schemes

benefit from applying consumer prediction.

The reason for this is that the number of sharing misses in Water was already very low (6.4 per 100,000 instructions). Thus there is little speedup that can be gained from consumer prediction, despite the fact that consumer prediction accurately identifies sharing patterns and acts on them.

This trend is repeated at a lower scale on every other benchmark. In the case of every benchmark tested, the total number of requests transmitted was reduced by at least one predictor. However, in many cases this did not translate into speedups.

Figure 4.16 also emphasizes a distinction between practical implementations of consumer prediction and the theoretical limit. The reduction in requests is much lower than would be indicated by the extremely high sensitivities of the Union predictor shown in Figure 4.6. The reason is that the practical implementation being evaluated has to wait for the data to be released by the cache. The trigger for data being released by the cache is a request message. Additionally, if the processor

that wrote the datum is one of the sharers, there is no action for speculation to perform. This means that the first sharer of each line is not predicted, nor is a second sharer in all cases where data is consumed by the processor that produced it.

The same problem exists if the consumer prediction is performed at the cache. No speculation is necessary to move data from the producing processor to itself, and the cache does not know to release data before a request arrives. Other methods to mitigate this issue are detailed in Chapter 7.

4.3.2.3 Bandwidth Usage Results Across Varying Predictor Designs

Figure 4.17 shows the resulting bandwidth use of each of the implemented prediction schemes. Overall, the only substantial decreases in bandwidth use are for the Volrend benchmark. The reason for this is the greatly decreased number of invalidation and request messages associated with this benchmark. In the case of the other benchmarks, bandwidth usage increases as the PVP of the predictor falls.

Overall, it can be seen that the trends in the bandwidth usage are closely tied to the number of extra invalidations that needed to be sent, as shown in Figure 4.15. This is to be expected, as the majority of the bandwidth change caused by consumer prediction should be due to the extra data messages transmitted. Each of these extra data messages should translate to an extra invalidation.

For several of the benchmarks where fewer requests were needed the amount of bandwidth consumed actually fell. One of two different effects causes this. In the

case of Volrend the total amount of interprocess communication decreases as the system becomes more efficient at handling global and false sharing. In addition, in some cases the PVP of the predictor is sufficient to reduce the amount of wasted bandwidth by extra messages to below the savings of the request messages.

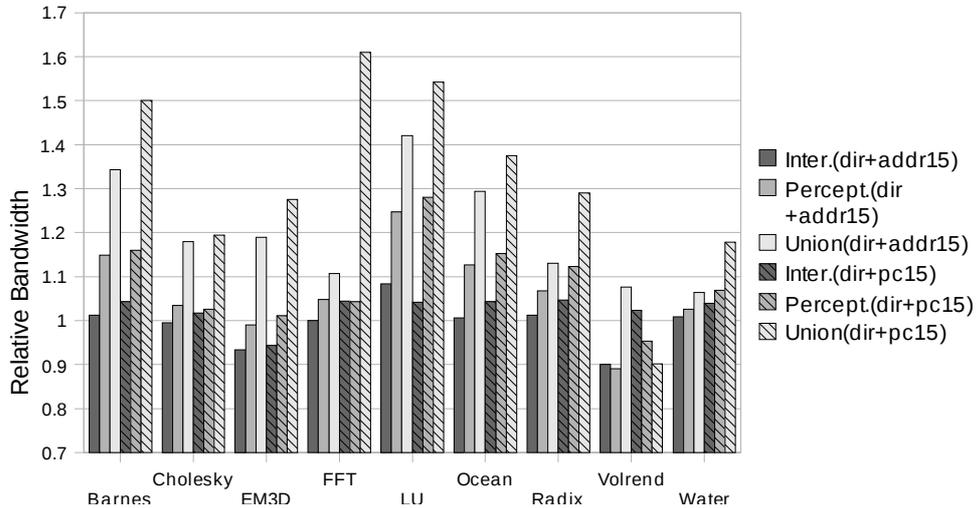


Figure 4.17: Effects of Consumer Prediction on Bandwidth Use

4.3.2.4 Miss Rates Across Varying Predictor Designs

Figure 4.18 shows the change in the miss rate of load instructions for each of the proposed predictors. With the exception of Volrend, the trends in miss rate for different predictors is related to the sensitivity of the predictor. In other words, the more data a predictor successfully transmits to consumers, the fewer the cache misses. Volrend performs the way it does because of the fact that Intersection best reduced the runtime of its critical path, removing large portions of the sharing that causes read misses.

Additionally, it is notable that, except in a few cases, consumer prediction

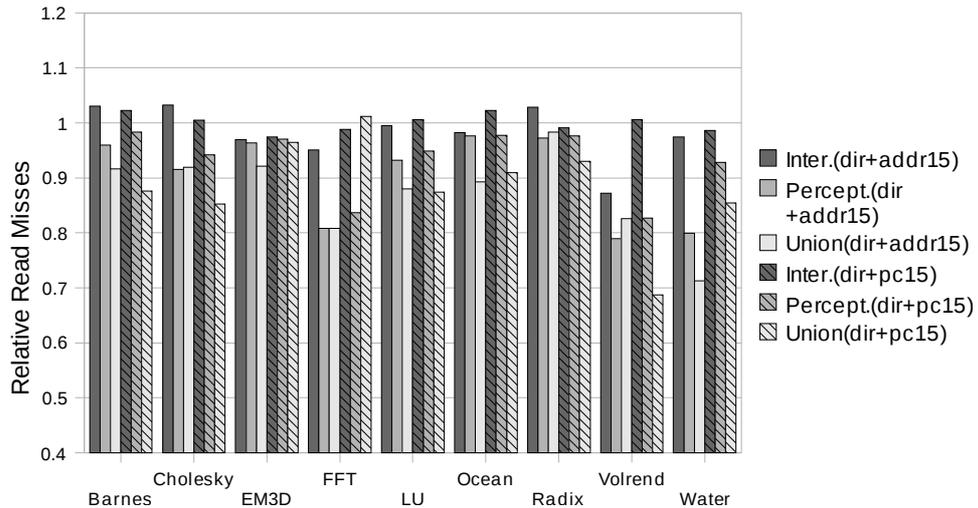


Figure 4.18: Effects of Consumer Prediction on Load Misses

reduced the load miss rate, even when no performance benefits were attained. This emphasizes the fact that speedups only occur when the thread that comprises the critical path is sped up. For example, the miss rate of Cholesky drops by nearly 10% for the Instruction-based Union predictor. Despite that, there is in fact a small slowdown in the runtime with this predictor. It is not enough to successfully predict some sharing; the correct threads must see speedups.

Figure 4.19 shows the change in miss rate of store instructions for each of the proposed predictors. All three of the benchmarks with the best speedups have a drop in the total number of Store Misses. For Volrend these are explained in the same way as the drop in invalidation count. For LU and FFT the reason is less clear.

This reduction in store misses is explained by a similar phenomenon that causes the decrease in write time for Volrend in the next section. When a line is being read by many processors and written by a few, consumer prediction can

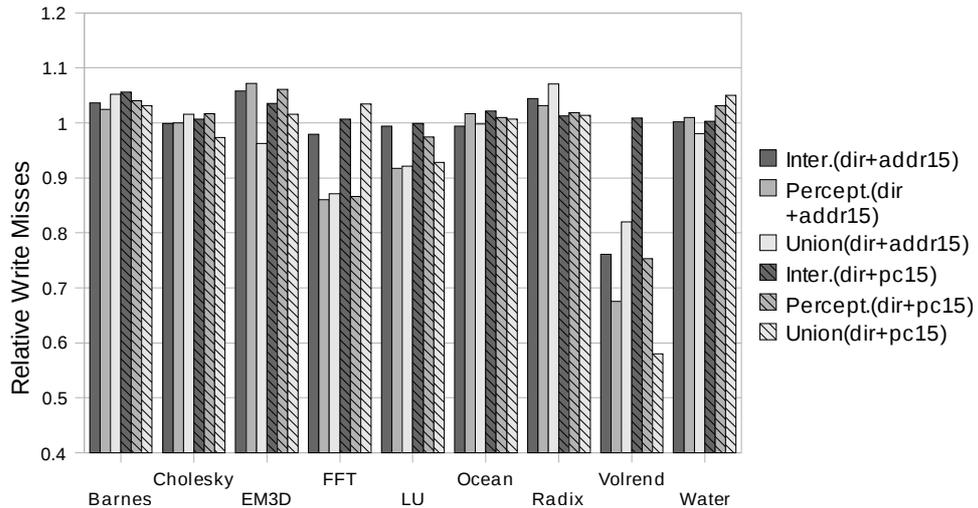


Figure 4.19: Effects of Consumer Prediction on Store Misses

reduce the number of store misses by allowing all the readers to read at once, and thus giving the writers access for a greater period of time.

4.3.2.5 Response Time Results Across Varying Predictor Designs

Figure 4.20 shows the latency of requests for read access on misses. Correct predictions that arrive in time will convert cache misses into cache hits. If they are placed in the L1 cache, there will be no change in this statistic. If they are placed in the L2 cache, the latency will fall.

Two of the benchmarks that show increases in read latency on miss (FFT and LU) are among the benchmarks that benefit the most from consumer prediction. The reason is that the speculative data in these benchmarks is more frequently being placed in the L1 cache, where no miss will be detected. When consumer prediction performs at its best, requests to the directory will require another processor give up Modify permission. If no processor has Modify permission, a prediction was already

made on this line. For this reason, the greatest increases in read latency can occur for the benchmarks where the reduction in execution time is highest. The average latency of a load instruction has fallen, as seen in the reduced load miss rate, but the reads that are left require greater service time.

In the case of every other benchmark, there is a decrease in the latency of requests for Shared permission. This means that correct predictions were being made, but were being placed in the L2 cache, rather than in the L1. Thus the total reduction in load time was comparatively smaller. An alternate strategy could be used to upgrade these lines into the L1 cache; however such a method could also result in moving important data out of the L1 cache.

Figure 4.21 shows the relative latency of store misses. This number should rise as more processors must be invalidated. However, these invalidations can be performed in parallel, and so the increase should be modest. As the PVP of the predictor falls the latency of requests for Modify access increases in most cases.

What is unexpected about these results is that in many cases the write latency fell. Because Modify permission is not sent speculatively, this decrease is not due to in-flight speculative data. By checking the state of the directory when different messages arrived, it is possible to confirm the reason for this change.

When accurate consumer prediction is in effect, requests for a Modify Copy are less likely to arrive while a line is in a transient state. Distribution of shared copies is completed earlier. When a request arrives during a transient state, the system cannot process the request until the previous transient has been resolved. Thus, consumer prediction is able to reduce the total write latency as well.

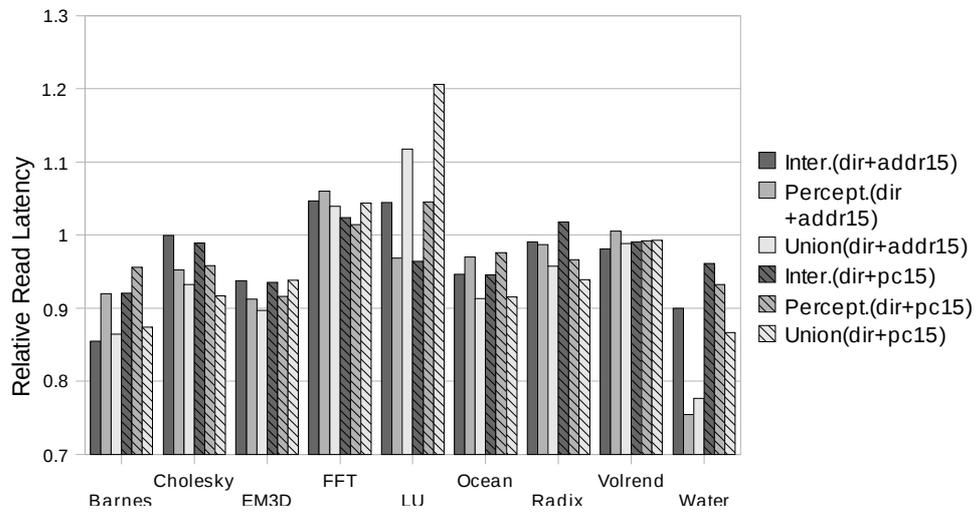


Figure 4.20: Effects of Consumer Prediction on the Read Latency of the Directory

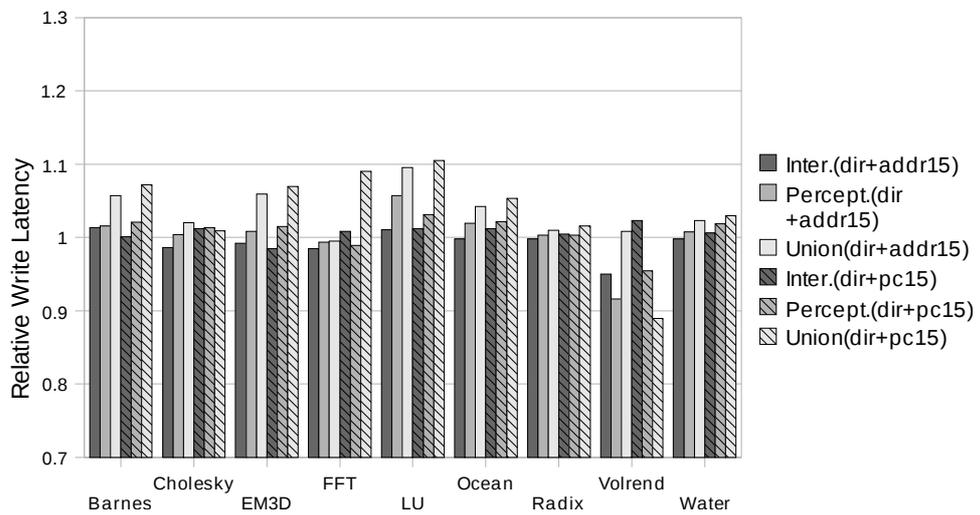


Figure 4.21: Effects of Consumer Prediction on the Write Latency of the Directory

4.3.3 Tying Processor Number Into the Indexing Scheme

The results in Figures 4.14 and 4.15 show that instruction based predictors have particularly poor performance in a number of cases. The number of false positives produced is substantially higher than is indicated by the trace based results. The most obvious explanation for this is that the trace-based instruction predictors were located at the caches, rather than at the directory. However, the difficulties of this kind of predictor placement were discussed earlier.

Fundamentally, one reason a pure instruction-indexed predictor performs so poorly is the nature of the benchmarks. In general, these benchmarks break the data up into regions, each processed by a different set of processors, but using the same code. Thus, while the instruction information may carry a great deal of weight, the information about which processors are used is not fully encapsulated by it. This intuition is confirmed by the data in Table 4.2. Including the id of the cache in the indexing scheme recovers a great deal of lost predictor performance on the traces.

This work considers two distinct methods of including this missing information into the indexing schemes: writer-mixing and address-mixing. Both of these indexing schemes focus on combining other information into the instruction signature.

4.3.3.1 Writer-Mixing

Writer-Mixing indexes the history table with both the processor that wrote the data and the instruction that wrote the data by replacing the top bits of the index with the cpu number. This removes three bits of instruction data (for eight

processors) in exchange for separating history information by writer. Conceptually, this last writer is strongly tied to the set of processors that act on the data, as it is one of them. The main difference between Writer-Mixing and the perceptron predictors indicated in the trace-based study is that by virtue of being split across the directories, these are effectively indexed by directory, processor, and instruction. This is equivalent to having a separate table at each directory for each of the processors. Earlier data has shown that the performance of this predictor is substantially closer to the ideal performance of a predictor located at the caches than the simply indexed schemes.

Figure 4.22 shows the speedups that are obtained by using a Writer-Mixing predictor. As can be seen, the slowdowns of the pure instruction-indexed table are mostly removed, except for a new one in Radix. The performance on FFT degrades slightly. Small speedups are attained on Cholesky, and EM3D that were not previously available. Figure 4.24 shows the reason for the spike in slowdown for the perceptron predictor for the Radix benchmark. The total number of requests necessary has actually increased. Overall, this change is not as effective as Figure 4.2 would suggest.

The number of extra invalidations needed, shown in Figure 4.23, is not particularly notable. In general there is no substantial change, except in the case of the union predictor. For the union predictor the number of invalidations grows in almost every case. This is related to the training order.

Any indexing scheme using instruction information will have entries placed in the history table in an order based on when the next request for Modify permission

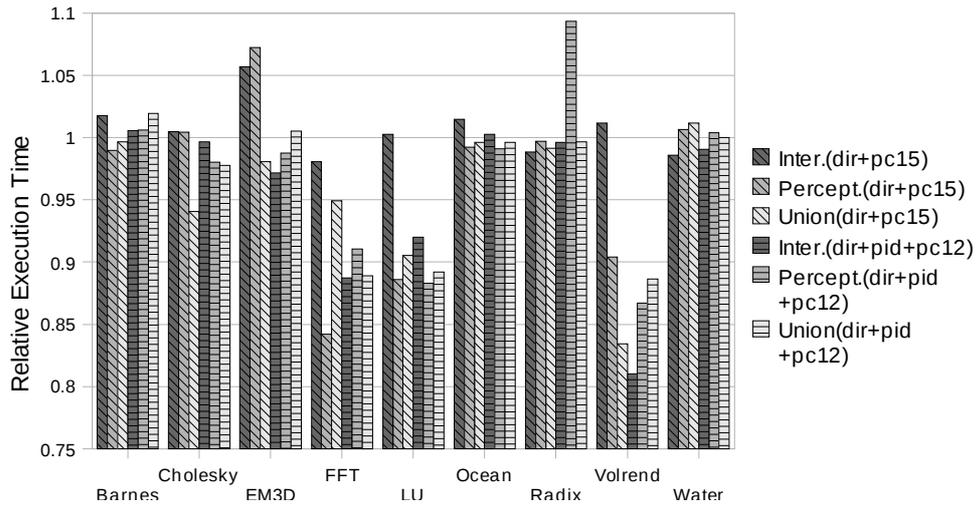


Figure 4.22: Runtime of Mixed Writer-Instruction Indexing

occurs for this line. These requests will not necessarily happen in the order the predictions were made. As a result the history table is poor at handling patterns that change over time. When the intersection predictor encounters such a pattern it will likely fail to predict anything. However, when a high-sensitivity low-PVP predictor such as union encounters such a situation it will produce extra incorrect predictions.

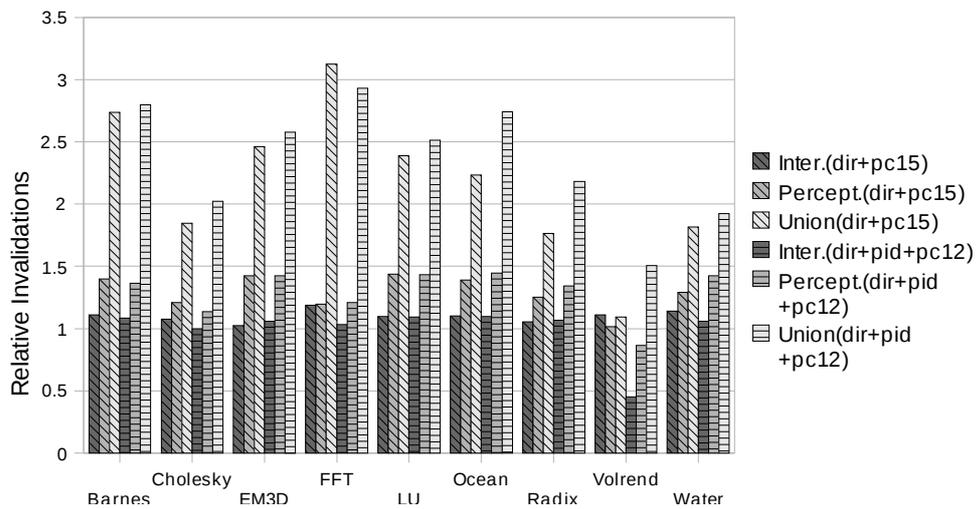


Figure 4.23: Invalidation Count of Mixed Writer-Instruction Indexing

From the change in request count shown in Figure 4.24 we can see that the increase in invalidations shown in Figure 4.23 has no correlation to decrease in requests. This means that despite the lower PVP a higher sensitivity was not achieved. The decrease in request count led to a decrease in runtime for both Cholesky and EM3D.

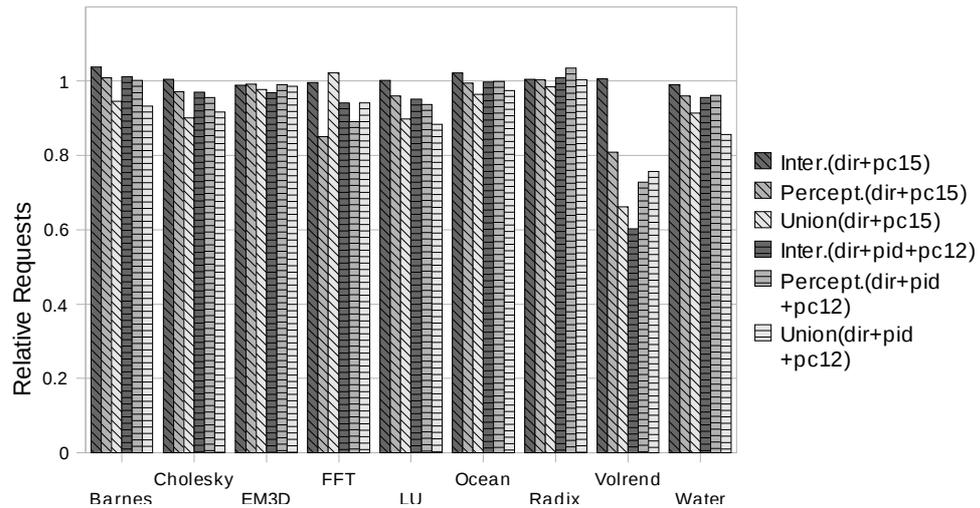


Figure 4.24: Request Count of Mixed Writer-Instruction Indexing

Finally, Figure 4.25 shows the relative miss rates for each of the benchmarks. As before, consumer prediction has performed properly and reduced the number of cache misses in the system even when speedups were not attained. Overall, including the CPU into the indexing scheme resulted in a slight decrease in the load miss rate. For the benchmark where this decrease was largest, Cholesky, there was a corresponding speedup.

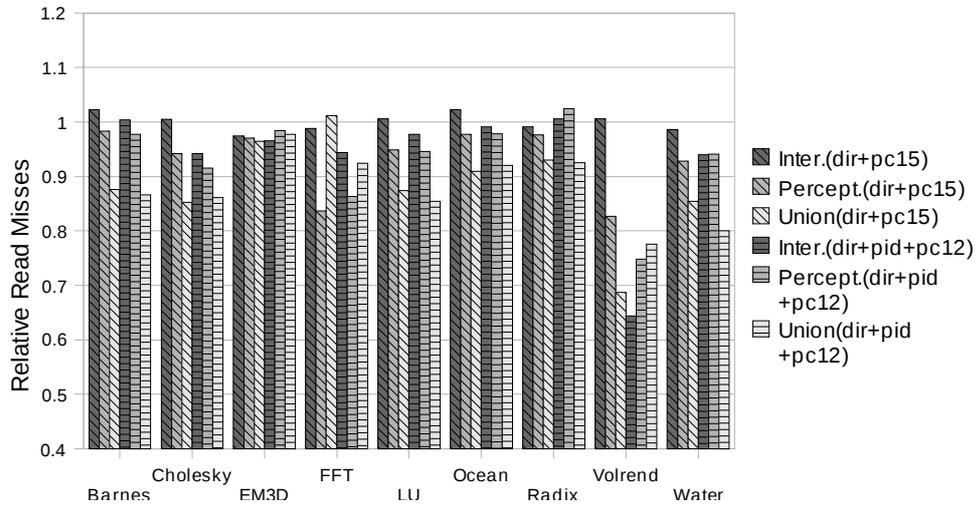


Figure 4.25: Read Miss Rate of Mixed Writer-Instruction Indexing

4.3.3.2 Address-Mixing

Address Mixing indexes the history table with a combination of address and instruction bits. With no collisions in the history table, the result of this will be that there is a separate table for each line that tracks the different instructions that acted on it. A major disadvantage with this method is that as fewer events are included in a single line of the history table the longer the training time will be.

Figure 4.26 shows the relative execution times of benchmarks running a mixed Address-Instruction predictor. As with the results shown in Figure 4.22 for a CPU indexed table, there is a loss of performance for those benchmarks where consumer prediction worked best. Unlike the Writer-Mixed table, the Address-Mixed table fails to remove the slowdowns in EM3d or Barnes, and does not achieve any speedups for Cholesky. What is particularly notable about this is that the perceptron predictor performs poorly for a number of benchmarks here, despite the fact that this indexing scheme was earlier identified as the best. Once real training times and

the feedback mechanism of speculation are included, simpler indexing schemes are superior for the perceptron.

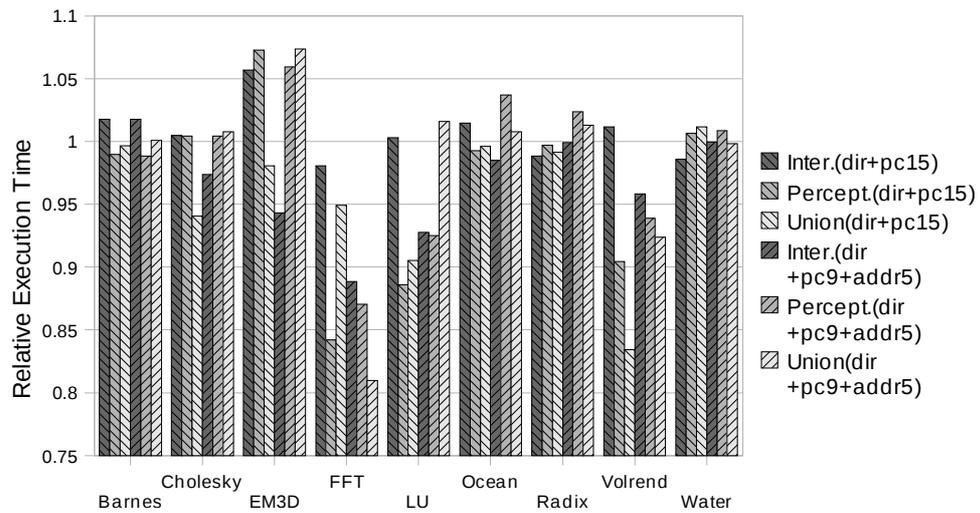


Figure 4.26: Runtime of Mixed Address-Instruction Indexing

4.3.4 Global Consumer Prediction

Another alternative predictor implementation considered is global prediction. This global predictor is similar to global predictors used for branch prediction. Rather than indexing a table to find an entry all events map to the same entry. A global predictor can take advantage of the fact that program execution occurs in phases. If there are specific sharing characteristics during one phase, they may apply to much of the data being traded during that phase. The disadvantage of a global scheme is that it is likely to produce more incorrect predictions. However, given that the penalty for an incorrect prediction is relatively small for data being shared among many processors, the possibility exists that there could be performance benefits.

Additionally, global prediction has advantages in terms of size and speed. The history table has only one entry, as opposed to thousands, and there is no table lookup latency. These two factors combine to make global prediction potentially attractive in terms of implementation considerations.

This work applies a global predictor to each of the three predictors studied, Intersection, Union, and Perceptron. The performance results are shown in Figure 4.27. In particular, global prediction can achieve small speedups on a number of benchmarks that address-indexed prediction had no affect on. The total speedup on those benchmarks where consumer prediction was most effective is lower, but still present.

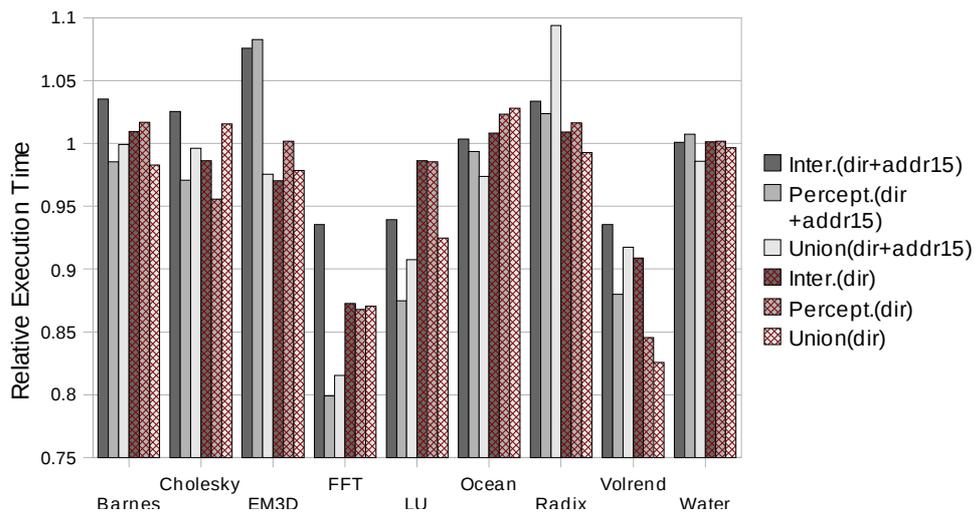


Figure 4.27: Runtime Benefits of Global Prediction

For the Volrend benchmark, Perceptron and Union prediction outperform their local counterparts. This follows the trend with Volrend, in which the important keys for speedups are getting falsely shared data back to their owners, and a small phase of wide sharing.

Figure 4.28 shows the relative number of invalidations transmitted. In general, this number is higher than that for the address based techniques. However, there are a few cases where the global predictor slightly outperforms the local predictor in terms of invalidation growth. In particular, the intersection predictor results in fewer invalidations on some of the benchmarks. The reason is that a globally indexed intersection will not predict any sharers during phases of when the sharers are nonconstant. These cases do not correspond to the cases where the global predictor outperforms the local because the misprediction penalty is relatively small in most cases, as many invalidations are performed in parallel.

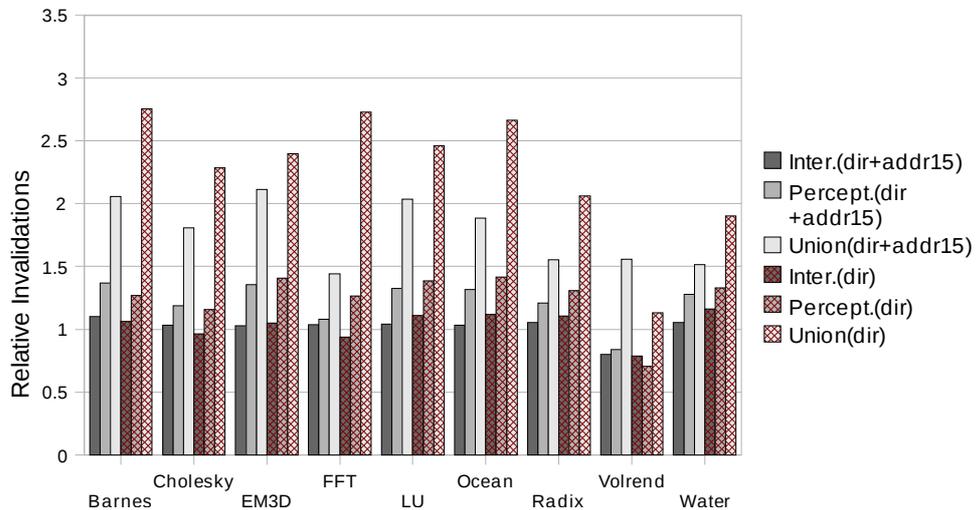


Figure 4.28: Global Prediction Invalidation Count Changes

4.3.5 Variance of Network Latency

This section studies the effect of changes in network latency on the benefits of consumer prediction. Ideally this information helps to provide information about the prediction algorithm that should be matched with each system. This information

can be used for system design choices, or even to provide adaptive behavior to the predictor.

4.3.5.1 Runtime Results

Figure 4.29 shows the relative execution time of an eight processor system using three forms of consumer prediction: $Intersection(dir + Address_{15})^4$, $Union(dir + Address_{15})^4$, and $Perceptron_{50}(dir + Address_{15})^4$. Program Counter information is left out of this study due to the potential cost of transmitting the additional information necessary to perform training and the issues with hysteresis in training order.

These benchmarks have been simulated at varying network latencies. The network latency given here is the amount of time a message needs to move between two nodes. In order for a request to be resolved, a message may need to pass across many nodes.

It is important to understand the effect that network latency has on the misprediction penalty and the benefits of correct prediction. A correct prediction has the potential to save a single thread two message latencies worth of execution time: a request for data and a response with those data. An incorrect prediction will usually cost an amount of time less than a single message in latency, as other invalidations may need to be sent. However, an incorrect prediction can cost more than a correct prediction if a line of migratory data has been given away to multiple sharers, when only the original thread would have wanted those data. In this case, invalidations

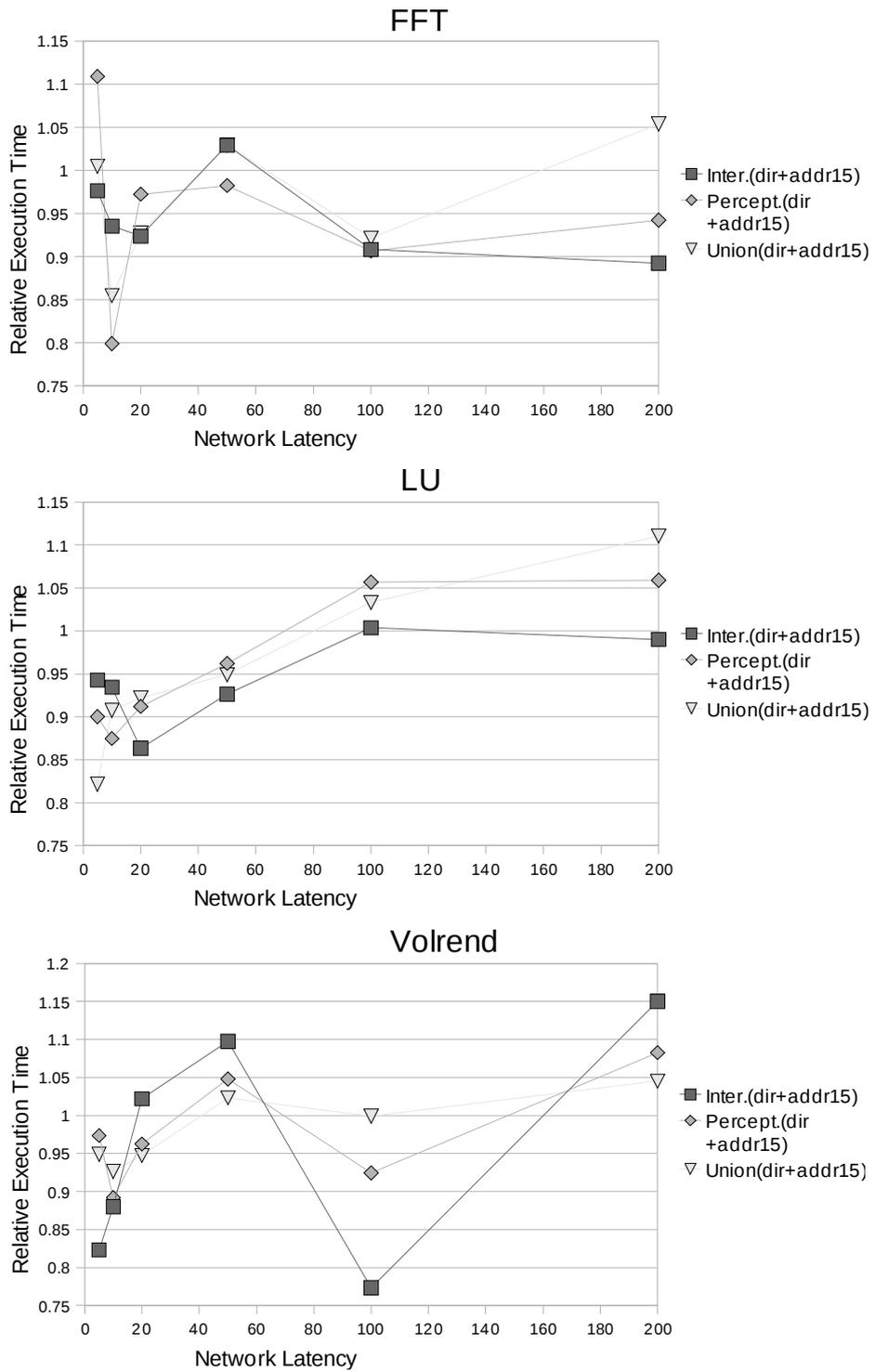


Figure 4.29: Relative Execution Time as Network Latency Increases. ($Intersection(Addr_{15})^4$, $Union(Addr_{15})^4$, $Perceptron_{50}(Addr_{15})^4$)

will need to be sent where none were needed before. This problem highlights the need to merge migratory prediction into any consumer predictor, a need that will be addressed in Chapters 5 and 7.

In general it can be expected that the penalties to any thread due to a misprediction will be smaller than the benefits due to a correct prediction. However, there are two additional factors that increase the danger of mispredictions. First, mispredictions use bandwidth. In particular, a correct prediction will save the bandwidth of a request message, whereas an incorrect prediction will use the bandwidth of a data carrying message and an invalidation. Second, speedups to any single thread are of relatively little value. Because the benchmarks studied here are based on barriers, a speedup must be attained on every thread or the time critical thread for the total system to experience a speedup.

Each of these benchmarks shows a slightly different behavior. However, in general, shorter network latencies result in better speedups and high PVP become more valuable as network latency increases. LU shows this trend most clearly. Both Volrend and FFT feature a similar pattern, with two distinct local minima. The reason is that the mechanism that is causing speedups changes as the network latency increases.

At low network latencies the speedup mechanism is moving data to their destination ahead of time. At high network latencies the speedup is due to the same effect that causes a decrease in the time it takes for the directory to provide write permission. Notice that for the second local minima in both, the time to achieve store permission drops. At the same time, more read requests begin to arrive when

speculative data are still in transit.

Thus, as network latency increases, the ability to provide timely forwarded data drops, while the penalty of being in a transient state when a write request arrives rises. Though both effects are present at all times, the first minima corresponds to benefits mainly from reduced read times and the second minima corresponds to benefits from reduced write times.

Each predictor considered has some benchmark and latency combination for which it outperforms the others. This emphasizes the need to synchronize the predictor design with the benchmark being executed and the system it is running on.

At high latencies the relative performance is clearer: the higher the PVP of the predictor the better the performance. However, this range is less interesting as in general speedups are either nonexistent, or smaller in this region.

4.3.5.2 Effect on Message Transmission

Figure 4.30 shows the relative number of invalidation messages transmitted for each of the benchmarks as network latency is varied. Figure 4.31 shows the relative number of request messages transmitted for each of the different benchmarks as network latency is varied. The expectation is that consumer prediction will cause the number of requests required to fall, and the number of invalidations to rise. Extra invalidations correspond to false positives, and reduced requests correspond to true positives. In general, high PVP predictors, such as Intersection predictors,

should result in fewer extra invalidates than low PVP predictors. High sensitivity predictors, such as Union, should result in a greater decrease in requests than low sensitivity predictors.

The number of extra invalidations needed is related to the PVP of the predictor. A low PVP, high sensitivity predictor, such as *Union*⁴ causes a greater increase in extra invalidations than a high PVP, low sensitivity predictor, such as *Intersection*⁴. The reasons that the number of invalidations falls so far on Volrend were discussed earlier. Here we can see that the trends they follow are the same two-minima trend as for runtime.

The number of fewer request messages needed is related to the sensitivity of the predictor. A high sensitivity, low PVP predictor, such as *Union*⁴, results in a greater reduction in total request messages sent than a low sensitivity, high PVP predictor, such as *Intersection*⁴. As with invalidations there is a brief range of network latencies in Volrend for which this rule is violated, and the pattern in which it is violated matches the pattern in invalidations.

The reason this trend is violated in Volrend is related to the reduction in total invalidations caused by the intersection predictor. Both the number of invalidations and the number of requests, fall due to a more efficient sharing pattern in which more instructions are executed before the status of a line changes.

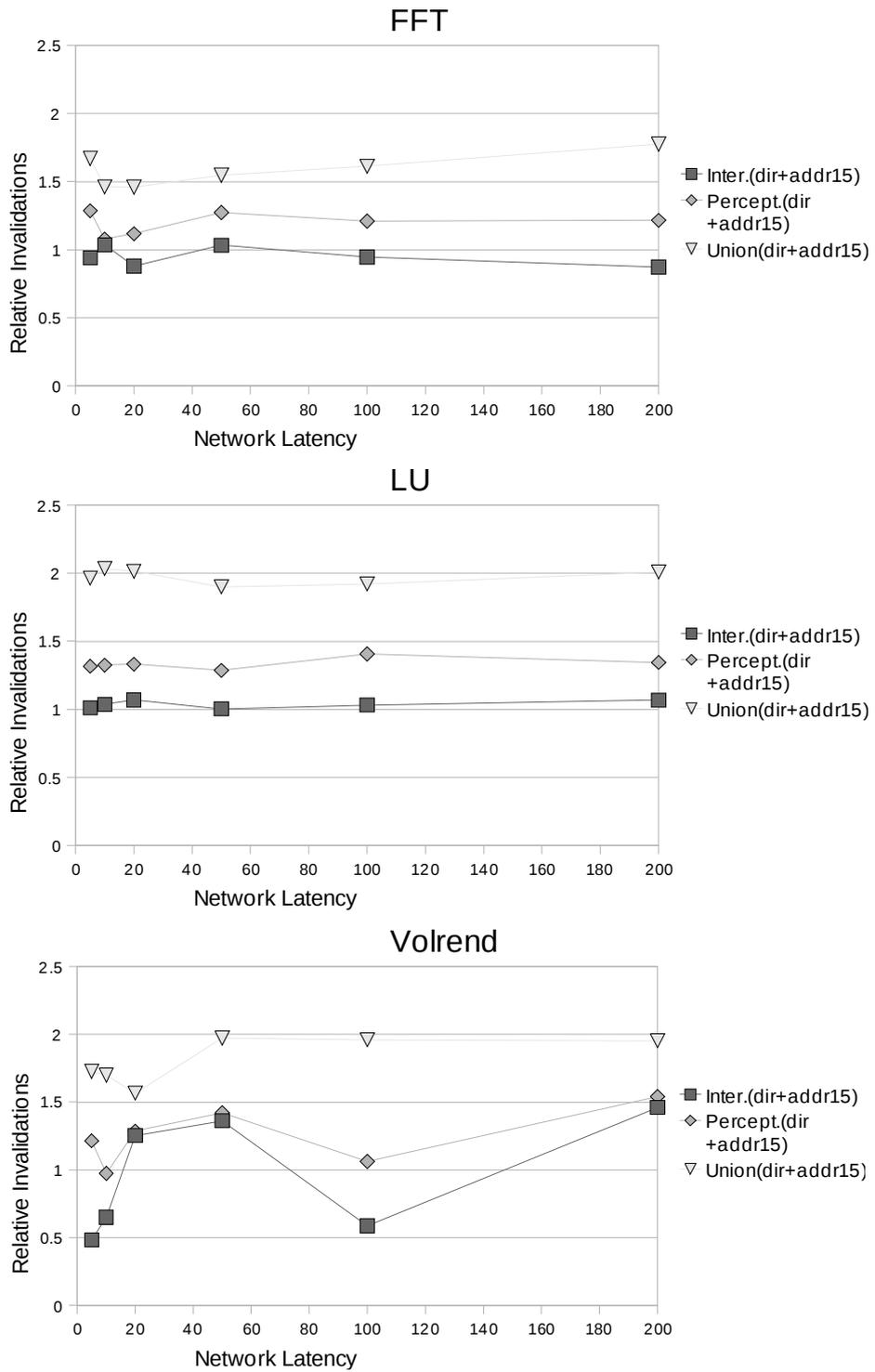


Figure 4.30: Relative Invalidation Messages Sent as Network Latency Increases. ($Intersection(Addr_{15})^4$, $Union(Addr_{15})^4$, $Perceptron_{50}(Addr_{15})^4$)

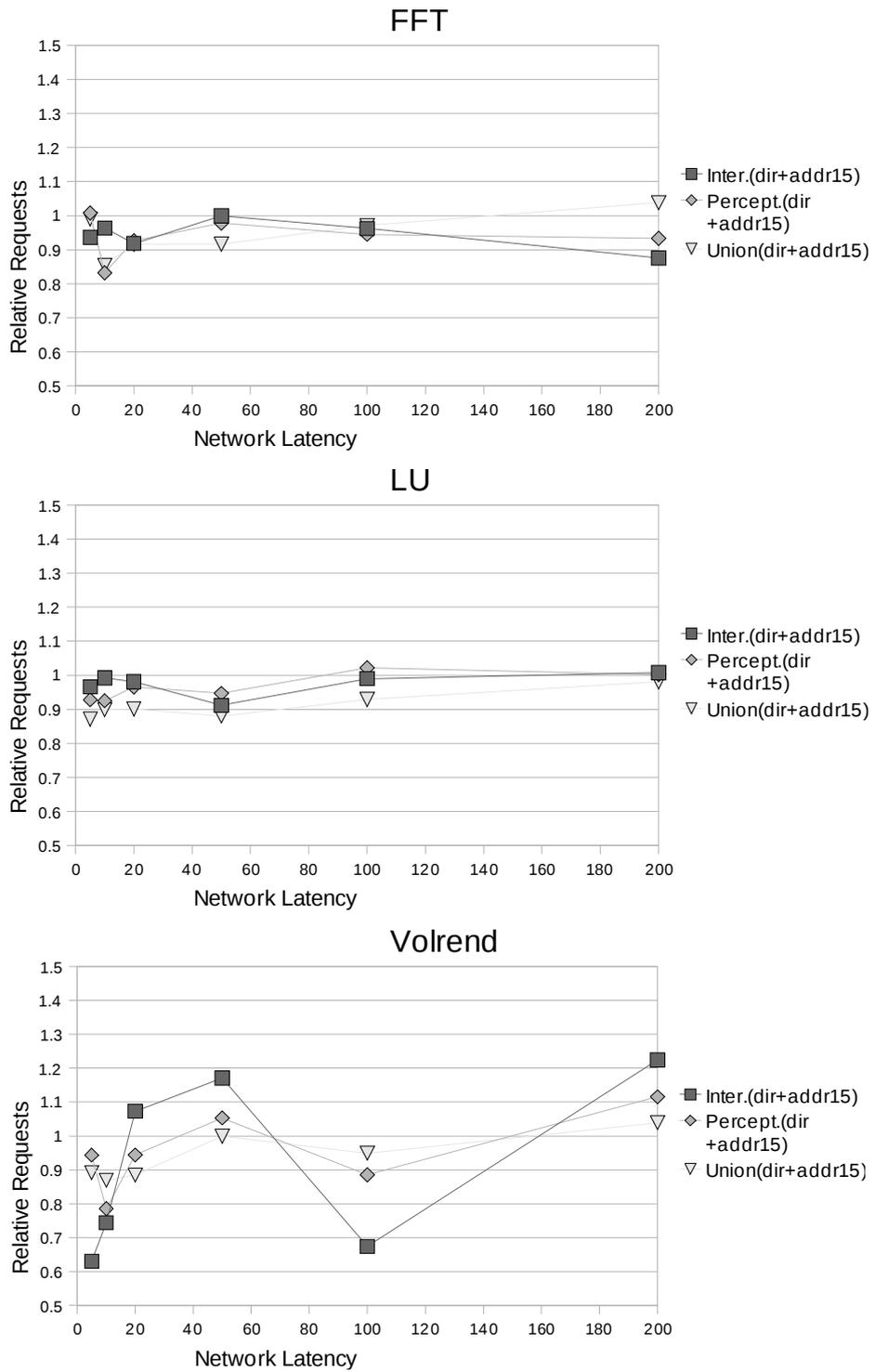


Figure 4.31: Relative Request Messages Sent as Network Latency Increases. ($Intersection(Addr_{15})^4$, $Union(Addr_{15})^4$, $Perceptron_{50}(Addr_{15})^4$)

4.3.5.3 Bandwidth Usage

Figure 4.32 shows the effect of changing network latency on the percentage of bandwidth consumed by the system. The expectation is that the highest PVP predictors may be able to reduce the total bandwidth used by avoiding the need for request messages. However, as data carrying messages are substantially larger than request messages, the required PVP may be very large.

Several features are apparent from these results. First, bandwidth use decreases as the communication time increases. This is because the simulator assumes that messages can be pipelined through the connections. That means that longer latency results in more empty stages. This is exacerbated by the in-order processors used, but would still be present in out-of-order processors.

Second, only the highest PVP predictor tested (*Intersection*⁴) results in a consistent reduction in bandwidth consumption. Notice that this reduction is consistently present, but shrinking, regardless of network latency. The reduction in benefit is to be expected as the total bandwidth used falls.

4.3.5.4 Directory Response Time

Figure 4.33 shows the change in the load latency of cache misses when consumer prediction is implemented and the network latency is varied. Figure 4.34 shows the change in the store latency of cache misses, when consumer prediction is implemented and the network latency is varied. It is expected that the latency of writes will increase slightly due to the need for more invalidation messages. Because

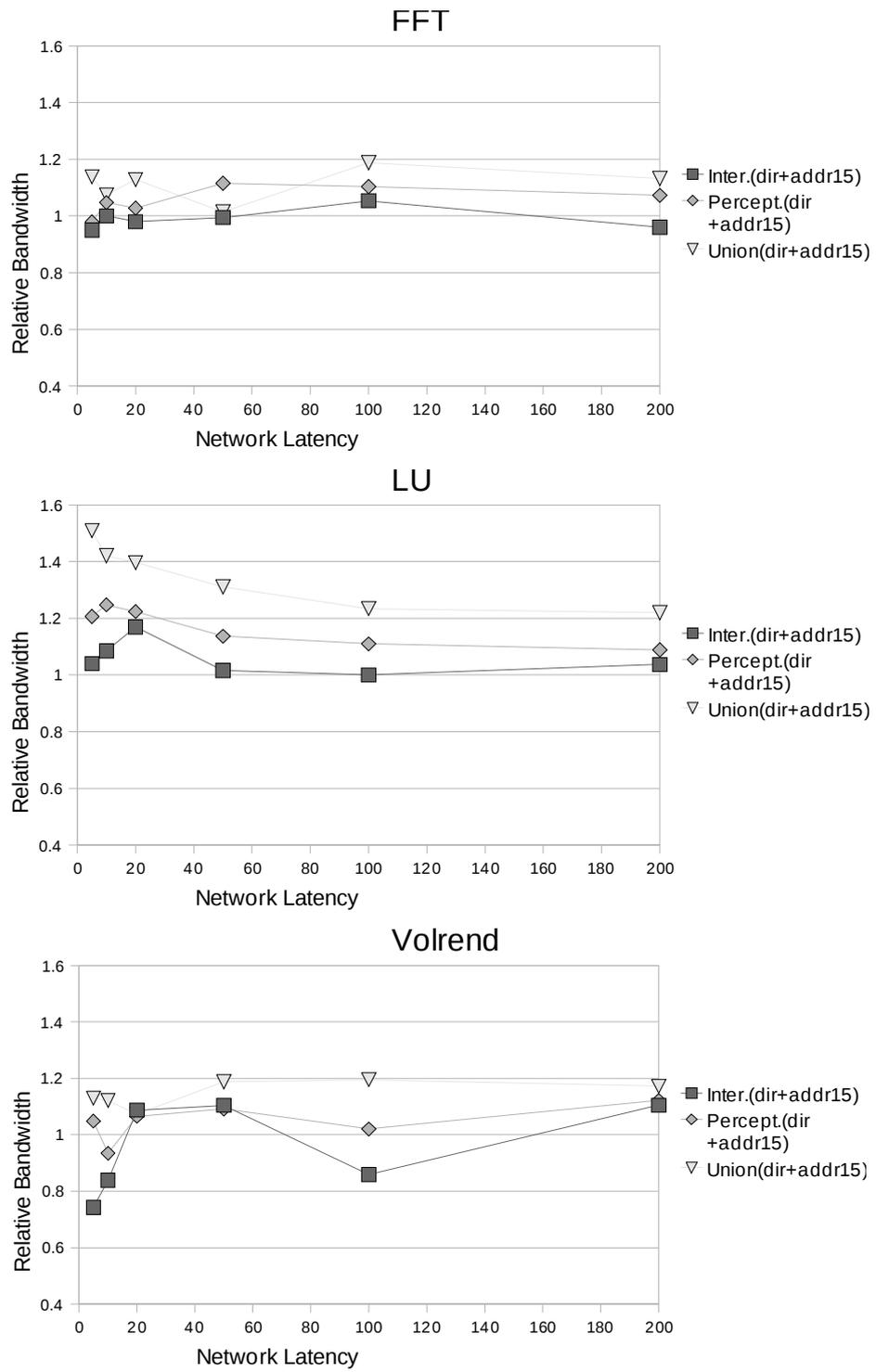


Figure 4.32: Bandwidth Use as Network Latency Increases. ($Intersection(Addr_{15})^4$, $Union(Addr_{15})^4$, $Perceptron_{50}(Addr_{15})^4$)

these extra invalidates are performed in parallel, there will only be a change when one turns out to be on the critical path. The latency of reads should be lower in those cases where the predictor made a correct prediction but the datum had not arrived by the time of the request.

It is clear from these results that many predictions do not arrive in time to avoid the transmission of a request. This explains the low reduction in the number of requests transmitted. It is also clear is that those benchmarks for which little performance enhancement was achieved still have a fair number of correct predictions, and a substantial reduction in read time. Nonetheless, these correct predictions are not sufficient to improve the speed of the benchmark.

Overall, there is little relationship between the read latencies shown here, and the speedup of the benchmark. This should not be surprising, as the best case of proper Consumer Prediction is not included in these figures. The only way to achieve a decrease in read permission is when a prediction fails to arrive in time to prevent any network latencies.

The overall trends shown in write latencies are not unexpected nor are they particularly interesting. The addition of extra sharers should have small effect on the total latency of requests for exclusive permission when other sharers are already present in the system. Consumer prediction slightly increases the overall write latency, with little real variation between schemes.

What is interesting about these data is that there is a clear relationship between the outliers in the invalidation messages sent and the total write latency. When the number of invalidation messages sent falls due to consumer prediction,

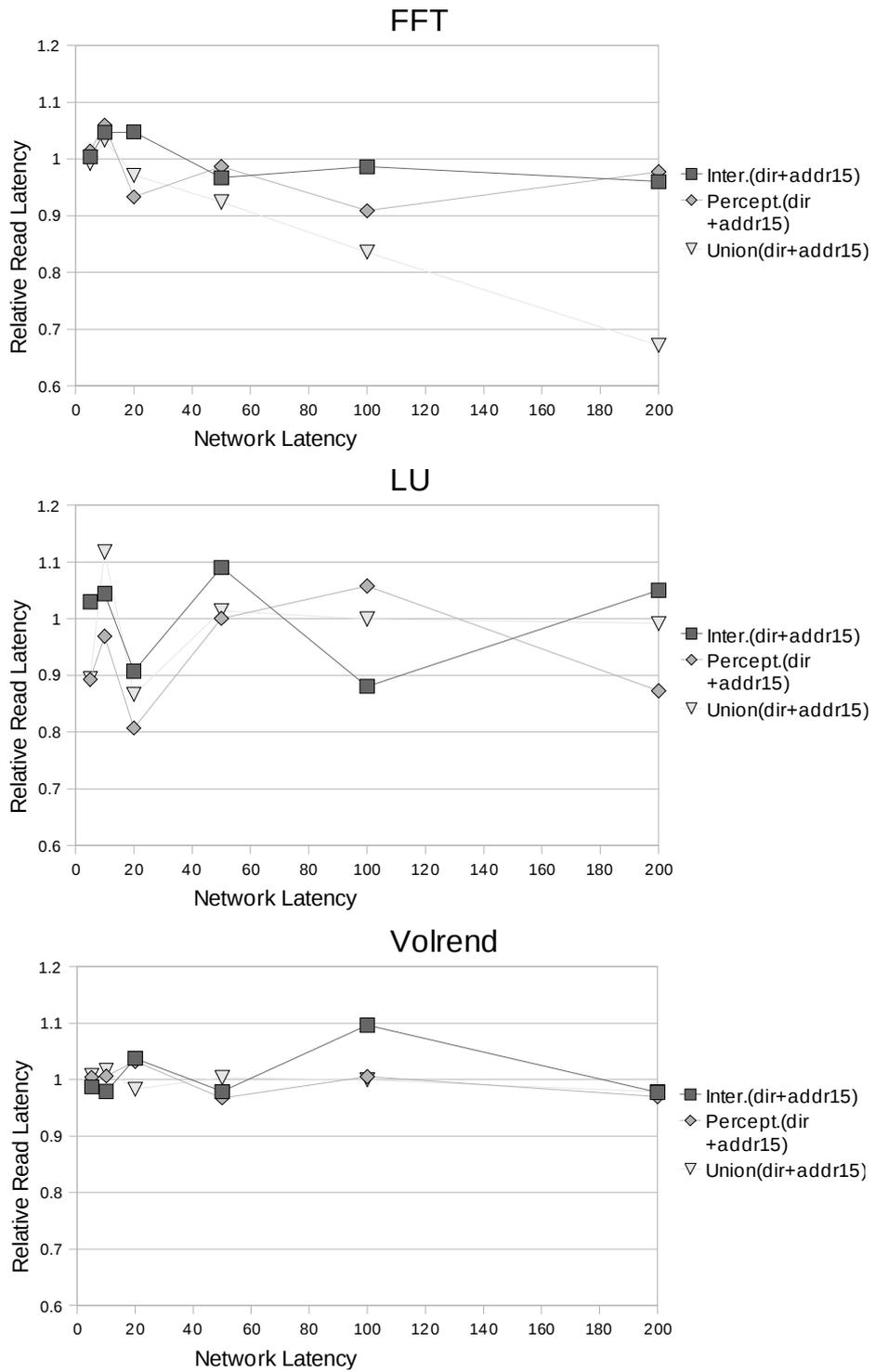


Figure 4.33: Latency of Read Requests to the Directory as Network Latency Increases. ($Intersection(Addr_{15})^4$, $Union(Addr_{15})^4$, $Perceptron_{50}(Addr_{15})^4$)

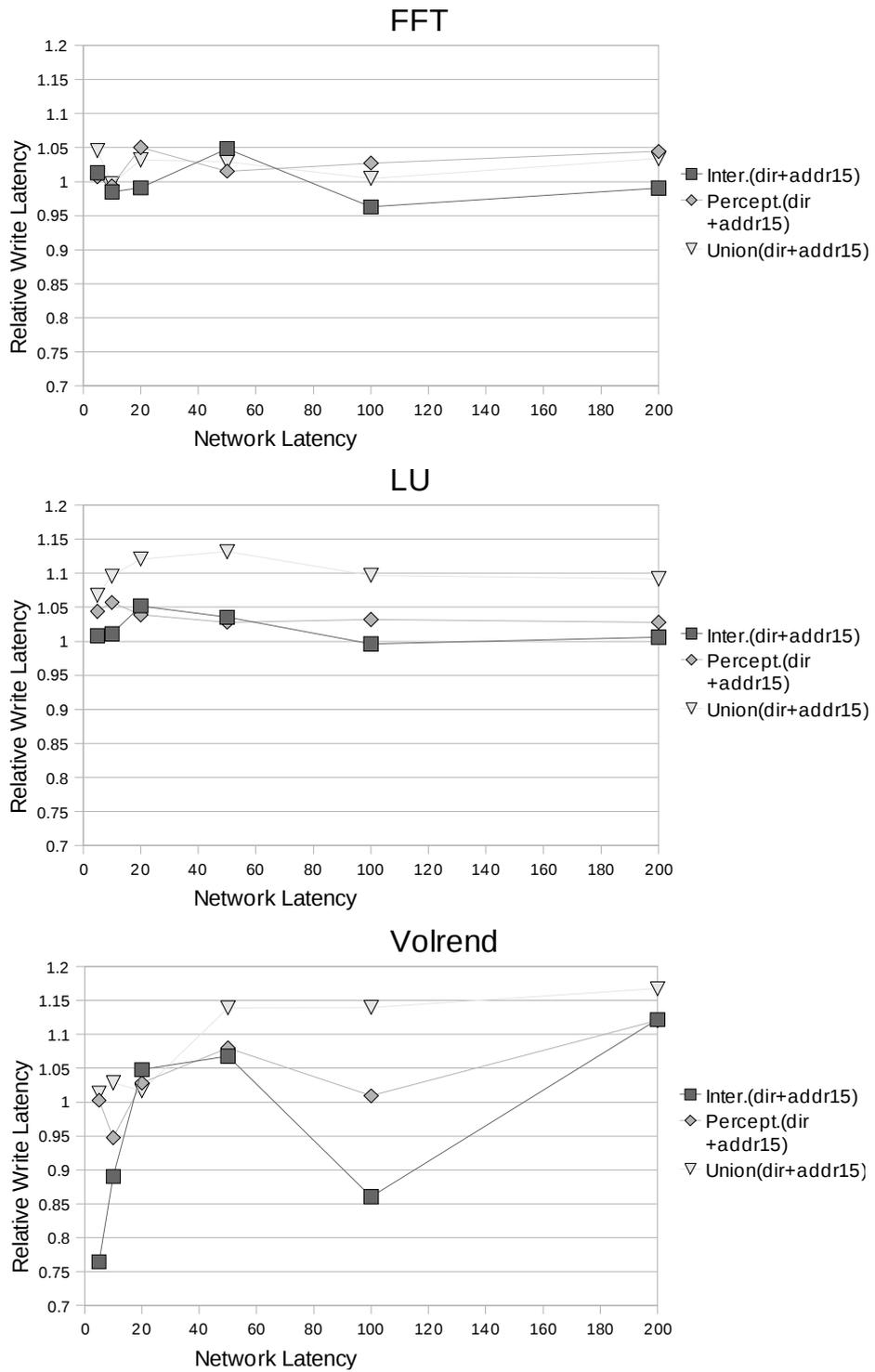


Figure 4.34: Latency of Write Requests to the Directory as Network Latency Increases. ($Intersection(Addr_{15})^4$, $Union(Addr_{15})^4$, $Perceptron_{50}(Addr_{15})^4$)

the write latency also falls substantially. As stated earlier, this effect is because timely consumer prediction can reduce the likelihood that a line is in a transient state when a request arrives.

4.3.6 Relationship Between Benchmark and Predictor Performance

In Figure 4.14 it is clear that some benchmarks benefit much more from consumer prediction than others. Figures 4.16 and 4.20 demonstrate part of the reason for this decrease in performance: late predictions. However, they do not address the reason that predictions are less timely on these benchmarks than the others.

This decrease in timeliness is related to the nature of the three benchmarks. Each of these benchmarks features a phased operation. In the case of FFT, each processor works on a block of data that no other processor reads. Once all processors complete a phase, those data are swapped and the process continues. In the case of LU, blocks are assigned to processors as needed. Once a block is completed, it is stored, and then read by other processors. In the case of Volrend, processing takes place in phases, with all processors completing a portion of the computation, then synchronizing to use those data elsewhere.

Of the remaining benchmarks, only Radix and EM3D could be described to share this behavior. However, the sharing pattern of Radix is determined by those data being sorted. As these data are random, the sharing pattern is also random.

Em3d is restricted for a different reason. In Em3d each node is linked to a certain number of other nodes, three in the case of the results included. Only the

owners of these three nodes will ever read the results of a given node. Only a few nodes are linked to nodes owned by another processor. This means that in general most lines are not shared. Of the shared lines, only a few are shared with more than one other processor. However, this implementation of consumer prediction can only predict when a line is requested. Thus, even though the sharing pattern is highly predictable, by the time a prediction is made there will be no other processors to predict as sharers.

4.4 Tuning the Perceptron

In the previous section a number of ways in which a perceptron can be further tuned were discussed. This section examines the actual results of these changes to the perceptron. In addition, this section discusses the possibility of teaching a perceptron to recognize inputs that are not yet present.

4.4.1 Varying the Perceptron Threshold

One key aspect of Perceptron-based prediction identified earlier was the ability to tune the performance using the threshold value. This criteria had the potential to provide a wide range of different tradeoffs in terms of PVP and sensitivity. The results of this variation are shown in Figure 4.35. They show no clear trend between perceptron threshold and system performance.

If the actual sensitivity was decreasing as the threshold increased, there should be a corresponding decrease in request messages to the system. However, from

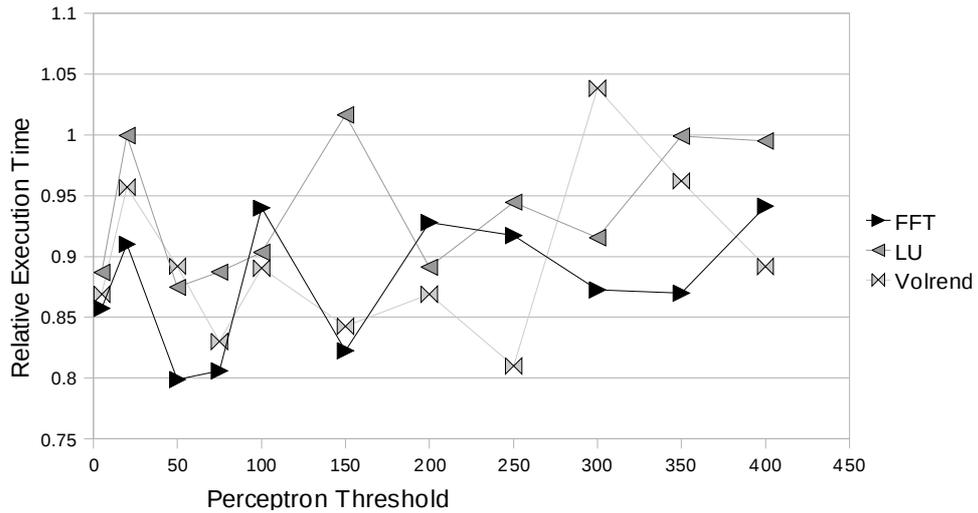


Figure 4.35: Effects of Varying Perceptron Threshold on Runtime

Figure 4.36 it can be seen that no clear trends exist. Similarly, if the actual PVP was increasing as the threshold increased, there should be a corresponding decrease in the number of invalidation messages sent. Figure 4.37 shows that this decrease is present.

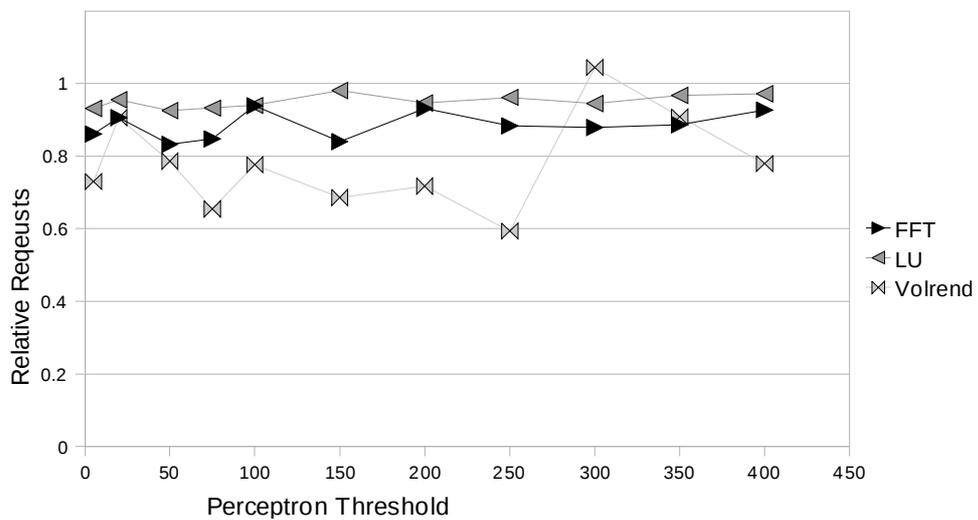


Figure 4.36: Effects of Varying Perceptron Threshold on Request Count

These results show that the amount of variation available in sensitivity and PVP by tuning the threshold of the perceptron when combined with the potential

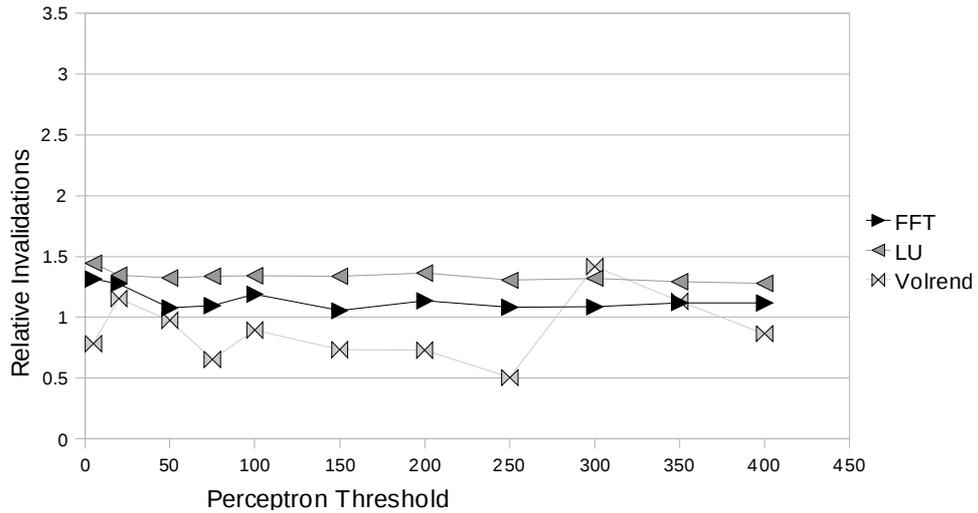


Figure 4.37: Effects of Varying Perceptron Threshold Invalidation Count

for incorrect ordering in training, is insignificant compared to the “noise floor” of execution in a multi-processor system.

4.4.2 Training a Perceptron With Non-Existent Data

One fundamental difference between this perceptron application and others is the presence of unused inputs. When the system is initialized it first trains with only a single history entry of data, regardless of the depth of the history table. The remaining history entries are fed into the perceptron in their initialized values of zero. These entries are “Non-Data”. They are not real training data for the perceptron.

One simple approach is to only use and train the perceptron weights corresponding to data that are actually present. The perceptron can recognize the “Non-Data” because it corresponds to sharing phases in which no processor had access. The implementation for this would be relatively simple, with the count being

processed in parallel to early additions in the tree and then gating them out before the weights corresponding to different history depths are combined.

The results of this are shown in Figure 4.38. From this it is clear that this approach is not helpful. While a few small slowdowns are reduced, the speedups on Volrend and FFT are lost. This leads to the conclusion that these deeper history bits are serving a different purpose.

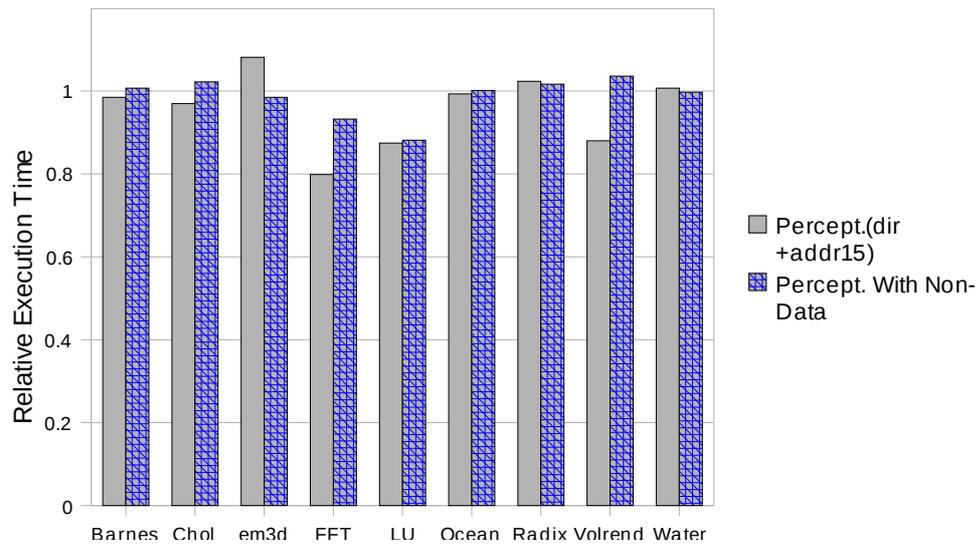


Figure 4.38: Benefits of Perceptron Recognizing "Non-Data"

4.5 Applying Confidence Estimation to Consumer Prediction

4.5.1 Supporting A Confidence Estimator

Historically, confidence estimators have not been attached to consumer predictors. Because consumer predictors were studied in terms of sensitivity and PVP, changes in the indexing scheme and depth of the history table were expected to provide the benefits of a confidence estimator. While such changes are possible, they

may unnecessarily restrict the performance of many predictions in order to make up for poor performance on a small number of predictions that a confidence estimator can identify.

The confidence estimator used in this section is similar to the ones proposed for branch prediction [3] and value prediction [5]. It contains a table of confidences, in most cases indexed by the same quantity as the original consumer predictor history table. For each index, this table contains a two bit confidence associated with every processor's history information. These confidences are updated as an up-down saturating counter.

Whenever a confidence is below the threshold, any predictions of forwarding data to that processor are overridden. Thus this system can transform True Positives into False Negatives, and False Positives into True Negatives. This will likely decrease the sensitivity of the system and increase its PVP. However, it will do it without applying the same strict standards to more predictable lines as it does to less.

This confidence estimation acts as a way to increase the PVP of a system without increasing the history depth. As previous results have shown, increases in history depth have diminishing returns. Moreover, as history depth grows the chance of out-of-order updates increases. Confidence estimation can provide a way to increase this sensitivity without increasing the effect of potentially out-of-date information.

4.5.2 Performance of Confidence Estimated Address-Indexed Consumer Prediction

Figure 4.39 compares the relative execution time of each predictor before and after confidence estimation is applied. These results are clearest for the intersection predictor. The reason is that the additional PVP provided by confidence estimation matches the speedup mechanism that Intersection prediction benefits from.

The performance of the system with a Perceptron predictor and a Union predictor have changes distributed evenly between positive and negative. In general, the performance improves when slow-downs had been present, and declines where speed-ups had been present. This is because these two predictors had lower PVPs. The benefits they received were not caused by transmitting highly confident lines. The penalties they received were caused by transmitting without sufficient certainty.

Most importantly, with confidence estimation in place an address-based intersection predictor is able to consistently provide some performance benefit to all of the benchmarks except for Radix, whose sharing is substantially random from the perspective of an address-indexed table.

4.6 Summary

This chapter proposed the perceptron consumer predictor, a novel consumer predictor that can provide a tradeoff in sensitivity and PVP when compared to previous techniques. The threshold of the perceptron can be adjusted in order to adjust this tradeoff.

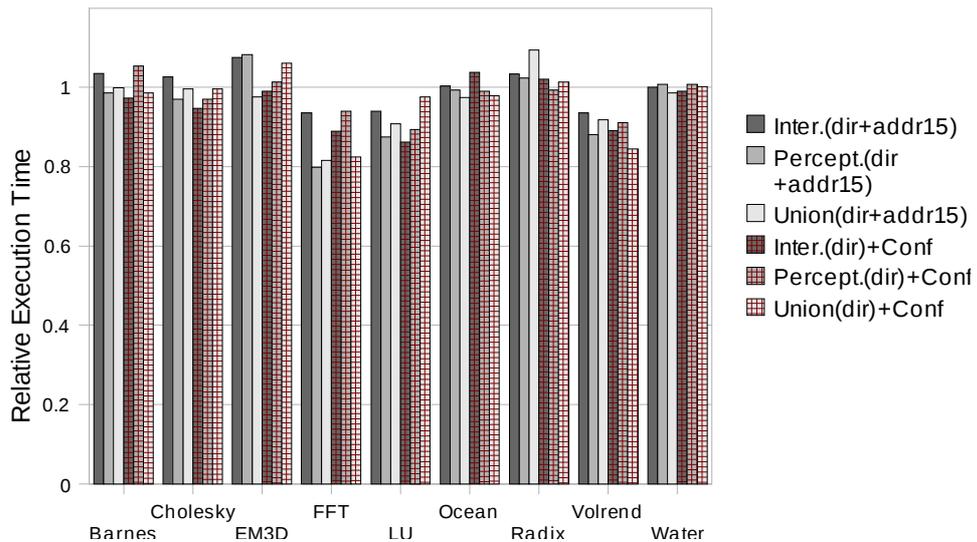


Figure 4.39: Effects of Applying Confidence Estimation to the Performance of Consumer Prediction.

A coherence protocol that can take advantage of a consumer predictor was developed. This protocol places a consumer predictor at each directory, and adjusts the state machine of the cache to handle the possibility of unexpected data arriving. This coherence protocol can provide speed-ups to benchmarks that regularly share lines among more than two processors in a predictable way.

Finally, this chapter analyzed the effects of this protocol as the latency of the interconnect between processors changes. The desired tradeoff in PVP and sensitivity is related to the latency of the network, with a high sensitivity being beneficial for lower latencies, and with a high PVP being beneficial for higher latencies. The exact tradeoff is dependent on both the latency of the network and the benchmark being executed.

Chapter 5

Migratory Prediction

Migratory data are data that are only “owned” by a single processor at a time. The data are read and written by one processor for one phase of execution, before another processor takes it to read and write. While at some time any processor in the system may touch it, any time it is read by a processor, it is then written by that same processor before another processor reads it. One example of a situation that produces this pattern is a locked structure. The lock prevents multiple processors from handling these data at one time.

Protocols to handle migratory data represent one of the earliest forms of speculation in the coherence system. These techniques typically perform speculation by adding states to the coherence protocol itself. While this approach is effective, it complicates verification, and makes it difficult to correctly implement other speculative techniques. Further, it can interfere with consumer prediction by modifying the sharer pattern. Similarly, failing to identify migratory sharing patterns can cause the additional transients necessary in a consumer prediction protocol to cause unnecessary slowdowns, by preventing requests for a Modify copy until the transients are resolved.

This chapter addresses this issue with a Consumer Predictor-based Migratory Predictor. This scheme takes advantage of the pre-existing data stored by the

consumer predictor detailed in Chapter 4 to identify migratory data. The objective here is not necessarily to outperform previous migratory predictors. Rather, the objective is to produce a migratory predictor that can be easily integrated consumer prediction without consuming additional storage space for memory structures. The data presented includes the results from Cox's migratory sharing predictor [7].

The end goal of this study of migratory prediction is to produce a scheme that can handle well both consumer prediction and migratory prediction. This section first studies migratory predictor designs that will lend themselves to being combined with consumer prediction. Later sections address the actual effects and benefits of combining the two.

5.1 Migratory Consumer Predictor Architecture

5.1.1 Predictor Architecture

Migratory data are characterized by a pattern of ownership in which only a single cache has control at any given time. The consumer predictor presented in previous chapters maintains a history table of previous sharers, addressed by either cache line or program counter information. Identifying migratory lines can be performed using an extra function that processes the same data as the consumer predictor.

This predictor identifies migratory lines by searching for the pattern they leave in the history table. When the history table indicates that a line has been possessed by only a single sharer in each phase for which it contains data, it is marked as

migratory.

Whenever a request for a Shared copy arrives, the history table is queried and if the predictor identifies the access as migratory, it is upgraded to a request for a Modify copy. This request is handled exactly as a request for a Modify copy would have been. Because the cache requested a Shared copy, there is no danger that the new copy will not fit in the cache, as was the case with consumer prediction.

Figure 5.1 shows this process in action on migratory data. The pattern shown is that of the accesses to a locked counter, such as the ones used by Volrend to distribute tasks among the processors. Each processor reads the counter, which acts as a queue, to retrieve their task, and then writes it to update the tasks left.

When the first processor requests access there is no history. It is given a Shared copy and quickly requests a Modify copy. Another processor requests a shared copy. The history table only has a single entry, and so it does not predict a modify. It then quickly requests a Modify copy. The third processor requests a shared copy. This time the system identifies the line as migratory. The second processor is invalidated and a Modify copy is sent to the third. When the third processor executes a store there is no delay.

As with Consumer Prediction, this scheme has the danger of “lock-in,” where a prediction is self-fulfilling. If a line is identified as migratory, the next time it is accessed, a Modify copy will be sent. Thus the behavior will be that of migratory even when the line was not written.

Fixing this issue is more complex than with consumer prediction, as the history table updates whenever Modify permission is granted. In other words, the history

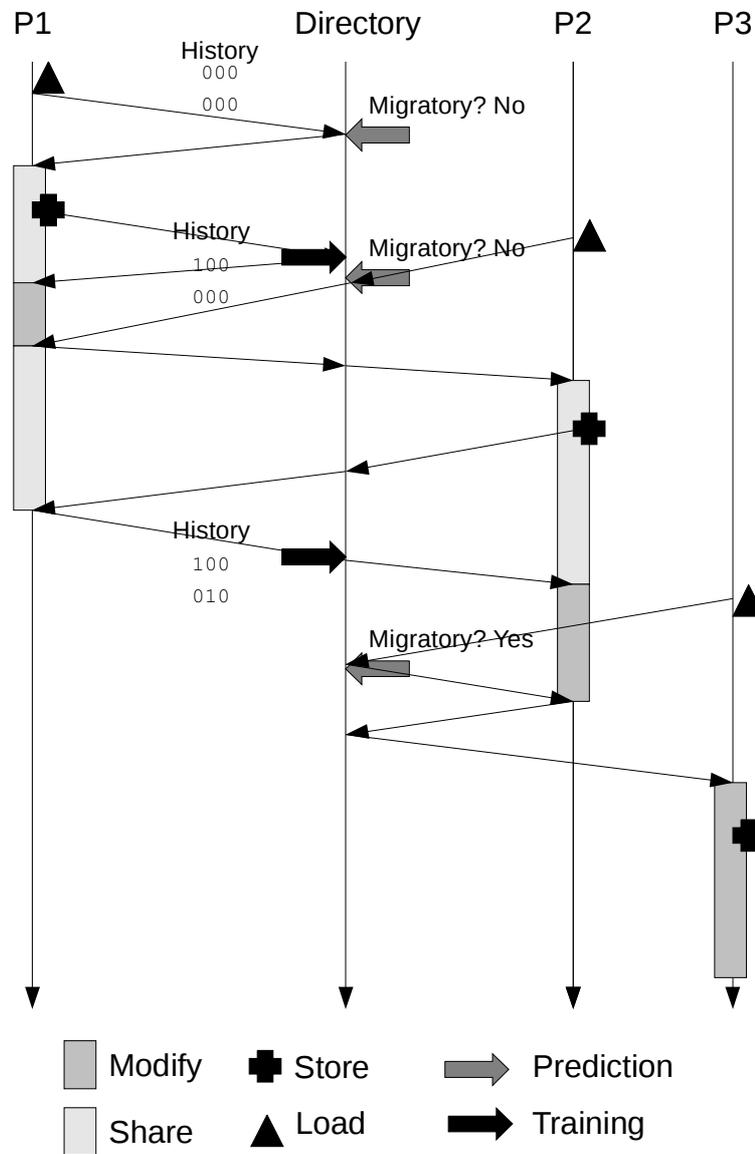


Figure 5.1: Example of the Behavior of Union Prediction on the distribution of a task queue.

table has already been updated when the error is detected. The results first shown in this chapter show this “lock-in” phenomenon. Once a line has been identified as migratory, it will always be identified as migratory. Later, Section 5.2.2 discusses alternative mechanisms to prevent “lock-in,”

When the history table is indexed using any form of instruction information the process of migratory prediction at the directory becomes more complex. When the request arrives, the position in the table to index is not necessarily known. Several options to handle this additional difficulty are discussed later in this chapter.

5.1.2 Protocol Modification

The coherence protocol developed in the previous chapter was modified to support migratory prediction. This requires the addition of a single event corresponding to a request for read access that has been predicted as migratory. When this happens the protocol responds to the request for a Shared copy with a Modify copy. In order to implement this, the cache must be adjusted to handle the possibility of these data arriving, and the directory must be changed to handle both normal requests for a Shared copy, but also requests for a Shared copy that have been speculatively selected for upgrade. The changes are shown in Figures 5.2, 5.3, 5.4, and 5.5.

<i>Event Name</i>	<i>Description</i>
GETS Spec Up	A cache has requested a Shared Copy of a predicted Migratory Line.

Figure 5.2: Additional Directory Events Necessary To Support Migratory Prediction in a Consumer Predictor

<i>Event Name</i>	<i>Description</i>
DataSpec	A speculatively transmitted Shared Copy has arrived

Figure 5.3: Additional Cache Events Necessary To Support Migratory Prediction in a Consumer Predictor

5.2 Address Based Migratory Consumer Predictors

This section focusses on migratory predictors based on a history table indexed by address. If this follows the trends seen in Chapter 4, address based predictors will perform well for a smaller set of benchmarks, while having no effect on, or slightly slowing, others. Section 5.3 discusses a migratory predictor based on an instruction indexed table.

5.2.1 Effects of History Depth on Migratory Prediction

Just as in Consumer Prediction, the choice of history depth represents a trade-off between PVP and sensitivity. A long history will result in fewer lines being identified as migratory, but also fewer lines being incorrectly identified as migratory. A short history will quickly identify migratory lines, but will also incorrectly label more lines.

Figure 5.6 shows the effects of a changing history depth on the history-table

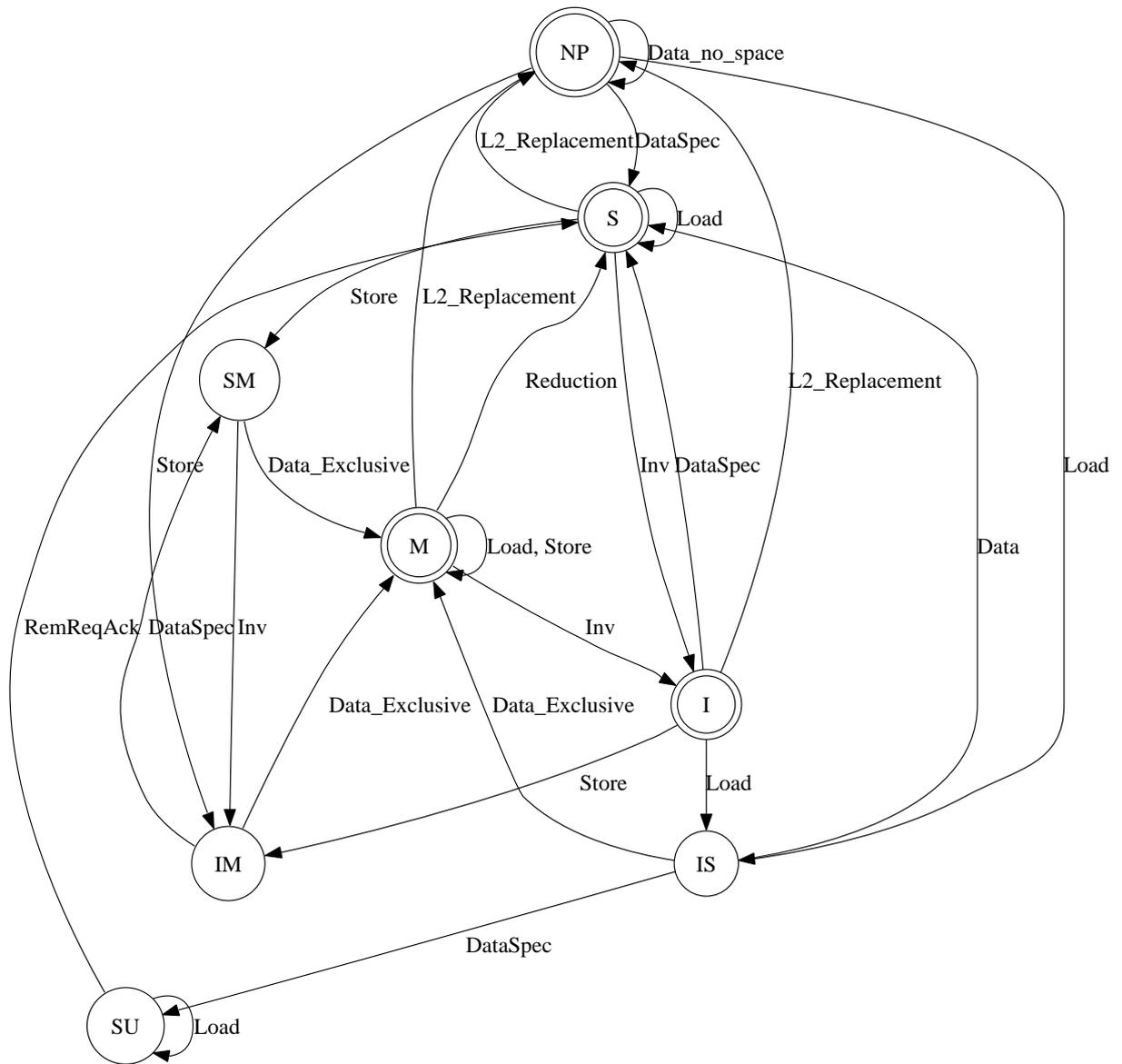


Figure 5.4: State Machine Representation of a Cache Supporting Migratory Prediction in a Consumer Predictor.

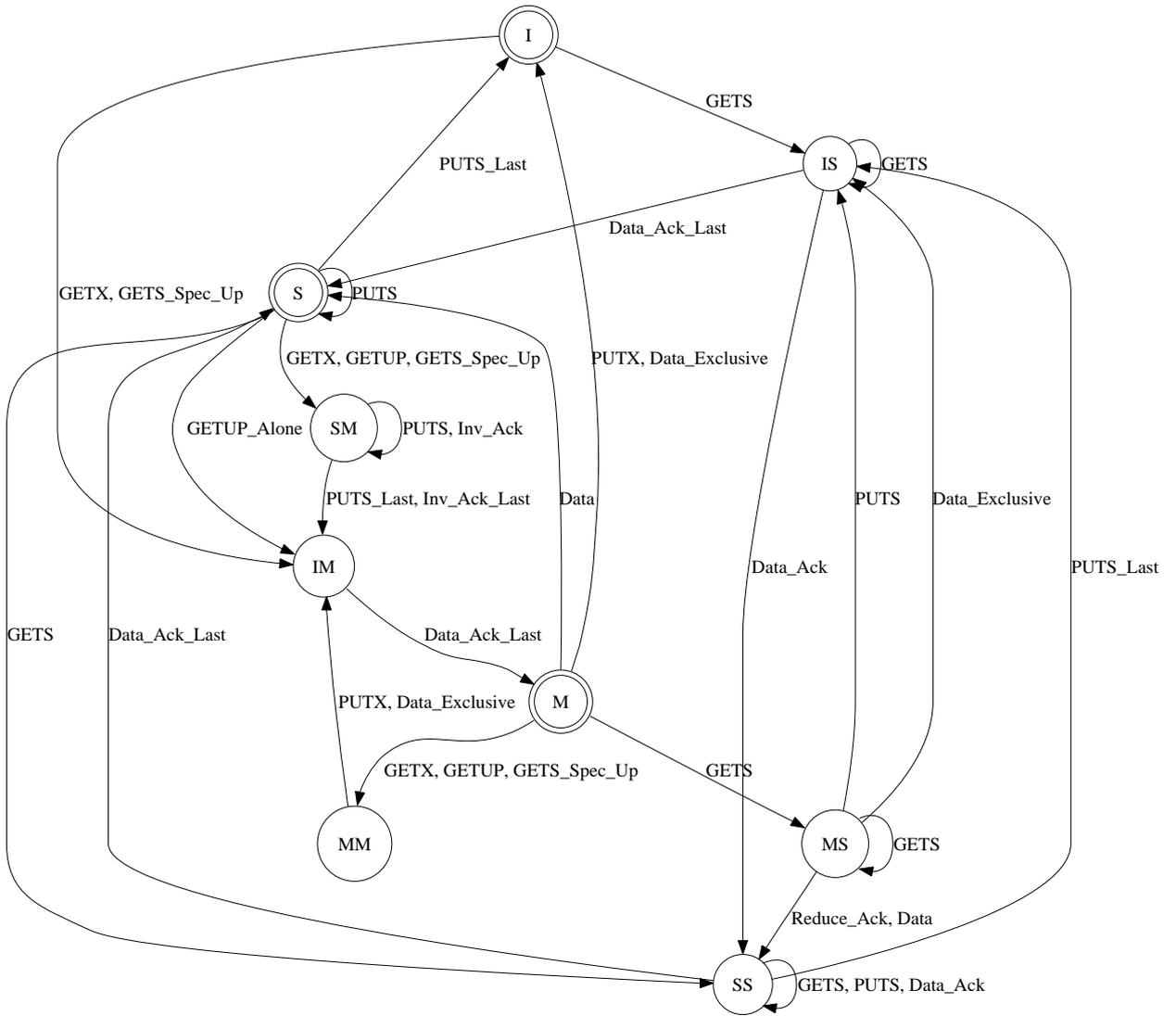


Figure 5.5: State Machine Representation of a Directory Supporting Migratory Prediction in a Consumer Predictor.

based migratory predictor. This graph illustrates the tradeoffs in design. A short history length can offer the greatest benefits in terms of true positives, but also has the greatest negative effects in terms of false positives. A long history offers substantially reduced benefits, but also removes the large penalties from some benchmarks. As you can see, in those few cases where migratory prediction is effective, the history table approach provides a slightly higher speedup than that provided by the original migratory predictor. In those cases where address information is less effective, lock-in causes slowdowns to occur.

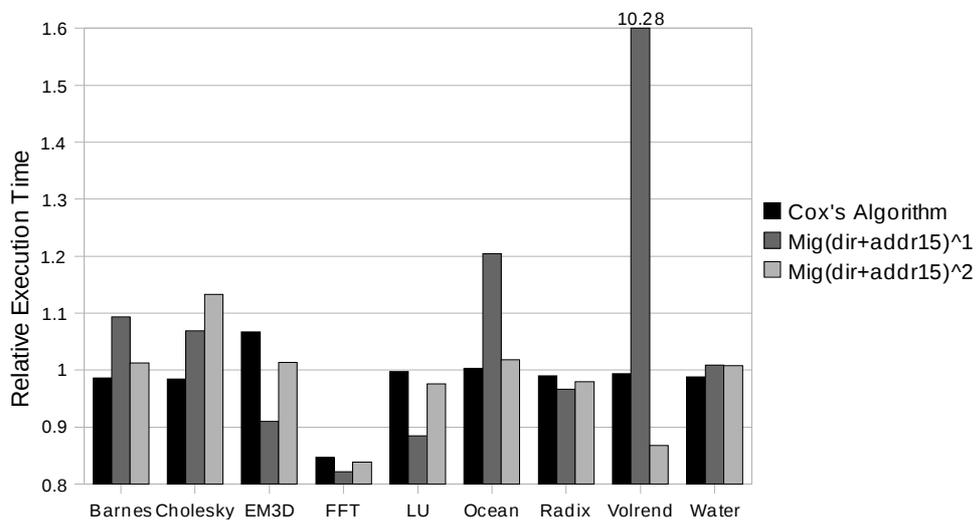


Figure 5.6: Execution Time Benefit of Address-Indexed Migratory Prediction as History Depth Varies

Volrend stands out in this example because it is a case of strong “lock-in.” Volrend is affected in this way because early in execution a few key pieces of widely-shared data are identified as migratory. The result is that all eight threads are constantly receiving Modify copies of a few lines that they will not write. Further, all the other threads will request access to the line shortly. This is a key example of the need for “lock-in” prevention mechanisms. Such a mechanism could enable

these speed-ups without the slowdowns on other benchmarks.

Figure 5.7 shows the effect of “lock-in”. On Volrend, the total number of requests raises by eighteen times. The other slowdowns on Barnes and Cholesky also show similar behavior, although not as drastic as that of Volrend.

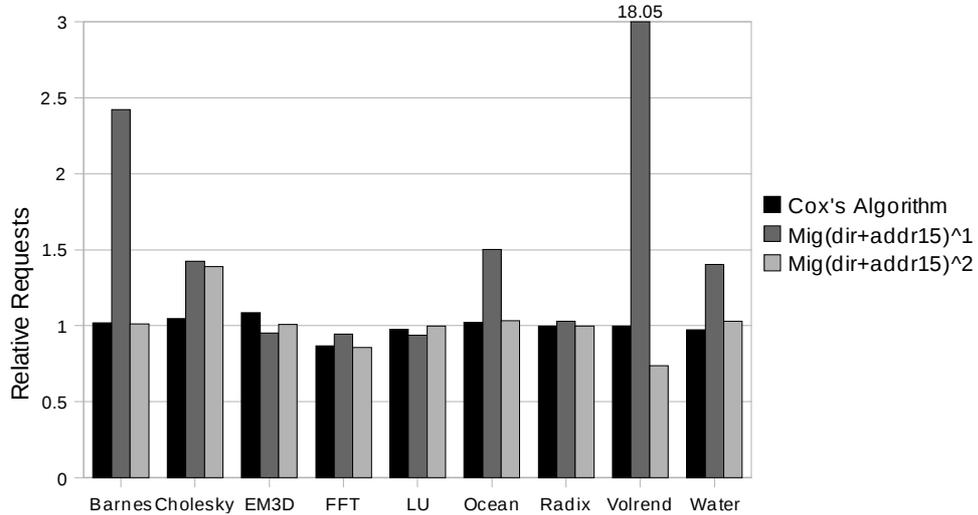


Figure 5.7: Change in Request Count With Address-Indexed Migratory Prediction

Additionally, it is clear that in those cases where migratory prediction can be beneficial, a history table approach outperforms the traditional state-based approach. A longer history depth reduces this advantage, but does not eliminate it.

5.2.2 Preventing Lock-In in Migratory Prediction

It is clear from Figure 5.6 that some scheme is necessary to remove the hazard of lock-in. This section discusses three schemes to address this issue.

- **State Additions:** This scheme is intended to represent the traditional method of coherence prediction. Additional states are added to the directory to identify incorrect migratory predictions.

- History Table Corruption: This scheme inserts incorrect data into the history table to escape from incorrect migratory predictions.
- Not-Migratory Counters: This scheme adds a small amount of memory to the history table to identify incorrect migratory predictions.

5.2.2.1 State Additions

In the state additions approach, lock-in is prevented by adding states to the directory protocol. When a request for a Read Copy is upgraded by the prediction logic, the line enters a new state, rather than treating the the message as a request for a Modify Copy as was done previously. The line then waits in the new state until the current modifier responds. If the responder has not modified the line, it is assumed that the modify prediction was incorrect, and only a Read copy is sent. Otherwise a modify copy is sent. In either case the current owner is invalidated, not reduced to a Shared copy. The new state machine is shown in Figure 5.8.

This method has the advantage of being direct. It does not require altering the behavior of the history table in any way, as the other methods do.

The disadvantage of this method is that it requires adding new states to the protocol. These new states will increase the difficulty of verification and potentially interfere with other optimizations or design choices.

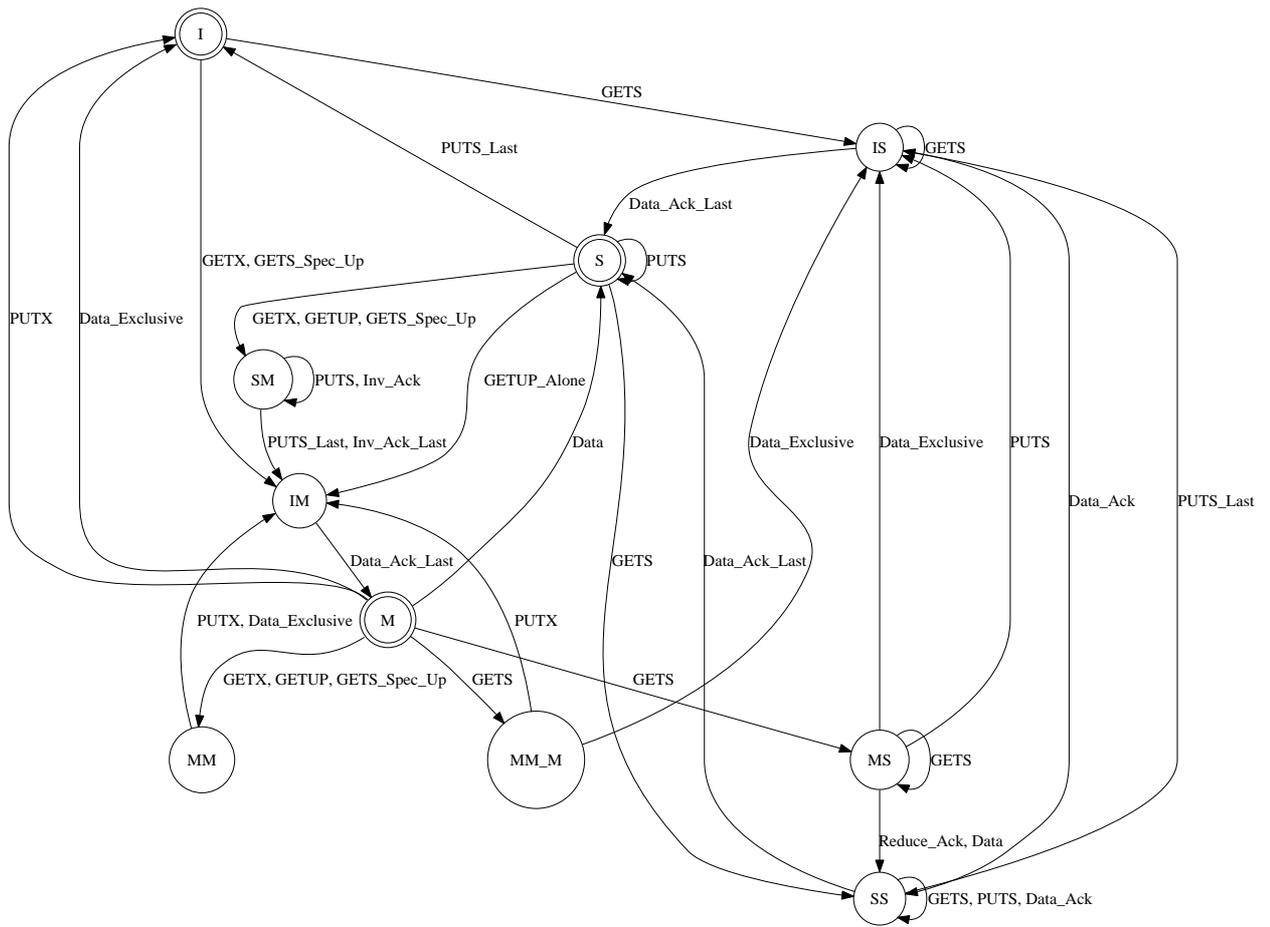


Figure 5.8: Directory Protocol of State Additions Scheme

5.2.2.2 History Table Modification

Both History Table Corruption and Not-Migratory Counters are examples of History Table Modification. These methods modify the behavior of the history table slightly to prevent lock-in. Normally, a block that fills the history table with lines corresponding to a migratory pattern will be predicted as migratory. This means that the next time a request for a Read Copy arrives, a Modify Copy will be sent. The directory determines the pass of sharing phases by the presence of a new Modifier. Thus, a new sharing phase is entered for every migratory prediction that is acted on.

There are several implementations possible here:

- The new sharing phase of the single processor that requested access is added to the history table.
- The behavior of the history table changes subtly in the presence of migratory predictions.
- The directory remembers the lines that have been sent as migratory. When it next sees them it updates the history table only if necessary.

The first option results in lock-in. The third option results in a larger history table that modifies the transient behavior of the system, and training data that is arriving at the same time predictions are needed. This is effectively equivalent to the State Additions mechanism proposed above, except that in the case of Program Counter based schemes it will require storing large amounts of additional training

information at the directory. The second option is the one explored in this dissertation. Two schemes that can correct for lock-in in the first two methods are detailed below: History Table Corruption, and Not-Migratory Counters.

5.2.2.3 History Table Corruption

In history table corruption, when an incorrect lock-in is detected (a line with Modify permission returns a clean line), an entry is inserted into the history table to escape this state. All that is required of this artificial sharing pattern is that it not have a single set processor. If consumer prediction is used in conjunction with migratory prediction, the entry chosen should have a desirable effect on the consumer predictor being used. In the case of a Union Predictor or Perceptron Predictor, a zero vector is added, and in the case of an Intersection Predictor a ones vector is added.

For the simple functions this provides a vector that is equivalent to not evaluating that entry, allowing predictions other than the null set and complete broadcast to be made. For the perceptron this vector is arbitrary, and the predictor should adapt to recognize this case.

5.2.2.4 Not-Migratory Counters

This scheme requires the addition of a counter comprised of a small number of bits to every set of the history table. These Not-Migratory counters are incremented when an incorrect Migratory prediction is detected. When a prediction is made

on this line and the Not-Migratory counter is below or equal to a threshold, the system performs as it did previously. When the Not-Migratory counter is above the threshold, the prediction is overridden and the line is not identified as migratory. Once the counter has overridden a prediction, it is reset to zero.

The effect of the Not-Migratory counters is that after a specific number of mispredictions the next access is not treated as migratory. If the behavior continues to be migratory the predictor will immediately revert to identifying it as migratory. By contrast, History Table Corruption will only revert to migratory prediction after the corrupt entry is shifted out of the history pattern. If the line does not continue to behave in a migratory fashion then lock-in has been escaped.

5.2.3 Results of Lock-In Prevention Mechanisms

Figure 5.9 shows the results of each of these mechanisms on the runtime of an application. The Not-Migratory Counters were run with a threshold of two. All of these mechanisms are able to remove the performance penalties when migratory prediction fails. The key difference between them is the extent to which they maintain the benefits that raw migratory prediction offered.

Surprisingly, the state based solution that most closely resembles previous solutions was the least able to accomplish these speedups. The “Not-Migratory” Counters were the most effective at this goal, although they required more storage space.

Figure 5.10 shows the total number of invalidations that were transmitted

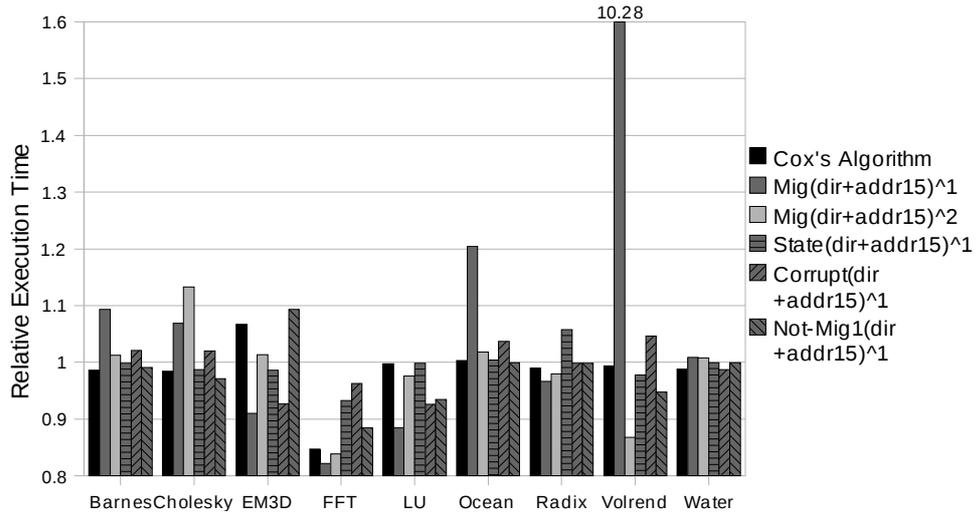


Figure 5.9: Runtime Behavior of the “Lock-in” Prevention Mechanisms

during execution for each of the schemes. All of the schemes were able to reduce the total number of invalidations in those cases where lock-in occurred to a reasonable level. It is interesting that some degree of lock-in was present on every benchmark, even those that received speedups.

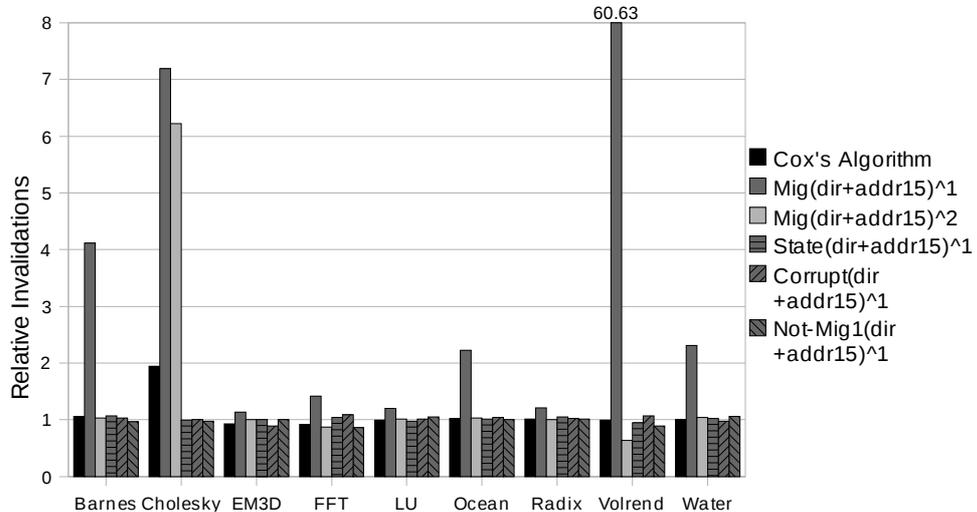


Figure 5.10: Changes in Invalidation Count Due to the “Lock-in” Prevention Mechanisms

Similarly, the number of requests necessary, shown in Figure 5.11 shows that lock-ins have been corrected, and correlates well to the overall performance. Where

the Not-Migratory counters perform best of the lock-in preventing mechanisms here they perform best in runtime.

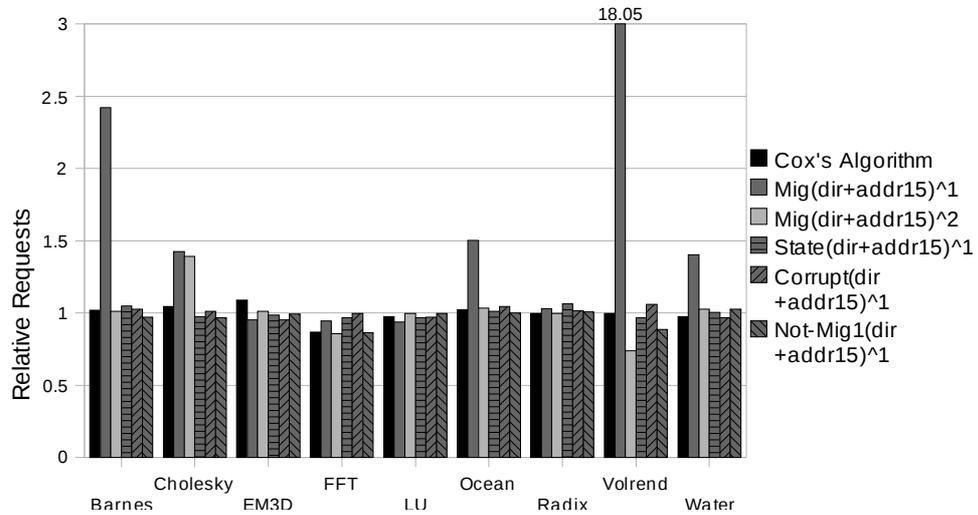


Figure 5.11: Changes in Request Count Due to the “Lock-in” Prevention Mechanisms.

Figure 5.12 shows the bandwidth consumed with the various migratory predictors in use. Migratory prediction produces small reductions in bandwidth consumption on all but a few of the benchmarks. The reason for this is the penalty for a misprediction. In the case of consumer prediction, a false positive resulted in the transmission of extra data. In the case of migratory prediction, a false positive results only in extra control messages.

Figure 5.13 shows the write latency. Overall there is a large benefit to write latency from lock-in. This is because the lines that lock-in to migratory pattern will only have to be requested for Modify permission if there is no load preceding the store.

Figure 5.14 shows the relative number of store instructions that generate messages to the directory. This includes both true misses and cases where the cache had

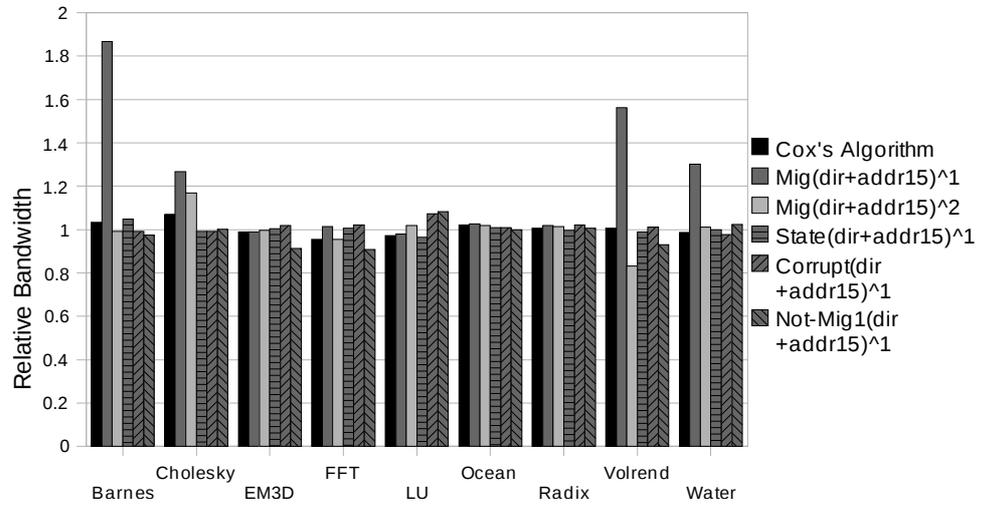


Figure 5.12: Changes in Bandwidth Consumed Due to the "Lock-in" Prevention Mechanisms

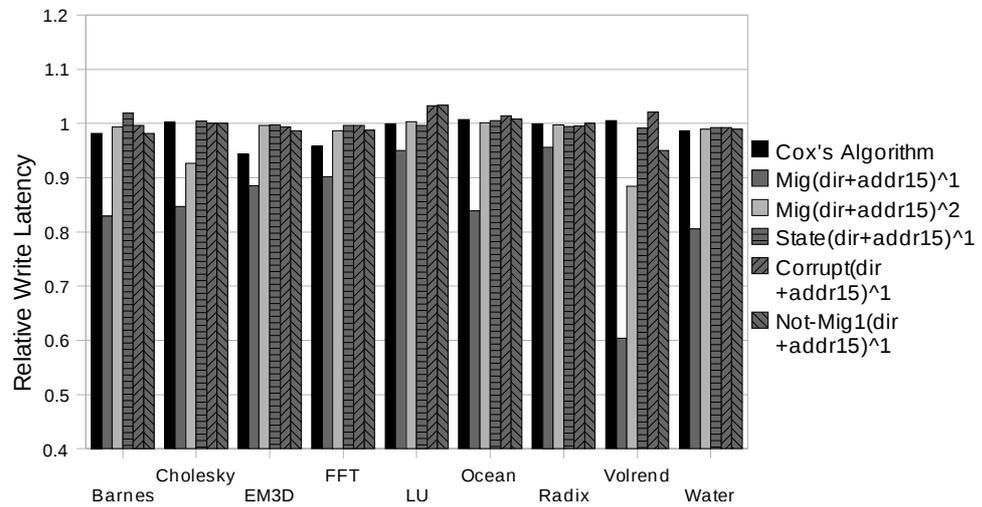


Figure 5.13: Changes in Write Latency Due to the "Lock-in" Prevention Mechanisms

a Shared copy of the line when the store was first encountered. As can be seen, the depth one migratory predictor with lock-in reduced the number of store misses by a substantial amount. It did this by transmitting a large number of Modify copies, regardless of whether or not the line is truly migratory. Effectively, it removed the Shared state from a large portion of memory.

All the three lock-in prevention mechanisms produce smaller speed-ups consistent with Cox’s algorithm. In most cases the not-migratory counters provided the best results in terms of store miss rate.

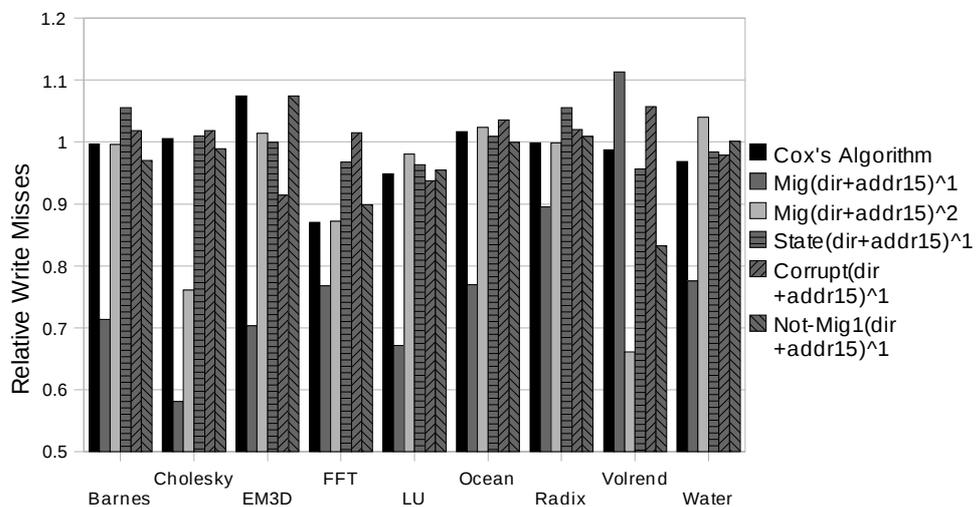


Figure 5.14: Changes in Write Misses Due to the “Lock-in” Prevention Mechanisms

5.3 Instruction Based Migratory Consumer Predictors

The first results presented in this chapter were for an address based prediction scheme because the address information is most readily available at the directory. However, it is possible to pass PC information to the directory. This was discussed at length in Chapter 4.

Using this information the history table is trained just as with consumer prediction. This history table is applied to the same function as above, looking for histories where only a single sharer was present in each phase. However, the task of Migratory prediction is more complex, due to the fact that the prediction needs to be made before the arrival of the response from the current holder of a Modify copy, so that an invalidate can be transmitted rather than a reduction. With consumer prediction, the speculation of which processors to target was only needed once the acknowledgment of invalidation had arrived at the directory. This meant that instruction information had already been transmitted to the system, and could be used to index the table. In this case the prediction needs to be made to determine whether the holder of the Modify copy will be sent an invalidate, or will be sent a reduction.

This section briefly outlines the effects of naively using old indexing information, and then move on to discuss two alternative methods that avoid this old information, at the cost of a greater latency on migratory predictions.

5.3.1 Naive Instruction-Based Migratory Prediction

In this naive version of Instruction-Based Migratory Prediction, whenever a request for a Shared Copy arrives at the directory, it is immediately evaluated to determine if it should be upgraded to a Modify Copy. This update is based on the table entry corresponding to the last instruction to write the line *that the directory is aware of at the time*. However, the directory is not aware of anything that has

happened to the line at the current owner of the Modify copy. When a Migratory line is identified, an invalidation is sent to the current owners, rather than a reduction. This means that unless the current modifier was executing the same code as the previous, the wrong history table entry is consulted.

Figure 5.15 shows the resulting change in execution time that results from this scheme. As we can see, there are substantial performance penalties for this scheme. Lines are being identified as migratory at the wrong times, specifically when the previous access was migratory. In fact, the only reason any speedups are achieved at all is that specific lines are always handled by the same small section of code, and thus always index to a small set of places in the table.

In addition, several benchmarks show signs of lock-in for low history depths. Overall, none of the penalties are as large as those in Figure 5.6, as a longer history depth is more effective for instruction based prediction than for address based prediction. Nonetheless, the possibility of lock-in should be avoided so that severe performance penalties are not incurred for specific cases of pathological code.

5.3.2 Second-Look Instruction-Based Migratory Prediction

Second-Look Instruction Based Migratory Prediction attempts to correct for the problem of indexing before the necessary information is available, by waiting until it knows the correct index to make a prediction. If a request for a Shared copy arrives and no one currently owns a Modify Copy, a prediction is immediately made. If a Modify copy is currently outstanding, the directory waits until the data

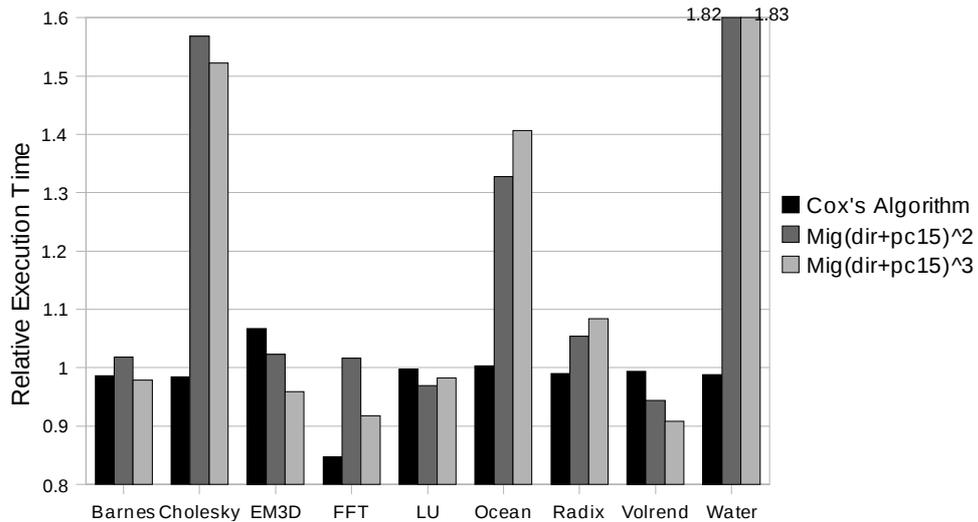


Figure 5.15: Execution Time of the Naive Instruction-Based Migratory Prediction Scheme

are returned from the modifier. It then uses the information about the last writer contained in this message to predict if the line is migratory.

If the line is not identified as migratory, a Shared Copy is immediately forwarded to the requester, only if a second request has not arrived. If it is identified as migratory, the directory waits to transmit a copy, first invalidating the distant copy, and then transmitting a Modify copy. This delay is incurred in order to avoid any possible race conditions that might arise from sending Exclusive Data unannounced. One advantage of this scheme is that it prevents a case where Modify copies are speculatively transmitted when additional requests are waiting to be processed.

Figure 5.16 shows the new events that implement this. These events are initiated when data arrive, and the instruction information it carries indicates a migratory line. As you can see from Figure 5.17 when these events arrive they are translated into pre-existing transient states. Data Exclusive Up translates to an im-

<i>Event Name</i>	<i>Description</i>
Data Up	Data Has Arrived, Instr info indicates Migratory Only one processor has requested a Shared Copy
Data Exclusive Up	Data Has Arrived, Instr info indicates Migratory Only one processor has requested a Shared Copy

Figure 5.16: Additional Directory Events For Second-Look Migratory Prediction

mediate transmission of exclusive data. Data Up is treated as though a new request has arrived from the target processor asking for migratory access. Invalidations must be sent first.

5.3.3 Forward-Ahead Second-Look Instruction-Based Migratory Prediction

One problem with Second-Look prediction is that when migratory prediction is made it takes longer to deliver the Shared copy than it would have otherwise. Forward-Ahead Second-Look prediction corrects for the extra delay added by Second-Look Migratory Prediction. It makes predictions in the same way, waiting for data to arrive from the requester before determining if the line should be migratory. If it is identified as migratory, the requested shared copy is immediately forwarded ahead, and then the directory speculatively initiates an upgrade to Modify permission. This scheme still distributes Modify permission slower than the Address based schemes, but it will not produce extra delays on read requests, as the Second-Look scheme will.

On the other hand, implementing this scheme results in unrequested Modify

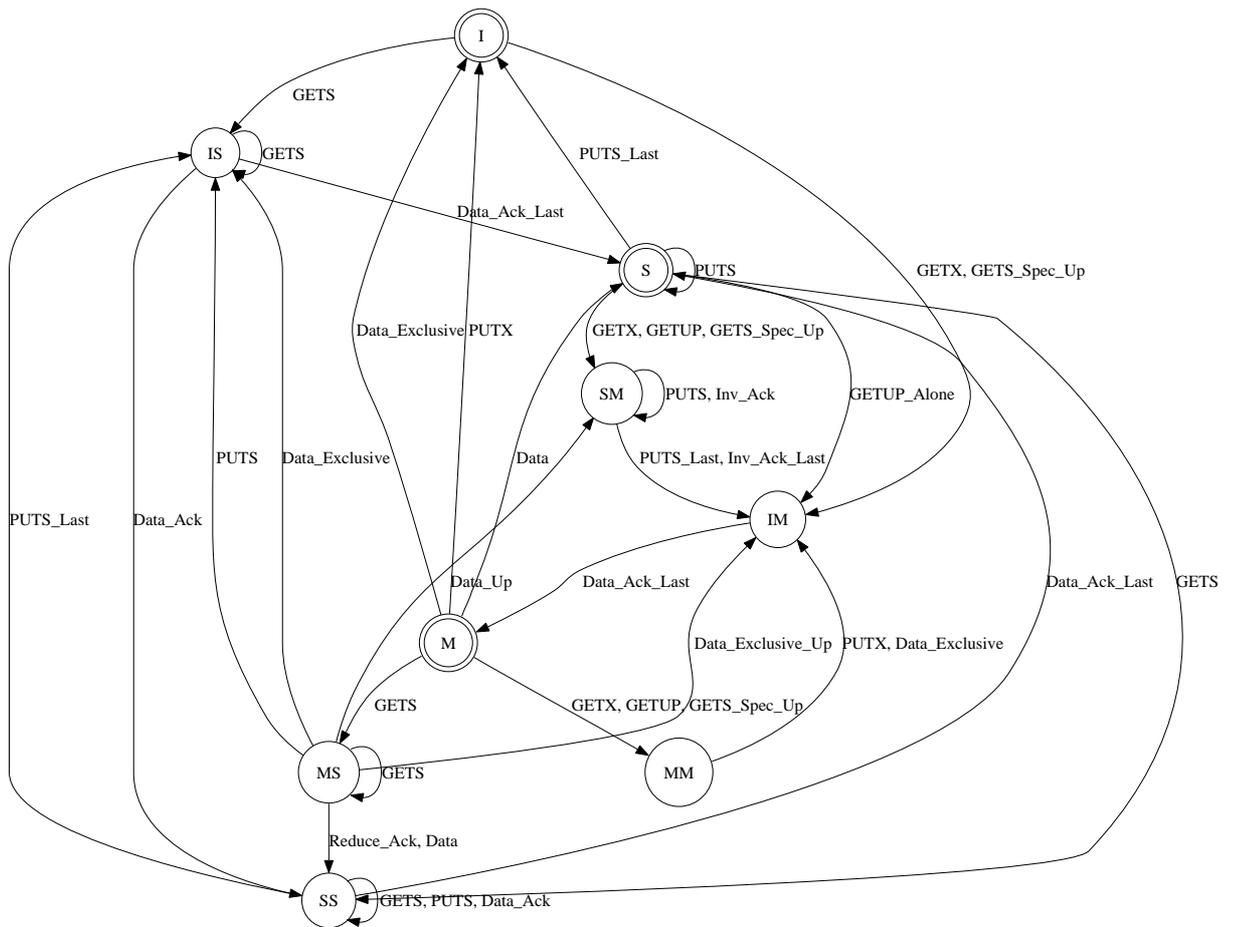


Figure 5.17: Directory State Machine Representation with Second Look Migratory Predictor

copies being transmitted to the cache. These data can result in new race conditions, just as unrequested Shared copies could in Consumer Prediction. This is handled in the same way as unrequested Shared copies in Consumer prediction. When speculative Modify Copies arrive and a request has already been transmitted to the directory, the cache holds off on acknowledging the data until this request has been removed by the directory. This effectively freezes the directory in the given transient state until all messages that could result in a race condition are resolved.

5.3.4 Second-Look Instruction-Based Migratory Prediction

Figure 5.18 shows the resulting execution time associated with Second-Look and Forward-Ahead Second-Look instruction-based migratory prediction. The large slowdowns caused by the naive implementation of migratory prediction have been removed in all cases. In fact, for each of the benchmarks which had these slowdowns there is at least one predictor that provides some speedup. For those benchmarks where the naive approach was effective, the Second-Look approach is superior. In every case except Barnes, there is some predictor out of the four evaluated here that will produce better results than Cox's scheme.

Overall, the penalties due to lock-in are smaller than in address based schemes. None of the benchmarks show the major performance degradation that was caused by lock-in previously. Unfortunately, none of the four schemes shown consistently performs the best across all of the benchmarks. In general, one of the forward-ahead benchmarks performs better than both of the plain Second-Look benchmarks.

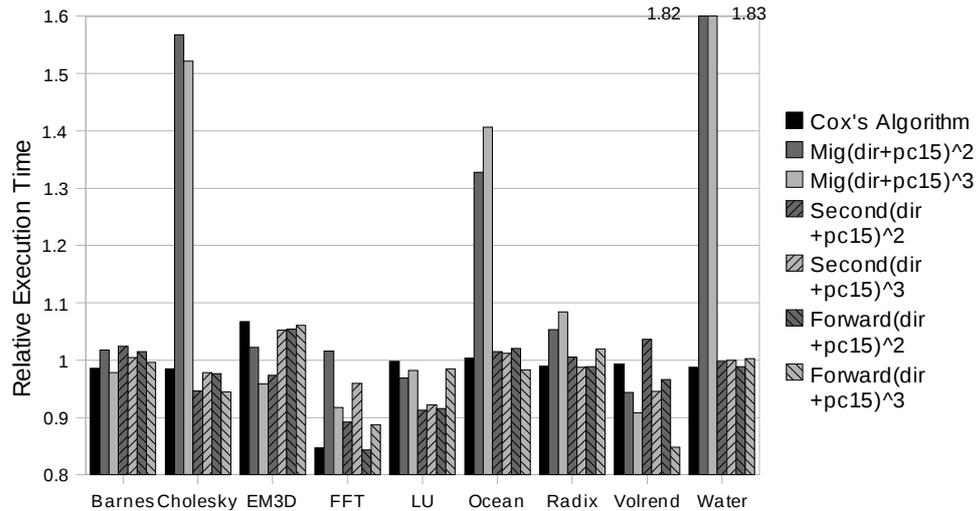


Figure 5.18: Execution Time Benefit of Migratory Prediction Using Instruction Based Addressing as History Depth Varies

5.3.4.1 Interprocessor Message Effects of Instruction Based Prediction

As Figure 5.19 shows, the number of invalidations required grows slightly over the control due to false positives. Each false positive corresponds to an invalidation sent to the sharer that previously wrote the line. However, this increase is much smaller than for address based techniques. In a few cases the total number of invalidations falls. In the case of Volrend, this effect is similar to the one described in an earlier chapter, where the invalidation's source is the distribution of work when one processor falls behind. The total number of invalidations here grows more than for Cox's algorithm. The reason for this is that Cox's scheme is extremely conservative by comparison.

Figure 5.20 shows the total number of request messages that were sent with instruction based migratory prediction in place. As can be seen, a decrease in

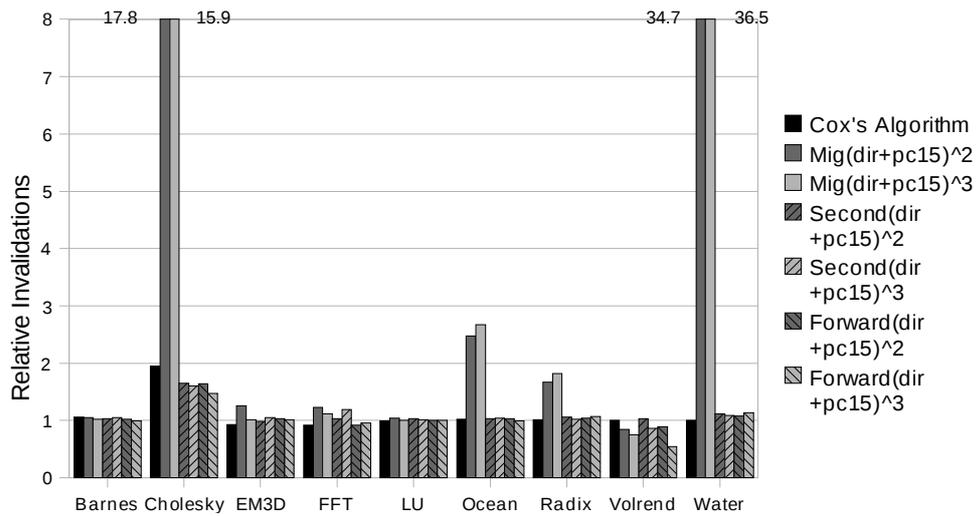


Figure 5.19: Changes in Invalidation Count Due to Instruction Based Migratory Prediction

this number correlates with improved performance. Correct migratory prediction removes the need for additional requests for modify access by the processor that issued the request for a Shared copy. In those cases where migratory prediction is least successful the number of requests rises, as processors that were incorrectly invalidated must request a new Shared copy.

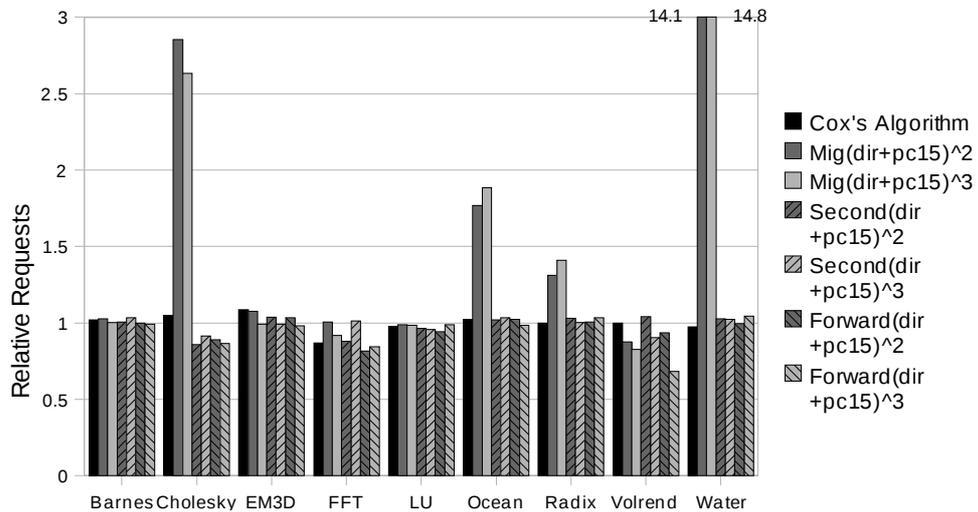


Figure 5.20: Changes in Request Count Due to Instruction Based Migratory Prediction

5.3.4.2 Bandwidth Effects of Instruction Based Migratory Prediction

Figure 5.21 shows the change in bandwidth consumption for each history depth and benchmark. In many cases, the bandwidth consumed falls slightly with migratory prediction in place. Unlike consumer prediction, an incorrect Migratory prediction results only in an additional two control messages (Reduction, Acknowledgment), rather than an additional data message. In consumer prediction, a correct prediction saved a single request message. That is also the case in Migratory prediction.

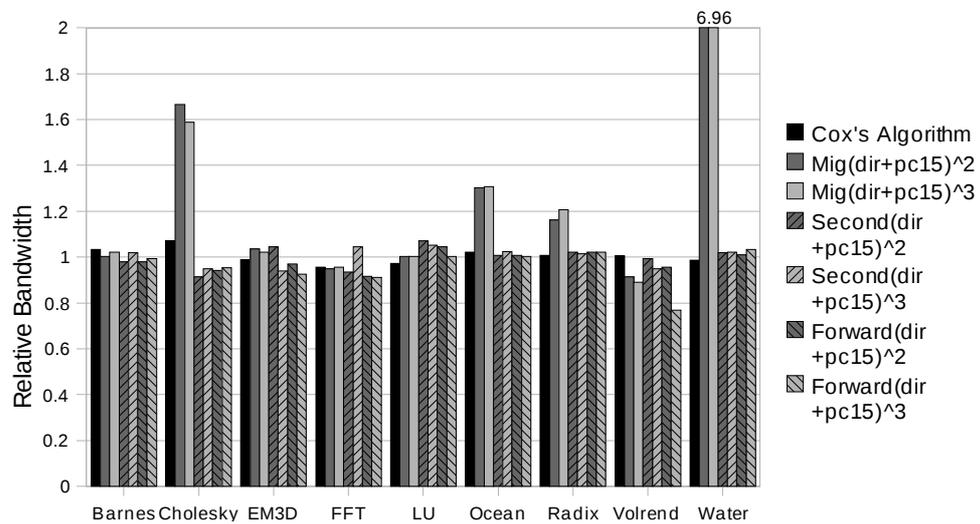


Figure 5.21: Changes in Bandwidth Consumed Due to Instruction Based Migratory Prediction

Thus the bandwidth penalty for a misprediction has decreased substantially, while the benefit has remained the same. Because of this, the migratory predictor is able to reduce the total bandwidth consumption where consumer prediction could not.

5.3.4.3 Directory Latency of Instruction Based Migratory Prediction

The change in the latency of requests for Shared copies is shown in Figure 5.22. The results here are similar to the results for write latency in the case of consumer prediction. For the most part they increase because the time required to service a request for a shared copy increases when another processor has a Modify copy.

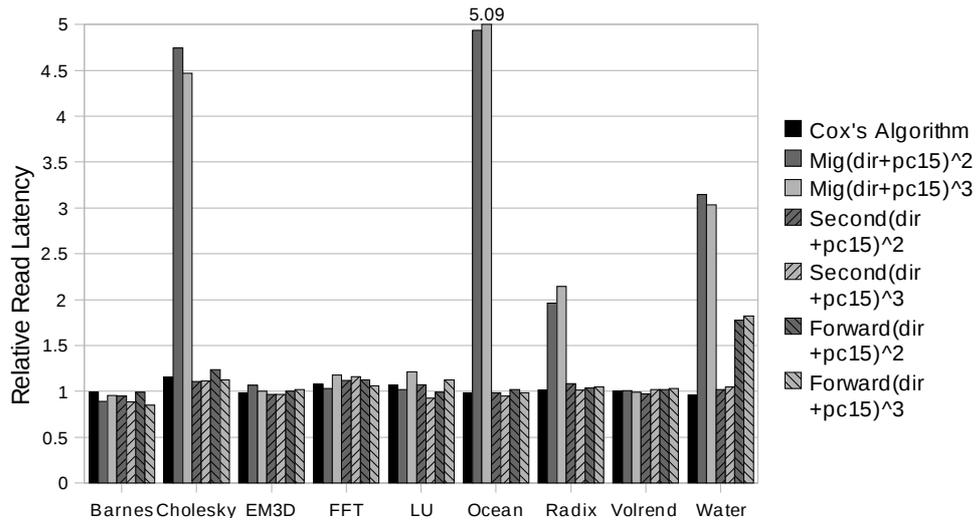


Figure 5.22: Changes in Read Latency Due to Instruction Based Migratory Prediction

In a few cases, the latency of requests for Shared copies decreases due to the fact that the requests are less likely to arrive during a transitory state. This effect is most pronounced in Barnes and LU for the plain Second-Look algorithm, but exists on some predictor for every benchmark.

Figure 5.23 shows the effect of the instruction based migratory predictors on the time it takes a cache to receive a modify copy. The total behavior of the system changes very little, save for a few benchmarks. Notice that these changes do not correlate to the runtime, the amount of time taken to obtain Modify access is not as important as how often that access needs to be obtained.

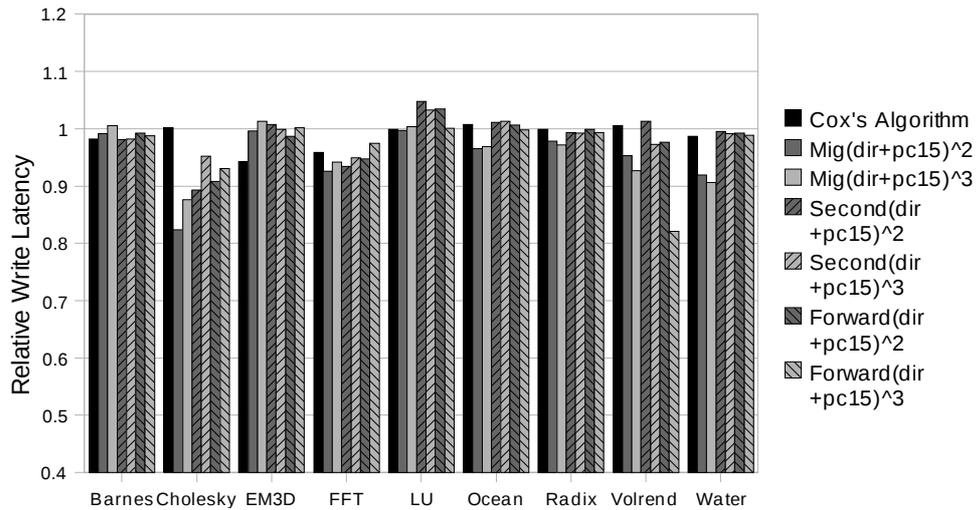


Figure 5.23: Changes in Write Latency Due to the "Lock-in" Prevention Mechanisms

5.3.4.4 Miss Rate of Stores With Migratory Prediction in Use

Figure 5.24 shows the number of stores that missed in the cache, or arrived when only a Shared copy was present for the instruction-indexed migratory predictors. The instruction-indexed predictors more effectively reduce the number of misses than the address-indexed predictors shown in Figure 5.14. Overall, the instruction based Second-Look, and forward-ahead protocols were able to identify substantially more migratory lines than Cox's predictor for all of the benchmarks.

Notice that the large reduction in write misses for Cholesky, combined with a small increase to read latency, and reduction in bandwidth produces only a modest speedup. This emphasizes the importance of preventing the rise in read time.

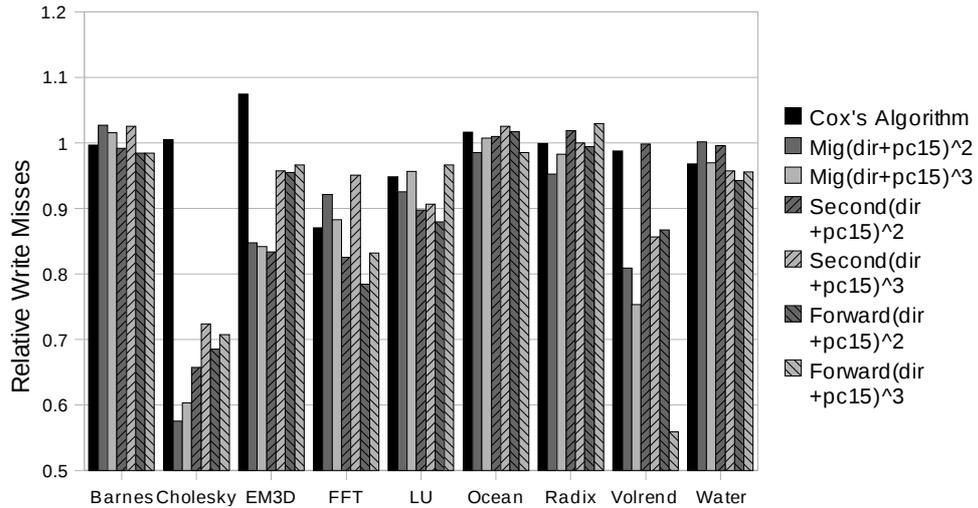


Figure 5.24: Changes in Write Misses Due to the "Lock-in" Prevention Mechanisms

5.3.4.5 Miss Rate of Instruction Based Migratory Prediction

5.4 Indexing a Migratory Predictor With Mixed Information

In consumer prediction there were some benefits to indexing a consumer predictor with a combination of instruction, CPU, and address information. This approach could be beneficial in migratory prediction for the same reasons it was beneficial in consumer prediction. In addition, the intention is to combine migratory and consumer prediction by sharing a history table. Given this, both predictors will have to use the same indexing mechanism. Because of that, the results of using Address Mixed, and CPU Mixed indexes are presented here.

5.4.1 Address-Mixed Migratory Prediction

In Address-Mixed Migratory Prediction, the history table is indexed using information about both the address and the last writer of the line. Conceptually

this can allow for a greater degree of accuracy on the part of the predictor as cases that may have resided in the same history line may now be split apart. This is an advantage when both the instruction and address are relevant to the behavior of a given line. However, when one is not a critical aspect this mixing is a disadvantage, as the predictor does not train as quickly.

With consumer prediction there was no clear benefit to using this scheme, and as Figure 5.25 shows there is still no clear benefit. A slight improvement is made to the EM3D benchmark, and the potential speedup on Volrend is lost, in this scheme. However overall no new benchmarks receive speedups that were not achieved by a previous scheme. The overall mechanism of these speedups has not changed.

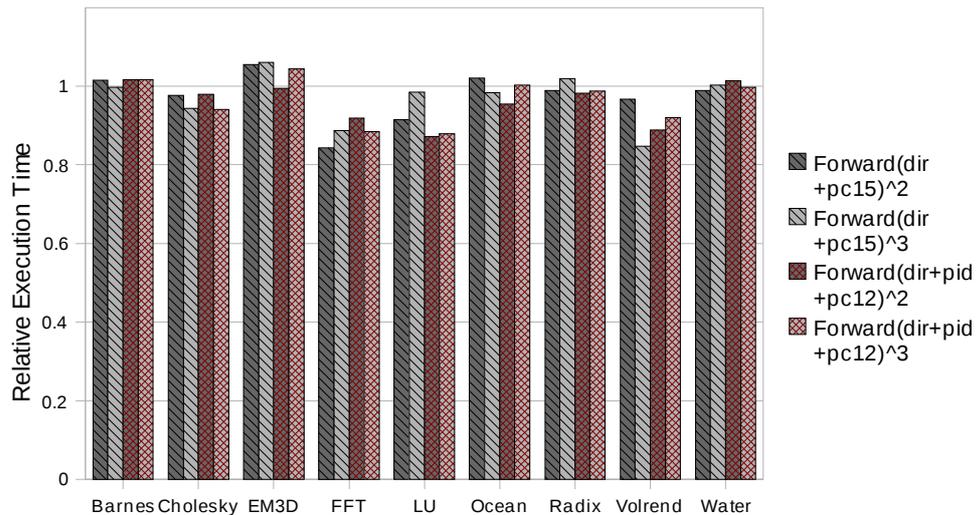


Figure 5.25: Runtime Effect of a Mixed Address-Instruction Migratory Predictor

5.4.2 CPU-Mixed Migratory Prediction

In CPU-Mixed Migratory Prediction, the history table is indexed using information about both the CPU that performed the last write, and the instruction

that last wrote the line. In consumer prediction this technique was more effective than Address-Mixed Migratory Prediction, because it allows more predictions to be made as the training happens more rapidly. Additionally, in many cases the writer contains more information about the program behavior per bit of index data than the address.

As with consumer prediction the positive changes here are relatively minor, but superior to Address-Mixing. Figure 5.26 shows the results of applying CPU-Mixed Migratory Prediction. As you can see several of the benchmarks improve slightly (Cholesky, Ocean, Radix), one improves measurably (LU) and several are held back (FFT, Volrend).

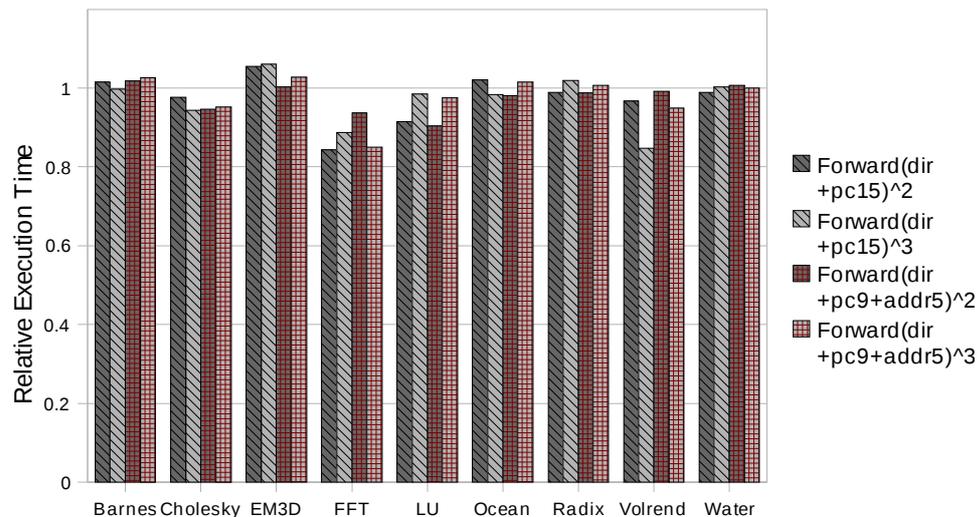


Figure 5.26: Runtime Effect of a Mixed CPU-Instruction Migratory Predictor

5.5 Summary

It is possible to use the data gathered by a consumer predictor to construct a migratory predictor that performs comparably to previous migratory predictors

that can be placed at the directory. There are two main challenges in building such a predictor. The first is detecting “lock-in”, false positives that force the system to continue making false positives. The second is acting on PC information when the information needed to index the table is not available when the prediction could most easily be made.

The most obvious method for correcting “lock-in” is to add a state to the system that remembers the mistake and prevents it in the future. Two other solutions that do not require adding additional states to the directory machine, but instead operate at the predictor level, were also discussed. One of these solutions is to corrupt the history table in such a way as to prevent the bad pattern of predictions from continuing. The other solution is to add a counter to the predictor that tracks false positives and overrides the predictor when enough have been detected. On the majority of the benchmarks both of these solutions outperform the “obvious” solution.

Two strategies are proposed to handle the use of PC information. One waits for the PC information to become available. If it then decides to pass migratory permission, it waits to send a Shared copy, invalidates the former Modifier, and then forwards a Modify copy. The other also waits for the PC information to become available. Rather than waiting to forward the copy until a Modify version is available, it sends the Shared copy ahead, and then sends the Modify copy when it arrives.

The first of these has a simpler implementation, and does not add new transient states or potential race conditions to the system. The second adds several new

transient states at both the directory and cache, and introduces potential new race conditions. However, it also satisfies the initial load condition faster. Both of these implementations are able to produce speedups comparable to, or exceeding the address based predictors.

Chapter 6

Timing Prediction

This chapter discusses two aspects of timing prediction: training of last touch prediction and use of last touch prediction signatures to enhance the performance of other predictors.

Timing prediction refers to prediction that estimates the time at which events will occur, rather than the events that will occur. For instance speculative self invalidation identifies the last memory access before an invalidation message will arrive. This prediction has the potential to be combined with consumer prediction, such that consumer prediction would not have to wait for a request to transmit data.

6.1 Self Invalidation

Self invalidation was first proposed by Lebeck and Wood [29]. In their system, specific lines were recognized by the directory to be self-invalidated and marked. Once the cache accessed the specific lines, it would release them based on one of two methods, either based on FIFO ordering or based on the detection of a synchronization action (such as a barrier). Thus the timing shown in Figure 6.1(a) could be transformed into that shown in 6.1(b), saving the latency of two messages across the network in those cases where the invalidation occurred early enough. This relatively simple scheme produced speedups on many benchmarks.

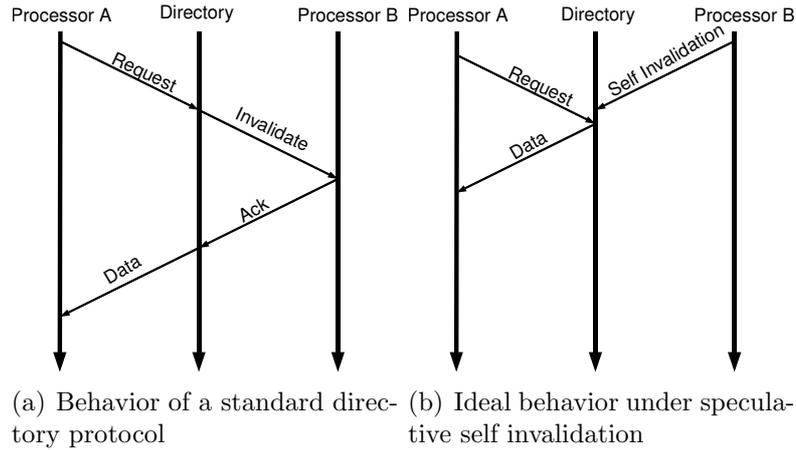


Figure 6.1: An example of the possible performance impact of speculative self invalidation. In the example on the right the current sharers speculatively self invalidate. This allows the directory to respond with the data more quickly.

Later, Lai and Falsafi took a different approach to self invalidation with a last touch predictor [28]. The idea behind the last touch predictor was to identify the last access a processor would make to a line before an invalidation arrived. Once such an access was complete, the cache would give up access to the line and notify the directory, just as in the scheme proposed by Lebeck and Wood.

In order to predict last touches, Lai and Falsafi used a table based predictor, storing the series of program counters that touched a line before it was invalidated when it was previously owned. An overview of this system is shown in Figure 6.2. Each time a line is accessed, an accompanying signature is updated by adding the program counter of the accessing instruction to the current signature and truncating the result. The signature is then compared to a table — either global or hashed based on the line address — of previous signatures corresponding to invalidations. When a match occurs, the line is flagged for self invalidation. The look-up table used had some level of associativity, and so multiple signatures could be stored for

any line address.

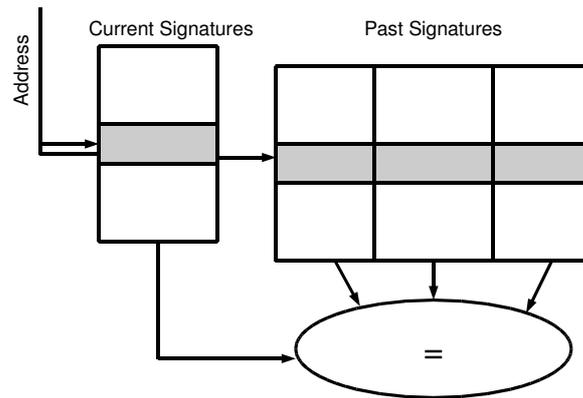


Figure 6.2: The structure of Lai and Falsafi’s Last Touch Predictor

Training of the system comprises two steps. First, whenever an invalidate message arrived, the signature currently stored with the line was exported to the table of known signatures. Second, a confidence estimator was maintained to identify signatures that were not accurate. The training of this confidence estimator was not discussed in detail, and the effects of several options are explored in Section 6.1.2.1.

Lai and Falsafi’s scheme had the advantage of being able to identify more opportunities to self invalidate than the scheme developed by Lebeck and Wood. Further, it demonstrated that program counter (PC) based techniques could be more effective in predicting the behavior of the memory system, a finding echoed by the work of Kaxiras and Young in their investigation of consumer prediction [19].

One of the main advantages of speculative self invalidation over other predictive schemes is the ease of protocol verification. Speculative self invalidation causes lines to become invalid, without a message from the directory arriving to demand it. This event also occurs whenever data are forced from the cache. Thus the capability to perform self invalidation is already built into any multiprocessor cache-coherence

protocol. Where other methods may require additional verification, this method does not.

Despite the fact that no new race conditions should be created by speculative self-invalidation, this form of prediction can cause race conditions to occur that were sufficiently unlikely that they had escaped notice. This predictor has a tendency to locate new race conditions that were not found in testing.

6.1.1 Changes to the Directory And Cache to Support Self Invalidation

The directory of a coherence protocol must be able to support lines returning from the caches at arbitrary times. Because of this, no additional race conditions develop from self invalidation. However, the predictions in this implementation are made when the request first arrives so that program counter information need only be stored for a minimal amount of time. This means that a number of additional transient states were added at the cache. Each of these states behaves exactly as its corresponding transient, except that when the request that caused it is satisfied, the cache immediately self invalidates. The new set of cache states and events as well as the new cache behavior can be found in Figures 6.3, 6.4, and 6.5.

<i>State Name</i>	<i>Description</i>
ISI	An Invalid copy is present, A Shared Copy has been requested That Copy will then be discarded
IMI	An Invalid copy is present, A Modify Copy has been requested That Copy will then be discarded
SMI	A Shared copy is present, A Modify Copy has been requested That Copy will then be discarded
SUI	Received speculative data after requesting a Shared copy, Self Invalidate once resolved

Figure 6.3: Additional Cache States Necessary to Support Self Invalidation

<i>Event Name</i>	<i>Description</i>
Load LT	Processor has performed a load, The line should be discarded
Store LT	Processor has performed a store, The line should be discarded

Figure 6.4: Additional Cache Events Necessary to Support Self Invalidation

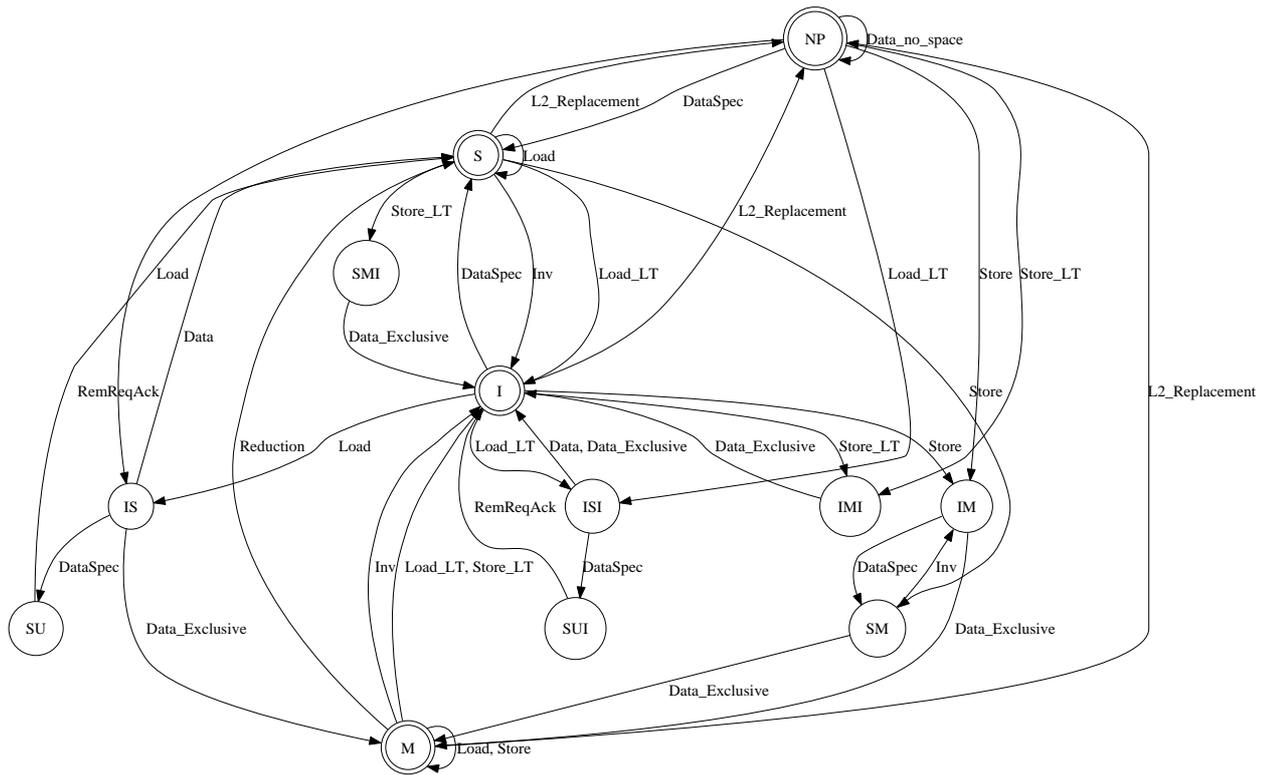


Figure 6.5: State Machine Representation of a Cache Supporting Timing Prediction.

6.1.2 Training The System

6.1.2.1 Confidence Estimator Training

Identifying signatures of interest in a coherence timing predictor is relatively simple. Whenever an event of interest (i.e., an invalidation message) occurs, the signature currently present can be recorded.

On the other hand, training the confidence estimator is much more difficult. While no additional information is necessary to identify a correct prediction to not self invalidate, or a correct prediction to self invalidate that was overruled by the confidence estimator, actual self invalidations are more complex. Once the cache speculatively self invalidates, it has no inherent way to identify when it was correct

and when it was incorrect. In order to accomplish this identification, additional information must be passed between the processors. In this work the coherence predictor is always updated as soon as possible. That means that whenever data are transmitted from the directory to a processor, an additional bit of data is sent identifying whether or not that processor self invalidated and no other requests for write permission on the data were subsequently seen. At the same time, acknowledgments are sent to all the other processors that self invalidated, telling them that they were correct. This differs from the previous work, which proposed that information would be “piggybacked” on other messages until it reached its target.

Sending these additional messages is important for several reasons. First, in the case of a global predictor, where the set of signatures being matched is applied to every line (or multiple lines mapping to the same entry in a local predictor), more predictions will be able to benefit from the training data. Second, any training will take at least some time, which may mean that it will not be possible to take the associated training data into account before making a decision on the data that were received when piggybacking is used. If predictions are made as the requests arrive at the cache, training data that come with a message will not be available. To take advantage of these data, predictions must be made after the training has completed, which will have occurred some time after the data arrive at the cache. This adds unnecessary delay to the system. Third, depending on the scheme used, information about predictions may be stored with the local predictors, and the resources used to do that are finite. By sending training information early, the information stored there can be used before it is overwritten.

While this method identifies invalidations, it does not provide information about the nature of the prediction that caused the last touch. In the case of a local predictor with no associativity, this may be a simple matter of hashing the line address onto the table to identify the confidence estimator. On the other extreme, in the case of a global predictor, without additional information the confidence estimator would only be able to operate on the entire predictor structure. In particular, the system must identify which associative entry was responsible for which prediction in any but the simplest local scheme. Further, the simulations showed (see Figure 6.6) that in the case of a local predictor with associativity, it was much more effective to keep a confidence estimator for each signature than for each line.

Thus, additional prediction information must either be stored locally or transmitted with the data and later messages. Keeping this information at the directory would require the storage of an associative set for every processor and every line, or $\log_2(\text{Associativity}) \times \text{Processors}$ bits for every line, in addition to the *Processors* bits needed to identify that had self invalidated. This could be modified to reduce the storage overhead by assuming that most lines would not have self invalidations and using an approach similar to that proposed by Nilsson, Landin and Stenstrom [39]. However, given such approximations would have to be made anyway, the extra training data that would be necessary to do this are not transmitted to memory at all. Instead, a local prediction history table is maintained at each processor that stores the tag of each line for which a self invalidation was made and the corresponding set in the look-up table.

Whenever training information arrives, it is matched to this table, and the

confidence estimator can be trained accordingly. The corresponding information in the local prediction history table is then removed, to free up the resource. Should the table fill up, the oldest entry is overwritten. Thus if training information does not return in time, it is rendered unusable.

A number of experiments were conducted to identify an effective training scheme. Figure 6.6 shows the result of keeping a single confidence for each signature, rather than one for each line. In both cases, the confidence estimator is trained only when a signature matched. These experiments showed that training the confidence estimator when the predictor correctly chooses not to self invalidate will quickly saturate all the confidences in the system in the highest state, thus rendering them ineffective. The system trains only when the predictor chooses to self invalidate, though it may have been overruled by the confidence estimator. In the case of keeping a confidence estimator for each signature, confidence estimator is reset whenever a signature is overwritten. Additionally, the signature that possesses the lowest confidence is replaced when a signature needs to be removed. This prevents a number of infrequent or unreliable signatures from interfering with a common reliable signature.

Given the advantages of keeping a confidence for each signature, it could be expected that confidence based signatures would outperform line based confidence significantly. While it does, the increase in performance is not as substantial as might be hoped. In neither case were the speedups originally shown for last touch prediction reproduced. This failure is very likely because the GEMS simulator includes simulation of the operating system, rather than simulating only the benchmark.

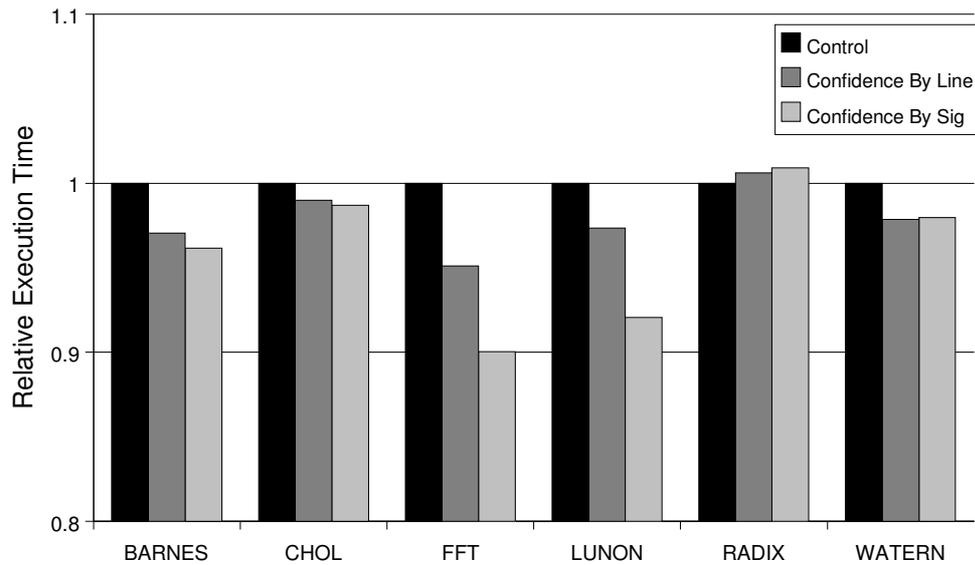


Figure 6.6: A comparison of the relative execution times depending on the location and coverage of the confidence estimator. (Network Latency = 40 cycles)

Note that in two cases associating a confidence with a line is preferable to associating a confidence with each signature. One is for Radix, a benchmark that is slowed slightly by self invalidation. The reason is that when a confidence estimator is associated with a line, the confidence estimator is more quickly able to deactivate the predictor. When associated with a signature, the confidence must be retrained for each new signature.

The other slowdown is for Water, where the increase is very small. This occurs because last touch prediction is extremely accurate on this benchmark, so the extra time for the confidences to grow sufficiently is a slight impediment to performance.

Figure 6.7 shows the effect of changing the size of the local prediction history table. As can be seen, the number of history entries needed is relatively small, meaning that there is little reason to rely on large structures at the directory to store it. In addition, it can be noticed that in all cases, an extremely long prediction

history table is not beneficial. For example, Cholesky decomposition reduces slightly in speed when the number of history entries is long. This is likely due to phase behavior, where there is an advantage to not training on data that are sufficiently old. A shorter history length causes the predictor to ignore old signatures that are no longer relevant if the program has passed into another phase.

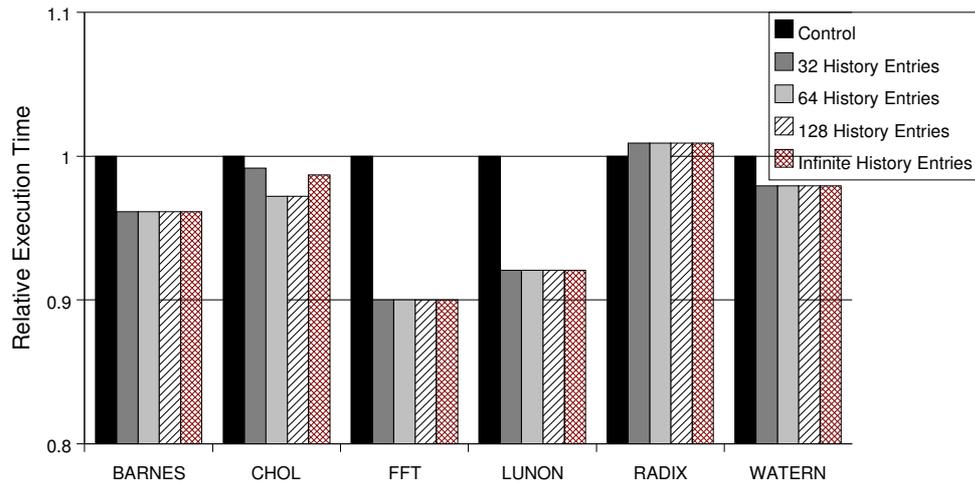


Figure 6.7: A comparison of the relative execution times as the amount of history information saved grows. (Network Latency = 40 Cycles)

6.1.2.2 External Training Information

The training scheme outlined above suffers from the need to separately train a predictor for each processor. This is a potential inefficiency because many multi-processor programs are symmetric; every processor is running the same code. Thus, it would be beneficial if the various predictors could share prediction data with one another.

This section identifies places where training data can be piggybacked on other messages. In general, the first corresponds to positive training data — signatures

that were seen as good — and the second corresponds to negative training data — signatures identified as poor. However, future implementations need not follow this pattern.

Figure 6.1(a) shows the messages passed by a directory protocol when ownership of data passes from one processor to another. Notice that when an invalidate arrives, all caches that possess a copy of the line must send a message to the directory. At the same time, every cache that possesses a copy of the line is training itself with a new last touch signature. Each of those messages is an ideal candidate to piggyback a signature to the directory, and from there to the requester. It is worth noting that the requester in this instance may get little use from these signatures because they correspond to the results of read access. However, the signatures acquired this way will then be available to forward to processors that later request read permission, where they are more useful.

The implementation used is straightforward. For each line, the directory remembers a single signature, the last one it observed. Whenever the directory sends data, it attaches this signature to the data. The receiving cache then processes this signature as though it had been observed locally.

As can be seen from Figure 6.8, including just the signature from a single processor produces sizeable speedups on three of the six benchmarks. It can be difficult to accurately state the coverage or accuracy of a last touch predictor, as the information about accuracy is distributed between the caches and directory. For example, a self invalidation at a processor, which would have been incorrect, may appear correct because the directory sees the request for access from another

processor first. However, had the processor held onto the line, the corresponding invalidation may not have arrived first.

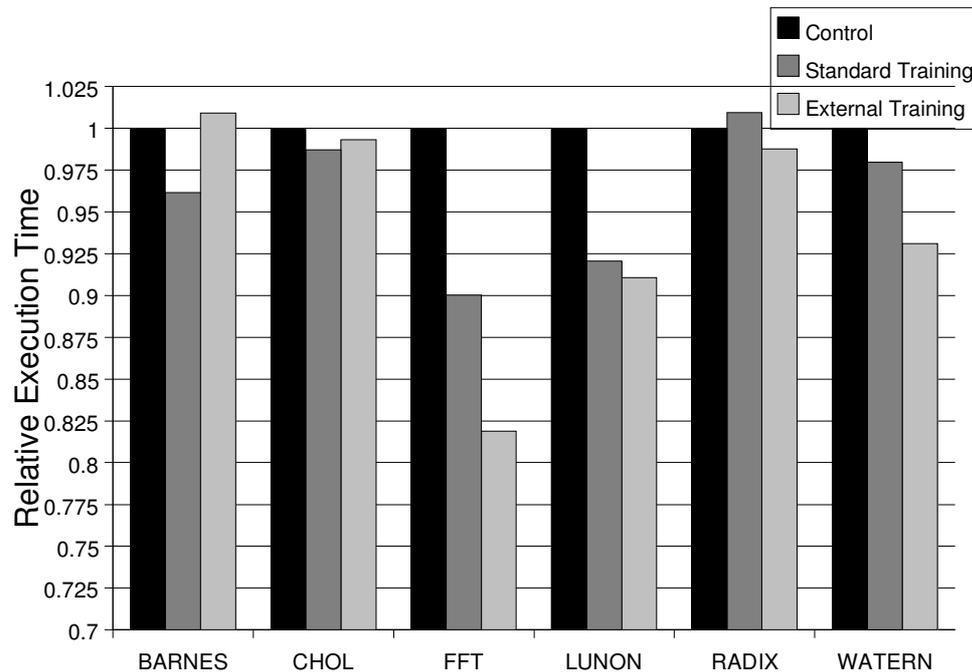
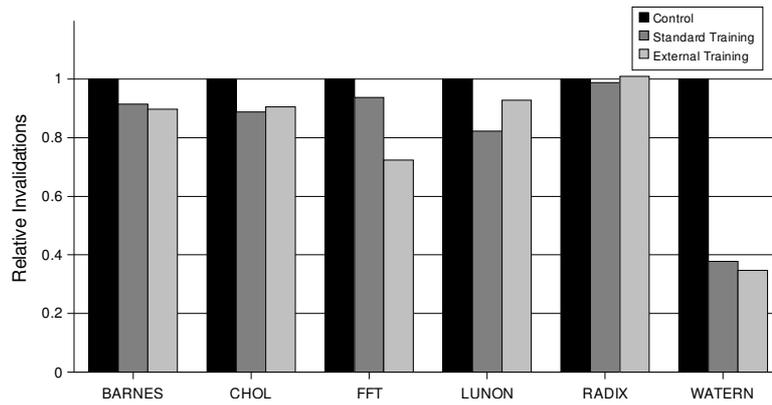


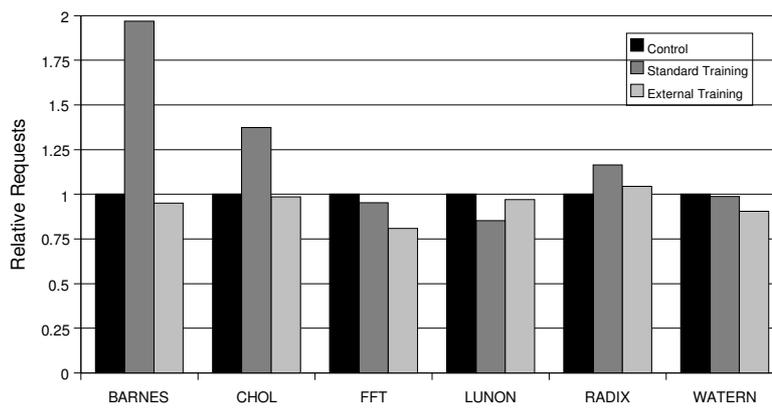
Figure 6.8: A comparison of the relative execution times with and without external training. (Network Latency = 40 Cycles)

Further, the behavior of multithreaded programs can depend on the performance of the memory system. This means that predictors can have feedback effects. Rather than looking at the predictor's accuracy or coverage, two other metrics are used, the number of invalidations required relative to the control experiment, and the number of access requests required relative to the control experiment. These are shown in Figure 6.9(a) and Figure 6.9(b).

The number of invalidations required relates to the coverage. As the predictor correctly self invalidates more frequently, the number of invalidations required will decrease. The number of requests required is related to the accuracy. As the predictor incorrectly self invalidates more frequently, the number of requests will



(a) Number of Invalidations required relative to control.



(b) Number of requests required relative to control

Figure 6.9: Change in protocol performance with the implementation of the two last touch prediction schemes.

increase.

Unfortunately, neither of these correlates directly with performance. For instance, if five invalidations are required for a particular request without last touch prediction, and four are required with it present, there will likely be little benefit, aside from the small bandwidth savings. Similarly, extra requests need not increase the execution time if they are not on the critical path. Nonetheless, these numbers show that the externally trained predictor reduces the number of invalidations. (It varies slightly between the two schemes, but is not conclusively better for either.) On the other hand, the number of requests increases greatly with the standard training method. When external training is used, the number of requests returns to the level of a non-speculative protocol, or even below. This would likely make a larger speed difference in systems where the bandwidth of the interconnect was more limited than in mine.

6.1.3 Summary of Training Variation

Speculation in directory protocols has been explored for some time. Historically, this speculation occurs either independently at each cache, or centralized at the directory. Cache based prediction has the advantage of extra knowledge about the current state of the thread executing at that processor, and is ideal in terms of determining timing. Directory based schemes have the advantage of being able to see the entire system. This chapter shows that it is possible to combine these two advantages by predicting at the cache, while passing additional information between

the caches to update predictors globally.

We have seen one way that this technique can be combined with last touch prediction to achieve additional speedups of up to 12% over previous schemes. In addition, we have seen that this technique reduces the demand on the interconnect by improving the accuracy of last touch prediction. In the best case, there was a reduction of over 50% in the number of requests the directory had to process, as opposed to previous training techniques. On average there was a reduction of 17%. This form of sharing training data could be similarly applied to other forms of coherence prediction.

As Figures 6.7, 6.8 and 6.9 mention, the network latency used in these experiments was 40, rather than the 10 used for consumer prediction. At shorter network latencies last touch predictions performance drops, and can even produce slowdowns. This is particularly unfortunate because the performance of consumer prediction falls for network latencies in the range of 40, and can even produce slowdowns. Because of this the apparently promising combination of these two forms of prediction is ineffective.

6.2 Indexing Other Predictors with Last Touch Signatures

In order to move signatures between the caches in the scheme discussed above, it was necessary to transmit these data through the directory. These last touch signatures have been studied previously as a mechanism for prefetching in single processor systems [14] and as a result it seems possible that they could contain

important information for consumer prediction. This section discusses the possibility of using last touch signatures to index into the history tables used in previous sections.

6.2.1 Consumer Prediction

Consumer prediction was described in depth in a previous chapter of this dissertation. In essence, it seeks to move data to sharers before the sharers issue a request. The history table in consumer prediction can be indexed with a variety of different metrics, including the last touch signature.

Figure 6.10 shows the relative execution time of each of the different predictors studied, indexed with the program counter of the last instruction to write, or the program counter of the last touch signature used. The results are fairly neutral. While performance benefits can be obtained, they are not consistent based on prediction function. FFT and EM3D benefit with the intersection function in place, Barnes and LU benefit most with the perceptron in place, and Cholesky and Ocean benefit most with the union predictor in place. This pattern connects to the fact that the best predictor used is strongly related to the code being run.

Figure 6.11 shows the total number of invalidations transmitted with each predictor in place. In all but a few cases the total number of invalidations fell in the transition from instruction to last touch signature indexing. This means that the PVP of the predictor has risen. The behavior on the sensitivity of the system is not as clear cut.

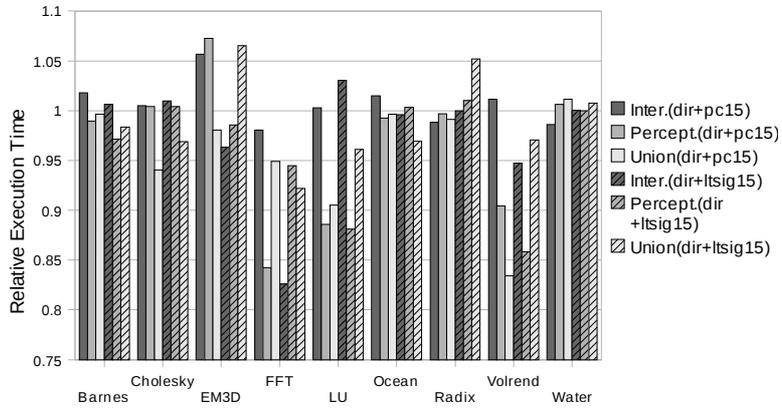


Figure 6.10: Runtime of Consumer Prediction Indexed By Last Touch Signatures

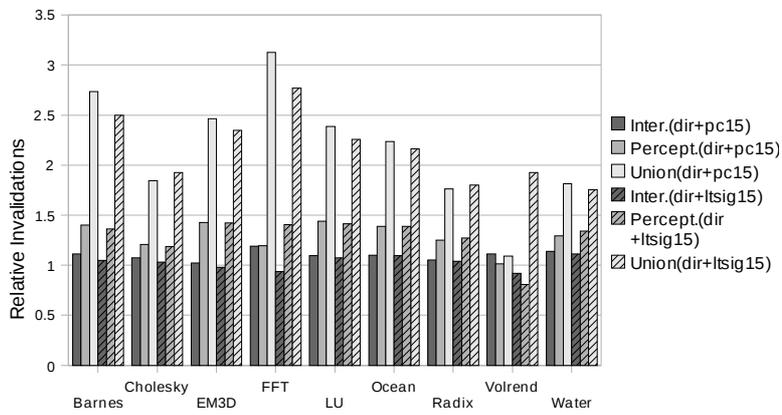


Figure 6.11: Relative Invalidations of Consumer Prediction Indexed By Last Touch Signatures

Figure 6.12 shows the total number of requests issued in the system. In a few cases there was a fall in the number of requests as well, indicating that both sensitivity and PVP improved. Barnes is the best example of this; both the number of invalidations and the number of requests fall for every predictor. However, in many cases the total number of requests actually rose.

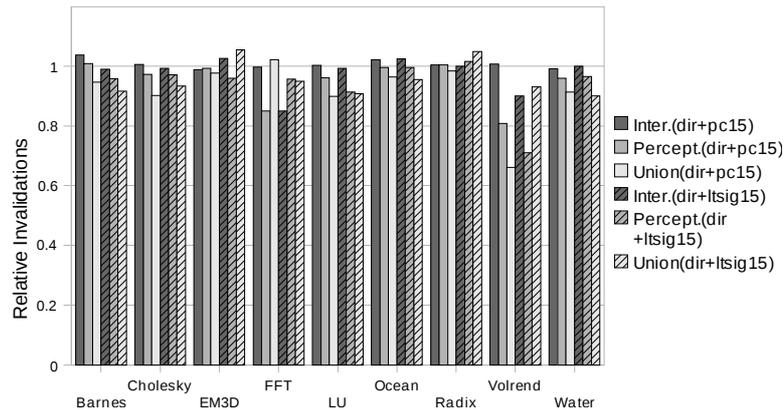


Figure 6.12: Relative Requests of Consumer Prediction Indexed By Last Touch Signatures

This is because of one of the flaws of last touch signatures as compared to the program counter of the last writer. Consider a sequence of instructions, followed by an "if-else" statement, followed by another sequence of instructions. With a last-writer based indexing scheme this codeset will always return the same value, unless the final load is within the "if-else" statement. With last touch signatures this sequence of code corresponds to two distinct values. If those two distinct paths have different sharing patterns, this will be beneficial. Given the clear benefits of this division, it is clear that in the Barnes benchmark they do have distinct sharing patterns. If they do not have different sharing patterns, this will result in increases in training time, and potentially predictors getting "out-of-date" with the current

phase of the program; we see this effect in Radix and Water.

The Barnes benchmark sees this clear improvement because of the way it subdivides “particles” at each step. Once the position of each particle is updated, it is evaluated against each of the limits of the region its processor is assigned to in order to determine if that particle might now belong to another processor. While all particles that are outside the region are not assigned to other processors, all particles that are assigned to other processors do have at least one of the if conditions in this section evaluate to true. Within each of these branches is an access to the particle itself. What this means is that the exact pattern of accesses to this branch determines which processors will share the line in the next processing phase.

By contrast, Radix has a similar structure in which a branch determines the next sharer of a line. However, in Radix, this branch also determines the last writer of the line. Thus, there is no additional information from the last touch signature, and the performance declines.

6.2.2 Migratory Prediction

The expectations from migratory prediction indexed by last touch signatures are lower than for consumer prediction. This is because migratory data are more likely to be indexed by address. The kinds of data that are migratory are less likely to change whether they are migratory, depending on their exact code flow through branch statements. However, with the goal of combining migratory and consumer prediction, it will be necessary to index the two tables with identical information.

Thus, performance impact of using last touch indexing for migratory prediction is evaluated here.

Figure 6.13 shows the results of this. In general the last touch signature has reduced the performance of the system at least slightly.

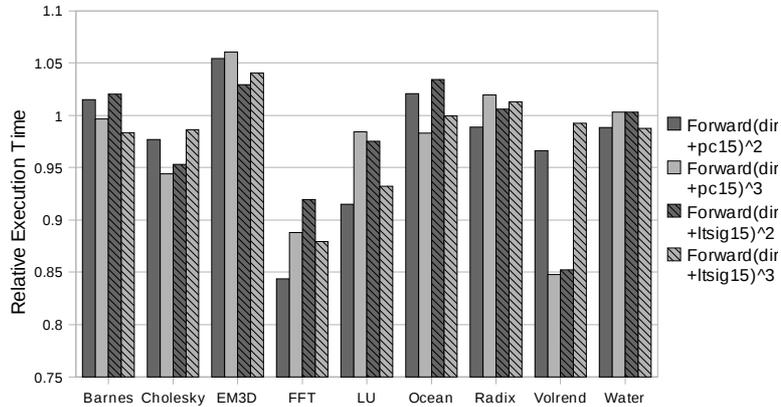


Figure 6.13: Runtime of Migratory Prediction Indexed By Last Touch Signatures

6.3 Summary

It is possible to use training information from around the system to improve the performance of the last touch predictor. This scheme benefits from the fact that the benchmarks being studied use symmetric code across the system. Thus the way one processor handles data is strongly related to the way other processors handle data.

Additionally, it is possible that the last touch signature itself can be used to improve the performance of a consumer prediction in some cases. This last touch signature carries information about the trace of instructions that accessed a line. This is beneficial in cases like Barnes, where the sharing pattern is dependent on

accesses other than the final load, but less so in other cases, where the exact path taken is not relevant.

Chapter 7

Combining Coherence Prediction Mechanisms

As implemented in the previous chapters consumer prediction has three main drawbacks. First, predictions are not necessarily made in a timely fashion. The results have shown that for several of the benchmarks, speculative data were already in flight when requests were made. This effect was present on Barnes, Cholesky, EM3D and Water.

Second, predictions cannot be made until a request for sharing permission has been made. This means that if no processors other than the writer and first requester will read the line, there is nothing for the processor to predict. This effect was present on EM3D, where most nodes that are shared are only ever accessed by two processors.

Third, additional penalties will be incurred in consumer prediction when the cache waiting for an extra request to clear the directory and a store occurs. This means that some quickly executed load-store pairs will be slowed if the data arrive at the cache after a request has been transmitted. While this is not the most frequent case, it is sufficiently common to result in net slowdowns in some cases, particularly for long network latencies.

Because of these three reasons a synergistic effect can exist between self invalidation, consumer prediction, and migratory prediction.

7.1 Potential Benefits of Combining Migratory Prediction With Consumer Prediction

One main drawback of consumer prediction is the way that it handles migratory data. Consider the example shown in Figure 7.1, that shows a migratory line and a Union based Consumer predictor.

When the line is first accessed no prediction is made. When it is accessed the second time a prediction is made, but the predicted consumer already has a Shared copy, so no action is taken. The next time the line is accessed the result of the prediction is that both of the previous owners receive a Shared copy. One of them does not have it, so the copy is forwarded. When the new processor seeks to write the line, the system must invalidate both of the processors, rather than just one. In this case that costs a small amount of extra time.

This pattern will continue for the rest of execution, with the line being constantly forwarded to the last two owners and then invalidated. If the history length was longer, the line would potentially be forwarded to more places, and thus be more likely to incur a penalty. Because of this, there is a potential benefit from merging these two predictors together.

7.2 Implementing a Joint Consumer/Migratory Predictor

Though the performance benefits of using a history table to implement migratory prediction are in some cases better than a traditional method, the original goal of the work on migratory prediction was to integrate consumer prediction and

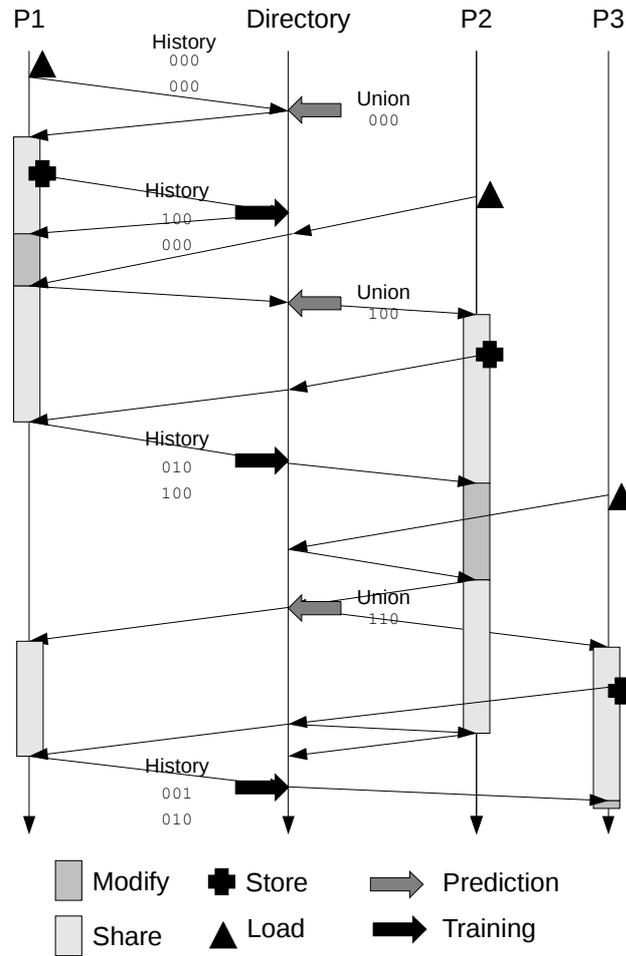


Figure 7.1: Example of negative impacts of consumer prediction in the presence of migratory data

migratory prediction. This section explores the implementation options available and demonstrate how these can be combined to produce further speedups.

A key difference between the consumer predictors and migratory predictors described so far is that the consumer predictors benefit from a long history, while the migratory predictors benefit from a short history. While it would be possible to settle with a tradeoff between the two, instead a simple strategy is implemented to correct this. The migratory predictor only operates on the first few history entries. The older entries are still available for other predictors.

The implementation for this is relatively simple, but imposes an additional limitation. Because both the migratory predictor and consumer predictor use the same history table, the two predictors must use the same indexing information. However, the previous chapters have already made it clear that these two forms of prediction react differently to indexing.

The key complication of combining these two forms of prediction is handling the cases where they disagree. For example, a line that has migrated between three different processors one at a time, would be flagged by a Union predictor to be distributed to all three of those processors. This can happen with any of the consumer prediction functions discussed earlier.

These conflicts can be resolved by giving precedence to migratory prediction. Without this, precedence, lines will never be identified as migratory. Thus consumer prediction is only applied after a line fails to be identified as migratory.

This chapter explores two distinct paths of combination. One uses only the address to index into the history table, and the other uses the last touch signature

of the line to index into the history table. No schemes are evaluated that use two separate predictor structures. Such an approach would be possible, but there would be a dramatic size increase associated with it. Such a structure would be better served by a migratory predictor specifically designed for that purpose.

7.3 Address-Indexed Joint Consumer/Migratory Prediction

Figure 7.2 shows the relative execution times of each benchmark with migratory prediction, consumer prediction, and both in place. The address-indexed migratory prediction uses Not-Migratory Counters with a threshold of two, as described earlier. For every benchmark except FFT and Cholesky, there is at least one combined predictor that outperforms the individual predictors. However, just as before the best predictor is dependent on the benchmark being evaluated.

Each of the prediction functions reacts slightly differently to being mixed with migratory prediction. The best results come from the Union predictor, under which at least one and usually both of the combined predictors outperform pure consumer prediction. This occurs because the worst case behavior of Union prediction occurs on migratory lines.

In many cases the total performance improvement for some combination of migratory and consumer prediction is better than either alone. In particular, both migratory predictors, when combined with Union prediction, result in an additional five percent speedup over the next best predictors from before. FFT benefited slightly from the depth one migratory predictor with Union prediction, but not the

depth two. Volrend, Barnes, and Cholesky on the other hand saw speedups over single predictor schemes for the perceptron predictor.

Combining these techniques can produce speedups over either individually. However, some additional mechanism is necessary to decide the predictive function to use. Without this the optimal predictor for one benchmark will result in relatively little speedup on other benchmarks.

7.3.1 Instruction-Indexed Joint Consumer/Migratory Prediction

The combined predictor detailed here uses the Second-Look Forward-Ahead predictor described earlier. In this lines are identified as migratory only after the current owner has updated them. A shared copy is then sent to the cache while the directory proactively attempts to obtain a Modify copy.

Figure 7.3 shows the effect of implementing both forms of prediction on the runtime of the final system. In general, when using the last writer to index the final table, the combination of migratory and consumer prediction performs *worse* than either of the two do individually.

The main exceptions to this rule occur for the Union predictor. For the same reasons as with an address-indexed table, migratory prediction is able to prevent the worst cases of Union prediction. However, though it is often better than either of its components individually, even this "best case" predictor is only more effective than all of the uncombined predictors on two of the benchmarks.

In general, when the code being executed on the line is used to index the table,

rather than the line itself, it is less likely that the negative case shown in Figure 7.1 will occur. This means that Union does not have as much to gain from adding migratory prediction in the case of an PC-indexed table, as opposed to an address indexed table.

Figure 7.4 and Figure 7.5 show the change in the number of invalidations and requests sent, respectively. The number of requests follows a similar trend as the runtime, with the addition of benchmarks such as Cholesky and Water, where prediction operates effectively, but the amount of benefit obtained is limited. As with both consumer prediction and migratory prediction, a reduction in the total number of requests correlates strongly with the presence of a speed-up in the total execution time. It can be seen that the combination of the two prediction schemes results in more requests and more invalidations for most of the benchmarks.

7.4 Summary

It is possible to combine a migratory and consumer predictor to achieve better performance on many benchmarks than either could individually. This combination can help prevent the worst case behavior of the system (as shown in Figure ??). On LU, a Union predictor, combined with either a depth one or two migratory predictor can provide an additional 5 percent reduction in runtime over either previous.

However, there is no clear pattern to which of the different predictors is best, combined or otherwise. While additional small speedups are possible, the exact predictor combination that attains them differs from benchmark to benchmark. This

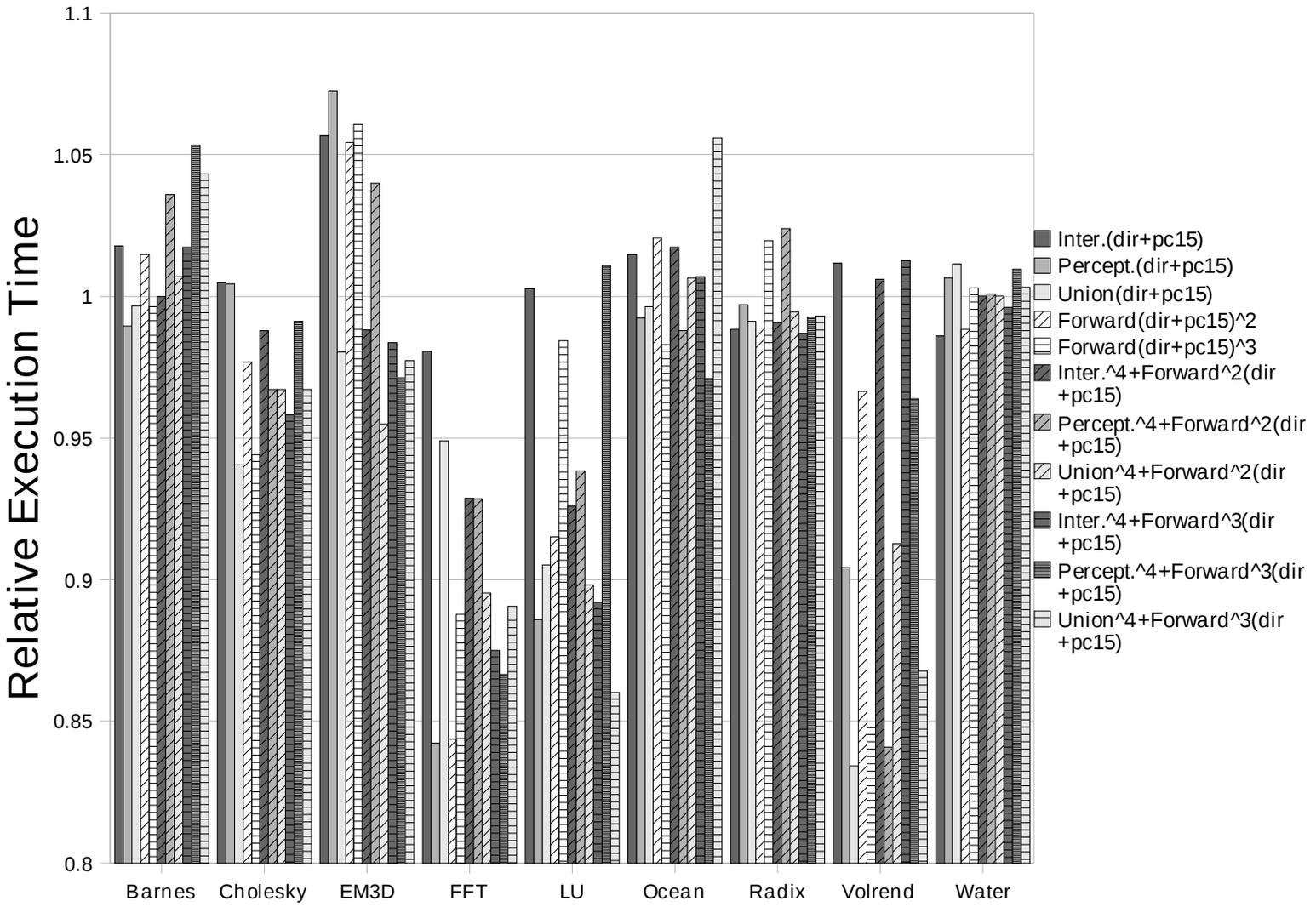


Figure 7.3: Runtime Behavior of Instruction Indexed Combined Migratory and Con-
 sumer Prediction

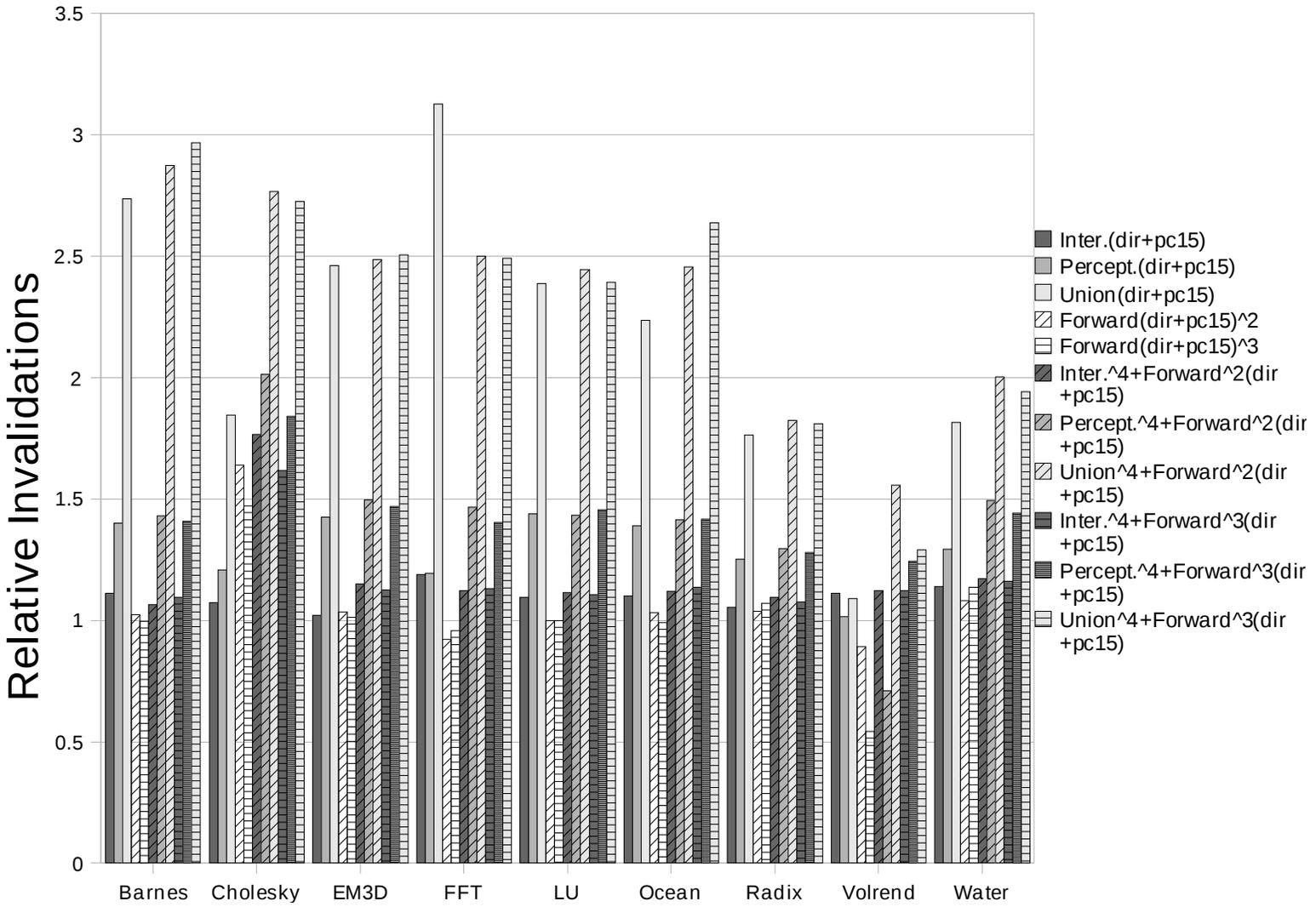


Figure 7.4: Invalidation Count of Instruction Indexed Combined Migratory and Consumer Prediction

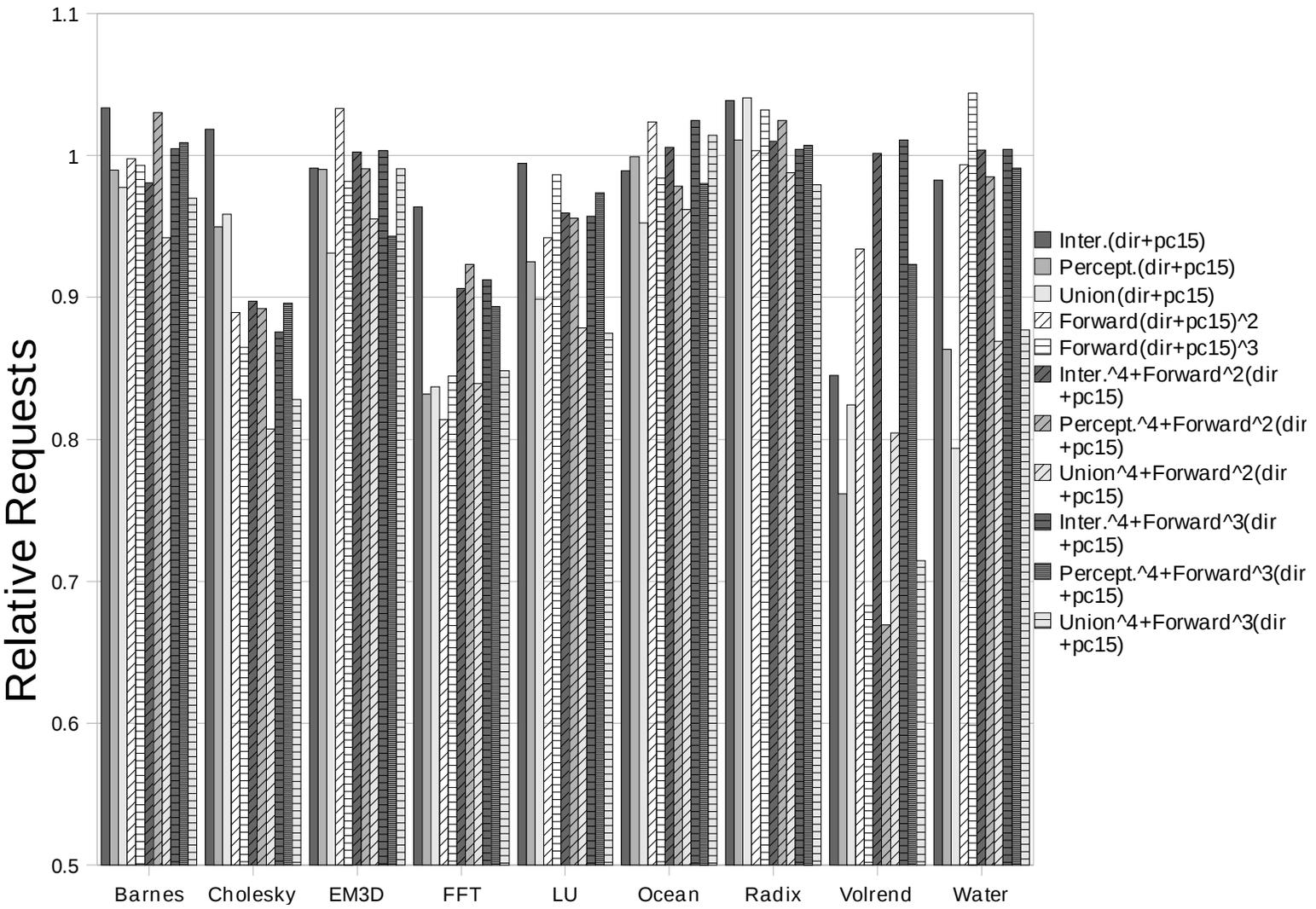


Figure 7.5: Request Count of Instruction Indexed Combined Migratory and Con-
 sumer Prediction

highlights the need for some form of hybridization, so that the correct predictor will be used at the correct time.

Chapter 8

Conclusions

8.1 Consumer Prediction

Previous work on Consumer Prediction assumed that the end goal of the predictor is to maximize either the Sensitivity of the predictor, or the Predictive Value of a Positive result. This led to a class of predictors that did not provide a tradeoff between these two. Using a perceptron based predictor it is possible to produce a predictor that can effectively provide a tradeoff between sensitivity and PVP.

Consumer prediction can be used to produce benefit to a specific class of application. In this class of application, data are regularly shared among more than two processors, and the behavior is constrained by a relatively small number of barriers, rather than frequent barriers. Given these two factors, consumer prediction is able to produce speedups of 10%-20%.

The studies detailed in this dissertation also show that an implementation that takes advantage of consumer prediction cannot achieve the speedups that the trace based results previously presented would indicate because the time at which to predict is not easily identified. The simplest way to identify the time is to wait for a request for the data. However, that means that identifying the requester will provide no speed advantages. Additionally, identifying the modifier as a consumer is not useful, as the modifier already has a copy of the line. Further, if other requests

for the line arrive while the system waits for the data, their senders cannot benefit from speculative identification. This means that Consumer Prediction's usefulness is limited to data that are shared among more than two processors, and where those data are not all requested at approximately the same time. This set of applications includes FFT and LU, as well as other tightly coupled multithreaded applications, or those with substantial amounts of widely shared data, such as Volrend.

8.2 Migratory Prediction

Migratory Prediction has been previously identified principally as a way to reduce the bandwidth load on systems, but can also produce speedups. A migratory predictor can be constructed that uses the same memory as a consumer predictor, thus allowing them to be combined at a low cost.

Migratory prediction implemented at the directory is limited by the fact that it does not have instruction information immediately. While it is possible to use this instruction information, the system must wait for the information to arrive before making a prediction. In the meantime the data can be transmitted to their destination.

Again, only a subset of benchmarks benefit from migratory prediction — those that exhibit migratory behavior that is responsible for a critical path in the system.

8.3 Timing Prediction

The benefits granted by last touch prediction occur for a different range of systems than consumer prediction. Specifically, last touch prediction is at its best for a higher network latency. By contrast, consumer prediction is best at low network latencies.

In addition, the performance of last touch prediction is harmed by full system simulation. When it was originally proposed, it ran on a simulator that did not simulate any portion of the operating system. The inclusion of the operating system harms last touch prediction by effectively mapping two distinct programs onto the same training table.

Nevertheless, last touch signatures can be used as an index to the history table of a consumer predictor. Using only the signatures, and making no last touch predictions, the overall performance of consumer prediction can be improved.

8.4 Combined Prediction

Combining consumer prediction and migratory prediction provides better results in many cases than either could individually. By allowing the migratory predictor to override the consumer predictor, the “worst case” of consumer prediction, where a migratory line is distributed to all of its previous owners every cycle, is removed. However, the exact predictor combination that is most effective varies from benchmark to benchmark. A hybrid prediction scheme might correct for this problem, causing the correct predictor to be applied to the correct benchmark.

Last touch prediction could have allowed consumer prediction to be substantially more effective by allowing the predictions to be made before any other requests. Unfortunately, the results show that last touch prediction is substantially less effective with an operating system in place. Additionally, most of the last touch events identified by last touch prediction correspond to shared copies, not Modify copies. As a result, last touch prediction is not effective at improving the performance of consumer prediction.

8.5 Future Work

This work identifies a set of benchmarks that can benefit from consumer prediction. Further, this work has shown how consumer prediction and migratory prediction can be combined at little extra cost to increase the speed further.

The main limiting factor in consumer prediction is the need to wait for a reader to request access to a line before predictions can be made. This limits its usefulness to those benchmarks where data are regularly shared with several processors. Unfortunately, last touch prediction is not sufficiently accurate to identify times to predict. Future work in this area should focus on identifying mechanisms by which the fundamental limitation of consumer prediction, waiting for a request before acting, can be bypassed.

Another future path this work could take would be adapting to prediction from a shared L2 cache to L1 caches. Future CMPs have discussed a number of different techniques for cache management. Generally, there has been discussion of having

some shared caches at the L2 or L3 level, and then unshared caches closer to the processors. From the perspective of a consumer predictor the shared cache could be treated like a directory, and the speculatively forwarded between the non-shared caches.

Bibliography

- [1] M. E. Acacio, J. Gonzalez, J. M. Garca, and J. Duato. "The Use of Prediction for Accelerating Upgrade Misses in cc-NUMA Multiprocessors". *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, Sept. 2002.
- [2] M. E. Acacio, J. Gonzalez, J. M. Garca, and J. Duato. "Owner Prediction for Accelerating Cache-to-Cache Transfers in a cc-NUMA Architecture". *Proceedings of High Performance Networking and Computing 2002*.
- [3] J. Aragon, J. Gonzalez, J. Garcia and A. Gonzalez. "Confidence Estimation for Branch Prediction Reversal". *Proceedings of the International Conference on High Performance Computing*, 2001.
- [4] R. Bianchini and T. J. LeBlanc. "Eager Combining: A Coherency Protocol for Increasing Effective Network and Memory Bandwidth in Shared-Memory Multiprocessors". *Proceedings of the Sixth IEEE Symp. on Parallel and Distributed Processing*, 1994.
- [5] M. Burtscher and B. Zorn, "Prediction Outcome History-based Confidence Estimation for Load Value Prediction." *Journal of Instruction Level Parallelism*, May 1999.
- [6] L. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems". *IEEE Transactions on Computers*, C-27(12):1112–1118, December 1978.
- [7] A. L. Cox and R. J. Fowler. "Adaptive Cache Coherence for Detecting Migratory Shared Data". *Proceedings of the 20th Annual Symposium on Computer Architecture* 1993, May 1993.
- [8] D. E. Culler, A. C. Arpaci-Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken and K. A. Yelick. "Parallel programming in Split-C." *Supercomputing*, p262-273, 1993.
- [9] F. Dahlgren, M. Dubois, and P. Stenstrom. "Combined performance gains of simple cache protocol extensions". *Proceedings of the 21st International Symposium on Computer Architecture*, Apr. 1994.
- [10] D. DeWitt and J. Gray. "Parallel Database Systems: The Future of High Performance Database Systems." *Communications of the ACM*, 35(6), 1992.

- [11] M. Dubois, J. C. Wang, L. A. Barroso, K. L. Lee, and Y. Chen. "Delayed Consistency and its Effect on the Miss Rate of Parallel Programs". *Proceedings of the 1991 Conference on Supercomputing*.
- [12] M. Dubois, F. Dahlgren, and P. Stenstrom, "Sequential hardware prefetching in shared-memory multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, 1995.
- [13] B. Falsafi and D. A. Wood. "Reactive NUMA: A Design for Unifying S-COMA and CC-NUMA". *Proceedings of the 24th Annual International Symposium on Computer Architecture*, June 1997.
- [14] M. Ferdman and B. Falsafi. "Last-Touch Correlated Data Streaming." *IEEE International Symposium on Systems and Software*, April 2007.
- [15] J. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic." *Proceedings on the 10th International Symposium on Computer Architecture*, 1983.
- [16] W. Gropp and E. Lusk, "A taxonomy of programming models for symmetric multiprocessors and SMP clusters," *Programming Models for Massively Parallel Computers*, 1995, pp. 2–7.
- [17] M. Gschwind, "Chip Multiprocessing and the Cell Broadband Engine". *Computing Frontiers 2006*, May 2006.
- [18] J. Huh, J. Chang, D. Burger, and G. S. Sohi. "Coherence Decoupling: Making Use of Incoherence". *Proceedings of the Eleventh International Conference on Architectural Support for Programming Languages*, October 2004.
- [19] S. Kaxiras and C. Young. "Coherence Communication Prediction in Shared-Memory Multiprocessors". *Proceedings of the 6th International High Performance Computer Architecture*, January 2000.
- [20] S. Kaxiras and J. R. Goodman, "Improving Request-Combining for Widely-shared Data in Shared-Memory Multiprocessors". *Proceedings of the Third International Conference on Massively Parallel Computing Systems*, April 1998.
- [21] S. Kaxiras, S. Gjessing and J. Goodman, "A Study of Three Dynamic Approaches to Handle Widely Shared Data in Shared-Memory Multiprocessors". *Proceedings of the 12th International Conference on Supercomputing*, 1998.

- [22] S. Kaxiras and J. R. Goodman. "Improving CC-NUMA performance using instruction-based prediction". *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*, Jan. 1999.
- [23] Stefanos Kaxiras, "Kiloprocessor Extensions to SCI." *Proceedings of the 10th International Parallel Processing Symposium*, April 1996.
- [24] Stefanos Kaxiras and James R. Goodman. "The GLOW cache coherence protocol extensions for widely shared data". *Proceedings of the 10th International Conference on Supercomputing*, May 1996.
- [25] D. M. Koppelman, "Neighborhood prefetching on multiprocessors using instruction history," *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, 2000.
- [26] R. Kumar, V. Zyuban, D. Tullsen. "Interconnections in Multe-core Architectures: Understanding Mechanisms, Overheads and Scaling," *Computer Architecture News*, 2005.
- [27] A. Lai and B. Falsafi. "Memory sharing predictor: The Key to a Speculative Coherent DSM". *Proceedings of the 26th Annual International Symposium on Computer Architecture (ISCA'99)*, May 1999.
- [28] A. Lai and B. Falsafi. "Selective, Accurate, and Timely Self-Invalidation Using Last-Touch Prediction". *Proceedings of the 27th Int'l Symposium on Computer Architecture*, May 2000.
- [29] Alvin R. Lebeck and David A. Wood, "Dynamic self-invalidation: reducing coherence overhead in shared-memory multiprocessors." *Proceedings of the 22nd annual international symposium on Computer architecture*.
- [30] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, and J. Hennessy. "The Directory Based Cache Coherence Protocol for the DASH Multiprocessor". *Proceedings of the 17th Annual International Symposium on Computer Architecture*, 1990.
- [31] S. Leventhal and M. Franklin. "Perceptron Based Consumer Prediction in Shared-Memory Multiprocessors". *Proceedings of the 24th International Conference on Computer Design*, 2006.
- [32] M. M.K. Martin, M. D. Hill, and D. A. Wood. "Token Coherence: Decoupling Performance and Correctness". *Proceedings of the 30th International Symposium on Computer Architecture*, 2003 (ISCA 2003).

- [33] M. M. K. Martin, P. J. Harper, D. J. Sorin, M. D. Hill, and D. A. Wood. "Using Destination-Set Prediction To Improve The Latency/Bandwidth Tradeoff in Shared Memory Multiprocessors". *Proceedings of the 30th International Symposium on Computer Architecture*, June 2003.
- [34] M. M.K. Martin, D. J. Sorin, M. D. Hill, and D. A. Wood. "Bandwidth Adaptive Snooping". *Proceedings of the Eighth IEEE Symposium on High-Performance Computer Architecture*, January 2002.
- [35] M.K. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset." *Computer Architecture News (CAN)* 2005.
- [36] M. R. Marty, J. D. Bingham, M. D. Hill, A. J. Hu, M. M.K. Martin and D. A. Wood, "Improving Multiple-CMP Systems Using Token Coherence". *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.
- [37] T. Mowry and A. Gupta. "Tolerating Latency Through Software-Controlled Prefetching in Shared-Memory Multiprocessors" *Journal of Parallel and Distributed Computing*, June 1991.
- [38] S. S. Mukherjee and M. D. Hill. "Using Prediction to Accelerate Coherence Protocols". *Proceedings of the 25th International Symposium on Computer Architecture*, June 1998.
- [39] J. Nilsson, A. Landin and P. Stenström, *The Coherence Predictor Cache: A Resource-Efficient and Accurate Coherence Prediction Infrastructure*. In Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS03.
- [40] R. Rajwar, A. Kagi and J. R. Goodman, "Inferential Queueing and Speculative Push for Reducing Critical Communication Latencies". *Proceedings of the 17th Annual International Conference on Supercomputing*, 2003.
- [41] P. Ranganathan, V. Pai, H. Abdel-Shafi, and S. Adve. "The interaction of software prefetching with ILP processors in shared-memory systems". *Proceedings of the 24th International Symposium on Computer Architecture*, June 1997.
- [42] S. Somogyi, T. F. Wenisch, N. Hardavellas, J. Kim, A. Ailamaki and B. Falsafi. "Memory Coherence Activity Prediction In Commercial Workloads". *Proceedings of the 3rd Workshop on Memory Performance Issues*, 2004

- [43] D. J. Sorin, M. M. K. Martin, M. D. Hill and D. A. Wood. "Using Speculation to Simplify Multiprocessor Design". *Proceedings of the 18th International Parallel and Distributed Processing Symposium*, 2004.
- [44] L. Spracklen and S. Abraham, "Chip Multithreading: Opportunities and Challenges." *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.
- [45] P. Stenstrom, M. Brorsson and L. Sandberg, "An Adaptive Cache Coherence Protocol Optimized for Migratory Sharing". *Proceedings of the 20th Annual International Symposium on Computer architecture*, 1993.
- [46] P. Stenstrom, E. Hagersten, D. Lilja, M. Martonosi, and M. Venugopal, "Trends in Shared Memory Multiprocessing." *IEEE Computer*, pages 44-50, December, 1997.
- [47] W-D. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Multiprocessors", *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Systems*, April 1989.
- [48] T. F. Wenisch, S. Somogyi, N. Hardavellas, J. Kim, A. Ailamaki and B. Falsafi, "Temporal Streaming of Shared Memory". *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, 2005.
- [49] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. "The SPLASH-2 Programs: Characterization and Methodological Considerations." *Proceedings of the 22nd International Symposium on Computer Architecture*, 1995.
- [50] Z. Zhang and J. Torrellas. "Speeding up irregular applications in shared-memory multiprocessors: Memory binding and group prefetching." *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, June 1995.
- [51] Intel Pentium Processor Extreme Edition 840 Δ . Intel Datasheet 306831-002. October 2005.
- [52] June 2006 Top 500 Supercomputing Sites. <http://www.top500.org/lists/2007/11>.
- [53] AMD Athlon 64 X2 Dual-Core Processor Product Data Sheet. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/33425.pdf

- [54] J. Brown. "Just like being there: Papers from the Fall Processor Forum 2005: Application-customized CPU design: The Microsoft Xbox 360 CPU story," <http://www-128.ibm.com/developerworks/power/library/pa-fpfxbox/?ca=dgr-lnxw09XBoxDesign>, September 2006.