Compilation-based Prefetching for Memory Latency Tolerance

by

Charles William Selvidge
M.S., Massachusetts Institute of Technology (1987)
B.S., Massachusetts Institute of Technology (1985)

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfullment
of the Requirements for the Degree of

Doctor of Philosophy

at the

Massachusetts Institute of Technology

May, 1992

Signature of Author [illegible]
Department of Electrical Engineering and Computer Science
18 May 1992

Certified by [illegible]
Stephen A. Ward
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by [illegible]
Campbell L. Searle
Chairman, Departmental Committee on Graduate Students

Compilation-based Prefetching for Memory Latency Tolerance

by

Charles W. Selvidge

Abstract

Performance impacts of memory latency can be mitigated with hierarchical memory systems which exploit locality and with concurrent memory systems which exploit parallelism. As the relative performance of processors improves with respect to memory components it will be advantageous to simultaneously exploit locality and parallelism in order to achieve memory latency tolerant computation, computation in which the runtime performance is relatively insensitive to high memory latencies.

Program locality is typically considered at a macroscopic level corresponding to aggregate program behavior. We investigate locality behavior at a finer granularity, considering the behavior of individual memory reference instructions. By grouping together the dynamic memory references produced by each static memory reference instruction in a program, a large amount of structure is exposed in locality behavior. Miss rates of dynamic memory references arising from the same static instruction are highly correlated; thus it is meaningful to ascribe an individual miss rate to each static memory reference in a program. These individual miss rates are highly polarized, often displaying a bimodal distribution consisting of a set of references with very low miss rates and a set with very high miss rates. As a result of this polarized, bimodal distribution, one finds that essentially all program misses occur as a result of executing an instruction from the set of static memory reference instructions exhibiting high miss rates. Low miss rate static instructions account for a large fraction of total memory references, however. This behavior is summarized in the **Static Locality Correlation Hypothesis** and the **Badref Hypothesis**. Empirical evidence validates these hypotheses.

Prefetching provides a mechanism for achieving concurrent activity in processors, caches and main memory systems. While this concurrency can produce latency tolerance, it requires program parallelism. Operations cannot be performed concurrently if they are ordered by dataflow dependencies. The insights above regarding program locality behavior can be exploited to leverage limited program parallelism. By restricting the focus of latency tolerance specifically to the subset of high miss rate static memory reference instructions, parallelism is targetted at the source of misses. Furthermore, low miss rate memory references eliminated from consideration become an additional source of parallelism for high miss rate references. The non-semantic nature of transactions between memory and cache serves to further increase parallelism for memory latency tolerance. An empirical measurement technique demonstrates that ample parallelism is displayed in programs to allow tolerance to memory latencies of hundreds of instructions.

These results are put in practice in a prototype compiler which generates memory latency tolerant code. Two distinct scheduling algorithms are developed and implemented in order to schedule cache miss processing activity to occur concurrently with computation and cache hits. One technique applies to structured accesses within loops, predicting future addresses in order to software pipeline cache miss latency with loop computation. The second technique merges prefetches into sequential code. Simulations of code produced by the prototype compiler demonstrate speedups of 10 to 30% over unmodified code for a 20 cycle memory latency and speedups of factors of 2 to 6 over unmodified code for a memory system with 160 cycle latency and bandwidth sufficient to allow 8 concurrent miss transactions.

Thesis Supervisor: Stephen A. Ward
Title: Professor of Electrical Engineering and Computer Science

Acknowledgements

I thank my wife Maureen for love, understanding, encouragement and much patience. Pursuit of a PhD has emotional ups and downs. The ups are focused at the end while the downs are distributed through the beginning and middle. Maureen helped me make it to the end.

I thank Steve Ward, my advisor, for providing me with a unique research opportunity. He allowed me great flexibility in the exploration of this topic while providing enough subtle direction to make sure I eventually finished. His many insights into the field of computer science as a whole and regarding the interactions of compilation and architecture, in particular, have helped form the basis of my own views.

A discussion with Greg Papadopoulos provided a catalytic moment for my thesis, helping to transform a set of vague ideas to a specific plan of action. This represented both the end of the beginning and the beginning of the end. Anant Agarwal, as a potential consumer of this type of technology, helped keep my work focused and realistic. I am grateful to both Greg and Anant for their participation on my thesis committee.

Andy Ayers has been my main sounding board for ideas and principle comiserant during the last 5 years. He also read a complete draft of the thesis and provided many useful comments. Larry Dennison provided me with constant prodding and encouragement to finish and graciously reviewed the document once I did.

Arthur Altman, my officemate, provided many needed diversionary discussions on work, politics and life.

Anne McCarthy and Sharon Thomas, each in their own way, oiled the wheels of progress in the research group in which I worked during this effort.

Over the years, students and staff have come, gone, and somtimes come again or changed from one to the other. Ken, Milan, John N., Pz, Russ, DG, Kubi, David Kranz and David Chaiken, Beng, Don, Mike, Peter, Steve and Rich all currently make the 6th floor of Tech Square a more interesting and enjoyable place. Joe and Joe, Cailein, Steve and Andy C. did the same during their stays.

Friends from BBN including Phil Carvey, Greg Bromley, Jerry Pianin and John Goodhue have encouraged me through the years.

Finally, I thank my parents for believing that I could finish and my brother and sister for graduating at the same time, thereby forcing me to do so as well.

Table of Contents

1 Introduction 10
- 1.1 Preview of Technical Contributions and Results 10
  - 1.1.1 Static Locality Behavior 10
  - 1.1.2 Parallelism Behavior 12
  - 1.1.3 Compiler Latency Tolerance Algorithms 14
  - 1.1.4 Performance Evaluation 14
- 1.2 Thesis Roadmap 15

2 Motivation and Background 16
- 2.1 The Memory Latency Problem 16
  - 2.1.1 Processors 19
  - 2.1.2 Memory 19
  - 2.1.3 Demand-Fetched Caching 19
  - 2.1.4 Interleaved Memory Pipelines 20
  - 2.1.5 Memory Interface Issues 21
  - 2.1.6 Memory Latency 22
- 2.2 Hardware Latency Tolerance Mechanisms 22
  - 2.2.1 Multilevel Caches 22
  - 2.2.2 Cache Prefetching 23
  - 2.2.3 Dynamic Scheduling 24
- 2.3 Software Memory Latency Tolerance 25
  - 2.3.1 Latency Tolerance Strategy 25
  - 2.3.2 Experimental Approach 26
- 2.4 Previous Work 27
  - 2.4.1 Cache Behavioral Simulation and Modelling 27
  - 2.4.2 Parallelism Measurement Techniques 27
  - 2.4.3 Compilation Techniques for Enhancing Locality 28
  - 2.4.4 Compilation Techniques for Memory Latency Tolerance 28
  - 2.4.5 Inline Profiling and Profile-based Compiler Optimization 29
  - 2.4.6 Code Scheduling for Hardware Concurrency 30
  - 2.4.7 Speculative Computation 30
  - 2.4.8 Analysis of Software Prefetching Behavior 31
- 2.5 Hardware Support for Software Latency Tolerance 31
  - 2.5.1 Memory System Support for Software Latency Tolerance 31
  - 2.5.2 Instruction Set Access to Latency Tolerance Mechanisms 32
  - 2.5.3 Processor Memory Interface Issues 33
  - 2.5.4 High-Bandwidth Memory Systems 34
  - 2.5.5 Target Hardware Model 35

3 Static Locality Correlation and the Badref Model 36
- 3.1 Memory Statistics of Individual Instruction 37
- 3.2 Exploitation of Static Locality Correlation 39
- 3.3 Program Structure 41
- 3.4 An Engineer's Description of Static Locality Correlation and the Badref Model 42
- 3.5 Validation of the Static Locality and Badref Hypotheses 44
  - 3.5.1 Miss Rate Variability of Individual Static References 44
  - 3.5.2 Cache Design Perturbations 50
  - 3.5.2.1 Impact of Cache Size 50
  - 3.5.2.2 Impact of Block Size 55
  - 3.5.2.3 Effects of Associativity and Conflict Misses 60

3.5.3 Perturbations From Data Set Variation 65
3.6 Summary 65

4 Parallelism For Memory Latency Tolerance 67
4.1 A Model of Parallelism and Latency Tolerance 67
4.1.1 Constraints on Parallelism 68
4.1.2 Dependence Constraints, Cache Misses and Speculation 70
4.2 Experimental Technique 71
4.3 Results of Parallelism Measurements 73
4.3.1 Parallelism 73
4.3.2 Forward versus Reverse Parallelism 77
4.3.3 Badref Measurements 81
4.4 Summary 85

5 Compiler Algorithms, Models and Heuristics 87
5.1 Memory Reference Behavioral Modelling 87
5.1.1 Badref Identification Techniques 88
5.1.1.1 Analysis: The Path Not Taken 89
5.1.1.2 Miss Rate Measurement in C-flat 90
5.1.2 Memory System Modelling 91
5.2 Latency Tolerance Optimization 92
5.3 Loop-based Prefetching Heuristics 93
5.3.1 RAAD Identification Algorithm 93
5.3.2 Array RAADs 95
5.3.3 Indirection Array RAADs 98
5.3.4 Linked-List RAADs 101
5.3.5 Prefetch Iteration Count Selection 104
5.3.6 Cache-block Based Optimizations 106
5.3.7 Prologue Prefetching 109
5.4 RAAD Prefetching in Conditional Loops 110
5.4.1 Conditional Strategy 112
5.4.2 Badref and IV Update on a Common Conditional Path 112
5.4.3 Conditionally Executed Badref with Unconditional IV Update . . 113
5.4.4 Conditionally Updated IV: Single Update 115
5.4.5 Conditionally Updated IV: Multiple Updates 116
5.5 Loop Dependence Analysis 117
5.6 Sequential Miss Scheduling 118
5.6.1 Sequential Miss Scheduling Algorithm 120
5.6.1.1 Address Interactions 120
5.6.1.2 Miss Latency Interactions 122
5.7 Choice of RAAD Prefetching or Miss Scheduling for Loops 122
5.7.1 Optimal Loop Scheduling 124
5.8 Summary 124

6 Compiler Results and Evaluation 126
6.1 Characterization of Badrefs 126
6.2 Optimized Code Size 130
6.3 Dynamic Performance 131
6.3.1 Processor and Interface Models 131
6.3.2 Memory System Modelling 132
6.3.3 Time Accounting 133
6.3.4 Performance Results 133
6.3.4.1 **FA** Data 134

6.3.5 DM Data . . . 139

6.3.5.1 Cache Pollution . . . 141

6.3.5.2 Memory Behavior Perturbation . . . 144

6.4 Summary . . . 145

7 Conclusions 147

7.1 Summary . . . 147

7.2 Program Behavior . . . 148

7.2.1 Static Locality Correlation . . . 148

7.2.2 Implications of Static Locality Correlation and Badref Behavior . 149

7.2.3 Parallelism . . . 149

7.2.4 Parallelism Measurement . . . 150

7.2.5 Implications of Forward vs Reverse Parallelism . . . 150

7.2.6 Badref Distribution . . . 151

7.3 Compiler Issues . . . 151

7.3.1 Compiler Memory Modelling . . . 151

7.3.2 Scheduling Algorithms . . . 152

7.3.3 Characterization of Badrefs . . . 152

7.3.4 Variations of Static Locality Correlation . . . 152

7.4 Compiler Performance . . . 153

7.5 Epilogue . . . 153

A Additional Static Locality Correlation Data 154

A.1 Baseline Locality Correlation Data . . . 154

Table of Figures

Figure 2.1	Component Performance vs. Time	16
Figure 2.2	Normalized Latencies vs. Time	17
Figure 3.1	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Single Hit Rate	37
Figure 3.2	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Two Distinct Hit Rates	37
Figure 3.3	$R_c(m)$ and $M_c(m)$ for Badref Behavior	38
Figure 3.4	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Doduc **FA**, **DM** and **DMV**	45
Figure 3.5	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Espresso **FA**, **DM** and **DMV**	46
Figure 3.6	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Tomcatv **FA**, **DM** and **DMV**	47
Figure 3.7	Doduc $R_c(m)$ and $M_c(m)$ variation with cache size	50
Figure 3.8	Espresso $R_c(m)$ and $M_c(m)$ variation with cache size	51
Figure 3.9	Fppp $R_c(m)$ and $M_c(m)$ variation with cache size	52
Figure 3.10	Tomcatv $R_c(m)$ and $M_c(m)$ variation with cache size	53
Figure 3.11	Doduc $R_c(m)$ and $M_c(m)$ variation with block size	55
Figure 3.12	Espresso $R_c(m)$ and $M_c(m)$ variation with block size	56
Figure 3.13	Fppp $R_c(m)$ and $M_c(m)$ variation with block size	57
Figure 3.14	Tomcatv $R_c(m)$ and $M_c(m)$ variation with block size	58
Figure 3.15	Doduc $R_c(m)$ and $M_c(m)$ variation with associativity	60
Figure 3.16	Espresso $R_c(m)$ and $M_c(m)$ variation with associativity	61
Figure 3.17	Fppp $R_c(m)$ and $M_c(m)$ variation with associativity	62
Figure 3.18	Tomcatv $R_c(m)$ and $M_c(m)$ variation with associativity	63
Figure 3.19	Miss Rate Variation with Input Data Perturbations in Espresso	65
Figure 4.1	Dataflow Constraints for Example 1	68
Figure 4.2	Dataflow Constraints for Example 2	68
Figure 4.3	Dataflow Constraints for Example 3	69
Figure 4.4	Windowed and Basic Block Parallelism for doduc, espresso and eqntott	73
Figure 4.5	Windowed and Basic Block Parallelism for fpppp, matrix300 and nasa7	74
Figure 4.6	Windowed and Basic Block Parallelism for tomcatv	75
Figure 4.7	Reverse and Forward Parallelism for doduc, espresso and eqntott	78
Figure 4.8	Reverse and Forward Parallelism for fpppp, matrix300 and nasa7	79
Figure 4.9	Reverse and Forward Parallelism for tomcatv	80
Figure 4.10	Windowed Badref Counts for doduc, espresso and eqntott	82
Figure 4.11	Windowed Badref Counts for fpppp, matrix300 and nasa7	83
Figure 4.12	Windowed Badref Counts for tomcatv	84
Figure 5.1	Characteristic IVs and SCCs in C-flat	93
Figure 5.2	RAADs in C-flat	94
Figure 5.3	Block Alignments of 5 word Data Structure	106
Figure 6.1	Doduc **FA** Performance	134
Figure 6.2	Eqntott **FA** Performance	134
Figure 6.3	Espresso **FA** Performance	135
Figure 6.4	Fpppp **FA** Performance	135
Figure 6.5	Matrix300 **FA** Performance	136
Figure 6.6	Nasa7 **FA** Performance	136
Figure 6.7	Tomcatv **FA** Performance	137
Figure 6.8	Doduc **DM** Performance	139
Figure 6.9	Eqntott **DM** Performance	139

Figure 6.10	Espresso **DM** Performance	140
Figure 6.11	Fpppp **DM** Performance	140
Figure 6.12	Matrix300 **DM** Performance	141
Figure 6.13	Nasa7 **DM** Performance	141
Figure 6.14	Tomcatv **DM** Performance	142
Figure A.1	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Eqntott **FA**, **DM** and **DMV**	154
Figure A.2	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Fppp **FA**, **DM** and **DMV**	155
Figure A.3	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Matrix300 **FA**, **DM** and **DMV**	156
Figure A.4	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Nasa7 **FA**, **DM** and **DMV**	157
Figure A.5	$R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Spice **FA**, **DM** and **DMV**	158

Table of Figures

Table 6.1	Badref Categorization for Doduc and Eqntott	126
Table 6.2	Badref Categorization for Espresso and Fpppp	127
Table 6.3	Badref Categorization for Matrix300 and Nasa7	127
Table 6.4	Badref Categorization for Spice and Tomcatv	128
Table 6.5	Fractional Increase in Optimized Code Size	130
Table 6.6	Ratios of Benchmark Speedups	137

Chapter 1

Introduction

Technological trends motivate a desire for memory latency tolerance mechanisms in computer system architectures. Memory latency tolerance in an architecture describes the ability to compute efficiently given an everincreasing ratio, over the lifetime of the architecture, between the latency required to access data from slow components of the memory and the time to perform elementary computations within the processor. Efficient computation in the face of high memory latency can be achieved by exploiting locality and parallelism inherent in a system's workload. Hierarchical, cache-based memory systems which dynamically exploit program locality have been the principle architectural tool for dealing with memory latency. Cache misses result in a residual memory latency component in cached systems. This thesis investigates the application of explicit, compilation-based management of processor/memory system concurrency, thereby exploiting parallelism to achieve tolerance to cache-miss latency.

1.1. Preview of Technical Contributions and Results

Cache-miss latency is a potential performance limitation in computer systems. Our contributions with respect to the problem of cache miss latency tolerance can be divided into four main categories.

- We identify, characterize and measure an important new aspect of the locality behavior of programs termed static locality correlation.
- We characterize and measure program parallelism relevant in the context of memory latency tolerance.
- We devise and implement scheduling algorithms which can be used by a compiler to produce latency tolerant code. These algorithms exploit our behavioral observations concerning locality and parallelism.
- We measure the performance of mechanically-generated, latency-tolerant code. This serves to demonstrate both the feasibility and effectiveness of compilation-based latency tolerance.

1.1.1 Static Locality Behavior

Architectural advances in computing, in contrast to technological advances, often arise from improved understanding of the behavior of programs. The study and identification of regularities in runtime program behavior allows better application of available technology. Analysis and characterization of program behavior relevant to memory latency and its tolerance leads to several insights which can be exploited in latency tolerant compilation.

The principle aspect of program behavior impacting memory performance in cached systems is locality. We make several related observations about the locality behavior of programs which are extremely important in the context of latency tolerance.

Distinct memory references at runtime which are produced by the same static instruction exhibit correlated locality behavior. Since caches adaptively modify their contents to reflect recent memory references, the hit/miss behavior of a particular memory reference at runtime is determined by the relationship of the addresses it uses to the addresses used by neighboring references. The juxtaposition of memory references does not occur by chance but instead occurs as a consequence of the structure of programs. Every time a particular memory reference instruction is executed the neighboring dynamic memory references are produced by the same neighboring static instructions. Thus the set of dynamic memory references produced by a particular static memory reference instruction all occur in a similar context, created by its neighboring instructions. As a consequence, these dynamic references are likely to exhibit the same hit/miss behavior systematically. Their locality behavior is correlated.

For many programs, the particular context in which references occur is independent or only weakly dependent on input data. References produced by the same instruction, even across multiple program runs, exhibit similar behavior. In contrast, even in the same program run, references arising from different static memory reference instructions have different contexts, each arising from its own local program structure. As a result, there is no reason to expect the behavior of dynamic references arising from different static instructions to be strongly correlated.

This behavior is elucidated through the **Static Locality Correlation Hypothesis.** This hypothesis states that program structure is a primary determinant of locality behavior. The hit/miss behavior of each individual static references can be characterized across multiple program runs while the behavior of different static references, even within a single program run, can vary widely.

The principle consequence of static locality correlation is to justify the distinct characterization of the hit/miss behavior of each static memory reference in a program with an individual miss rate. In the absence of static locality correlation the behavior of all static memory references would be essentially equivalent. This is simply not the way programs behave. Hit/miss behavior varies significantly between different static references and can be individually ascribed to each reference. Individual characterization of locality behavior of references is extremely useful in the context of generation of latency tolerant code.

Building on the static locality hypothesis, additional aspects of program locality are observed. Program structure frequently guarantees that some static memory reference instructions cannot produce misses by construction. Given that some fraction of static references produce no misses, it is an inevitable consequence for programs with a non-zero average miss rate that some other static references must have individual miss rates which exceed the program average. We observe two aspects regarding the distribution of miss rates of individual static memory references. First, static reference miss rate behavior is highly polarized, exhibiting some references with very low miss rates and some with relatively high miss rates. Second, far more dynamic references are produced by low miss rate references than high miss rate references, in most cases. The **Badref Hypothesis** encapsulates these observations regarding polarization of miss rates and distribution of references by instruction miss rate.

The **Badref Hypothesis** observes that static memory references can be partitioned into a low miss rate set, goodrefs, and a high miss rate set, badrefs. Goodrefs, static references with negligibly small miss rates, often account for a significant fraction of all dynamic memory references, between 60 and 90% or even more. Badrefs typically account for a much smaller fraction of dynamic references. Notice that 100% miss rate static references can account for a fraction of dynamic references no greater than the average program miss rate. Thus most dynamic references occur in conjunction with the execution of goodrefs.

The number of misses produced by a particular static reference is the product of the number of references and the miss rate exhibited by the reference. Even a large set of dynamic references produce zero or very few misses if they are produced by a static reference with a very low miss rate. A relatively small set of dynamic references which miss very frequently compared to the program average may well account for the preponderance of program misses. In fact, in many programs, this is precisely the behavior exhibited. Badrefs, a subset of static references with high miss rates, produce a small set of dynamic references while accounting for most of the misses experienced by programs. Notice that this behavior reflects the three distinct aspects described. Not only do static references exhibit distinct miss rates but these miss rates are polarized far from the program average. Furthermore, the dynamic size of the low miss rate set dominates that of the high miss rate set, but not by enough that misses produced by the low miss rate set represent a signficant fraction of total misses.

These insights into memory behavior can be exploited in latency-tolerant compilation. Static locality correlation suggests individualized behavior modelling of each memory reference in a program, rather than the uniform treatment accorded all memory references in most compilers. Based on the polarization of miss rates described by the **Badref Hypothesis**, we develop a compiler model for memory system behavior, the **Badref Model**. This model partitions memory references into two sets, goodrefs and badrefs. Goodrefs are modelled as cache hits while badrefs are modelled as cache misses. The predominance of goodrefs in many programs serves to focus the problem of miss latency tolerance onto a small set of memory references, greatly simplifying the development of compiler techniques for scheduling this latency. Exploitation of these phenomena enables the practical application of software-scheduled concurrency as a memory latency tolerance technique.

With regard to understanding and exploiting locality behavior of programs, this thesis makes several distinct contributions. We develop and elucidate the concept of static locality correlation and the further refinement regarding the polarization of this behavior in the static locality and badref hypotheses. These insights are supported through qualitative and quantitative empirical evidence. Qualitative evidence takes the form of examples of frequently occurring program structures which lead both to goodrefs and badrefs. Quantitative evidence takes the form of benchmark simulation studies for a set of programs across a variety of cache configurations and focused studies addressing the impacts on badref behavior of changing cache design parameters including cache size, block size and the level of cache conflict misses. Finally, we demonstrate the technique by which this behavior can be exploited for latency tolerant code generation in the form of the badref model.

1.1.2 Parallelism Behavior

Concurrency between the servicing of miss transactions and other computation and/or concurrency between the servicing of multiple misses is a means for achieving latency tolerance. Concurrent processing of distinct operations in programs requires parallelism or independence between the operations. Investigation of program behavior turns then from locality to parallelism. Do programs exhibit sufficient, accessible parallelism that software latency tolerance is a practical technique?

Measurements of instruction-level parallelism in the literature are typically relatively low, between two and five. These numbers should not be interpreted to mean that memory latencies beyond five cannot be tolerated through concurrency. Average instruction-level parallelism is simply not a suitable metric for evaluating parallelism for memory latency tolerance.

To allow complete concurrency between a processor and memory during the servicing of a 20 cycle long miss transaction, 20 instructions must be identified which are independent of the miss transaction. These instructions cannot produce data needed to initiate the miss transaction and similarly cannot use data fetched from memory by the miss transaction. It is by no means necessary for these instructions to be independent of one another, however. In fact, a linear sequence of 20 mutually dependent instructions is quite suitable for overlap with a 20 cycle miss transaction as long as each instruction in the sequence is independent of the miss. Such a construction involving a single miss operation and 20 dependent instructions, each independent of the miss, might be characterized as exhibiting an average parallelism of 1.05.

The observation above regarding parallelism can be coupled with the behavior described by the badref hypothesis, namely the fact that miss latency occurs primarily in conjunction with a small set of identifiable, memory reference instructions. This conjunction of ideas is essential. The key to finding adequate parallelism to tolerate very high memory latencies lies in the specific ability to pinpoint this latency to a small, fixed and identifiable set of static memory references. Having isolated the sources of misses, limited program parallelism can be devoted exclusively to masking the associated latency, rather than squandered in untargetted efforts to tolerate potential latency for all memory references. Moreover, goodrefs, those memory reference instructions which produce no misses, become an additional source of parallelism for badrefs rather than a consumer of parallelism.

Thus the problem of latency tolerance can be viewed as finding parallelism between a small set of very slow but identifiable operations and a large set of fast operations. The relevant question to ask about parallelism in this context is how many fast operations are independent of each slow operation. Rather than looking at average program parallelism one should focus specifically on the issue of whether a group of instructions consisting of goodrefs and non-memory operations can be matched with each dynamic instance of a static badref, the source of memory latency. These instructions must be independent of the candidate badref instance but need not be independent of one another since they can be executed sequentially.

The insight above makes the problem of parallelism identification for latency tolerance somewhat more tractable. The operation of servicing a miss transaction has additional properties which can be exploited to further reduce constraints limiting parallelism. Caches operate transparently with respect to processors so the servicing of miss transactions has no semantic impact on programs. This property makes the speculative initiation of miss transactions an attractive option for eliminating both constraints associated with program control and potential but unlikely data dependencies for miss addresses. Furthermore, the disambiguation function performed by caches decouples the transfer of data from memory to cache from the act of modifying this data. As a consequence, one need not consider potential dependencies for transferred data in the form of undisambiguated store operations. Finally, storage reuse dependencies, *i.e.* antidependencies and output dependencies, can be eliminated when desired through storage replication. The only dependencies which remain after application of these techniques are real dataflow dependencies for addresses used to initiate miss transactions and dataflow dependencies resulting from the actual use of transferred data.

We measure parallelism for a set of benchmark programs using a measurement technique targetted specifically at identifying parallelism relevant for latency tolerance. Anticipating the application of constraint reduction techniques, program parallelism is measured in a context which relaxes away many constraints and pseudo-dependencies. Dependencies are only recognized for actual runtime dataflow dependencies for addresses used to initiate miss transactions and from transferred data to uses of the data. The measurement technique tabulates non-dependent instructions with respect to each dynamic badref instance in a program run.

The first place to look for parallelism, since it yields parallelism most easily exploited in the framework of static code scheduling, is in the neighborhood of instructions adjacent to badrefs in a sequential program schedule. In this context we find substantial amounts of parallelism, often comparable in size to the neighborhood or window over which we look. In a 256 instruction window adjacent to a badref it is not uncommon to find that 90% of the instructions in the window are independent of the badref.

In the context of discussing parallelism we contrast statically scheduled latency tolerance, *i.e.* latency tolerant code, with single-threaded hardware dynamic schedulers. We observe that hardware schedulers probably cannot exploit key aspects of program behavior and properties unique to cache transactions which make latency tolerance tractable. In particular, it is difficult for a hardware scheduler to distinguish badrefs from goodrefs, at which point the battle may already be lost. Even given such a differentiation, it would be difficult for a hardware scheduler to look far into the future to find badrefs and to resolve necessary intermediate dependencies by executing those key instructions which lead to computation of badref addresses. As a consequence, hardware schedulers are restricted primarily to exploiting parallelism between badrefs and instructions closely following them in sequential program schedules. In benchmark measurements we find that this is the least effective place to look for parallelism.

1.1.3 Compiler Latency Tolerance Algorithms

Guided by insights regarding program locality and parallelism behavior, and emboldened by measurements indicating large quantities of parallelism, we proceed to develop code scheduling algorithms to access this parallelism to achieve concurrency for memory latency tolerance. These algorithms achieve their effectiveness by modelling memory behavior using the badref model. Focusing on badrefs leads to a viewpoint for the problem of generating latency tolerant code as one of explicitly scheduling miss activity within the memory system. This viewpoint proves to be effective.

In pursuing the problem of developing miss scheduling algorithms it is helpful to turn once again to the structure and behavior of programs. Upon close examination, it turns out that in many programs badrefs occur primarily in conjunction with accessing large, aggregate, data structures within loops. Loops traversing arrays typify these accesses although other patterns exist as well. In loops traversing aggregate data structures, some set of the static memory references in each iteration perform the first access to components in the newest data structure element. These references often systematically result in misses, producing badrefs. References which perform subsequent accesses to components never miss and are goodrefs. Accesses to these Regularly Accessed, Aggregate Data structures will be referred to as RAADs. RAAD prefetching is a technique developed specifically to target this key class of badrefs. RAAD prefetching applies software pipelining via prefetching to overlap overlap computation from one loop iteration with miss transactions scheduled for one or more future iterations.

While RAAD prefetching addresses an important class of badrefs for many benchmarks, it is not a panacea. By specifically targetting references inside of explicitly recognizable loops it fails to be effective when badrefs arise in other contexts or when procedure boundaries obscure explicit looping constructs from their contained badrefs. A second scheduling technique termed sequential miss scheduling addresses a more general class of badrefs. From a long sequence of code a shorter sequence of miss transactions is constructed, one per badref in the original sequence. These two sequences are then merged together, subject to limited rescheduling of the original code sequence in order to satisfy address dependencies for the miss schedule. A technique akin to trace scheduling is used to construct long code sequences for miss scheduling.

A prototype compiler implementation performing RAAD prefetching and sequential miss scheduling and a suitable simulation environment provide a mechanism for measuring the runtime performance of latency tolerant code. Code for this system is generated completely mechanically, thus the system provides a proof-of-concept for compilation-based tolerance to memory latency for data accesses.

1.1.4 Performance Evaluation

Despite the relatively primitive status of the prototype compiler, performance improvements are exhibited by benchmark programs. The specific speedup values depend upon the memory configurations assumed. For a small, fully-associative cache combined with a memory system with a latency of 20 cycles, speedups between 10 and 30% are exhibited by programs from the SPEC benchmark suite. For the same cache, coupled with a memory system with a latency of 160 cycles, but equivalent bandwidth, latency tolerant code outperforms unmodified code by factors between 2 and 6, *i.e.* modified code is sometimes 600% faster than unmodified code. Modified code for the 160 cycle memory system typically achieves performance within 25% of that of modified code for the 20 cycle memory system.

Performance is mixed for direct-mapped caches. While some benchmarks exhibit performance improvement comparable to the fully associative case for the same memory parameters, some benchmarks are actually degraded by latency tolerance optimization. We believe that this degradation reflects limitations in the current prototype compiler rather than fundamental problems associated with direct-mapped caches.

On balance, the results for the fully-associative cache test demonstrate the effectiveness of compilation based latency tolerance. Tests based on 20 cycle memory latencies exhibit modest improvements in conjunction with current memory latencies. Tests using 160 cycle memory suggest that our techniqes can provide significant tolerance to much higher levels of relative memory latency which will accompany very fast processors in the future.

1.2. Thesis Roadmap

Subsequent chapters address the topics alluded to above in significantly more detail. This section provides a description of the forthcoming chapters in order to allow the intrigued reader to skip directly to chapters targetting his or her interest.

Chapter 2 provides introductory, background and motivational material. The problem of memory latency is motivated as an inevitable peformance limitation for computing given current technological trends. Caching and interleaved memory systems are described, as well as difficulties in coupling these two techniques arising from limitations in current processor/memory system interfaces. Alternative memory latency tolerance techniques are surveyed, as is other related literature. This chapter is non-essential to subsequent chapters.

Chapter 3 discusses the static locality correlation and badref hypotheses. The hypotheses are developed and supported qualitatively, based on an intuitive, empirical analysis of program structure. The significance of these phenomena with respect to latency tolerant compilation is explained. Finally, quantitative, empirical evidence in the form of benchmark simulation data is provided to indicate the degree to which programs exhibit static locality and badref behavior for a single input data set across a variety of cache configurations. Evidence is also provided supporting the hypothesis that static locality behavior is consistent across multiple input data sets.

Chapter 4 analyzes parallelism requirements for memory latency tolerance. It explains why average parallelism and similar operation-blind metrics are not appropriate for characterizing parallelism for latency tolerance. A suitable measurement technique which relaxes away avoidable constraints and specifically measures parallelism with respect to badrefs in their neighborhood in a sequential program is developed. This measurement technique is used to characterize parallelism in benchmark programs. Empirical evidence shows that substantial parallelism exists.

Chapter 5 describes modelling techniques, algorithms and heuristics relevant to the problem of design and implementation of compilers to produce latency-tolerant code. The chapter describes two scheduling algorithms termed RAAD prefetching and sequential miss scheduling. It details heuristic tests to determine an appropriate scheduling technique in situations when both can be applied.

Chapter 6 provides empirical results gathered using a prototype compiler implementation based on the algorithms and techniques from Chapter 5. Two forms of results are provided. The compiler generates statistics describing the forms of badrefs encountered and the frequency of use and expected performance costs and benefits of the two code scheduling techniques. Code generated by the prototype compiler is executed in conjunction with a timing simulator to assess the latency tolerance of the code using a variety of cache configurations, processor/memory interface models and memory latency and bandwidth characteristics. In each case, the compiler is supplied with an accurate parameterization of the target memory system. Data on the speedups alluded to above is found in this chapter.

Chapter 7 draws conclusions based on this work.

Chapter 2

Motivation and Background

2.1. The Memory Latency Problem

Computer architecture and system design is a field which is partially driven by economics, and justifiably so. A frequently stated goal of system designers is to provide a system which maximizes performance subject to cost constraints or minimizes cost subject to performance constraints. Despite a general lack of agreement about precisely how to measure either cost or performance, economics often motivates the use of high-bandwidth, high-latency components in various subsystems within computers. High bandwidth is important since bandwidth imposes a hard upper bound on the performance of machines. High bandwidth can be achieved by repeatedly using a single low-latency component or concurrently using multiple, high-latency components. The latter solution may be more economical when it is applicable.

High bandwidth represents potential, but not necessarily actual performance, when it is achieved at high latency. Insofar as a computation is forced to wait for a data item from a high-latency component and this results in idle bandwidth, high bandwidth has not resulted in high performance. Significant effort in the computer research community is being applied to various incarnations of the problem of efficient use of high-bandwidth, high-latency computational resources and software latency tolerance can be classified as part of this effort.

The latency problem specifically addressed is that of memory latency for data accesses, latency associated with movement of data within the memory hierarchy of machines. We focus on uniprocessors with a memory hierarchy consisting of a single cache and an underlying main memory although latencies in memory systems with more caching levels or in multiprocessor systems should to some extent yield to a similar approach.

Latency for memory accesses associated with data are specifically addressed, as opposed to instruction accesses. Instructions show noticably higher spatial locality than data and sometimes higher temporal locality as well. If one measures spatial locality as the change in cache hit rate as a function of cache block size, data in [49] illustrates the heightened spatial locality associated with instructions. Enhanced spatial locality behavior for instructions makes them more amenable to various hardware prefetching mechanisms. Data latency tolerance is a harder problem.

With regard to data memory latency, one might ask a variety of questions.

1. Is there a memory latency problem? Under what situations does memory latency limit computational performance? To what extent does memory latency limit the achievable memory bandwidth of current and future machines?

2. Can high memory latencies be masked by overlap with other concurrent activity in systems which process miss transactions sequentially?

3. Can high latencies be masked by overlap with concurrent activity in systems which can concurrently process multiple miss transactions?

The first questions above will be answered within this chapter. The remainder of the thesis addresses the latter two questions. In order to assess the first question above, one can examine the relative changes in performance of processors and memory over time.

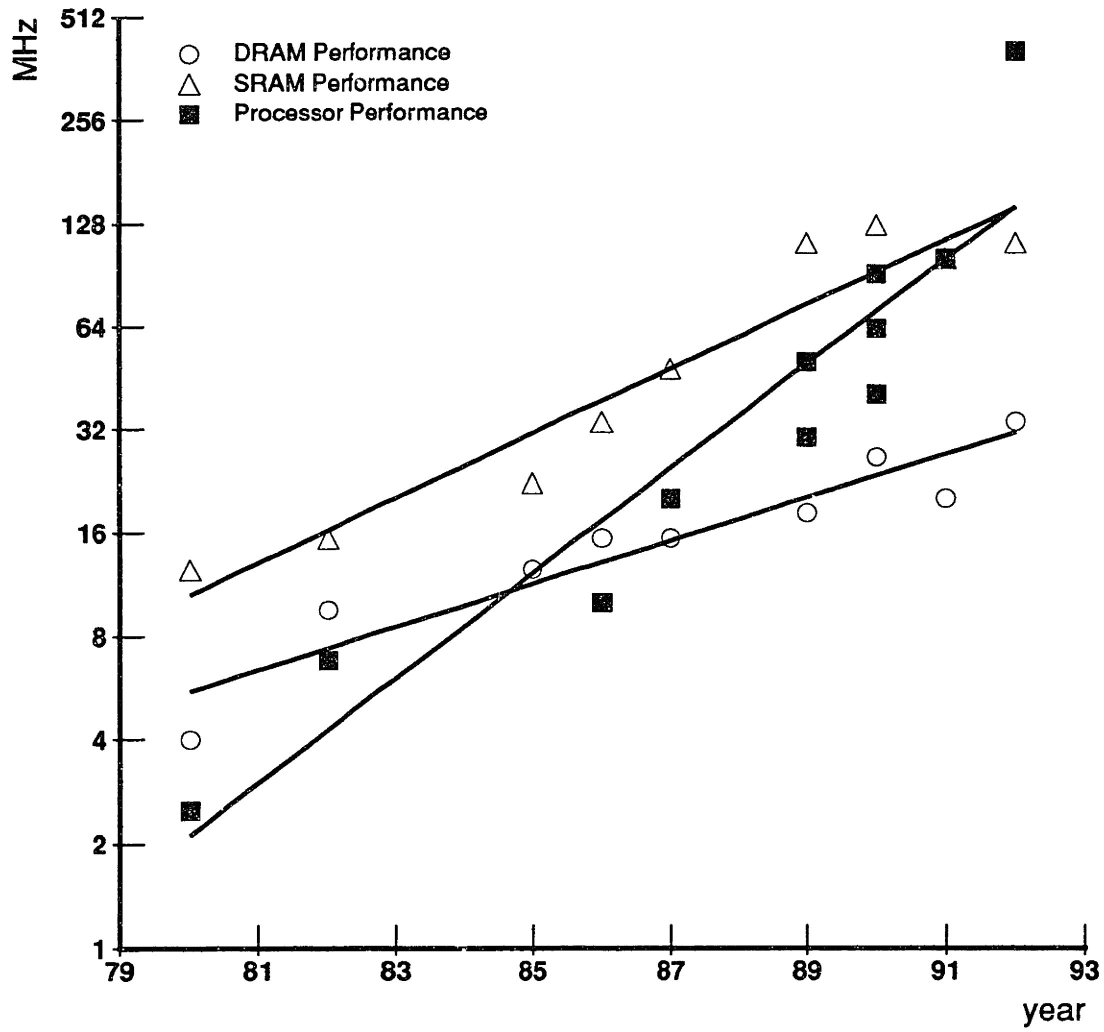


Figure 2-1: Component Performance vs. Time

Figure 2-1 shows the performance as a function of time of a sample of processor and memory components. The chart includes data for processors and CMOS DRAM and SRAM components. In order to get a uniform timebase, data is collected for each type of component from ISSCC proceedings [22]. Processor performance is measured as the bandwidth at which the processor can compute integer add instructions. Memory performance is measured as the inverse of the access time, address access time for SRAMs and row access time for DRAMs. The maximum density component reported each year is used since these components have the lowest per bit cost in production. Because of its source, data is based on research components rather than commercial availability. Bias produced by differing times for components to move from research into the commercial marketplace may skew the results slightly in comparison to a graph based on performance and availability of commercial products but trends in the relative performance of components should not reflect this bias. The graph shows a significant relative increase in processor performance in comparison to memory components over the period examined. Based on this trend, is is evident that either now or in the future, there will be a memory latency problem.

Figure 2-2 presents the same data in another way. In this graph all performance has been normalized to a time unit of processor cycles, the inverse of the integer add bandwidth of the fastest processor in a given year. Where the other graph showed inverse latencies, this graph shows relative latencies. DRAM and SRAM performance numbers indicate the relative latency of these components as measured in processor cycles. In this data it is even more evident that memory components, particularly DRAMs, are losing ground relative to processor performance. Since the scale is logarithmic, an exponential increase in relative latencies is indicated. The numbers in this graph do not reflect any time associated with bus access or address translation time. This further increases relative latency.

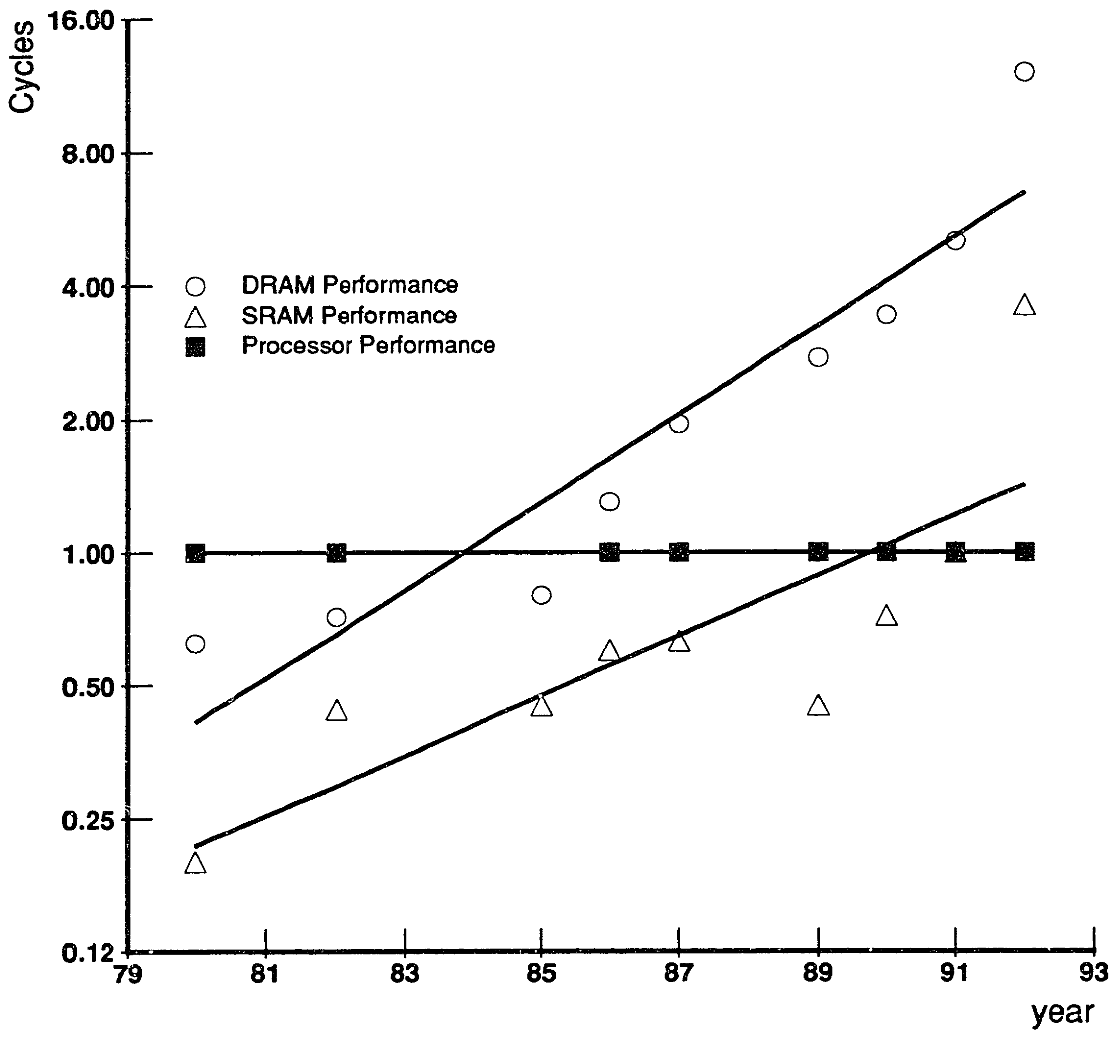


Figure 2-2: Normalized Latencies vs. Time

2.1.1 Processors

Recent advances in processor technology have led to a large drop in computational latencies and a corresponding increase in bandwidth. Measured in terms of potential instructions or elementary computations per unit time, computational bandwidths have increased dramatically. Furthermore, architectural changes toward concurrent instruction issue have led to increases in computational bandwidth beyond that associated with decreased clock period. The result of these trends is that computational bandwidth costs, measured as dollars per Mips, have dropped.

The RISC trend in processor architecture [17] [42] which has occurred over the last ten to fifteen years can be attributed with producing significant computational latency and bandwidth improvements by the simple technique of allowing higher clock speeds. RISCs are characterized by load/store instruction sets, isolating interactions with external memory to a small set of instructions which initiate either read or write transactions. Computations are performed on data buffered in fast registers. The repertoire of elementary computational operations directly provided by these processors is limited to those which are both heavily used and simply implemented. This economy of functionality has produced simpler, less cluttered processors, allowing for clock speed improvements. Pipeline concurrency has been heavily employed, exploiting parallelism between instruction fetching, decoding and computation, inherent in the interpretation process performed by processors. This type of processor architecture has leveraged advances in semiconductor technology, resulting in the hundred-fold increase in bandwidth noted in Figure 2-1 from 1-2 Mips 10 years ago to hundreds of Mips today and potentially thousands of Mips in the near future.

A more specialized trend which has produced additional bandwidth increases above and beyond those which accrue from simple clock rate increases is the advent of superscalar machines. A byproduct of the RISC architectural trend has been the use of specialized dedicated hardware resources for various types of functionality; thus a RISC processor might have a data unit which provides for external data transactions, an integer computation unit, floating point computation units, etc. Superscalar machines provide a means for exploiting the potential concurrency between these independent, often pipelined, functional units, either explicitly through instructions sets which specify multiple actions with a single instruction or implicitly through the simultaneous issue of multiple instructions. The DEC alpha [9], at the far right corner of Figure 2-1, is a superscalar processor.

2.1.2 Memory

Adequate memory systems must be provided if the low-cost computational bandwidth discussed above is to be used. Computer system architects, particularly those designing high clock speed systems, have properly turned to hierarchical memory systems to provide the memory bandwidth and latency characteristics required. Hierarchical memory systems incorporate two or more heterogeneous levels of storage with differing speeds, sizes and relative costs. At one end of the hierarchy is a modest amount of relatively expensive, low-latency, high-bandwidth storage, often managed as a cache. Further levels increase in size, but have increasing latencies and lower per bit costs. Hierarchical memory systems exploit locality in computations to leverage the limited quantities of low-latency, high-cost storage. Caches provide a hardware mechanism for dynamically exploiting locality. Interleaved memory describes a mechanism for providing memory systems with high bandwidths in comparison to their latencies. These systems exploit potential concurrency which often exists in memories constructed from many individual components, forming a high-bandwidth pipeline from multiple, high-latency components. Merging these two techniques potentially represents an attractive option for building memory systems with low, average latency and high bandwidth for processing miss transactions.

2.1.3 Demand-Fetched Caching

Caches store a dynamic collection of recently used data, using hardware to maintain a dynamic mapping between cache locations and aliased locations at lower levels of the storage hierarchy.

Caches typically use a demand fetching policy in acquiring their collection of data. Under this policy, if a non-resident datum is needed during a computation, the value is fetched from high latency storage and added to the collection, replacing some other value.

Demand fetching results in a multimodal distribution of service times for memory transactions. Transactions for which the desired storage location is dynamically aliased by the cache (*i.e.* hits) are satisfied quickly, exhibiting the storage latency of the cache. Transactions which miss the cache exhibit the storage latency of underlying storage levels. Average memory response times depend on the sum of the product of the service times associated with satisfying requests at each level of the hierarchy and the relative frequency at which those levels service requests. Two-level systems are characterized by a hit rate, measuring the fraction of requests satisfied by a cache, and by hit and miss times corresponding to the times to service transactions by the cache and by the underlying storage.

Inherent in typical, demand-fetched cache memory systems is a necessity to pay the latency penalty associated with cache misses. As the relative latency of low-level storage in comparison to computational speed increases, time spent waiting for misses to be processed accounts for a larger fraction of program time. Demand-fetched caches rely on high hit rates to amortize miss costs. There is a class of applications, characterized by very large data sets, for which even rather large caches do not exhibit high hit rates. Furthermore, as the relative penalties of misses increase, ever higher hit rates are required to amortize these penalties. Higher hit rates require larger, more expensive caches or further levels of cache hierarchy. Latency tolerance techniques exploiting concurrency offer a potential escape from this cycle. A goal of our research is to measure the extent to which high latency memory transactions, corresponding to cache misses in a cache system, can be tolerated through the software-based exploitation of computational parallelism.

2.1.4 Interleaved Memory Pipelines

Interleaving is a mechanism for using a set of high-latency storage components to construct a pipelined, high-bandwidth memory system. The components are organized into a set of banks. A high-bandwidth interface capable of supporting multiple pending transactions is provided between the processor and memory system. Transactions are initiated without waiting for the completion of prior transactions if they can be serviced by a bank not required by any pending transaction. In an interleaved system, a unit change in address results in a change of bank; thus for a typical N-way interleaved system, all addresses held by a bank are equal modulo N. This arrangement exploits spatial locality, the propensity for values adjacent in address space to be used together, although it is also somewhat effective under random transactions. Structured, transaction patterns with strides which are a divisor or factor of N result in overuse of banks and do not fully utilize the potential bandwidth of the interleaved memory system.

Large storage systems, consisting of many identical components, can be structured in an interleaved fashion. This has some impact on the medium connecting the storage and processor since transactions from multiple banks must be multiplexed over this channel, but it allows the use of unused bandwidth existent in many current systems. Further opportunities for pipelining exist within storage devices and even within the wires used for connecting components. [54] discusses a design for pipelined DRAMs to provide high-bandwidth, high-latency special purpose storage in supercomputers. [45] describes a new DRAM interface technology utilizing narrow datapaths at very high clock frequencies to achieve high bandwidth at relatively high latency.

An additional important source of increased bandwidth is the simultaneous use of multiple layers in hierarchical memory systems. Increases in latencies associated with data transfer and storage, while decreasing in absolute terms, seem unavoidable in relative terms compared with computational latency. Relative bandwidths, in contrast, can potentially retain more parity. Combining caching with some form of high-latency, high-bandwidth underlying memory is an attractive option for implementing economical, high-performance memory systems. Unfortunately, a variety of limitations in typical cache-based systems primarily associated with the processor/memory interface, coupled with models for code generation which are not cognizant of memory latency, conspire to make this difficult.

2.1.5 Memory Interface Issues

Memory interfaces couple processors to memory systems, supporting a series of transactions consisting of requests and replies. These interfaces provide needed synchronization and flow control mechanisms. Required synchronization functionality includes both a mechanism for matching replies with their requests and a mechanism for propagating data availability and transaction completion information to the processor. Flow control ensures that the processor does not produce more requests than the memory system can service. Synchronization, matching and flow control mechanisms can be provided either implicitly or explicitly. Many implicit mechanisms cause bandwidth to be directly limited by latency by preventing concurrent transaction processing.

If request and reply matching is achieved implicitly then reply order is highly constrained. Reply ordering can only be dependent on information implicitly available to both sides of the interface, which is limited essentially to the history of prior and pending transactions. The simplest implicit ordering mechanism imposes a limit of a single pending transaction. Reply delivery orders such as FIFO or LIFO are feasible for multiple transaction interfaces. Explicit matching, accomplished by tagging transactions, allows for flexible delivery order at a cost of some additional hardware complexity.

Communication of completion information can similarly be achieved either implicitly or explicitly. Implicit synchronization often takes the form of a hardware enforced upper bound on memory latency in processor virtual time. Transactions are guaranteed to be completed after a given number of processor cycles. Any transactions which would violate this guarantee produce stalls, stopping processor virtual time. A common upper bound used is one, whence transactions not completing in a single cycle produce stalls. Implicit data synchronization also achieves flow control since the number of memory transactions which can be produced before a stall occurs is limited.

Implicit mechanisms for reply matching and data synchronization are largely incompatible with software techniques for achieving concurrency between computation and slow memory transactions. Synchronization achieved by stalling to bound apparent latency at one provides no possibility for software latency tolerance. An interface which allows only a single transaction, avoiding the need for request/reply matching, does not provide any possibility for overlap of several slow transactions or simultaneous fast and slow transactions.

More sophisticated implicit mechanisms can be devised which might be appropriate for a pipelined memory system exhibiting uniform latency behavior. These mechanisms are incompatible with systems consisting of a cache overlaying a pipelined main memory system. If a stall enforced time bound provides synchronization, a dilemma arises in choosing this stall due to the bimodal response time of the memory. If a small bound value is used then high latency transactions unavoidably produce stalls. If a large bound value is used this eliminates the benefit of the cache since data values cannot be safely used until the bound has passed. Choice of a suitable ordering to allow implicit request reply matching is similarly confounding given bimodal response times. If two adjacent requests exhibit the same latency or a slow transaction follows a fast transaction then a FIFO ordering is desirable since the first request completes before the second. If a fast transaction follows a slow transaction, a LIFO ordering allows the latter to bypass the first, but LIFO is not suitable for the other three scenarios.

Interfaces providing explicit synchronization, matching and flow control provide the most flexible basis upon which to build software latency tolerance mechanisms. Explicit synchronization can be achieved through register scoreboarding, while explicit matching can be achieved by appending small tags to transactions and replies, allowing replies to cross the processor memory interface in any order. While scoreboarding is not uncommon in current RISC processors, tagged, multi-transaction, memory interfaces are essentially non-existent, at least in commercial processors.

2.1.6 Memory Latency

A variety of limitations in most current processors prevent more than one memory transaction from occuring at once. This directly couples achievable memory bandwidth to average memory latency. Even based on current technology, memory latency produced stalls can account for a noticable fraction of program runtimes. As relative memory latencies, defined as the ratio of memory latency to the inverse of processing bandwidth, increase with time, the problem will only become worse. In the absence of revolutionary technological changes resulting in large, inexpensive, fast storage components, the memory latency problem will exist.

2.2. Hardware Latency Tolerance Mechanisms

A variety of hardware mechanisms exist to exploit either locality or parallelism to tolerate residual latency associated with cache misses. Most of these techniques have been implemented in research or commercial machines. Multilevel caching attempts to extract and exploit further locality in the transaction stream resulting from primary cache misses. Hardware prefetch techniques use speculative concurrency, exploiting parallelism to overlap movement of data blocks with computation. Dynamic rescheduling techniques depart from sequential program order, either reordering instructions within a single thread of program control or between multiple threads, to find instructions which can be executed concurrently with slow memory transactions.

2.2.1 Multilevel Caches

As the relative latency of processors (and thus first-level caches), to main memories increases, adding additional levels of caching between the first-level cache and the main memory seems like a natural solution. If one cache is good, two caches must be better. (Use of multiple, hierarchical caches is different from the use of separate data and instruction caches. Separate data and instruction caching is implicitly assumed everywhere in the thesis). Additional levels of caching are quite likely to improve performance but this improvement must be justifiable when compared with its cost. Caching obeys a well known law of diminishing returns. An increase in cache size does not necessarily result in an equivalent relative increase in performance. This behavior applies to multilevel caches just as it applies to changes in capacity of a single cache.

Consider a two-level cache system composed of two direct-mapped caches with equal block sizes. Such a system conveniently obeys an inclusion property guaranteeing that any block in the smaller, first-level cache is also contained in the larger, second-level cache. Assume that the first-level cache exhibits a miss rate of m_1 and in the absence of the first-level cache, the second-level cache exhibits a miss rate of m_2 for the same workload. Since no value can be present in the first-level cache which is not present in the second-level cache, the second-level cache experiences an equivalent number of misses independent of the existence of the first-level cache. In a system containing both caches, the input stream to the second-level cache consists of misses from the first-level cache. The miss rate exhibited by the second-level cache when the first-level cache is included is thus m_2/m_1. If m_1 is a small number, 0.1 or 0.01 for instance, the miss rate of the second-level cache is magnified by a factor of 10 or 100, respectively.

The phenomenon discussed above can be viewed in terms of locality. The locality in a stream of addresses produced as misses from a cache is much lower than the locality in the address stream initially applied to the cache. As a consequence of this behavior, second-level caches must be significantly larger than primary caches to have a large impact on residual latency associated with primary cache misses. Large, secondary caches potentially represent a large system cost.

2.2.2 Cache Prefetching

Cache prefetching uses speculative concurrency to attempt to mask latency. Prefetch engines attempt to predict future memory needs using heuristic analysis of current memory transactions. Based on these predictions, the prefetch engine uses apparently idle bandwidth to move potentially useful data into the cache.

The simplest and most common form of prefetching is the use of large cache blocks. Transactions between high latency storage levels and the cache involve blocks of adjacent data larger than the transaction size between cache and processor. This strategy exploits spatial locality in programs, the phenomenon that adjacent data are frequently used together. When multiple words within a block are used, miss latency is avoided on accesses after the first, increasing the hit rate and amortization of the initial latency penalty. Utilization of blocked caches also allows for the amortization of cache management hardware in the form of tag storage and comparators over blocks of data, decreasing the cache cost. While this scheme only moves data as a direct consequence of a processor request, it can be viewed as prefetching. Concurrency is exploited to move data into the cache which has not been explicitly demanded.

More sophisticated hardware prefetching approaches asynchronously initiate transactions to fetch cache blocks which are not the target of a pending cache miss. These schemes are usually limited to the prefetching of blocks adjacent to either the block of the current processor cache transaction or the block most recently fetched due to a demand miss. Pure hardware-based prefetching is limited by the ability to determine implementable prefetch heuristics which contribute much over demand fetching performance. Smith describes hardware prefetching as well as other aspects of cache design in [47] and [48]. [12] investigates vector cache hardware prefetching in multiprocessors.

In an effort to produce a more intelligent, hardware, cache-prefetch mechanism, [4] proposes a technique which uses a separate, programmable prefetch processor which executes prefetch code generated by a compiler in conjunction with code for the main processor. This represents a mixed hardware, software technique. Like our techniques, a software mechanism controls the prefetching strategy, which allows the incorporation of knowledge available to the compiler. In our model, no separate, asynchronous prefetch engine exists. Instead, latency tolerance is achieved directly through processor executed code.

In [30], Lee describes a hardware prefetching technique which fetches data into buffers rather than into a cache. This prefetching technique uses addresses computed from prefetched instructions in a CISC style processor. It is more closely related to dynamic scheduling within RISC processors than cache prefetching. Dynamic scheduling is discussed in the next section.

Porterfield evalutes the effectiveness of hardware data cache prefetching for several supercomputer applications in terms of its effects on program hit rate in [43]. His data indicate mixed performance of hardware prefetching. For programs with data references predominated by unit stride array accesses hardware prefetching is very successful. These programs exhibit strong spatial locality, the property exploited by hardware prefetching. In the absence of needed spatial locality, hardware prefetching is less able to hide latencies.

The performance impacts of several complicated cache fetch strategies are investigated in [44]. Although this study does not specifically investigate prefetching techniques, the results are likely to apply to hardware-based prefetching as well. The performance metric used in this study is simulated processor time, in contrast to the hit rate measurements used in many prefetch studies. Complicated fetch techniques, including prefetching, may not produce performance benefits, in terms of execution time, commensurate with their increases in hit rate. Prefetching can tie up memory resources penalizing unrelated misses. Furthermore, prefetches which successfully avoid misses may not have finished moving data before the data is needed, resulting in stalls even for ostensible hits. Data in [44] indicates a clustering behavior of cache misses leading to the problems described above.

2.2.3 Dynamic Scheduling

Hardware prefetching operates on the memory side, speculatively producing memory traffic to achieve concurrency between the memory system and processor. In contrast, dynamic scheduling techniques operate on the processor side to find independent instructions which can be executed to achieve the same goal of processor/memory system concurrency. Dynamic instruction scheduling can be applied within a single control thread or across control threads. It can be applied to individual instructions or at a coarser level of granularity.

Hardware scheduling mechanisms which seek to extract parallelism within a single thread will be termed windowed, dynamic schedulers. These schedulers have a finite instruction buffer, providing a window into the future. Non-dependent instructions identified within the instruction buffer can be issued. While this technique may be suitable for tolerating short latencies of several instructions, it is highly inadequate for the task of tolerating very long latencies, on the order of a hundred instructions, for a variety of reasons. First, buffers used by the technique must grow proportionally with latency to be tolerated. The buffer providing the instruction window must be large enough that it doesn't fill up with dependent instructions. In addition to the instruction buffer, buffer storage is required for results from all out of order instructions since input operands to dependent instructions which are delayed cannot be overwritten. Given huge instruction and result buffers, it becomes impractical to determine whether new instructions are independent and if so where their operands can be accessed. The problem is rendered more intractable due to the many levels of branches that may need to be crossed speculatively in order to execute instructions out of order across a very large span.

Latency tolerance can be achieved by delaying instructions which require unavailable results and issuing non-dependent instructions. This approach moves the use of high latency results later in time. Alternatively, latency tolerance can be achieved by moving the latency producing instructions ahead of non-dependent instructions in order to start the slow operations earlier in time. Optimal schedules will exploit both of these techniques. Windowed, dynamic scheduling cannot easily move slow memory instructions ahead in time since it has no basis upon which to differentiate slow memory instructions from fast memory instructions. In contrast, software-scheduled, memory latency tolerance can effect the early initiation of high-latency operations, representing a fundamental advantage over window-based, hardware techniques.

Alternative hardware techniques for achieving memory/processor concurrency exploit parallelism across multiple threads of control. High-latency memory transactions from one control thread are overlapped with processing and additional high-latency transactions from other threads. Context switches between threads can occur at instruction-level granularity or at a higher level such as being triggered by cache misses. The HEP, desribed in [50], and various dataflow machines such as [41] support instruction level interleaving of contexts within pipelined processors. Machines such as [7] perform context interleaving at a coarser granularity. These machines are all multiprocessors and use their context switching mechanism to tolerate long latencies associated with network traversals as well as unbounded latencies associated with synchronization operations.

Support for multiple contexts is a costly proposition. Fast storage resources such as register files and caches must either be shared or replicated. In either case, the resources available for a single thread are decreased relative to single context machines in which all resources are devoted to a single thread. Fine-grain, context switching can provide tolerance to latencies with a spectrum of characteristics. [39] proposes that this mechanism should be used to hide latencies ranging from fixed latencies of several instructions associated with pipelined functional units to very large, potentially unbounded latencies associated with synchronization. It is our premise that software scheduling can adequately deal with predictable latencies and that memory latencies are largely predictable. The main benefit associated with multithreading then is tolerance to highly variable or unbounded latencies. Results of ongoing research projects will indicate whether multithreading mechanisms justify their costs for multiprocessor architectures.

We present evidence that for uniprocessors with hierarchical memory systems, despite the fact that memory response times may be multimodal in general, the references produced by any particular static instruction exhibit relatively predictable timing behavior. As a consequence, memory latency tolerance can be achieved without the need for dynamic scheduling techniques, either within a single thread or across multiple threads.

2.3. Software Memory Latency Tolerance

Software memory latency tolerance describes techniques in which latency tolerance is scheduled into code rather than achieved through some runtime hardware manipulation. Latency tolerant code initiates slow transactions well in advance of the need for the associated data. Processor/memory concurrency for latency tolerance is explicitly managed by scheduling code between the initiation and termination of high latency transactions. Concurrency between several, high-latency transactions can similarly be achieved by appropriate code scheduling.

Software latency tolerance requires a combination of hardware and software support. The basic hardware functionality required is a mechanism for initiating high-latency transactions without immediately incurring processor stalls. This functionality can be provided in a number of different ways including fetching data into a cache, into dedicated buffers or into registers. It can similarly be made accessible to software through several different mechanisms including non-blocking load operations or explicit prefetch instructions.

Thornier issues in software memory latency tolerance occur on the software side. Some entity, either compilers or users, must take responsibility for utilizing whatever hardware mechanism is provided to ensure that slow transactions are properly scheduled for latency tolerance. Several problems are involved in producing latency-tolerant code. First, a model of memory which recognizes the existence of slow memory transactions is needed. This model must not assume that every single reference which occurs is slow. Second, based on a model of memory behavior a scheduling technique is needed which can produce code with the desired characteristics. For very high latencies, this scheduling task can be difficult. We primarily address the software issues described above.

2.3.1 Latency Tolerance Strategy

As a fundamental premise, it is assumed that a compiler must be the entity described above which is responsible for producing latency-tolerant code. In order for hardware support for latency tolerance to be justified the mechanism must benefit a reasonably large set of programs. Latency tolerance techniques can only be applied in a wide-scale fashion if they can be applied transparently, or at least relatively effortlessly, through compilation.

The first step in developing a strategy for compiler support for latency tolerance is to develop a model of memory behavior suitable for the compiler. The simplest model which assumes every reference results in a cache hit is inadequate since it does not model the latency to be tolerated. Another simple model assumes every reference is a miss. This results in an intractable scheduling problem when latencies are very high. One can turn to the phenomenon of static locality correlation to help find a solution to this modelling problem. For programs exhibiting static locality correlation, static memory reference instructions exhibit either very high or very low individual miss rates relative to the average. Under this type of behavior, an appropriate model assumes some set of static references always hit while another set always misses. This model is termed the **badref model**.

A compiler based on the model above must perform two, high-level tasks in order to produce latency tolerant code. The first task must determine which memory references are to be modelled as misses and which references are to be modelled as hits. This modelling decision can be based either on static analysis or dynamically measured information. The second task is to apply scheduling algorithms to produce latency tolerant code based on modelled latencies.

The scheduling problem can be addressed on two fronts. One finds empirically that many references which can profitably be modelled as misses appear in loops. A loop-based scheduling technique similar to software pipelining [31] is applied in order to overlap the latency of memory transactions for one loop iteration with computation from a different loop iteration. This can be achieved when addresses for references for a particular loop iteration can be predicted one or more loop iterations in advance. To provide latency tolerance for references not occuring within loops, a second scheduling technique tuned to references in long sequential code regions is also employed.

2.3.2 Experimental Approach

The primary experimental technique used in our research is benchmark simulation. The benchmark suite used consists of the SPEC benchmarks. Three forms of simulation experiments are performed. Two of these simulations are used to empirically characterize relevant locality and parallelism properties of programs in the benchmark suite. The final simulation measures the performance of code and is used to evaluate the latency tolerance of code produced by a prototype compiler implementation.

The first simulation provides empirical evidence for the existence of static locality correlation. Static locality correlation can be assessed by measuring the individual hit rates of memory reference instructions within programs. Individual hit rates are measured under a variety of memory system configurations both to demonstrate that static locality correlation exists and to measure the impacts on this correlation of changes in cache design parameters and input data sets.

An additional aspect of program behavior which is germaine to software latency tolerance is parallelism. Latency tolerance is achieved through concurrent operation of memory systems and processors. Parallelism must exist in code in order to utilize hardware concurrency. Most techniques for measuring program parallelism do not adequately characterize constraints inherent in exploiting parallelism for latency tolerance. Average parallelism across an entire program is not an appropriate metric. The requirement for latency tolerance is parallelism specifically occuring between high latency transactions and other forms of instructions. When using latency tolerance techniques the concurrency at any given moment is low, perhaps a single memory transaction and a single processor operation. Successful latency tolerance requires a consistent low level of parallelism which can be maintained for the duration of a long transaction. A new simulation technique and parallelism metric are used to characterize parallelism for latency tolerance.

A final simulation technique is used to evaluate latency tolerance within code produced by a prototype compiler. This simulation includes a detailed model of cache and memory system timing. Performance of latency tolerant code cannot be adequately described by a cache hit rate. A simulation environment capable of accounting for various latency and bandwidth induced stalls is essential to have a realistic estimate of the performance of latency tolerance mechanisms.

The simulation forms described above share a common infrastructure. A C compiler capable of producing normal and latency tolerant code is the first component in the simulation process. An assembly code post-processor is used to instrument code with extra instructions to gather relevant data. In the case of cache simulations to measure static locality correlation this data includes a trace of memory addresses, for instance. In all cases this data is buffered and processed as it is gathered. Generating and consuming simulation data within a single program run allows large programs with billions of instructions to be simulated. (It is not feasible to generate and store an address trace with several billion memory addresses.) The final component of simulation infrastructure consists of a set of simulation routines for processing the various buffered data traces produced by each form of simulation.

2.4. Previous Work

Study of software, memory latency tolerance touches on a rather wide spectrum of topics, from cache performance to code scheduling. This section briefly overviews previous work in some of these fields in a progression that corresponds to its order of presentation in the rest of the thesis.

2.4.1 Cache Behavioral Simulation and Modelling

Cache evaluation based on simulation of behavior for address traces is an important and frequently used technique. [47] and [46] use this technique in a general context. [49], [19], [20], [44] evaluate aspects of cache design such as block size or associativity using trace driven simulation. [1] describes a technique for generating address traces for both user and system references and uses these traces for a comprehensive cache performance study. Study has even been devoted to fast techniques to simulate multiple caches simultaneously for the same addres trace, [19], [58].

Trace driven cache simulation is applied in our work to measure an aspect of cache behavior which has been largely ignored. Traces are extended within our simulation technique to include both addresses and unique identifiers for the specific memory reference instructions producing those addresses. This allows the investigation of the existence of static locality correlation by measuring hit rates of individual instructions. Porterfield investigated the same phenomenon in [43], work closely related to ours.

In addition to cache simulation, research has been conducted to develop analytical or approximate models for characterizing cache behavior. [2] describes an analytical model for cache behavior based on a probabilistic model for the input address stream and cache behavior. [56] and [57] describe an alternative modelling technique based on fractal behavior.

We also propose a model for cache behavior in the badref model. Prior models primarily focus on estimating hit/miss performance of caches in aggregate. The badref model, rather than modelling aggregate cache behavior, provides a framework and justification for individually modelling distinct memory reference instructions in programs.

2.4.2 Parallelism Measurement Techniques

Researchers have characterized parallelism in programs. In [29], Kumar uses an inline simulation technique to measure parallelism in Fortran programs when storage related dependencies, antidependencies and output dependencies, are relaxed. Kumar measures and presents results as parallelism profiles, graphs of inherent parallelism versus time. In [28] and [8], parallelism is characterized in terms of the maximum sequential path through programs and average parallelism over the duration of programs. Parallelism profiles and average parallelism metrics do not characterize the parallelism behavior of interest in software latency tolerance. Unstructured parallelism between completely unrelated regions of code cannot realistically be exploited without multithreaded techniques.

Parallelism of interest is more akin to instruction-level parallelism which has been measured by many researchers. [23] measures local parallelism available for multiple instruction issue. The results of this experiment would drastically underestimate relevant parallelism if extrapolated to apply to software, memory latency tolerance, however, because parallelism is only measured between independent instructions which are adjacent in a sequential schedule. Additionally, non-dataflow register constraints are included in the measurement, further limiting parallelism. [51] measures instruction level parallelism exploitable by window-based dynamic rescheduling mechanisms. This measurement technique identifies more parallelism because it relaxes away some unnecessary dependencies. It still underestimates parallelism relevant to memory latency tolerance because it uses a small window of instructions and only considers delaying instructions, ignoring parallelism available by moving instructions ahead in time. [59] measures instruction level parallelism using large windows and using techniques to relax away many dependencies. Again, this data is misleading if applied to memory latency tolerance. Average parallelism is not really an appropriate parallelism metric for latency tolerance.

We use a technique for estimating parallelism that shares some similarities with both macroscopic and instruction level estimates. The technique relaxes away storage and microscopic ordering constraints while maintaining macroscopic ordering constraints. This is achieved by measuring instructions which are independent of badref memory references within large windows in both directions around the instructions.

2.4.3 Compilation Techniques for Enhancing Locality

A number of researchers have investigated the use of compiler transformations to improve the cache or virtual memory performance of programs. The simplest such transformations involve reordering of code in order to enhance spatial locality. [16] describes a static technique, based on the structure of program call graphs, intended to improve virtual memory performance by managing the layout and potential duplication of code for procedures. [32] and [35] describe similar techniques which also incorporate basic-block, profiling information to guide the placement of code for basic blocks so as to achieve long sequential runs, thus exhibiting enhanced spatial locality. These latter techniques allow direct-mapped instruction caches to perform very competitively with fully-associative caches.

A much more challenging problem is addressed in [13]. This research aims to improve data cache performance through code transformation. The authors propose a static technique for analyzing cache behavior. Based on the fact that data dependence implies repeated use of the same address, they propose a technique for determining the locality for behavior of memory references using a technique related to static dataflow analysis. Based on this behavioral model, they propose algorithms for managing the contents of explicitly controllable caches or local memories.

A similar dataflow analysis technique is used to model memory behavior in demand-fetched, hierarchical memory systems in [43]. Based on statically predicted memory behavior, the profitability of program transformations such as loop fusion, loop splitting and blocking techniques can be assessed. Algorithms for evaluating the application of these transformations to nested loops in Fortran programs are presented.

One might characterize the latter two techniques as latency avoidance as opposed to latency tolerance. The analysis techniques and resulting transformations improve machine performance by decreasing the net flow of data between memory and cache. Latency tolerance, in contrast, masks latency associated with this flow of data witout decreasing the aggregate size of the flow. When applicable, latency avoidance is superior to latency tolerance. The two techniques are complimentary, however. Code can be first transformed to maximize its hit rate and minimize the information flow between memory and cache. Afterwards, latency tolerance techniques can be applied to lessen the impact of the residual flow.

2.4.4 Compilation Techniques for Memory Latency Tolerance

A number of researchers have investigated the problem of compilation support for software memory latency tolerance. This exploration has taken a number of different paths, based on differing models for underlying hardware mechanisms for data movement.

The first path investigates the use of software-controlled, hardware, block-prefetch engines capable of asynchronously moving large blocks of data. [21] and [14] address this problem in the context of multiprocessor systems. The hardware, latency-tolerance mechanism, a prefetch engine, is software activated and moves a block of data into or out of a locally connected memory on a multiprocessor. The goal of prefetching is to transfer arrays used in inner loops using the prefetch engine some time prior to execution of the loop. The local memory into which data is moved has neither a disambiguation mechanism with respect to locally produced writes nor a coherence mechanism with respect to memory references produced by other processors. As a consequence, the techniques must be conservative. The focus of the work is analysis techniques for identifying when an array reference within a loop can safely be prefetched and an evaluation of the expected performance impact of the technique.

As mentioned earlier, [4] explores the design of a programmable cache prefetch engine and compilation of code aimed at this engine. The engine executes a program generated automatically during compilation of an associated program for the main processor. The prefetch program specifies cache blocks which may be used by the main program. Additional prefetch code helps the prefetch engine track the state of the main processor based on its interaction with the memory system.

In the hardware model targetted in our work, data movement is directly initiated through specific instructions produced by a processor. Porterfield, in [43] and [5] describes a compiler algorithm targetting such a hardware model. Porterfield's basic algorithm is augmented by Klaiber in [25].

Porterfield's algorithm focuses specifically on Fortran array references within loops. He performs a syntax based transformation, adding prefetch instructions for array references within do loops which have a linear subscript computation involving the most recent loop induction variable. An optimized version of the algorithm uses a static estimate of memory behavior to eliminate prefetches associated with memory references unlikely to generate misses. Within the regime of Fortran programs, Porterfield's algorithm targets a likely source of low-hanging fruit. Array references generally account for a large fraction of misses. Klaiber's extensions to this algorithm modify prefetching strategy based on a parameterized model of memory behavior.

Porterfield's work touches on many of the important issues addressed in our research. He observes and describes behavior in benchmark programs which in our parlance would be described as static locality correlation. He employs a compiler model which partitions array references into a set categorized as misses and a set categorized as hits. The primary use of this cache model is in algorithms for assessing the locality impact of loop transformation in highly structured Fortran programs. Porterfield also discusses the use of his memory behavior modelling to decrease software overhead in the prefetching algorithm described above.

We address memory latency tolerance and data prefetching in a somewhat more general context than Porterfield. Algorithms applicable to a much wider variety of data references are presented and the problem of latency tolerance is presented as one of explicit, miss scheduling. Static locality correlation is presented as a fundamental form of program behavior and characterized for a variety of benchmarks and memory system configurations.

2.4.5 Inline Profiling and Profile-based Compiler Optimization

Efficient profiling can be achieved through the modification of code via a compiler or assembly postprocessor to include extra code specifically added to gather some form of desired statistics inline with program execution. [26] describes a compiler-based approach to gather multiprocessor address traces. The pixie system from MIPS [36] modifies object code to perform inline, basic-block profiling. [53] describes a similar basic-block, profiling technique based on an assembly post-processor. We rely heavily on inline profiling and simulation techniques, both for gathering data for profile-driven compiler optimization, as well as gathering basic, empirical data used to evaluate locality and parallelism behavior and compiler performance.

Profile-based compiler optimization techniques have been explored by several researchers. Two techniques mentioned previously, those in [32] and [35], for improving direct-mapped instruction cache performance, modify the layout of code within memory based on basic-block, profiling statistics. In [33], McFarling investigates the application of profiling statistics to branch prediction. He compares the performance of branch prediction based on profiling with hardware, branch-prediction techniques.

The compiler discussed herein utilizes basic-block, profile information. This information is secondary to dynamically gathered, memory performance information. Memory performance information, consisting of individual hit and writeback rates for ea^h memory reference instruction, is gathered using inline simulation techniques.

2.4.6 Code Scheduling for Hardware Concurrency

Once a method of modelling memory latency is devised, generation of memory latency tolerant code is reduced to a schedul:ng problem similar to that for machines with multiple, pipelined functional units. Compilation techniques for such machines have been investigated by other researchers.

When viewed in the correct light, the loop-based prefetching techniques implemented in our compiler bear a strong similarity to software pipelining techniques discussed by Lam in [31]. Software pipelining is a technique for exploiting interloop parallelism to mask high-latency, computational operations. High-latency operations, primarily floating point operations in Lam's work, can result in a critical path for a single loop iteration which substantially exceeds limits on loop performance due to instruction issue, (*i.e.* the number of instructions in the loop divided by the instruction-issue concurrency of the machine is less than the critical path latency). Software pipelining spreads this latency across two or more successive loop iterations by mixing instructions from different virtual iterations within a single loop.

Ellis, in [6], describes a compiler for VLIW machines based on trace scheduling. Trace scheduling identifies sets of basic blocks which are likely to be executed in sequence. These sequences, or traces, are scheduled using scheduling techniques for linear code, treating the entire trace as an extended basic block. After scheduling, cleanup code must be added at basic-block boundaries leading onto or off of a trace. The technique of trace scheduling is also discussed in [10] and [11].

The hardware model targetted in our compiler is rather different than that of Lam or Ellis. VLIW machines have many, pipelined, functional units, each potentially exhibiting multicycle latency. The latency of these functional units is not large, typically a handful of cycles. The target model for our compiler has only two functional units, a non-miss unit and a miss unit. The latency of the non-miss unit is only one and only a single instructions, targetting one of the units, can be initiated in a cycle. The latency of the miss unit, rather than being 4 or 5, as might be characteristic of a functional unit on a VLIW machine, is a very large number, as high as 160 in some of our experiments. Scheduling for one, very high-latency, functional unit differs from scheduling for a multiplicity of lower latency units. An additional point of contrast between our work and the work of Ellis or Lam is the domain of computation targetted. The VLIW techniques target highly structured, scientific code. Our technique, while most successful on structured code, targets a more general class of computations.

2.4.7 Speculative Computation

One might observe that the trace scheduling technique above is simply a controlled form of speculative computation. Speculative computation has been investigated by other researchers at a variety of granularities. [52] describes an architecture for efficient, fine-grained, speculative computation for instructions with side-effects and notes moderate increases in instruction-level parallelism using speculative, instrution execution.

Prefetching in any form can be viewed as speculative computation. It is difficult to be certain that a memory reference will result in a miss. Given this uncertainty, prefetching is a speculative expenditure of computation in order to avoid an expected cost in terms of memory latency. Prefetch instructions are a natural target for more dramatic speculative execution since they are free of visible side-effects, (or can be made so through appropriate treatment of various faults). In order to tolerate very high latencies it is necessary to cross block boundaries, resulting in speculative computation. Given high latencies, the payoffs for successful prefetching can justify aggressive speculation.

2.4.8 Analysis of Software Prefetching Behavior

Performance of user-introduced, non-binding, software-prefetch operations is analyzed in [38] and [25]. [38] examines several multiprocessing benchmarks while [25] looks at uniprocessor benchmarks. These studies tend to support our observations regarding software prefetching in several ways. In both studies, the addition of a relatively small number of prefetch operations to programs improves performance. This supports the conjecture of static locality correlation and also indicates that sufficient parallelism exists in the programs analyzed to allow prefetching to be a succesful technique.

2.5. Hardware Support for Software Latency Tolerance

Software latency tolerance requires hardware support. This section will briefly overview the necessary hardware functionality and discuss the software implications of various hardware implementation choices. It will also describe and motivate the particular hardware models targetted by our prototype compiler implementation.

The essential hardware functionality for software latency tolerance is a non-blocking mechanism by which a processor can initiate high-latency memory transactions and continue to process other instructions. This mechanism is used in code to trigger memory transactions for addresses which would otherwise result in cache misses. Non-dependent instructions can then be scheduled during the transaction, achieving processor/memory system concurrency.

A primitive form of this functionality exists in several current commercial RISC processors such as the Motorola 88100, [37], and Intel i960, [34]. These processors use register scoreboarding techniques to implement hardware, pipeline interlocks for variable-latency, functional units. In such systems, issued load instructions do not produce stalls until the target register is used as an operand in a subsequent instruction. Independent instructions continue to issue. In these systems, a resource conflict at the memory system prevents any further memory transactions from being initiated during a pending transaction, irrespective of whether these are high or low-latency transactions. As a consequence, concurrency between high-latency memory operations and other computation is restricted to computation involving only data maintained in registers. While this functionality is better than nothing at all, it is highly restricted in its ability to achieve tolerance to high, memory latencies through software techniques.

2.5.1 Memory System Support for Software Latency Tolerance

Successful software latency tolerance requires memory system support in order to relax the resource constraint at the memory system mentioned above. Memory systems must have a mechanism by which they can continue processing cache hits concurrently with a pending miss. Caches which can process hits in the face of an outstanding miss have been termed lockup-free caches by Kroft in [27]. The primary, additional hardware to produce lockup-free caches is a set of storage buffers, one per outstanding reference, which have an associative, address register to identify hits on data from pending transactions and which have buffering space for arriving data. Kroft moves data from these buffers into the cache proper upon its receipt.

An alternative strategy suggested by Klaiber in [25] uses a larger number of buffers and does not move buffered data into the cache unless this data is modified. These buffers are filled through explicit prefetch instructions which differ from memory loads. This latter strategy avoids the cost of cache bandwidth cycles required to move buffered data into the cache when the data is not written. It also lessens the potential for interference between cached data and prefetched data since prefetched data has its own buffering space independent of the cache.

Based primarily on intuition, one might suspect that data in buffers will often be modified, necessitating movement into the cache and eliminating some advantages of the separate buffering technique. As a consequence, our model assumes a lockup-free cache implementation, as opposed to an implementation with separate dedicated prefetch buffers. From a software viewpoint, however, both implementations are very similar. They both provide an enabling feature for memory latency tolerance, the ability to process cache hits during miss transactions.

2.5.2 Instruction Set Access to Latency Tolerance Mechanisms

Instruction set access to underlying hardware latency tolerance mechanisms can be provided on a typical RISC architecture in one of two ways. Non-blocking load instructions have the semantics that a data value from memory is moved into a processor register. In order to be non-blocking the load must not stall the processor in the face of a cache miss until after the transaction can be processed. A synchronization mechanism must be provided to insure that non-blocking loads have completed before their associated data is used at some later time. Explicit prefetch instructions, in contrast, initiate memory transactions as a semantically transparent operation. A prefetch instruction typically provides the same addressing capabilities as a load or store instruction. It produces a memory address and causes a cache access. If the access is a miss then a main memory transaction is initiated for the appropriate cache block. No value is returned to the processor in the event of either a cache hit or miss. Misses in a lockup-free cache can be triggered by either non-blocking loads or explicit prefetch instructions. A memory system which differentiates between normal and prefetch requests needs an explicit prefetch instruction as its software interface.

Gupta draws a distinction between binding and non-binding prefetching mechanisms in [15]. Data accessed with a binding operation is not subject to coherence and disambiguation mechanisms with respect either to the accessing processor or to other processors if the system is a multiprocessor. Thus, in the uniprocessor context considered, data accessed by a binding prefetch is not modified by write operations to the same address which occur between initiation of the operation and use of its data. Non-blocking loads are logically implemented as binding operations whereas explicit prefetch instructions which move data into a cache are non-binding.

Modifying code for software latency tolerance using non-blocking loads, a binding operation, potentially results in lower software overhead than similar code using non-binding, explicit prefetch instructions. A single non-blocking load instruction initiates a memory transaction and moves the resulting value into a register. In order to achieve the same effect with explicit prefetch, a prefetch instruction must be used to initiate the memory transaction followed by a load (either blocking or non-blocking), to move a value from cache to a processor register. Additionally, if a complicated address computation is required then with explicit prefetch this computation must either be performed twice or the result must be maintained in a register. Since a non-blocking load ties up its target register for the duration of a memory transaction the cost of keeping an address in a register under non-binding prefetch is equivalent to the cost of tying up the target under binding prefetch.

While decreased software overhead is an advantage of binding prefetch, this advantage comes at a cost in terms of flexibility of use. Disambiguation describes the compiler process of determining if two memory references may use the same address. A binding prefetch cannot be reordered with respect to a write operation with which it cannot be successfully disambiguated. Non-blocking loads can be used to implement non-binding prefetch operations in addition to binding prefetch operations, however. Non-binding prefetch can be implemented with non-blocking load by ignoring the result value delivered by the load. A non-blocking load used in this fashion avoids write reordering constraints but loses the overhead advantage associated with binding prefetches. Non-blocking load thus provides a flexible mechanism which can be used as either a binding or non-binding operation.

Several additional issues arise since non-blocking loads require explicit register targets. It is not possible for two memory transactions which arise from the same instruction in two different loop iterations to be outstanding simultaneously, under binding prefetch, since they would target the same register. This necessitates loop unrolling to tolerate memory latencies in excess of the latency of a single loop iteration. Similarly, the maximum number of outstanding, non-blocking loads at any time is limited by the total number of distinct processor registers.

Explicit prefetch instructions are somewhat different from non-blocking loads. The do not have semantic significance. They can added to or removed from a program without changing its results, with the possible exception of producing spurious memory faults. As a consequence of this absence of semantics, explicit prefetch instructions have several potential advantages. First, with appropriate hardware modifications, they can be made fault-safe and thus truly absent of semantic significance. Address faults generated by explicit prefetch instructions can be ignored by the processor by suppressing the generation of a memory transaction, thus converting the prefetch to a no-op. If faults are suppressed, explicit prefetches can be used freely in speculative contexts without the worry of introducing spurious faults. Even though non-blocking loads can be used as non-binding prefetch operations, they cannot easily be made fault-safe. A processor has no way to tell whether a load is being used in a semantically meaningful way or not. While techniques such as those described in [52] can make all instructions safe for speculative use, this occurs at a more substantial hardware cost than that associated with ignoring faults for explicit prefetches.

An additional consideration arises favoring explicit prefetch as a mechanism for tolerating very high latencies (say a hundred cycles), particularly on processors supporting precise interrupt models. One technique for supporting precise interrupts in the face of high operation latencies is to buffer side-effects produced by instructions in shadow buffers. Side-effects from an instruction which modify either the register file or memory can be safely retired after previously issued instructions can no longer produce faults. When a fault occurs, the contents of the buffer can be discarded after the fault point, effectively rolling back time to the occurance of the fault. Given such a scenario, the latency which can be tolerated by an instruction which can produce faults is limited by the shadow-buffer size. Non-blocking loads can produce faults whereas explicit prefetch instructions can be implemented to be fault-safe. Buffering issues such as these may arise for other reasons on processors without precise interrupt models. One can argue that it would not be a good engineering tradeoff to dramatically increase the size of shadow buffers simply to allow non-blocking loads to tolerate latencies of hundreds of cycles when the same latency tolerance functionality can be achieved using explicit prefetches.

The apparent efficiency advantage of non-blocking load may be artificial in many situations. As effective latencies of memory and computational units increase, many programs may be limited not by instruction issuing bandwidth, but rather by latencies or bandwidths of relatively slower components. In this scenario, excess instruction issuing bandwidth exists for extra instructions required for explicit prefetching. Nonetheless, it would be desirable to lessen this efficiency difference if possible.

In order to lessen the potential efficiency difference a slight variant of the explicit prefetch described above might be implemented. Memory instructions typically provide for some primitive form of arithmetic computation for addresses. Typical computation might be the sum of a register and a constant, the sum of two registers or perhaps the sum of a register and a scaled register. One might consider implementing a prefetch which had the additional functionality of storing the computed address into the register file. (The HP precision architecture already offers such a feature for regular load operations [18].) In this case a prefetch instruction would be similar to an integer add instruction which had the additional transparent side effect of producing a memory transaction if the resultant sum represented a valid address. Induction variable optimization inside of loops frequently produces pointer variables which must be updated in each loop iteration. Straightforward code transformations could be applied so that prefetch instructions could be used for induction variable updates in addition to their latency tolerance function. By allowing prefetch instructions to replace another necessary instruction, the efficiency gap between explicit prefetch and non-blocking load is closed. This is accomplished while maintaining the advantages of explicit prefetching described earlier. Under this new form, prefetches have a semantically meaningful component, the add, which cannot produce faults, and a second non-visible operation for which faults can be squashed.

2.5.3 Processor Memory Interface Issues

Processor memory interface issues impact software latency tolerance. Some interface issues primarily influence the choice of instruction support while others influence the code generation and scheduling process. Interface issues include synchronization, buffering and flow control characteristics of memory transactions.

It was mentioned previously that a lack of explicit mechanisms for synchronizing loaded data to the instruction stream and matching transaction replies with their requests led to constraints on latency tolerance. Non-blocking load instructions cannot be used effectively for software latency tolerance without explicit mechanisms for synchronization and reply matching. Explicit synchronization is needed since with a non-blocking load, hardware stalls do not guarantee data availability. Explicit reply matching allows subsequent references producing hits to return ahead of misses. Explicit prefetch circumvents these problems without the need for explicit mechanisms since prefetch instructions do not return data to the processor from the memory system. These synchronization mechanisms primarily influence the choice between prefetch or non-blocking load. Beyond the choice of a form of instruction access to hardware latency tolerance mechanisms, these issues do not strongly influence software.

In contrast, buffering and flow control characteristics of the operation providing latency tolerant memory access do potentially impact code generation strategy. In the absence of flow control, memory operations could be issued by a processor at a rate exceeding the bandwidth of the underlying memory system. Flow control provides a mechanism for stalling the processor in order to keep the memory request rate within the limits of memory bandwidth. Buffering of memory requests allows the instantaneous rate of memory request production to exceed the allowable bandwidth for a short period of time without incurring flow control stalls. Semantically meaningful, memory references, including loads and stores, whether blocking or non-blocking, must be flow controlled. (Blocking operations need no additional flow control since this is achieved by blocking.) Prefetch operations, since they have no semantic significance, do not necessarily require flow control. If their generation rate exceeds the bandwidth of the memory they can be flow controlled or simply ignored.

Prefetch buffering impacts code generation since it is undesirable to generate a series of prefetch instructions exceeding the buffer capacity of the memory. This has unattractive consequences whether or not the prefetch operations are flow controlled. If the operations are flow controlled, exceeding buffer capacity results in a processor stall. If the operations are not flow controlled, exceeding the buffer capacity results in prefetch operations dropped by the memory system. Thus code schedulers should be cognizant of buffer limits.

Based on the statement above, one would naturally assume that our prototype compiler implementation models the prefetch buffer size. In fact, buffering behavior is ignored in the prototype. Compiler implementation is simplified by abstracting away this behavior. In the absence of flow control, buffer overflow causes a prefetch to be dropped, eventually resulting in a stall equal to the memory latency lessened by the probability that the prefetch accesses a value already in cache. With flow control, a buffer overflow results in a stall until a buffer becomes available. This requires a variable amount of time bounded by the memory latency. Given a memory system which can process several misses concurrently, the resulting stall is likely to be substantially less than the memory latency. Flow control does run the risk of producing stalls for prefetches for values already contained in the cache, however. Which of these consequences is more severe depends both on the memory miss-processing concurrency and the likelihood that prefetches target addresses which are cache hits.

2.5.4 High-Bandwidth Memory Systems

A high-bandwidth, memory system refers to a memory system with a latency-bandwidth product exceeding unity. This can be accomplished with pipelining or some form of interleaving. Code which is not explicitly scheduled for latency tolerance potentially has some intrinsic capability to utilize a high-bandwidth memory system. This primarily arises through overlap of multiple write transactions or read transactions and write transactions. Cache misses requiring writebacks produce two memory transactions. In a memory system interleaved by a power of two these transactions are both likely to require the same bank. With a pipelined system or an interleave which is prime with respect to the mapping used for cache-set selection, both transactions may be initiated simultaneously. Even if the mappings are not prime, if writebacks can be buffered there is potential for overlapping writebacks with later cache fills. There is little intrinsic ability in typical code to exploit more than two-way memory system concurrency. Latency tolerant scheduling, based either on explicit prefetch or non-blocking load, provides a mechanism for taking advantage of high-bandwidth memory systems of higher concurrencies if this added bandwidth is modelled within the scheduling algorithm.

2.5.5 Target Hardware Model

Our research includes the implementation of a prototype latency tolerant compiler. The hardware latency tolerance model targetted by this compiler is chosen primarily on the basis of its percieved ease as a compilation target. As a compiler implementer, as opposed to a hardware designer, this seems like a natural basis upon which to make decisions. Having taken on the burden of latency tolerance in the first place, doesn't a compiler implementer deserve whatever help can be provided by hardware? There is an ongoing debate over software versus hardware functionality to which this question can be added. We find ourselves in the camp which would answer no, advocating a strategy of making hardware cheap and letting compiler writers earn their salaries. Fortunately, in this case, many of the tradeoffs that appear desirable from a software perspective are not inconsistent with the choices which might be made by a hardware implementor.

Tolerance to very high latencies requires aggressive code motions and probably speculative computation. Explicit prefetch is more suited to speculative use than non-blocking load. Explicit prefetch is probably also easier to implement on the hardware side since it avoids difficulties associated with supporting multiple outstanding transactions across the processor memory interface.

The implementation of explicit prefetch as a special form of integer add, as suggested above, imposes costs on both the hardware and compiler. While this technique has some potential benefit in terms of software overhead, it is not included in the model.

The target model assumes a flow control mechanism for prefetches. This is a tradeoff where software and hardware viewpoints might diverge. One can speculate that dropping prefetches overflowing buffering capacity might be easier than providing flow control. Since loads and stores already require flow control, added flow control for prefetches can probably leverage the available mechanisms to some extent. The advantage of flow control for the compiler is that it allows prefetch buffer limits to be abstracted away.

Memory latency and bandwidth characteristics are not specifically included as components of the target hardware model. Instead, it is assumed that these numbers are provided to the compiler in a parameterized form at compilation time. This allows code to be specifically optimized for particular memory system behavior while the compiler addresses a family of memory systems exhibiting a broad range of performance characteristics.

Chapter 3

Static Locality Correlation and the Badref Model

Computer memory systems are often hierarchical. A two-level hierarchical system typically consists of a cache and underlying main memory. The timing behavior of such a memory system is nonuniform, depending on which component satisfies a request. Requests satisfied by the cache, termed hits, are relatively fast while those satisfied by the main memory, termed misses, are somewhat slower. For programs executed on machines with hierarchical memory systems the dynamic memory system behavior is typically characterized by a hit rate or equivalently a miss rate. Hit and miss rates, denoted by h and m, are statistics identifying the fraction of memory requests satisfied by the cache and main memory components of the memory system respectively.

Processors produce memory requests for both instructions and data values. The following analysis focuses specifically on data requests. In high-performance RISC architectures instruction and data requests are frequently satisfied by distinct caches. When this is the case, one can mostly ignore instruction requests in an analysis of the performance of data requests. Henceforth, all discussion of memory performance refers specifically to data requests.

A static program is a set of instructions. For load/store architectures, the topic of this analysis, data memory requests can only be produced by load and store instructions. Each such instruction produces a single request to the memory system. The dynamic memory behavior associated with the execution of a program is an average of the dynamic behavior of the individual load and store instructions comprising the static program weighted by the relative frequency with which these instructions are executed. Suppose that execution of some program for a particular input data set exhibits a 90% hit rate for data requests, thus 9 out 10 requests are satisfied by the cache. This average behavior can arise from two markedly different forms of behavior when examined at a finer granularity.

One possible scenario is that each memory request, irrespective of the particular instruction which produces it, acts essentially as a statistically independent trial with a probability h of being satisfied by the cache. Under these conditions, each memory reference instruction in the static program would exhibit an individual hit rate close to the average over the entire program. Based on the laws of probability, one might expect an approximately Gaussian distribution of individual hit rates, centered on the average hit rate with a width dependent on the number of individual instructions considered and the number of times each instruction executes. Given no evidence to the contrary, one might assume that this scenario reasonably modelled program behavior.

As an alternative scenario, assume that hit rates exhibited by individual memory reference instructions are either 1 or 0. Thus upon execution of a particular load or store instruction the corresponding request is always satisfied by the same memory system component, cache or main memory. If appropriately sized sets of the static memory references exhibit these two forms of behavior a 90% average hit rate can be achieved. These sets must be sized such that the set which always hits accounts for 90% of the dynamically occuring references while the set which always misses accounts for 10% of the references.

These two scenarios are distinguished by the absence or presence of a correlation between particular static memory reference instructions and dynamically occuring locality behavior. The scenarios are idealizations of two extremes of possible memory behavior. The extent to which behavior of real programs is characterized by one or the other of these scenarios is a question of both theoretical interest and practical significance. This chapter presents arguments and empirical evidence that to a significant degree, static locality correlation is a fundamental property of programs. Static locality correlation is the phenomenon that cache misses for programs when executed on machines with hierarchical memory systems are predominantly associated with a subset of memory reference instructions, or equivalently that memory reference instructions exhibit a predisposition towards either high or low hit rates relative to the average. The chapter also outlines how this behavior can be exploited to dramatically increase the effectiveness of software latency tolerance techniques.

3.1. Memory Statistics of Individual Instruction

Exploration of static locality correlation requires analysis of the locality behavior of the set of dynamic references arising from a single static memory reference instruction. In the following discussion, the term static reference will be used to refer to a memory reference instruction in a program and the term dynamic reference will be used to refer to a memory reference instruction executed while running a program. Each static reference has an individual miss rate for some program execution corresponding to the fraction of the dynamic references it produces which result in misses.

Define two functions $R(m)$ and $M(m)$ which are statistical properties associated with the execution of a program. $R(m)$ is a density function of dynamic references produced by a program, parameterized by the miss rates of the static references from which they are produced. Thus $R(0.01)$ corresponds to the fraction of references produced by static references with miss rates of 0.01. Since $R(m)$ for a program run only exhibits density at the finite set of m values corresponding to the miss rates of each static reference in the program it is a discrete density function. The m values with non-zero density are non-uniformly spaced and vary across program executions. For convenience, $R(m)$ is treated as a continuous density function which implies that non-zero density samples are impulses. This impulse issue is ignored. Similar to $R(m)$, $M(m)$ is a density function of dynamic misses incurred by a program, parameterized by the miss rates of static references producing the memory requests resulting in the misses. As with $R(m)$, when modelled as a continuous density function $M(m)$ also really consists of impulses.

The miss behavior characterized by $M(m)$ can be derived from $R(m)$. The number of misses produced by a static reference is simply the total number of dynamic references produced by the static reference multiplied by the references miss rate. Thus multiplying $R(m)$ by m and rescaling by the integral of m $R(m)$ produces $M(m)$.

$$M(m) = \frac{mR(m)}{\int_0^1 mR(m)dm} \qquad (3-1)$$

Cumulative representations of these statistics, $R_c(m)$ and $M_c(m)$, can be produced by integrating each function with respect to m.

$$R_c(m) = \int_0^m R(m')dm' \qquad (3-2)$$

$$M_c(m) = \int_0^m M(m')dm' \qquad (3-3)$$

These latter representations of the same information can lend additional insight into techniques for exploiting static locality correlation.

The two scenarios of memory behavior described at the beginning of this chapter exhibit very different patterns of $R(m)$ and $M(m)$. In the first situation a tight distribution of $R(m)$ centered at the average program miss rate of 0.10 is assumed. Figure 3-1 illustrates $R(m)$, $M(m)$ and the corresponding $R_c(m)$ and $M_c(m)$ for a distribution with all mass at a single miss rate. The second scenario describes behavior in which 90% of the weight of $R(m)$ occurs at miss rate 0 while 10% occurrs at miss rate 1.0. Figure 3-2 shows $R(m)$, $M(m)$ $R_c(m)$ and $M_c(m)$ for this situation.

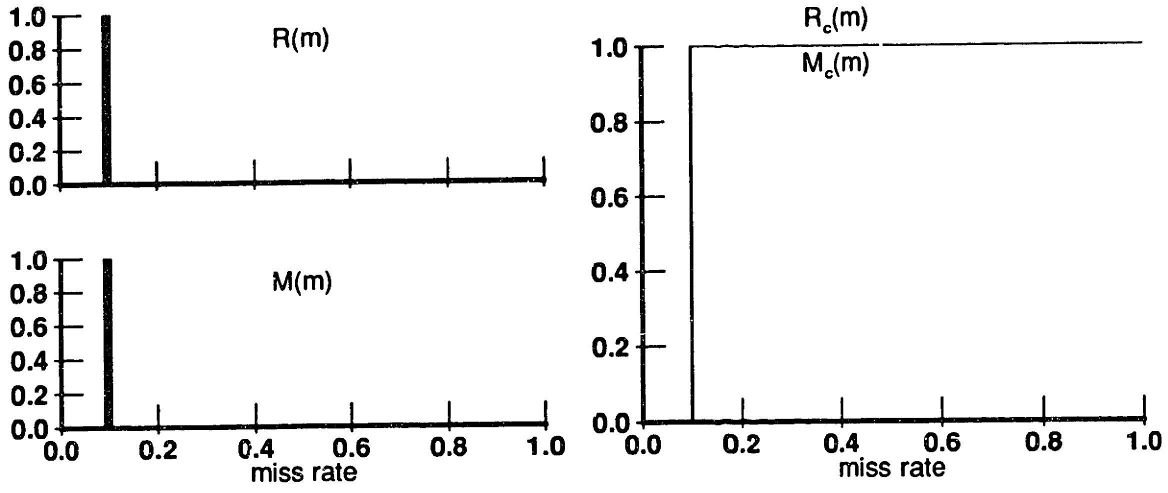


Figure 3-1: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Single Hit Rate

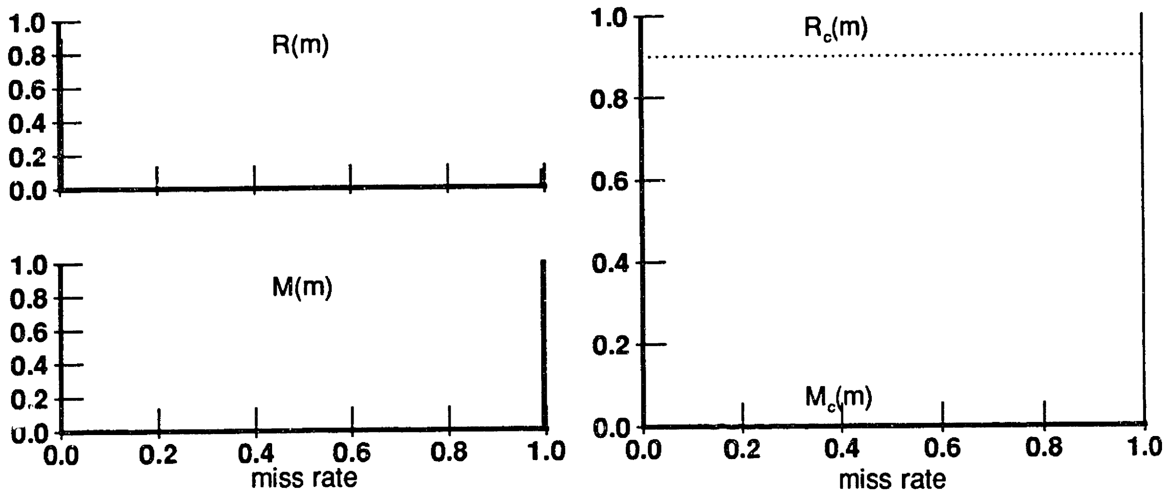


Figure 3-2: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Two Distinct Hit Rates

The statistics $R(m)$ and $M(m)$ or their cumulative forms indicate where dynamic references and dynamic misses are produced with respect to with respect to the miss rates of static references. In the initial uncorrelated case dynamic references and misses are produced in approximately equal proportion by any given static reference. In the latter cases, some references, those which always miss, account for a disproportionate fraction of the misses of the program in comparison to their share of the references. While accounting for only 10% of program references, static references with miss rates of 1 produce all the misses.

3.2. Exploitation of Static Locality Correlation

Static locality correlation can improve the effectiveness of software latency tolerance techniques. Oftentimes, different static references produce dynamic references with strikingly different behavior with respect to their cache performance. If this behavior can be anticipated, it can be exploited by treating static references differently based on the expected cache performance of their dynamic references.

Consider again the behavior illustrated in Figure 3-2, repeated below in Figure 3-3. Partition the static references of this program into two sets based on some miss rate threshold. Static references in these two sets will be referred to as goodrefs and badrefs, goodrefs corresponding to those references with the low miss rates. Each set accounts for some fraction of the total dynamic references and total dynamic misses. At the miss threshold, the height of $R_c(m)$ and $M_c(m)$ correspond to the fraction of references and misses respectively produced by goodrefs. Similarly, the distance above these curves up to 1 corresponds to the fraction of references and misses associated with badrefs. For any initial distribution of $R(m)$ similar to that assumed the badref set accounts for a relatively small fraction of dynamic references but a large fraction of dynamic misses. To produce this behavior, $R(m)$ must be bimodal with a predominance of its mass concentrated at low values of m.

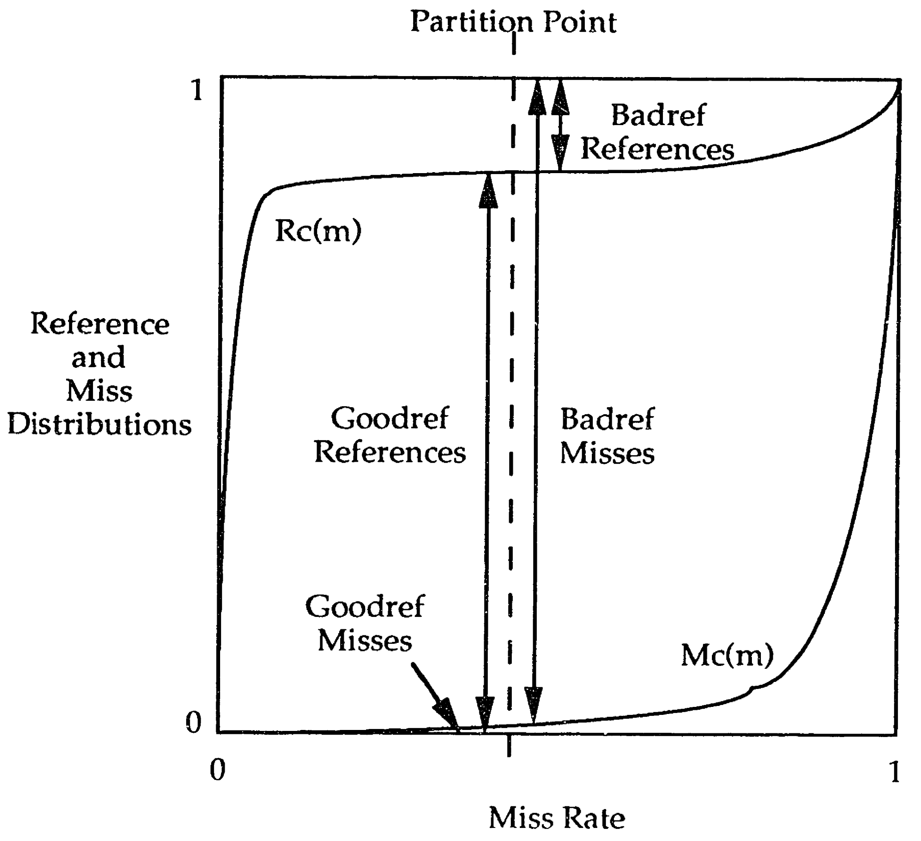


Figure 3-3: $R_c(m)$ and $M_c(m)$ for Badref Behavior

Software latency tolerance can exploit this behavior by applying differential treatment to badrefs and goodrefs. Goodrefs are not likely to produce misses and do not benefit from any treatment aimed at latency tolerance. Badrefs, in contrast, are much more likely to produce misses and should be the targets of applicable latency tolerance optimizations. Two primary hardware techniques can be used to support software memory latency tolerance, either explicit prefetch instructions or non-blocking loads. In each case, careful scheduling can ensure that transferred values are not used prior to their arrival. Application of either technique in an informed way, exploiting static locality correlation by partitioning static references, produces a superior result.

Using explicit prefetches for latency tolerance, static locality based partitioning allows software overheads to be significantly decreased. In adding a prefetch for some static reference a software overhead of O is incurred, where O is the number of instructions added. With explicit prefetches, O can be no less than 1, the prefetch instruction itself. It may be more if additional computation must be duplicated to produce a prefetch address. If prefetching is applied blindly to all static references then a breakeven point at which one might consider prefetching occurs when the following relation is true.

$$O < m * L \qquad (3-4)$$

In the formula, O is the dynamic average of overhead for individual static references. Dynamic overhead refers to a weighted average based on dynamic execution frequency. L is the latency of main memory, the latency to service cache misses, and m is the dynamic average of static reference miss rates, which is equal to the program miss rate. Optimistically assuming O is 1, if m is 10% as analyzed above, prefetching is a wash when memory latency is 10 cycles. At 20 cycles overhead potentially eliminates half of any gains associated with prefetching. For programs with 99% hit rates and thus 1% miss rates, prefetching does not even reach a breakeven point until memory latencies exceed 100.

For a program exhibiting static locality correlation, using an informed application of prefetching, much unnecessary overhead can be eliminated. Prefetches for static references which never or almost never miss, goodrefs as described above, are useless and should be eliminated. In a partitioned approach no overhead is incurred for the goodref set. Overhead is only incurred for the badref set. Derating overhead by the dynamic fraction of badrefs, b, produces the relationship below.

$$O * b < m * L \qquad (3-5)$$

For the $R_c(m)$ distribution from Figure 3-3 the dynamic fraction of badrefs, b, is exactly equal to m, the program average miss rate, and the relation can be reduced to the following.

$$O < L \qquad (3-6)$$

This latter relationship is only true for a perfectly bimodal distribution in which references either always miss or always hit. In this case the dynamic average miss rate for badrefs is 1. An alternative way to arrive at this formula is to use Equation 3-4 with an m value corresponding to the dynamic average miss rate for badrefs. In any event, opportunities for beneficial prefetching occur at much lower memory latencies when references are partitioned since overheads are reduced.

The formulas above underestimate the costs of prefetching and optimistically estimate the benefit. If prefetching is universally applied to a program it produces a noticable increase in size. This results in more misses for instruction references. The estimates above optimistically assume that the benefit of a prefetch is equal to the memory latency when a miss occurs. On a machine with non-blocking loads and an interface intrinsically providing some level of latency tolerance, the benefit is decreased. These effects only serve to increase the importance of limiting the application of prefetching to those references for which it will be productive because of static locality correlation.

Under a software latency tolerance strategy based on non-blocking loads, static locality correlation is also very important although the benefits are less easily quantified. Latency tolerant scheduling attempts to separate the use of loaded values from the static references loading those values by a distance at least equal to the memory latency. For latencies of any significant size, trying to perform this scheduling for all static references is simply an impossible task. Latency tolerant scheduling uses inherent program parallelism to overlap slow operations. For large latencies there simply is not enough parallelism in most programs to perform this scheduling. A scheduler unaware of static locality correlation is unable to focus available parallelism on those references where it is profitable. It might choose to spread available parallelism evenly across references to gain some benefit no matter where misses appear. Alternatively it might focus parallelism on a subset of references and choose unproductive references. Reference partitioning allows parallelism to be applied specifically to those references for which it will produce the maximum benefit, *i.e.* badrefs which produce most misses.

In addition to rendering an essentially impossible problem soluble, or perhaps less impossible, informed application of scheduling has other semi-quantifiable benefits. In addition to consuming parallelism, scheduling for high latencies increases the lifetimes of registers involved. The resulting register pressure increases the need for spill code, producing more static memory references. By confining latency tolerance optimization to a small subset of memory references these costs are minimized.

Static locality correlation, assuming it exists, can be a great boon to software latency tolerance. Its existence and exploitation is, in fact, what renders these techniques feasible and effective.

3.3. Program Structure

Static locality correlation isn't magic even if its application has almost magical results. It occurs as a logical consequence of program structure. Identifiable program patterns can be associated with static references that cannot miss by construction. These references form the core of goodrefs. Similarly, various program constructions can be identified which are frequently associated with badrefs.

Much of the program structure which leads to high temporal locality of programs results directly in zero miss rate goodrefs. The most important source of goodrefs is short term variable reuse. Consider execution of code produced by translating a program statement like X=X+1. If variable X is allocated in memory the statement produces a sequence containing two memory references, a load followed shortly thereafter by a store. Ignoring highly unlikely events such as interrupts between the two references, the second reference, the store, cannot miss. If X is also used in a preceding or following statement another reference to X is a goodref.

Given caches with block sizes larger than the units of memory accessed by memory references, some forms of spatial locality also lead to goodrefs. Consider a cache block aligned aggregate data structure with several components. Assume static references accessing these components occur in a fixed order over a short span of time, not an uncommon pattern. The first reference to some component in a cache block moves the block into the cache. Subsequent static references to other components in the same block never miss and are goodrefs.

Caches typically can hold a relatively large number of values simultaneously and as a consequence address reuse over longer time frames can result in systematic hits. Access patterns such as those described above can produce goodrefs even when relatively widely spaced, as long as the pattern occurs systematically with the same order and the spacing is not so wide that the jointly used value is flushed from the cache between the two references. Over wider time frames, goodrefs may be produced by different variables which share a common address. Consider two small procedures called from another in short succession and called nowhere else. Some set of stack accesses in the latter procedure are likely to be rendered goodrefs by corresponding stack accesses in the former. Thus goodrefs arise not only from fundamental algorithmic structure but also from storage allocation conventions which favor the short term reuse of storage in a systematic way.

Badrefs are somewhat harder to motivate from a fundamental analysis of program structure. Badrefs are in some sense a residual, the static references left over after eliminating structural goodrefs. Any static reference can lie in the cache shadow of some other static reference which uses the same address. If a static reference follows shortly after another static reference which uses the same address it is a goodref irrespective of its structuıe. As a consequence, there are no specific program structures which can be pointed to as always producing badrefs when they occur. Despite this, there are structural forms in programs commonly associated with badrefs.

The most common program structure associated with badrefs is the occurance of aggregate data structures such as arrays within loops. Assume the elements of an array are not contained within the cache. Consider a loop which traverses the elements of the array. A single static reference often produces the first dynamic reference to each element of the array. In successive array iterations it accesses new array values, systematically producing misses. Other static references accessing the array values are rendered goodrefs by this single badref.

Generalizations of the array and loop behavior described above similarly lead to badrefs. Large aggregate data structures interconnected with pointers exhibit the same form of behavior when traversed within loops. As another example, consider a self-recursive procedure which frequently performs deep linear sequences of recursive calls. The stack, in this situation acts as a generalization of a large array, while the recursive pattern acts as a generalized loop. Static references which produce the first access to a particular stack location within the procedure may be badrefs.

While not corresponding directly with the notion of goodrefs and badrefs it is possible for misses incurred by a particular static reference to be correlated with a region of code which does not include the corresponding memory reference. Turning again to loops, it is not uncommon that a set of misses occur systematically on the first iteration of a loop and then do not recur on subsequent iterations. This can happen when a static reference to a single variable or location occurs within the loop and the corresponding location is not used anywhere directly preceding the loop. Misses of this form are statically correlated to the code directly preceding the loop since execution of this code implies the eventual occurrence of the misses. One can view this situation as the aliasing of a virtual badref and a virtual goodref onto a single static reference. Applying loop peeling to the first iteration of the loop separates these virtual references into two distinct static references, one of which is a badref and one of which is a goodref. Static correlations of this form can be exploited in addition to goodref/badref correlations when they can be identified.

Although an example was presented above in which cache blocks larger than referenced data produced goodrefs, more often than not cache blocks serve to decrease apparent static locality correlation. Blocking in caches tends to alias virtual sets of goodrefs and badrefs onto the same static reference for array accesses in loops. Cache blocking may, for instance, cause every other dynamic reference produced by a static array reference to hit if the reference has a stride equal to one half the cache block size. For data structure references which are unaligned with respect to cache blocks, blocking tends to smear a set of logically related misses across a set of memory references. Other cache implementation issues, such as limited associativity tend to decrease observed static locality correlation as well. A question then arises whether sufficient static locality correlation inherently associated with program structure exists after obfuscation by non-ideal caches. This question will be addressed with empirical data in an upcoming section.

3.4. An Engineer's Description of Static Locality Correlation and the Badref Model

High school dances are sometimes classified as formals and semiformals. This section will attempt to produce a semiformal definition of static locality correlation, an engineer's definition rather than a theoretician's, worthy of a suit and tie if not a tuxedo. Semiformally speaking, static locality correlation states that the program behavior known as locality is correlated with particular static memory references and that each individual reference may exhibit its own distinct locality behavior.

In order to define static locality correlation, one is first forced to venture onto thin ice and define locality. [60] qualitatively describes locality as follows:

Locality of Reference

> Reference to location X at time t implies that the probability of access to location $X + \Delta X$ at time $t + \Delta t$ increases as ΔX and Δt approach 0.

This definition is characteristic of the semiformality often associated with descriptions of locality. Locality is a somewhat squishy concept.

Engineers prefer things which can be quantified and measured. Since most people agree that locality is the phenomenon which makes caches work well we will quantify locality in terms of cache behavior. The locality exhibited by a stream of addresses will operationally be defined as the hit rate exhibited by an ideal test cache for the stream. Thus a test cache acts as a locality meter. The test cache used to quantify locality is fully associative and uses an optimal replacement policy subject only to the constraint that data for each reference in the stream must be resident in the cache when the transaction is satisfied. Because of their optimal replacement policy, locality test caches cannot be implemented online but they can be simulated. Since cache hit rates depend on both total cache size and cache block size, locality will be parameterized in terms of the cache and block size of the test cache. $L(C, B)$ represents the locality of an address stream measured using an ideal test cache with total size C bits and blocksize of B bits and is equal to the hit rate of the stream in a test cache of the appropriate size.

Using this definition of locality, the notions of temporal and spatial locality correspond to partial derivatives of L with respect to C and B. The incremental temporal locality of a stream measured at point $L(C, B)$ is $\frac{\partial L(C,B)}{\partial lgC}$ and the incremental spatial locality is $\frac{\partial L(C,B)}{\partial lgB}$. The logarithms arise due to the fact that C and B typically grow as powers of two.

Under this operational definition, the locality of a subset of references within a stream can be defined similarly as the hit rate of references in the subset, when measured in the context of the stream as a whole. The locality of a static memory reference instruction can be defined as the locality of the set consisting of the dynamic references produced by the static reference.

Based on the operational definition of locality above, the following static locality correlation hypothesis acts as a definition for static locality correlation:

The Static Locality Correlation Hypothesis

> Locality behavior of individual static memory reference instructions is primarily influenced by the static structure of programs rather than the dynamic behavior of programs. As a consequence, within a single program, individual static references can and will exhibit highly varying locality behavior. Furthermore, the locality behavior of any particular static reference is not strongly influenced by perturbational changes in input data.

Based on this foundation, we now build the badref hypothesis. The badref hypothesis incorporates two additional observations. Static miss rates are polarized with respect to the average. Also, low miss rate references often produce more dynamic references than high miss rate references. Not only is locality behavior statically correlated, it is bimodally partitionable. Many static references exhibit very high locality, *i.e.* low miss rates, while others exhibit low locality and high miss rates relative to the average miss rate of a program. If static references are partitioned into two sets, one set will account for most misses while the other accounts for most memory references.

The Badref Hypothesis

> Most misses incurred by programs are produced by a small subset of all static references, termed badrefs, which have miss rates well in excess of the average miss rate exhibited by the program as a whole. These references account for a

disproportionately high fraction of misses; the fraction of dynamic misses generated by badrefs is much larger than the corresponding fraction of dynamic references. Further, because of static locality correlation, the identity of the badrefs within a program is not a strong function of the input data set.

3.5. Validation of the Static Locality and Badref Hypotheses

This section presents empirical evidence to validate the static locality and badref hypotheses. Two different forms of experiments address two different aspects of program behavior captured by the hypotheses. First, data is presented illustrating the variability of miss rates of static memory reference instructions in programs. Static locality correlation predicts that miss rates of static references should be varied rather than focused near the average program miss rate. The badref hypothesis further predicts that this variation should be bimodally partitionable, with a large component at very low miss rates and a smaller component at much higher miss rates relative to the program average.

Static reference miss rate variability is examined for our complete benchmark set for several baseline test cache configurations. Rather than directly measuring locality by the definition above involving optimal caches, we measure the behavior of several realizable cache configurations. One configuration is fully associative with LRU behavior and should serve as a good approximation to the optimal test caches in the locality definition. Empirical data is also provided to illustrate the impact on static locality correlation of non-idealities in cache behavior associated with decreased associativity and effects of operating in different regions of $L(C, B)$ locality space.

The second aspect of behavior studied is variability of static reference locality associated with perturbations in input data. Most benchmarks in the initial study only have a single input data set so several UNIX text processing benchmarks have been added in order to gather input variation data.

3.5.1 Miss Rate Variability of Individual Static References

The first experiments to validate the static locality and badref hypothesis measure the variability of miss rates of individual static references. Static reference miss rates are measured for three baseline cache configurations. These configurations are denoted **FA**, **DM** and **DMV** and are described in more detail below.

FA is a fully associative cache with an LRU replacement policy. The size of the cache is 1024 4-byte words and the block size is 2 words.

DM is a direct-mapped cache. The size of the cache is 16 Kwords and the block size is 4 words.

DMV is a direct-mapped cache with a victim cache. The main cache is identical to that of **DM**. The victim cache is fully associative and holds 8 blocks. References are considered to be hits if they hit in either the main cache or the victim cache. Victim caching is a technique aimed at reducing conflict misses in direct-mapped caches. Jouppi describes victim caching in [24].

The benchmarks studied in this experiment are gathered from the SPEC benchmarks [55]. Data is presented for 7 of the 10 benchmarks in the SPEC suite plus one additional program derived from a SPEC benchmark. The simulation environment used to gather data does not support multiple processes. SPEC benchmark eqntott makes calls to the Unix fork command. Benchmark data for eqntott is based on a modified version of the actual SPEC eqntott benchmark which excludes work which occurs in forked processes in the original version.

Data for three sample benchmarks illustrating a range of static locality behavior is illustrated in the following figures. For each benchmark, graphs are provided indicating $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ vs m for each of the three baseline caches. Corresponding data for the remainder of the benchmarks is provided in Appendix A.

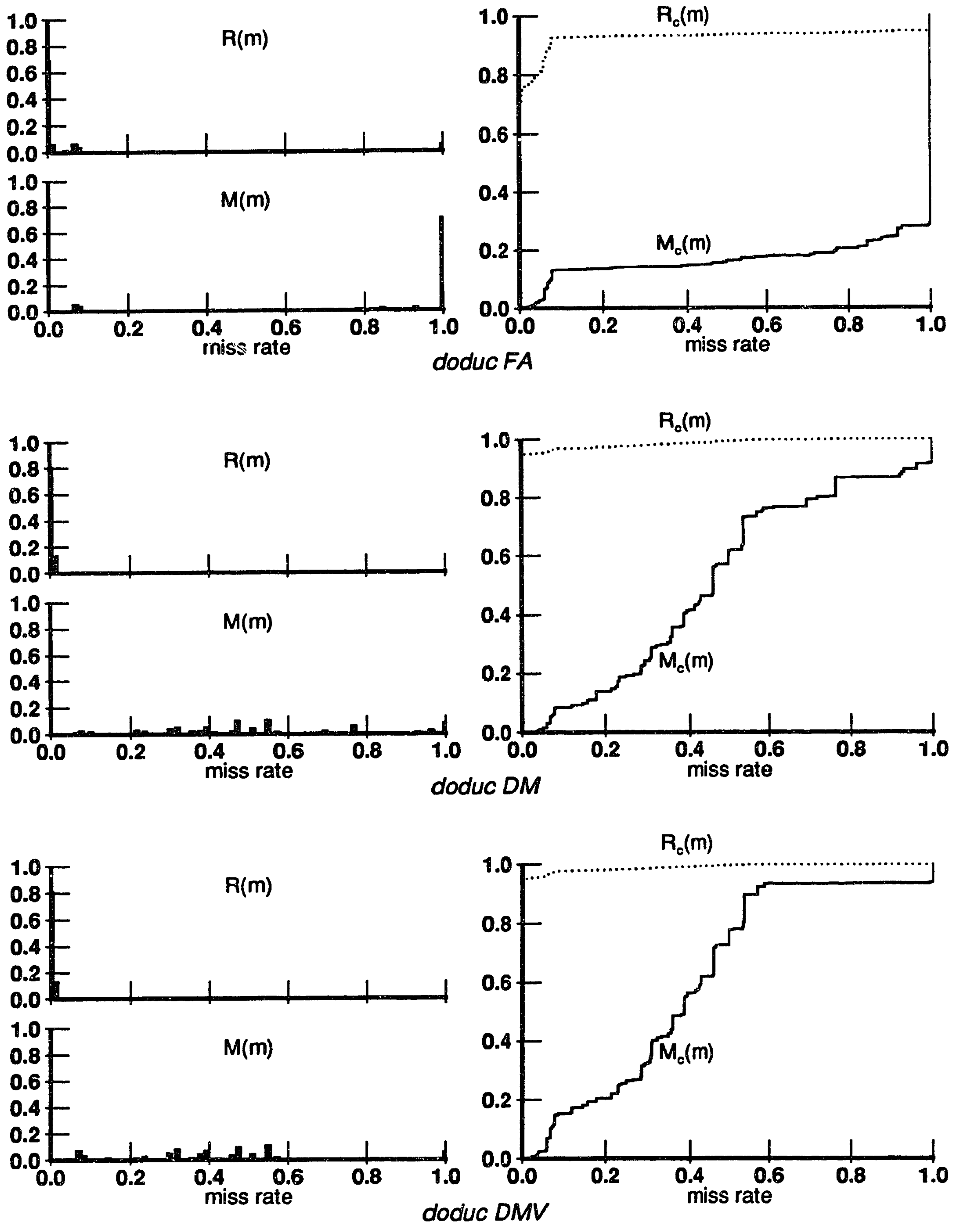


Figure 3-4: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Doduc **FA**, **DM** and **DMV**

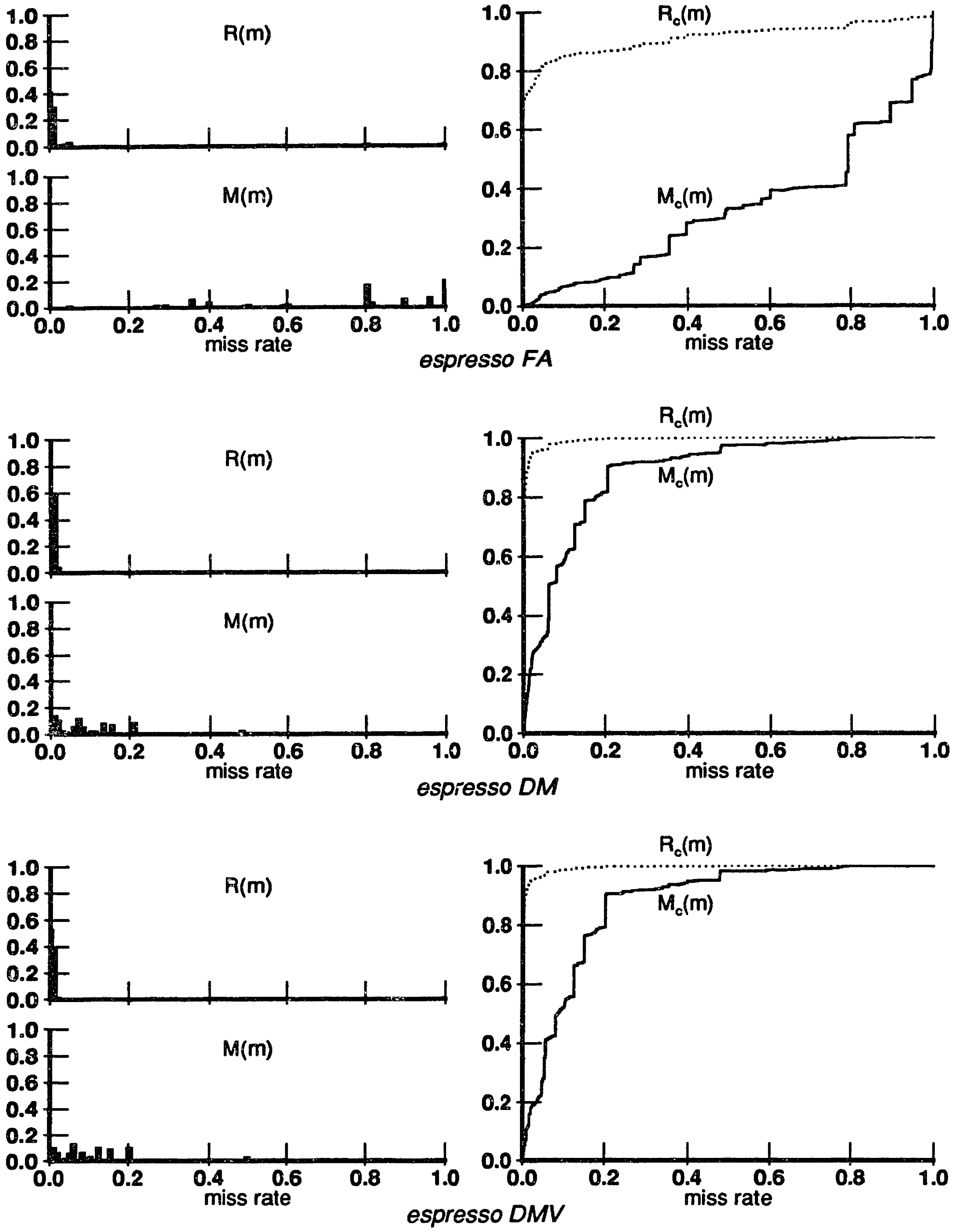


Figure 3-5: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Espresso **FA**, **DM** and **DMV**

The simulation data in the figures above and in Appendix 1 supports the hypotheses of static locality correlation and badref behavior. For the **FA** model, representing a relatively small cache,

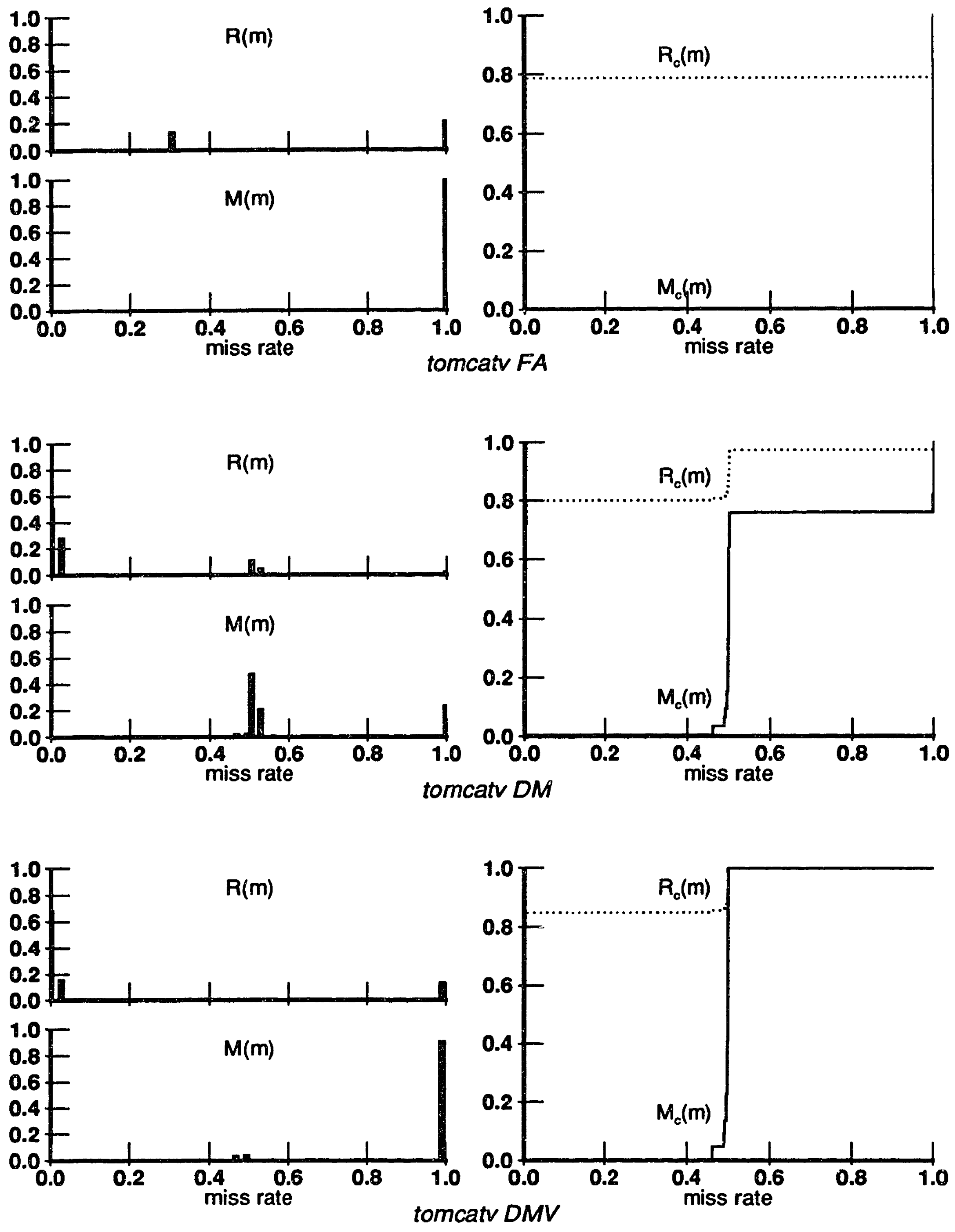


Figure 3-6: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Tomcatv **FA**, **DM** and **DMV**

two of the benchmarks illustrated above and in all about half of the benchmarks examined exhibit extremely pronounced badref behavior. For these benchmarks a large fraction of misses occur for static references with 100% miss rates while the majority of references occur for static references with essentially 0% miss rates. For the other **FA** experiments badref behavior is slightly less pronounced but is still quite noticable. Based on $R_c(m)$ and $M_c(m)$, it is clear that references can be partitioned into two sets, one of which accounts for most references while the other accounts for most misses.

Simulations using the **DM** and **DMV** models also support the static locality correlation hypothesis. Due to the large fraction of references at or near 0% miss rate in several of these experiments, it is not obvious from $R(m)$ data that static miss rates are varied. Other references account for too small a fraction of total dynamic references to be noticable in the graph. If one instead turns to $M(m)$ data, two observations can be made. First, it is apparent that static references exhibit a spectrum of miss rates, even in benchmarks with relatively high average hit rates. (It is necessary to have some static reference with a miss rate corresponding to that exhibited by any component of $M(m)$.) Furthermore, due to scaling by m in $M(m)$, it is precisely these infrequently occuring static references with miss rates above average which account for a large fraction of misses. This is the essence of the behavior described by the badref model.

Data based on **DM** and **DMV** cache models support the badref hypothesis. In some of the benchmarks the **DM** and **DMV** models still exhibit average miss rates between 5 and 10%. These benchmarks exhibit fairly pronounced badref behavior just as data in the **FA** tests. In other benchmarks, including espresso and fpppp, average hit rates have fallen to below 1%. These data illustrate the low miss rate manifestations of static locality correlation and badref behavior. Data from these tests exhibits a large spike of goodrefs, references with miss rates at or near zero which do not contribute at all to program misses. For these low miss rate tests, misses are no longer primarily associated with references with very high miss rates in absolute terms. It is still true, however, that most misses are associated with references with hit rates 10 to 100 times higher than the average program miss rate. Looking specifically at espresso, about 80% of misses occur for static references with miss rates between 5 and 20% while the average miss rate is about 0.5%. References producing misses account for a negligible fraction of total dynamic references. Once again, static references can be partitioned into one set producing most dynamic references and a disjoint set producing most dynamic misses.

Having claimed that the data above supports the badref hypothesis, we propose the following specific operational test for determining when badref behavior exists. Examine a plot of $R_c(m)$ and $M_c(m)$ and consider drawing a vertical line across the plot partitioning it into two regions. Badref behavior exists when it is possible to draw a vertical, bisecting line which crosses $R_c(m)$ at a high value and crosses $M_c(m)$ at a low value. The degree to which badref behavior can be productively exploited depends on the position of this bisecting line. References with miss rates larger than the bisection value are considered badrefs. If the partitioning occurs for a large miss rate value then all badrefs exhibit high miss rates and optimizations applied to badrefs target only high miss rate references. If appropriate partitioning can only be achieved using a low miss rate value then selective optimizations applied to badrefs target lower miss rate values.

In addition to static locality behavior associated with high and low average miss rates, comparison of **FA**, **DM** and **DMV** illustrates the interaction of static locality correlation and changes in cache block size and associativity. The **DM** and **DMV** models have a block size which is twice that of the **FA** model. In the tomcatv benchmark above, as well as several others in Appendix 1, the increased block size exploits spatial locality to transform references with 100% miss rates into references with 50% miss rates. While this has changed the specific miss rates of some references, it has not diminished static locality correlation. In a sense, the fact that cache blocks interact in a predictable way with the miss rates of specific static references further drives home the point that static program structure plays a determining role in cache performance, the main premise of static locality correlation.

DM and **DMV** data for tomcatv illustrates one potential consequence of associativity changes on badref behavior. In the **DM** data, conflict misses have produced some 100% miss rate static references which would otherwise exhibit a 50% miss rate as indicated in **DMV**. The 50% miss rate arises due to the blocksize which is twice the size of a double precision floating point value, the predominant form of data used in tomcatv. In this particular experiment conflict misses have occurred in a structurally predictable way, increasing the polarization of miss rates, resulting in behavior closer to the badref ideal. This behavior is somewhat counterintuitive. In general, one might expect that conflict misses would represent a relatively uncorrelated component of misses and thus act to increase the miss rates of goodrefs more significantly than those of badrefs, lessening the degree of badref behavior.

3.5.2 Cache Design Perturbations

The static locality and badref hypotheses are not intended to imply that cache behavior is completey independent of the particular design parameters of a cache. Instead they claim that given a fixed program and a fixed cache, the hit/miss behavior of individual static reference can be characterized in a meaningful way which will be valid for repeated runs of the program.

While changes in behavior of individual references are expected to accompany changes in cache design, one might anticipate that in some regions of $L(C, B)$ locality space or in the presence of cache non-idealities such as limited associativity the static locality and badref hypotheses might start to break down. Some forms of misses are likely to be more highly structurally correlated than others. In a regime of behavior in which uncorrelated misses outweigh correlated misses, the static locality and badref hypotheses will no longer accurately characterize program behavior.

As an example, consider a very large cache in which the predominance of misses occur in conjunction with multiprogramming context switches. If context switches occur asynchronously with respect to program behavior then the basic premise upon which the static locality hypothesis is founded may no longer be true. While the example above is not specifically addressed in this section, data is presented examining the impacts on static locality correlation of changes in total cache size, cache block size and cache associativity. In each case the **FA** model above is used as a baseline model and the specific parameter of interest is varied over a range of values.

3.5.2.1 Impact of Cache Size

Cache misses are sometimes classified as compulsory, capacity and conflict misses [19]. As the size of a cache is varied, the relative fraction of these different forms of misses shifts. One can view a study of static locality correlation as a function of cache size as an investigation of the relative degree of correlation of different forms of misses. In this study, fully associative caches are used, eliminating conflict misses. As cache size is increased, the relative fraction of compulsory misses increases and the fraction of capacity misses decreases.

The figures below indicate the results of simulations for caches based on the **FA** model parameters, fully associative caches with a block size of 2 words. The cache size is varied from 256 to 32K words or 1 Kbyte to 128 Kbytes. The study is restricted to 4 of the benchmarks, doduc, espresso, fpppp and tomcatv. These benchmarks are chosen because they exhibit noticeable variation in behavior in the initial study between the **FA** and **DM** models.

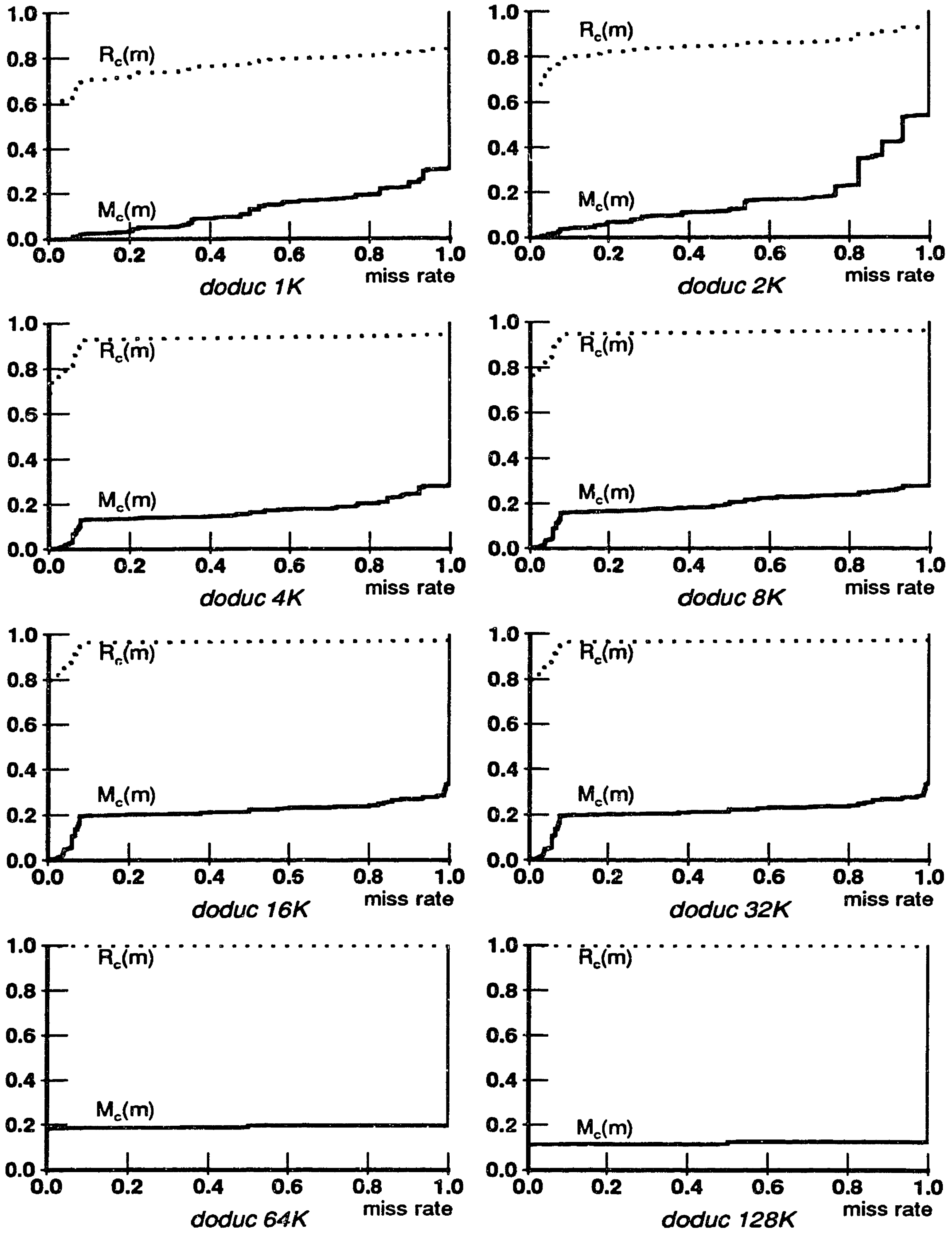


Figure 3-7: Doduc $R_c(m)$ and $M_c(m)$ variation with cache size

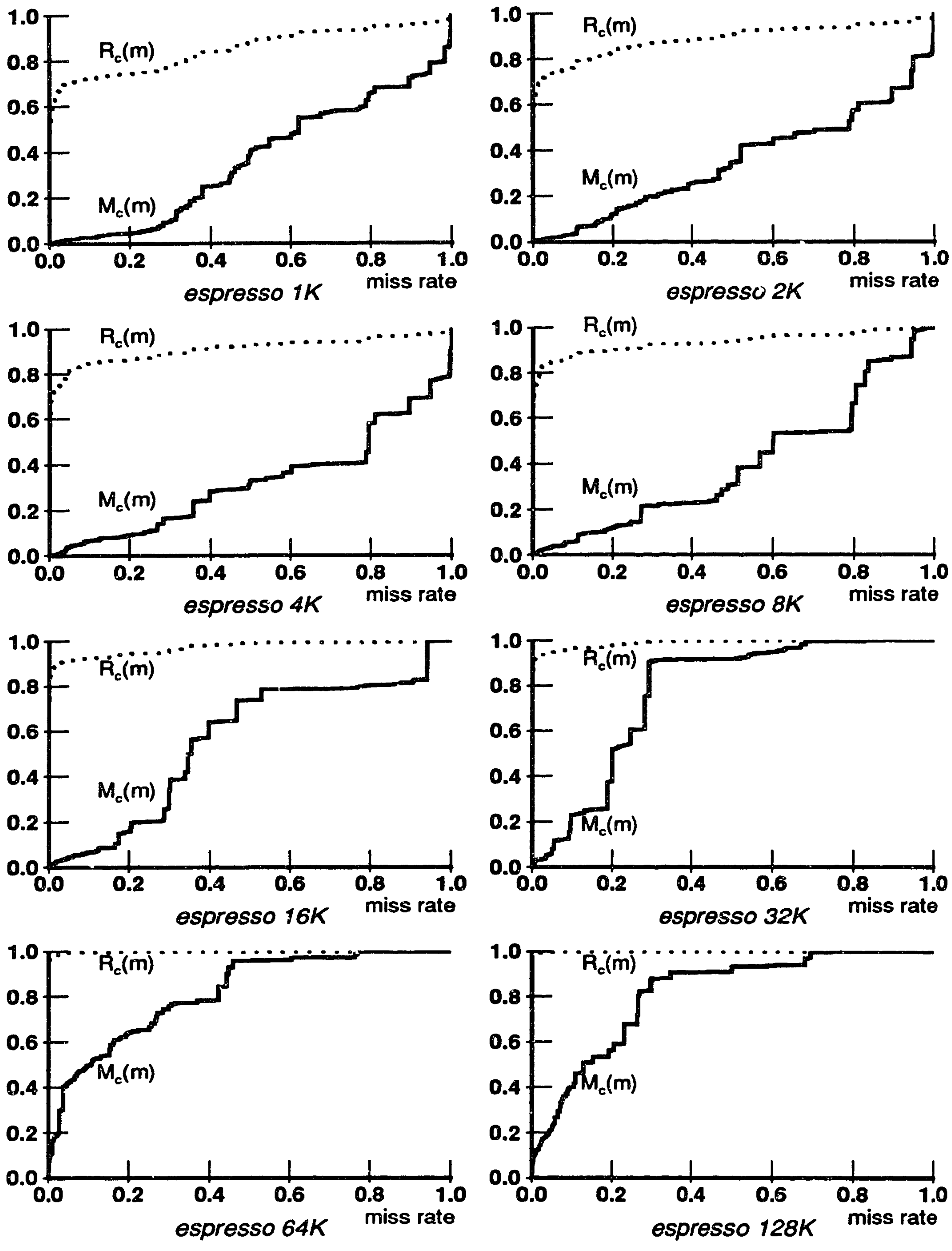


Figure 3-8: Espresso $R_c(m)$ and $M_c(m)$ variation with cache size

This data does not conclusively indicate an increased propensity for static locality correlation in either compulsory or capacity related misses. In the case of doduc for the largest caches most misses

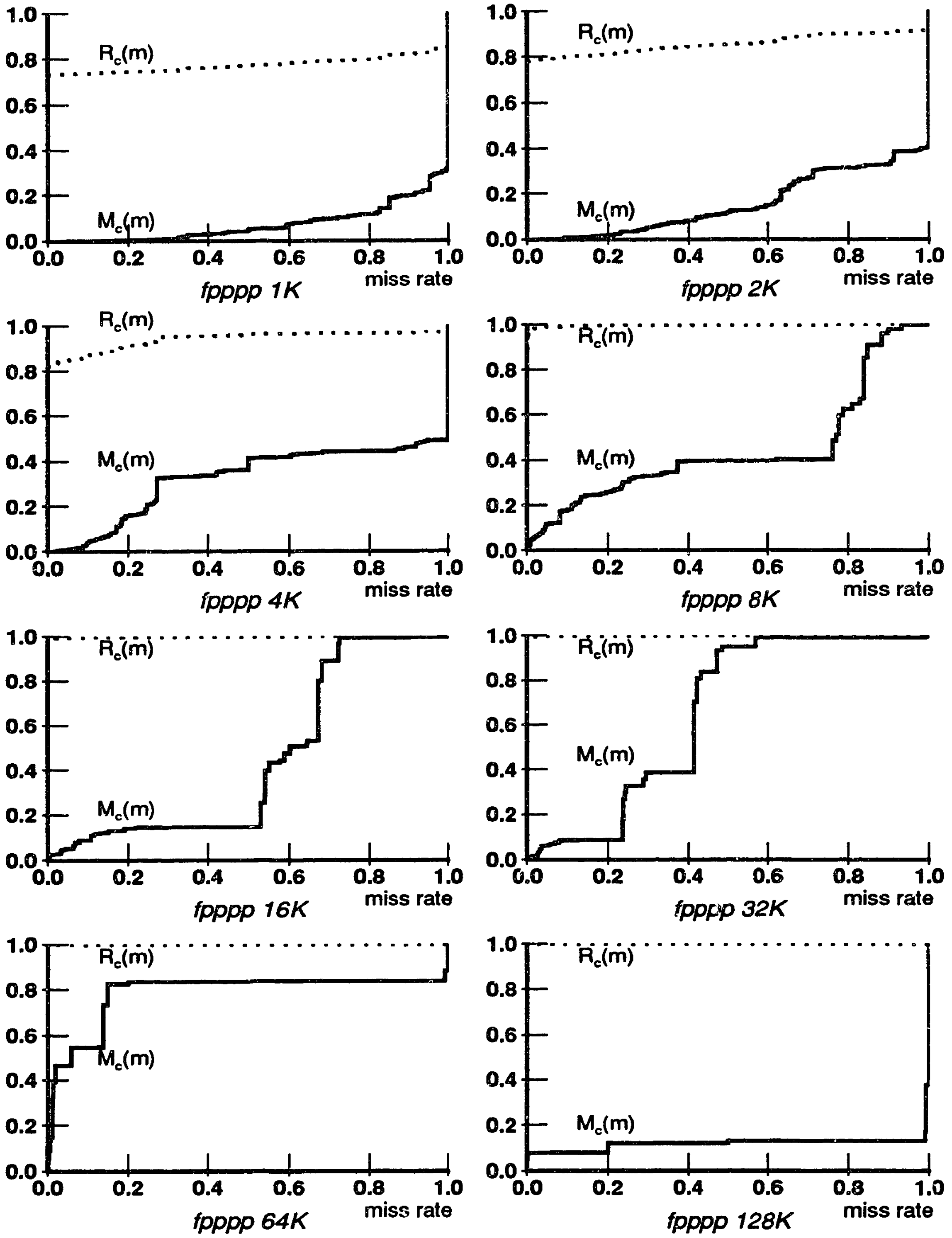


Figure 3-9: Fppp $R_c(m)$ and $M_c(m)$ variation with cache size

are compulsory misses. The benchmark is operating in a regime of very low average miss rate, 0.002%, yet static locality correlation and badref behavior is exhibited even to the point of showing

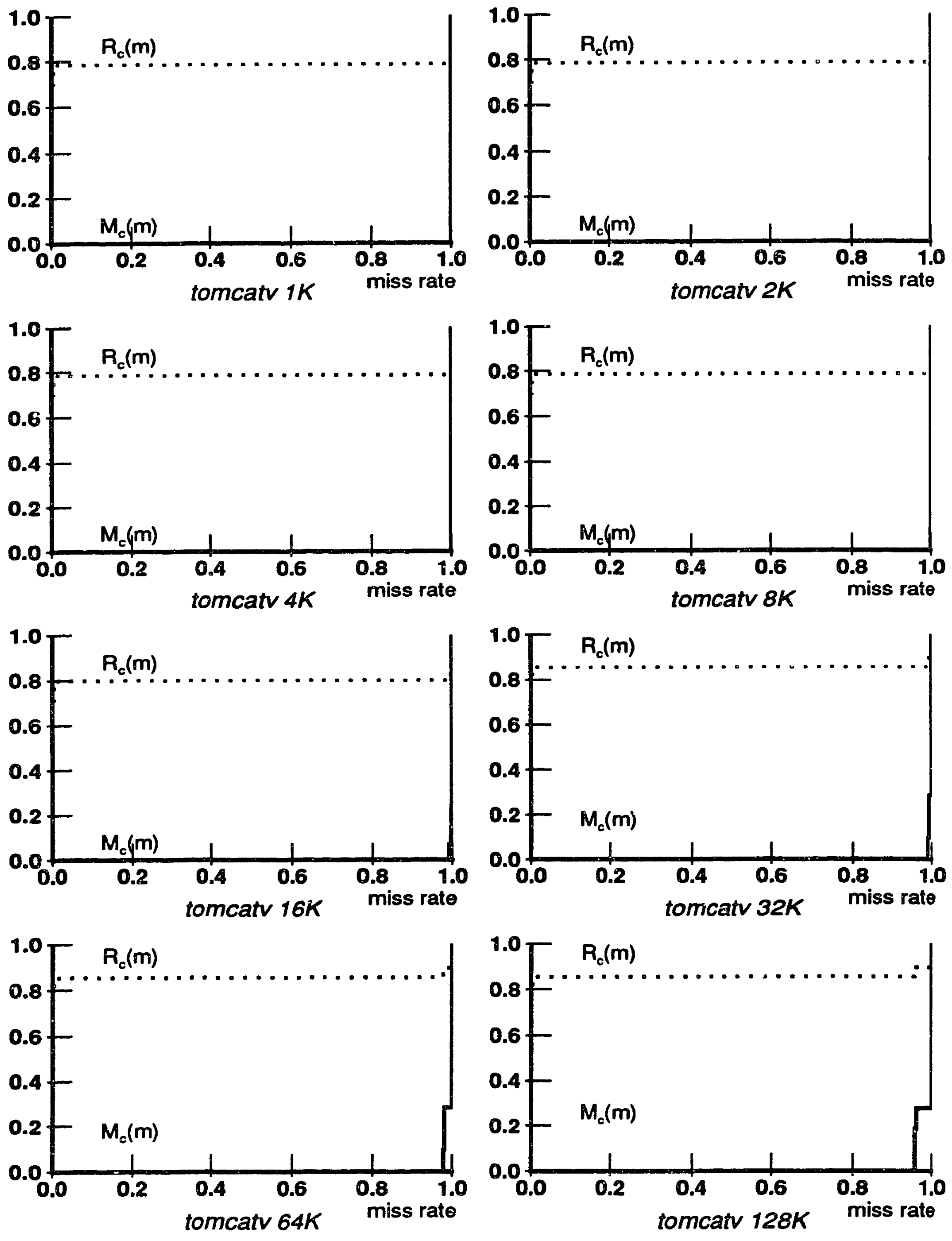


Figure 3-10: Tomcatv $R_c(m)$ and $M_c(m)$ variation with cache size

100% miss rate static references. An initialization section of the program which is not repeatedly executed is the source of the 100% miss rate static references . In contrast, for large cache sizes espresso and fpppp do not have an appreciable number of high miss rate references. Using the operational badref test described, it is still possible to partition the references of these benchmarks but the partitioning must occur at a low miss rate value. The locality behavior of programs in high locality areas of $L(C, B)$ space depends on the way in which the static references exhibiting the remaining misses are used by the program.

One might make the observation that static locality correlation behavior in high locality regimes is somewhat less interesting than behavior in lower locality regimes from a latency tolerance standpoint. If locality is very high, indicating a very high cache hit rate, then the time associated with residual misses is not likely to be a substantial component of program execution time. Where locality is not as high, cache misses represent a more noticable performance component. In situations where caches are small enough to exhibit lower hit rates, static locality correlation and badref behavior is pronounced in the data.

3.5.2.2 Impact of Block Size

Block size variations can interact with static locality correlation in a number of ways. The most noticable effect occurs for small stride array references within loops. Increases in blocksize exploit the spatial locality inherent in these references. When these references are badrefs, increased block size decreases the miss rate of the references, decreasing the overall polarization of miss rate values.

Block size can have a variety of effects in the context of multiword aggregate structures other than arrays. If structures are aligned with cache blocks then increased block size can produce goodref and badref accesses to structure components, increasing hit rates but potentially resulting in strong badref behavior with 100% miss rate badrefs and 0% miss rate goodrefs. Unaligned data structures in the context of blocks are likely to result in a smearing of inidividual hit/miss behavior of static references to structure components.

Finally, increases in block size at constant cache size decreases the total number of independent data values which can be stored in a cache, potentially leading to decreases in hit rate if this effect dominates hit rate increases associated with increased exploitation of spatial locality. In general, when hit rates decreases, static locality correlation and badref behavior tend to increase.

Block size impacts on static locality and badref behavior are studied using a set of caches produced by manipulating the block size of the baseline **FA** model. At constant total cache size, the block size is varied from 2 words to 64 words. Single word block size is not investigated because then double precision memory references could produce multiple cache misses, complicating simulation behavior. Data for experiments using benchmarks doduc, espresso, fpppp and tomcatv are presented in the figures below.

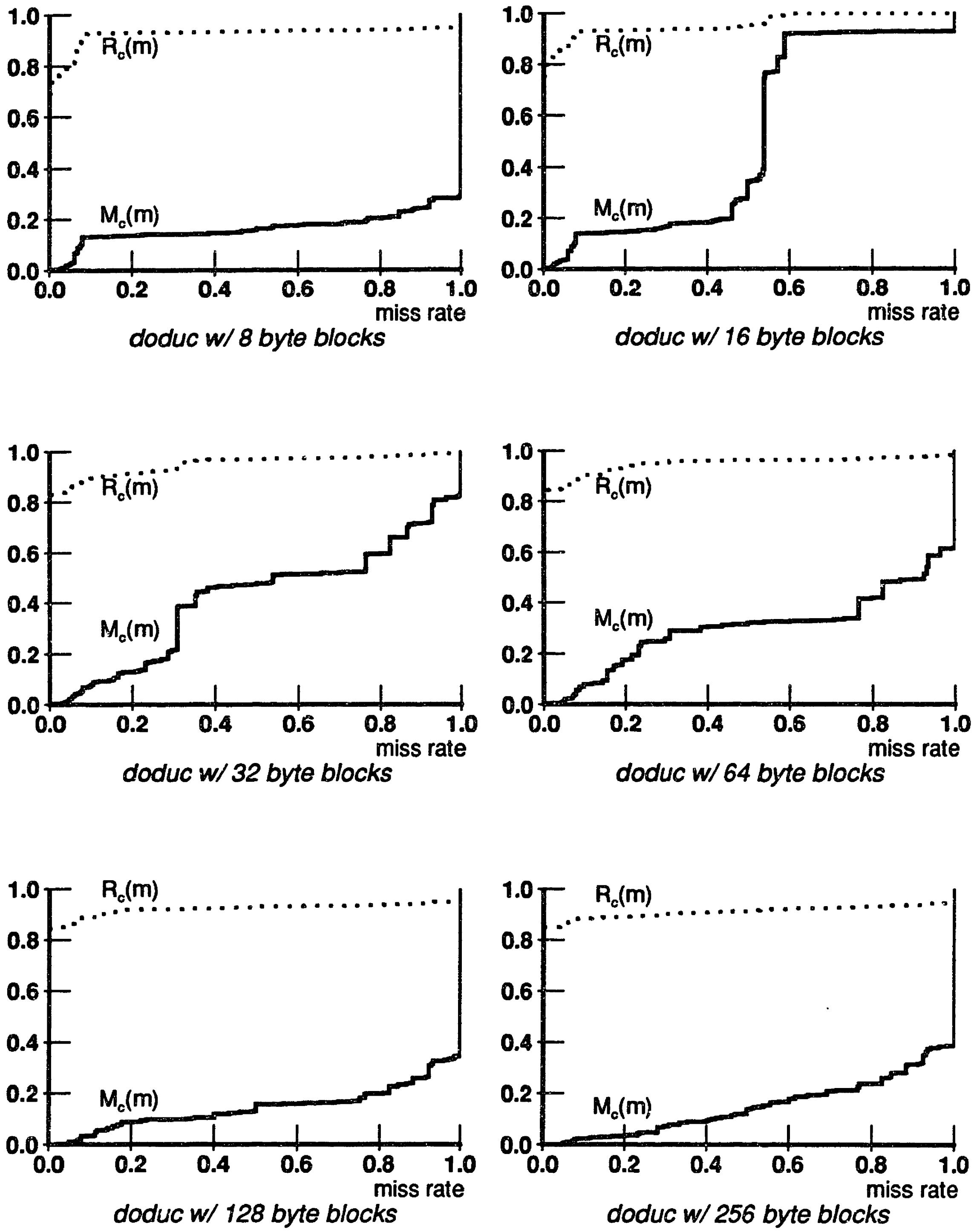


Figure 3-11: Doduc $R_c(m)$ and $M_c(m)$ variation with block size

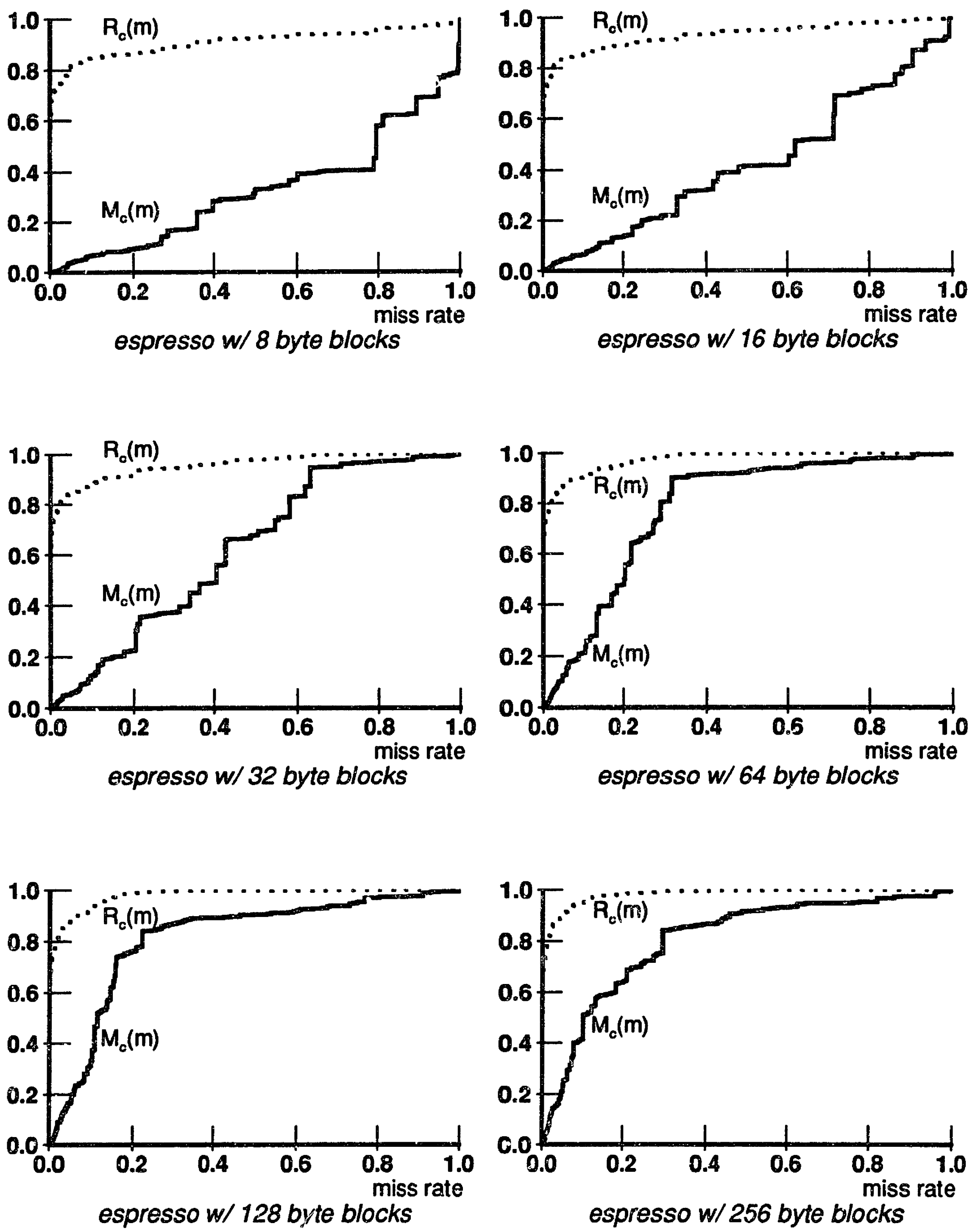


Figure 3-12: Espresso $R_c(m)$ and $M_c(m)$ variation with block size

Two effects are exhibited in the doduc data. In moving from block size 8 to 16 and then to 32, noticable shifts in miss rate of references occurs. 100% miss rate references are converted to 50% miss rate references and then to 25% miss rate references. This has the effect of moving the required

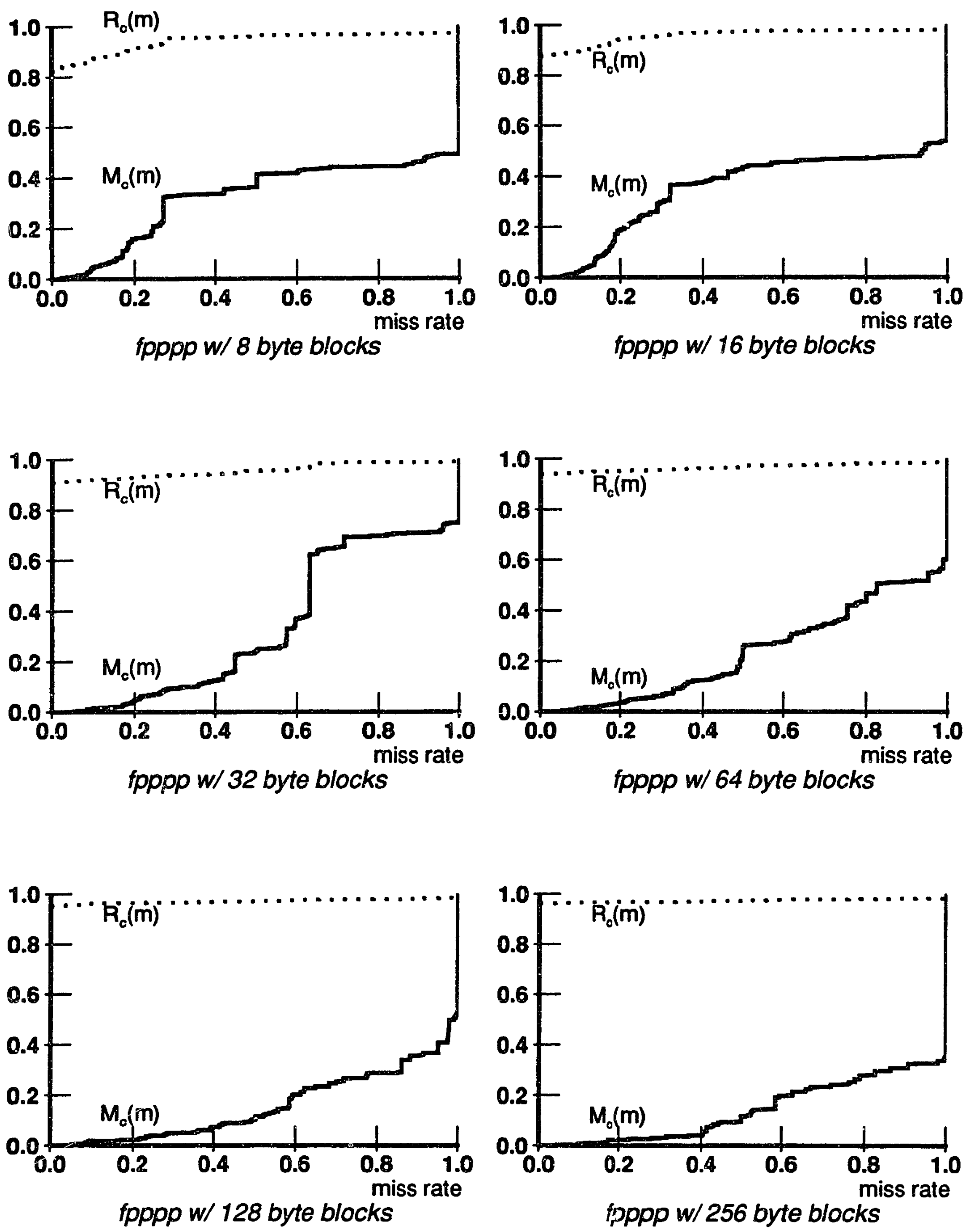


Figure 3-13: Fppp $R_c(m)$ and $M_c(m)$ variation with block size

partition point to the left, decreasing the polarization of miss rates and the effective utility of badref behavior. While it is not directly evident from the graphical data, a second effect is that after an initial drop in overall miss rate of almost a factor of two when moving from 8 to 16 byte blocks the

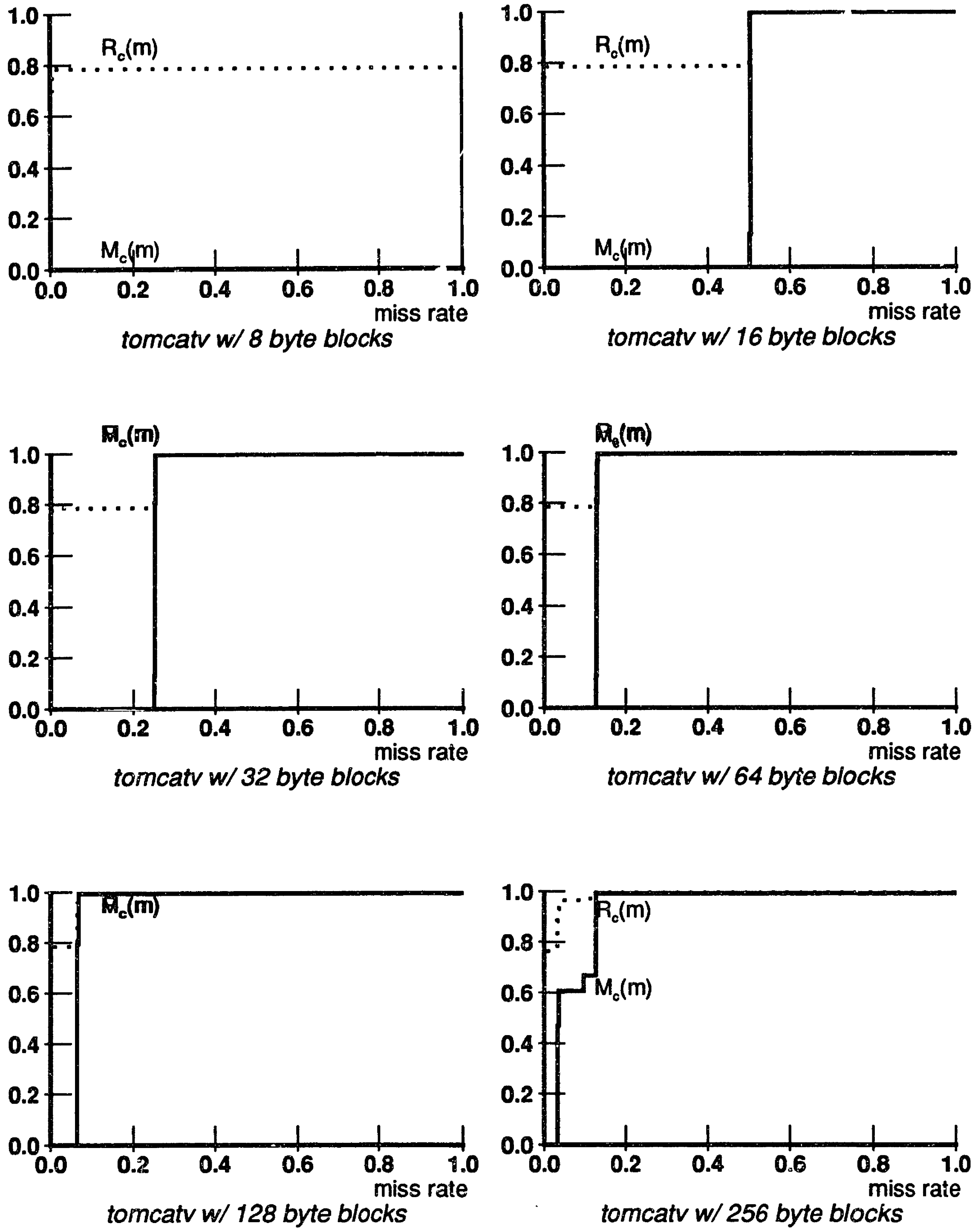


Figure 3-14: Tomcatv $R_c(m)$ and $M_c(m)$ variation with block size

miss rate rises with further increases in block size. These increases in miss rate tend to increase the polarization of miss rates, enhancing the utility of badref behavior.

In the case of fpppp, block size variations do not dramatically impact badref behavior, serving primarily to enhance it rather than detract from it despite a gradual decline in overall miss rate with increasing block size. Performance improves for the lower miss rate references rather than higher miss rate references as block size is increased.

The average hit rate for espresso is improved for each increase in block size and in each case this is accompanied by a decrease in miss rate polarity and badref behavior. This behavior is characteristic of what one might intuitively expect to occur in general, even though it is only exhibited by one of the three benchmarks examined.

The benchmark most noticably effected by block size variations is tomcatv. Each increase in block size, up to 64 word blocks, leads to a decrease by a factor of two in the miss rates of essentially all badrefs and in the average program miss rate. The next chapter on compiler heuristics discusses loop unrolling techniques which can be applied when large cache block sizes lead to low to moderate miss rate badrefs in loops. These techniques can lead to polarized miss rate behavior even when overall miss rates are low.

3.5.2.3 Effects of Associativity and Conflict Misses

While changes in cache size shift the balance of compulsory and capacity misses, variations in associativity impact conflict misses. One might expect that conflict misses would represent a relatively uncorrelated form of miss and that as a consequence miss rates would become more polarized with increases in associativity, leading to more pronounced badref behavior.

Victim caching is a technique which can be applied to lessen the impact of conflict misses on direct mapped caches. A victim cache is a small fully-associative cache which is loaded with blocks which are replaced in the main direct-mapped cache. Rather than studying set-associative caches of varying degrees of associativity, the performance of conflict misses can be examined by using victim caches with varying sizes.

This study uses a family of caches with the cache and block size parameters based on the **FA** model, namely a 1024 word capacity and a blocksize of 2 words. The study includes direct-mapped caches with no victim cache, with victim caches of 2, 4, 8, 16, and 32 blocks and finally a fully associative cache.

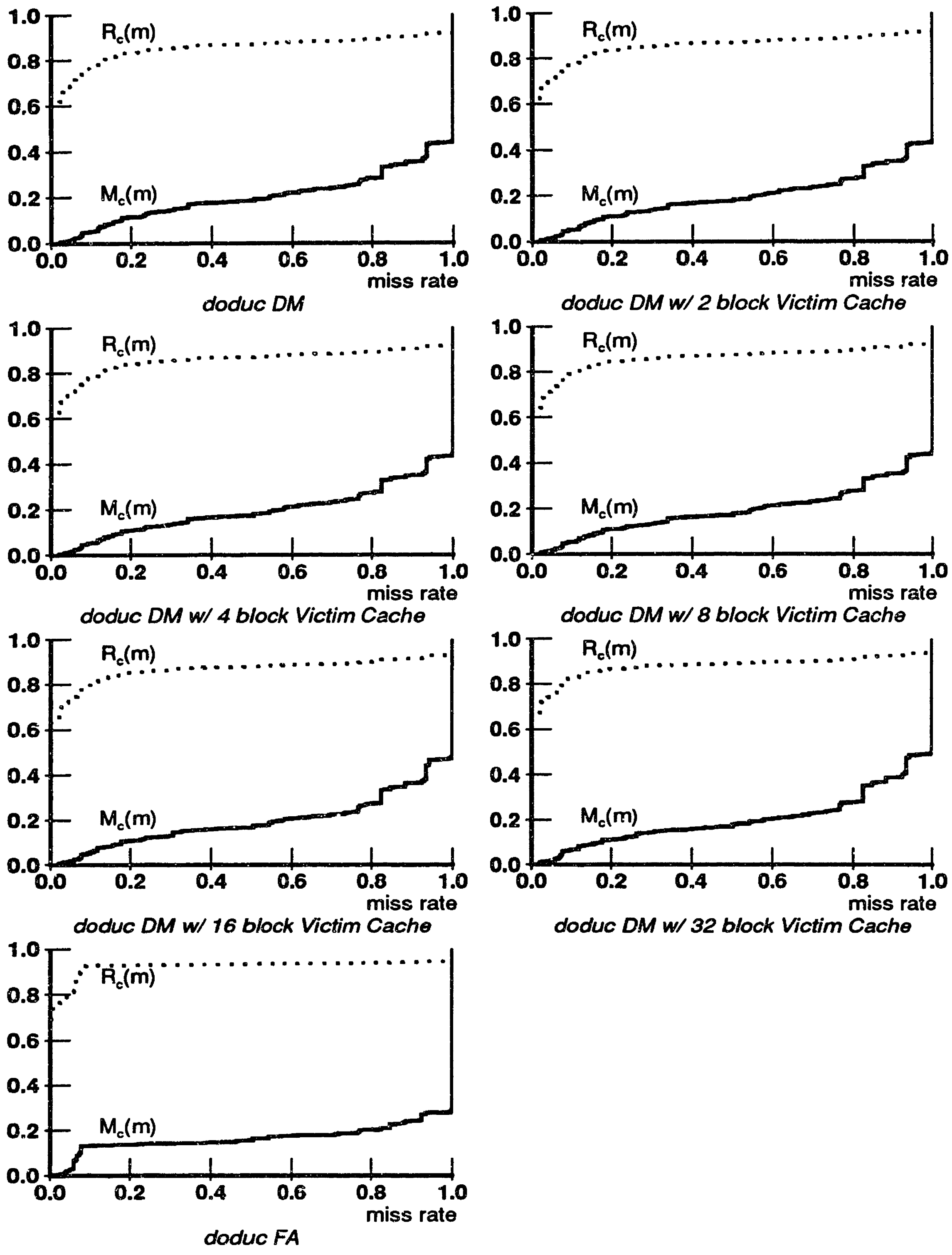


Figure 3-15: Doduc $R_c(m)$ and $M_c(m)$ variation with associativity

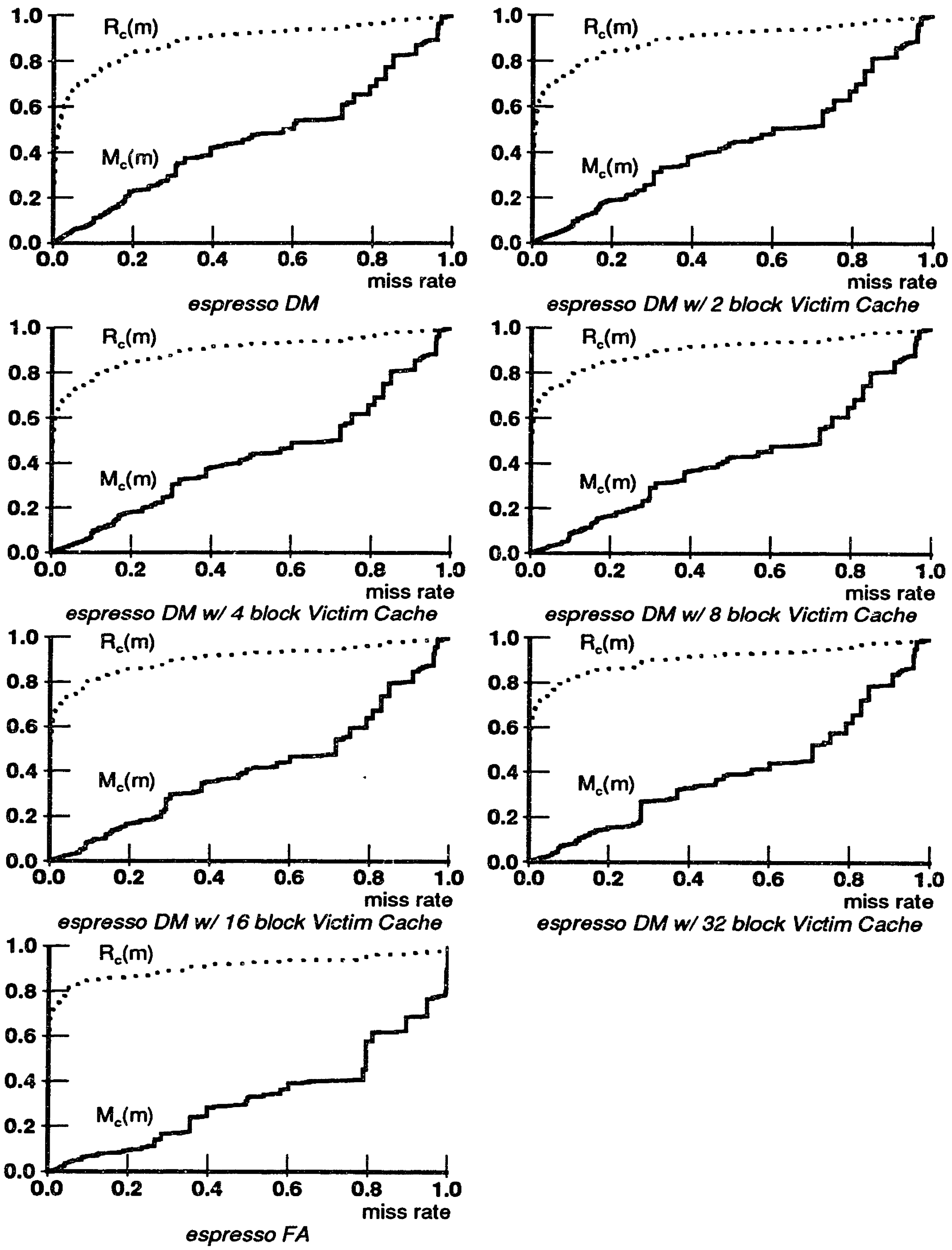


Figure 3-16: Espresso $R_c(m)$ and $M_c(m)$ variation with associativity

Varying the relative fraction of misses caused by conflicts has some impact on badref behavior but not a dramatic impact. In the data for doduc, in moving from the direct-mapped to fully associative

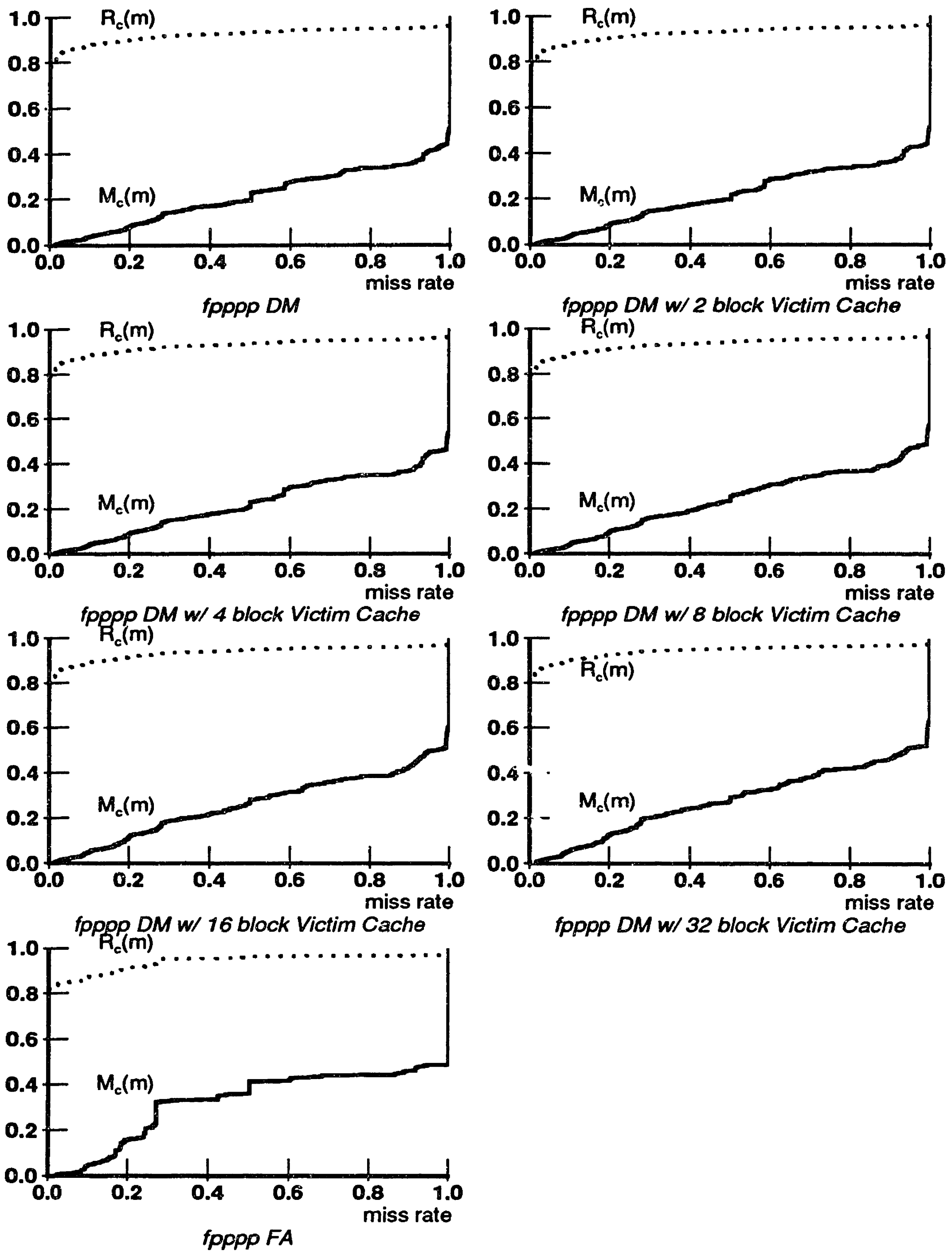


Figure 3-17: Fppp $R_c(m)$ and $M_c(m)$ variation with associativity

caches the miss rate changes by a factor of two from 14.6 to 7%. Despite the fact that half the misses in the direct-mapped data for doduc arise from conflicts, the benchmark exhibits zero miss

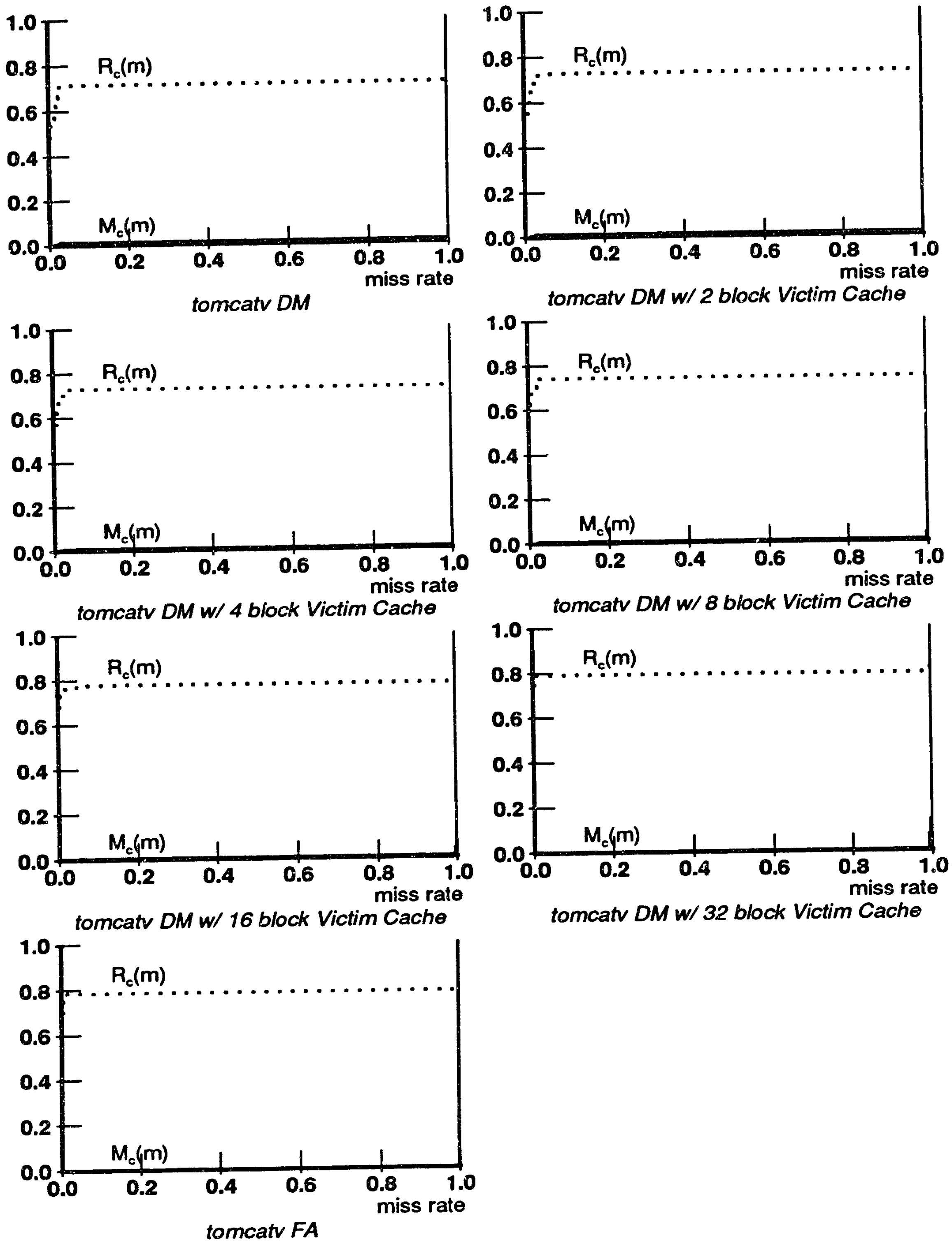


Figure 3-18: Tomcatv $R_c(m)$ and $M_c(m)$ variation with associativity

rate goodrefs accounting for 60% of memory references and 100% miss rate badrefs accounting for about 60% of misses.

Other benchmarks result in a smaller relative change in miss rate between direct-mapped and fully-associative caches. Changes in badref behavior for these benchmarks are similarly undramatic. In general, associativity does not have the same level of impact on badref behavior as block size or cache size.

3.5.3 Perturbations From Data Set Variation

The experiments above address one aspect of static locality correlation and badref behavior, namely the fact that in a single program run using some specific memory system, individual instruction miss rates are varied and typically polarized. This section addresses a second aspect of static locality correlation, specifically that the behavior of individual references, as described above, is not strongly influenced by perturbational changes in input data.

One must be somewhat careful in claiming that memory system behavior is independent of program input. In fact, this is by no means the claim that is intended in the static locality correlation hypothesis. As illustrated in data above, substantial changes in cache size can lead to dramatic changes in memory system performance. Similar effects can be achieved by leaving cache size fixed and radically modifying input data. Clearly, memory system behavior is likely to be different if programs use 100 Kbyte vs 100 Mbyte input data sets. The assumption of the static locality correlation hypothesis is that memory system behavior exhibits regimes of operation, demarcated by various critical data set sizes. Input changes which do not cross such a threshold will be termed perturbational input changes. The static locality hypothesis characterizes behavior of individual memory references as being relatively insensitive to perturbational input changes.

This aspect of the static locality correlation hypothesis can be tested by comparing the behavior of specific individual memory reference instructions under a variety of input data sets. The figures below show scatter plots of individual reference miss rates. Each point corresponds to a single static reference. The x-coordinate of each point in a plot corresponds to the miss rate of some static reference for a single data set. The y-coordinate represents the average miss rate of the static reference across all input data sets. Thus a point at $(0.25, 0.30)$ corresponds to a static reference with a miss rate of 25% for one data set and a miss rate of 30% averaged across all data sets. The points displayed for each input set correspond to the most frequently executed static references for that input. Point size is scaled by the frequency of the corresponding reference in the sample data.

While individual static reference miss rates do change as a function of inputs the data presented in Figure 3-19 show that for the benchmark tested, changes in miss rates are not substantial. In particular, in the data above it is very unusual for a reference to exhibit either a very high or very low miss rate in average data and a miss rate polarized in the opposite direction for some particular input data set. Consider the result of partitioning references into goodref and badref sets based on a threshold value in the average data. This technique will properly classify most references.

3.6. Summary

This chapter has introduced the static locality correlation and badref hypotheses. The static locality correlation hypothesis states that the hit/miss behavior of dynamic memory references occuring during program execution is strongly correlated with the particular static memory reference instructions from which the references arise. As a consequence it is meaningful to consider the memory behavior of particular static references. Furthermore, the hypothesis states that this behavior is insensitive to perturbational changes in input data.

The badref hypothesis extends static locality correlation. Not only is it meaningful to consider the hit/miss performance of static references but this behavior is frequently polarized with some references exhibiting essentially zero miss rates while others exhibit much higher miss rates, often

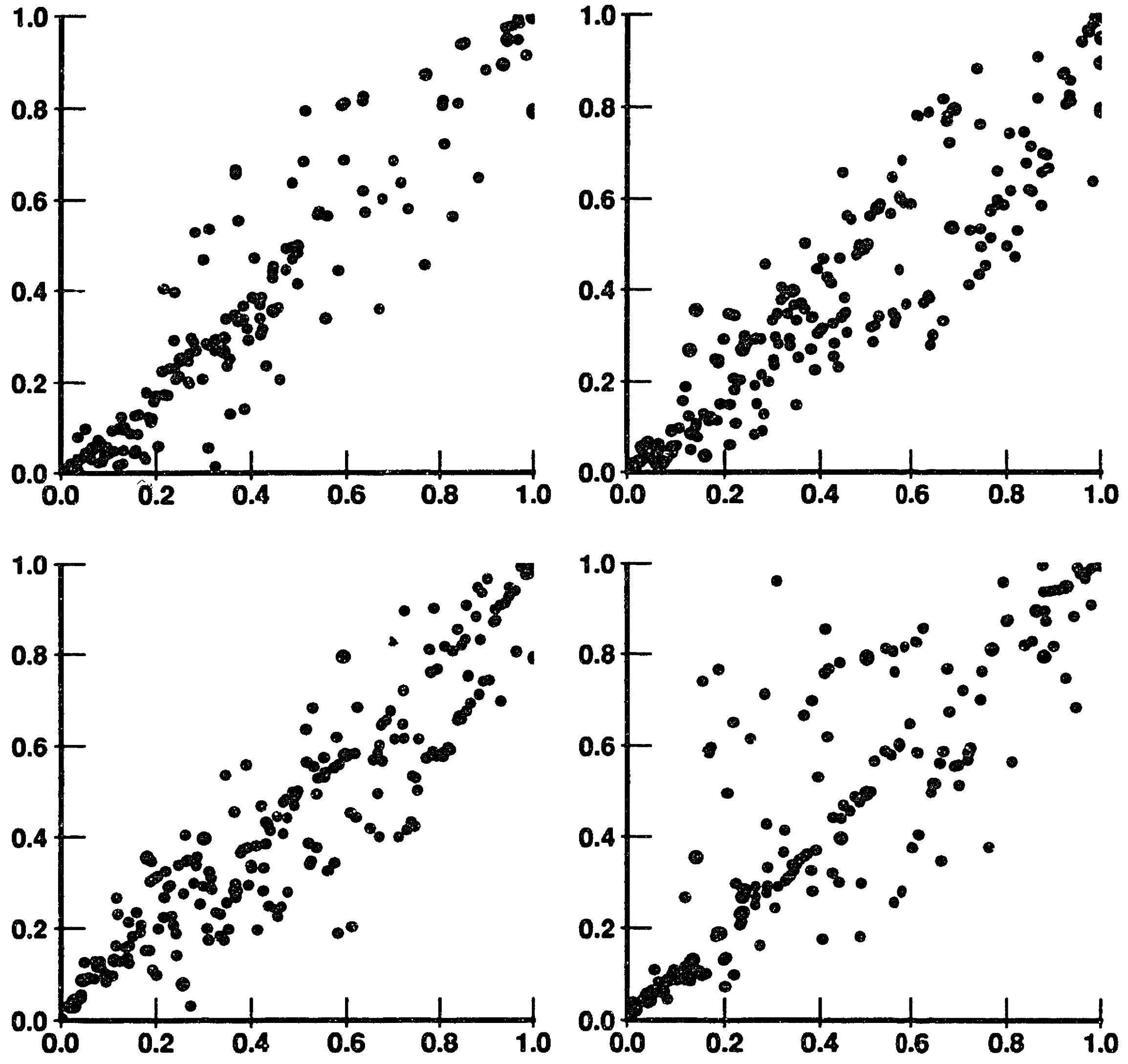


Figure 3-19: Miss Rate Variation with Input Data Perturbations in Espresso

approaching one. As a consequence of polarized static miss rates, a large fraction of dynamic memory references arise from references which never miss while a large fraction of dynamic misses arise from references which miss much more frequently than the average. This observation regarding the miss behavior of programs provides a cornerstone for software latency tolerance techniques.

Empirical evidence is presented to validate these hypotheses. This evidence takes the form of measured miss rates for static references for a set of benchmark programs. Statistics $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ are computed from simulated data. This data indicates that the behavior described in the static locality correlation and badref hypotheses is exhibited to a substantial degree by many programs, particularly in the context of relatively small caches. Parameterized cache simulation studies are included to assess the impacts on static locality correlation and badref behavior of changes in cache size, block size and associativity. Sensitivity of static reference behavior to input variation is also tested.

Chapter 4

Parallelism For Memory Latency Tolerance

Memory latency tolerance can be achieved by processing one or more high latency memory requests concurrently with low latency requests and non-memory computation. Overlap through concurrency can only occur when sufficient program parallelism exists. Measurement of program parallelism and evaluation of any obstacles to its effective exploitation can help lead to a characterization of the domain under which software memory latency tolerance techniques can be expected to be effective.

4.1. A Model of Parallelism and Latency Tolerance

Like locality in the previous chapter, parallelism and concurrency are slippery and sometimes ill-defined or overloaded terms. In the following discussion, use of the word parallelism will be restricted to refer to an intrinsic property associated with the dynamic execution of a program. To clarify further, the word execution will be used to describe an abstract model of dataflow constraints associated with the dynamic execution of a program equivalent to a dynamic dataflow graph. Executions consist of a set of primitive computational operations and a set of dataflow precedence relations between these operations. Parallelism is interpreted within the context of this model and refers to the intrinsic property of executions that some of the primitive operations may be unordered with respect to one another within the partial ordering induced by dataflow precedence relations. A machine consists of a set of resources capable of performing various primitive operations. When an execution occurs on a machine some scheduling of the primitive operations comprising the execution consistent with the dataflow partial ordering and also consistent with any constraints imposed by the availability of computational resources within the machine is induced on the primitives of the execution. Concurrency will refer to the simultaneous advancement of more than one primitive operation within a schedule of an execution on a machine.

For the purpose of analyzing parallelism for latency tolerance, consider a machine with two resources. One resource processes all primitive execution components with a latency of 1 cycle, the basic time unit, and a throughput equal to 1 inverse cycle. High-latency memory references, ie. cache misses, require additional processing by the second resource which must occur directly after processing of these operations by the first resource and which exhibits a latency of $L-1$ additional cycles and a bandwidth or throughput of BW operations per inverse cycle. (The machine could equivalently be modelled using $BW*L$ miss processing components each able to process a single operation at a time.) If an execution consists of I instructions of which M are cache misses, machine resources impose a lower bound on performance, measured in cycles, of the maximum of I or M/BW. All primitive operations can be processed by the first resource in no fewer than I cycles. The M miss operations can be processed by the second resource in no fewer than M/BW cycles. A complete scheduling of the execution requires both forms of processing to be completed and thus has a minimum execution time identified by Equation 4-1.

$$T_{min} = max(I, M/BW) \qquad (4-1)$$

In order to achieve either of these limits all required processing of the smaller component must be performed concurrently with processing of the larger component. Additionally, if bandwidth in the miss processing resource limits performance, all miss processing must be performed with maximum concurrency with respect to other miss processing based on bandwidth of the miss processing resource in order to meet the lower bound on performance. In the absence of all concurrency, a lower bound on performance can be computed by allocating L cycles for the M misses and one cycle for the remaining I-M instructions for a total value as indicated below.

$$T_{min} = M * L + (I - M) * 1 \qquad (4-2)$$

Latency tolerance techniques attempt to produce performance equal to that expressed in Equation 4-1. Concurrency in an execution schedule requires corresponding parallelism within the execution. In order to achieve the performance bound of Equation 4-1, two forms of parallelism are required. Parallelism must exist between miss and non-miss primitive operations in order to overlap miss processing with non-miss instructions. Additionally, if the product of L and BW exceeds unity and $M * L$ is larger than I, parallelism must exist between miss operations in order to exploit the miss processing bandwidth of the system.

4.1.1 Constraints on Parallelism

Parallelism cannot be uniformly exploited to achieve concurrency. Constraints on the particular forms of parallelism which can be effectively exploited arise for a variety of reasons. One form of constraint arises due to resource limitations in the machine. The second form of constraint arises as a result of the fact that, although not modelled above, static programs are used to produce dynamic execution schedules. Static programs act as templates for the schedules for executions and thus impose ordering constraints in addition to those associated with either the dataflow of the execution or the resource limitations of the machine.

Due to resource constraints, all parallelism is not created equal with respect to its ability to support the concurrency needed for latency tolerance. Two executions with the same amount of required computation and comparable parallelism by some metric may exhibit different achievable performance if machine resource constraints permit exploitation of the parallelism in one case but prevent it in the other.

Consider a machine model in which $BW = 1/L$ and the number of instructions and misses are related by $I = M * L$ instructions. The first assumption implies that concurrency is not permitted between the processing of multiple misses while the latter implies that miss processing and instruction processing performance bounds are equal. The lower bound on execution based either on instructions or misses is $M * L$.

Consider an execution characterized by a partial order in which the set of M miss operations must occur in sequential order and the set of $(L-1) * M$ non-miss references must also occur in sequential order. Dataflow contraints exist from miss operation $i-1$ to non-miss operation $i * (L-1)$ and from non-miss operation $(i-1) * (L-1)$ to miss operation i. Thus operations are partitioned into groups containing 1 miss and $L-1$ non-misses. Each miss can be overlapped with the non-misses in its group and no others. The constraints described are illustrated in Figure 4-1. Circles enclosing an M represent miss operations while circles enclosing a + represent all non-miss operations including memory references producing hits.

In an alternative execution, suppose that all non-miss operations can be executed in parallel with each other and in parallel with a single miss operation. Miss operations are sequential. These constraints are illustrated in Figure 4-2.

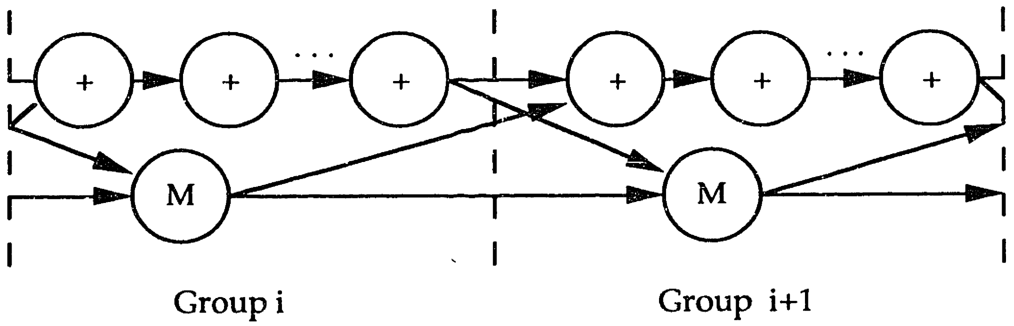


Figure 4-1: Dataflow Constraints for Example 1

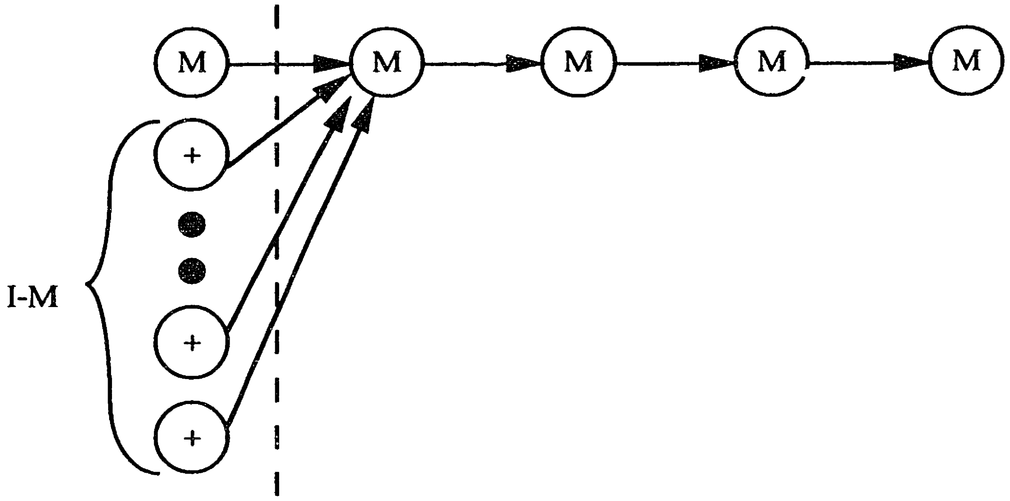


Figure 4-2: Dataflow Constraints for Example 2

Based on some quantifications of parallelism one might conclude that these two examples exhibit comparable parallelism or perhaps even that example 2 has more parallelism. Example 2 has an instantaneous parallelism of $I - M$ at some point while instantaneous parallelism in example 1 never exceeds 2. In each case, the critical path based on dataflow contstraints is equal and so is average parallelism computed as total work divided by critical path length. This average parallelism score is $\frac{I+1}{L-1}$, a number very close to 1 for large L. On the machine described above, however, example 1 can be scheduled perfectly for maximum concurrency while example 2 permits essentially no concurrency. These examples demonstrate two things. First, average parallelism measured in terms of instruction issue is not a particularly relevant metric for latency tolerance. Second, machine imposed constraints must be quantified in any realistic parallelism metric for latency tolerance.

Suppose instead that parallelism is measured as the average number of non-miss operations in parallel with each miss. Again, under this metric, both example 1 and example 2 exhibit equal parallelism scores despite the fact that only example 1 has appropriate parallelism to exploit machine concurrency. One might modify the parallelism metric so that there is a threshold for any single miss of $L - 1$. Under this metric example 1 has a score of $L - 1$ while example 2 has a score of $\frac{(L-1)}{M}$. Consider a third example, illustrated in Figure 4-3 in which the same $L-1$ non-misses exhibit parallelism with $M-1$ misses while the remaining non-misses exhibit parallelism with the remaining miss. Under the metric above which distinguishes examples 1 and 2 this new example exhibits a score equivalent to example 1 but permits essentially no concurrency on the modelled machine.

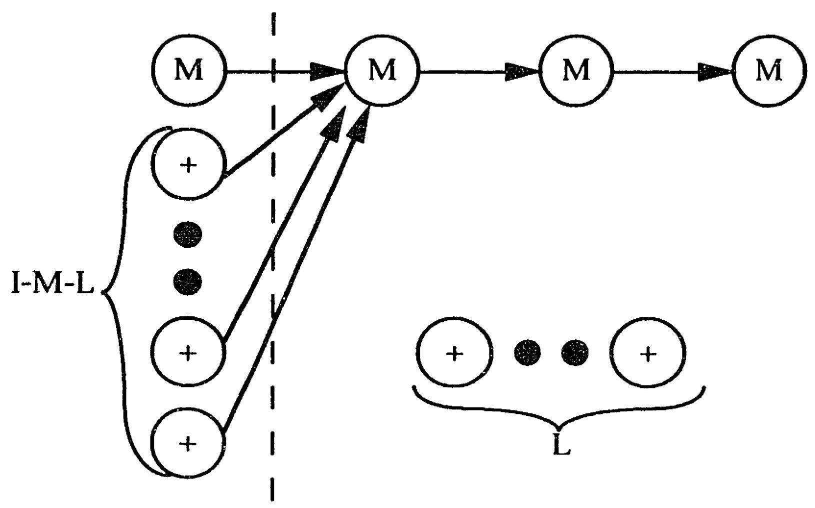


Figure 4-3: Dataflow Constraints for Example 3

Parallelism which can be exploited for latency tolerance requires a form of bipartite independence in which each miss operation exhibits parallelism with some distinct set of non-misses which are not credited as being parallel with other misses. The examples above are intended to illustrate that graph based parallelism metrics which stop short of assessing schedulability based on machine imposed resource constraints do not accurately characterize the parallelism required for latency tolerance.

In addition to machine imposed constraints on the specific form of parallelism required to enable the concurrency required for latency tolerance, constraints on exploitation of parallelism are introduced because executions arise from static program code. Program code acts as a template both for dynamic dataflow dependencies, the abstract concept modelled by executions, and for schedules for these executions on machines. Changes to static code modify the set of schedules which can occur for executions arising from the code. Some schedules which might be legitimate based on data dependencies in an execution and which would be desirable from the standpoint of concurrency may not be consistent with any transformed code. The set of schedules which can occur is limited by the set of code transformations which can be applied.

Consider, in particular, code transformations which do not perform code motion across basic block boundaries. In this case, despite substantial parallelism when considering only dataflow produced constraints, concurrency which can effectively be used for latency tolerance may be limited by the amount of parallelism within basic blocks. Similarly, code transformations which are highly constrained with respect to reordering memory references may also result in limited parallelism available for concurrency despite an abundance of actual parallelism. Constraints introduced by basic block boundaries and ambiguous memory references arise in many compilation techniques.

4.1.2 Dependence Constraints, Cache Misses and Speculation

Cache-fill transactions occupy a rather unique position among the set of operations occuring during program execution in that they are not semantically significant. Cache operations are program transparent. While these operations have performance implications, they have no correctness implications.

References resulting in cache misses typically have no short term memory carried dataflow constraints to prior references. That is, memory loads and stores which produce misses do not have real dataflow constraints, dependencies in the abstract execution model described above, with respect to preceding memory references. If a memory carried dataflow constraint existed for a reference, the reference would not produce a miss as it would be rendered a hit by the source of the dependency. Constraints may well exist with subsequent references, but these references will be hits.

Cache-fill transactions, since they are transparent, do not produce ordering constraints. Misses, when they occur, imply an absence of dataflow constraints with preceding references. Given this lack of constraints and thus apparent parallelism, what limits the concurrency achievable for miss transactions?

Two potential limitations exist. One limitation arises through constraints on legitimate static code transformations. A pair of memory instructions which includes at least one store operation cannot be safely reordered unless unequivocal proof exists that the two references use different addresses. The term reference disambiguation refers to this proof process. The presence of a few, scattered, ambiguous store operations within a program potentially serves to tightly limit the schedules attainable from safely transformed versions of the code. The second limitation arises from dataflow dependencies for address computations. In order to produce a memory reference and induce a miss an appropriate address is required. Miss processing cannot be carried out concurrently with operations computing the address for the operation invoking the miss. These two limitations can each be circumvented to some degree due to the special nature of cache transactions.

The first limitation arises when memory references producing misses cannot be reordered with respect to other memory references. The problem can be avoided by decoupling the miss from the memory reference. Decoupling is achieved by performing a non-binding prefetch, either with an explicit prefetch instruction or with a non-blocking load reference whose result is disregarded. In either case, the miss producing instruction is not semantically significant and thus can be added, moved or eliminated without any constraints involving other references.

The second limitation really is more fundamental. Addresses are needed to induce cache misses by any software mechanisms. Address computation can result in dataflow constraints between adjacent miss operations or between non-miss and miss operations. True, runtime data dependencies for addresses cannot be avoided. Static programs may contain potential address dependencies which rarely if ever result in actual dynamic dataflow constraints in executions. These artificial constraints can be avoided using speculative execution. Since cache miss behavior does not affect program correctness, code whose sole purpose is to provoke cache misses can ignore potential but unlikely dependencies.

4.2. Experimental Technique

Adequate exploitable parallelism is essential for the concurrency which can lead to latency tolerance. The discussion above argued that a direct measurement of parallelism, particularly one that can be reduced to a simple metric, is an elusive and perhaps unrealistic goal. This section describes a measurement technique which attempts to model and rationalize the various constraints to which parallelism, when it exists, is subjected in order to be usefully exploited for latency tolerance.

Parallelism is measured using a constrained traversal of a dynamic dataflow graph corresponding to a scheduled program execution. Individual instructions are represented as nodes in this dataflow graph and actual dataflow dependencies corresponding to the runtime production and use of data values are represented as edges. Within this graph both register carried and memory carried values explicitly produce dependence edges. The node representing an instruction has an explicit incoming edge from each instruction producing a value used as a register operand. Memory load instructions also have an edge from the memory store operation which produced the value loaded. Dependencies arising from storage reuse, *i.e.* antidependencies and output dependencies, are not represented as they can be avoided through storage replication.

In addition to explicit dataflow dependence edges, graph nodes have an ordering corresponding to their position in a legal sequential schedule of the execution. This ordering scheme can be used to identify nodes occuring in the same general vicinity within the sequential schedule whether or not these nodes share common dependencies. Traversal of the graph based on this sequential ordering is used to model constraints limiting parallelism which arise from the fact that the graph corresponds to a static program. Control flow nodes are explicitly represented within the graph and are linked by a seqential chain of graph edges. These nodes and edges can be used to identify basic block boundaries within the graph. Dependence arcs are added to ensure that memory stores occur in their sequential basic block.

Potential parallelism for latency tolerance is estimated by performing a windowed graph traversal in the vicinity of each dynamic memory reference node associated with a badref memory reference. Windowed parallelism is measured by marking all nodes with a transitive dependence relationship with a selected badref within a window of instructions based on the sequential order. Nondependent, parallel instructions can be identified by counting unmarked instructions within the window. Parallelism is measured in a range of window sizes around each badref and parameterized by its window size.

A graph traversal is performed in both directions for each load badref. The traversal directions correspond to parallelism exploitable through two different rescheduling techniques. One traversal begins at the badref and moves back in time, marking nodes upon which the address associated with the badref depends transitively. This traversal direction estimates parallelism which can potentially be exploited by triggering the miss associated with a badref instruction earlier in time relative to the sequential schedule. A second traversal in a direction corresponding to positive flow of time marks nodes which transitively depend on the result of a badref load. Parallelism in this direction is exploited by delaying the use of results of badref loads relative to the schedule. Graph traversal for stores is only performed to evaluate moving these instructions earlier in time.

In addition to assessing windowed parallelism, other instances of badref memory references within each window are counted and classified as either dependent or non-dependent with respect to the target badref. This data can be used to evaluate the potential for concurrency between the processing of multiple cache miss operations.

The information described above is gathered using an assembly postprocessor based simulation technique. Assembly code is transformed so that at the end of each basic block a section of graph structure corresponding to instructions within the basic block is added to a large graph data structure. Arcs for register carried dependencies within the block can be generated directly. Arcs corresponding to cross block register carried dependencies are produced using a table which records the node identifier corresponding to the last instruction to write to each register. Memory reference addresses and their graph node identifiers are added to a trace buffer. When the buffer becomes full a simulation routine is called which patches the graph to include memory carried dependencies for the references in the buffer and then performs graph traversal and parallelism measurement for selected badref occurrences from the buffer. Storage for the trace buffer and graph structure corresponding to sufficiently old instructions is reclaimed after a simulation call.

Data collection is performed based on a statistical sampling of memory reference nodes which are marked as badrefs. Badrefs are identified within programs using an initial program execution and cache simulation to determine static reference miss rates. References are partitioned based on a miss rate threshold. No cache simulation is performed in the parallelism simulation. Data is collected not specifically for misses but rather for occurances of references deemed likely to produce misses. The set of references produced by badrefs forms a fairly conservative estimate of the misses within a program. It errs much more frequently in the direction of identifying dynamic hits as misses, as opposed to identifying misses as hits. Since latency tolerance optimization is applied to static references and not specifically to those dynamic references which miss it seems appropriate to investigate parallelism in the neighborhood of occurances of badref references irrespective of whether they produce misses on the specific dynamic references measured.

Statistical sampling rather than exhaustive analysis for all badref occurances is applied for two reasons. The first reason is one of practicality. Graph traversal for windows of the size measured is a process consuming many thousands of instructions. Parallelism simulation is a much more time consuming process than the other forms of simulation used to gather data. It is impractical to perform a lengthy simulation operation for a non-negligible fraction of the memory references in a large program. The simulation takes too long. Given the practical neccssity of sampling versus exhaustive measurement, one then attempts to rationalize this as a good thing. By sampling sufficiently far apart, one insures that parallelism is not overcounted with respect to machine resource constraints. There is no added benefit to parallelism between a single specific instruction and more than the number misses of which can be simultaneously processed. Sampling sufficiently widely spaced badrefs insures that any given instruction is only credited to a single badref.

4.3. Results of Parallelism Measurements

The simulation procedure described above performs three distinct measurements for each sampled badref. The resulting statistics will be referred to as windowed parallelism, basic block parallelism and windowed badrefs. Windowed parallelism is a measurement of instructions which are independent with respect to the badref, parameterized by the distance with which the badref and independent instruction are separated within a sequential schedule of the program. Windowed badrefs are a similar tally of other badrefs, parameterized by their distance from a target badref. Basic block parallelism measures the number of independent instructions within the same basic block as a badref.

4.3.1 Parallelism

Total windowed parallelism and basic block parallelism for each benchmark is indicated in Figures 4-4 to 4-6. Total windowed parallelism measures independent instructions both before and after sampled badref instructions. Each graph shows six bold lines and one dotted line. The bold lines represent windowed parallelism measurements and the dotted line represents basic block parallelism.

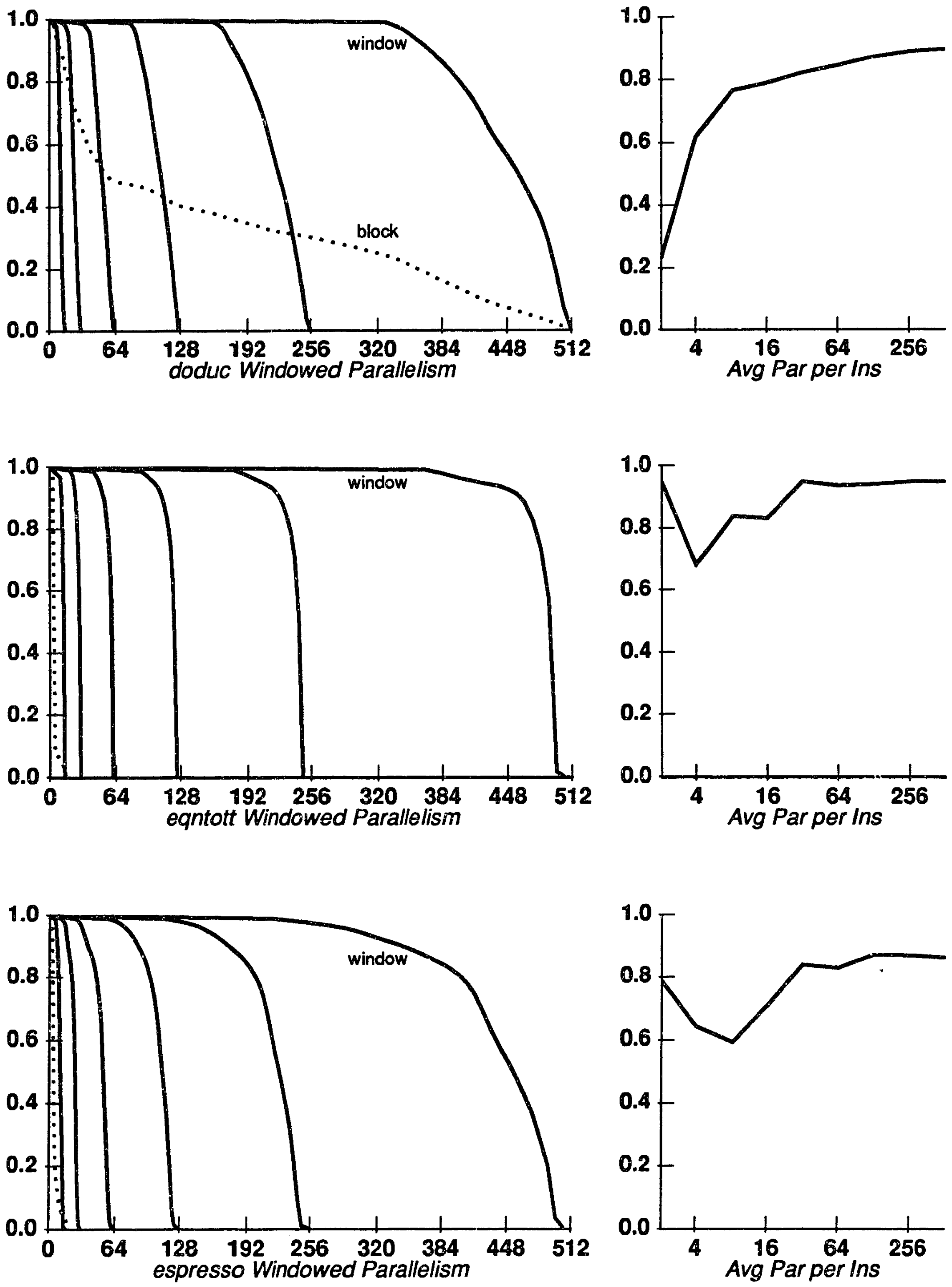


Figure 4-4: Windowed and Basic Block Parallelism for doduc, espresso and eqntott

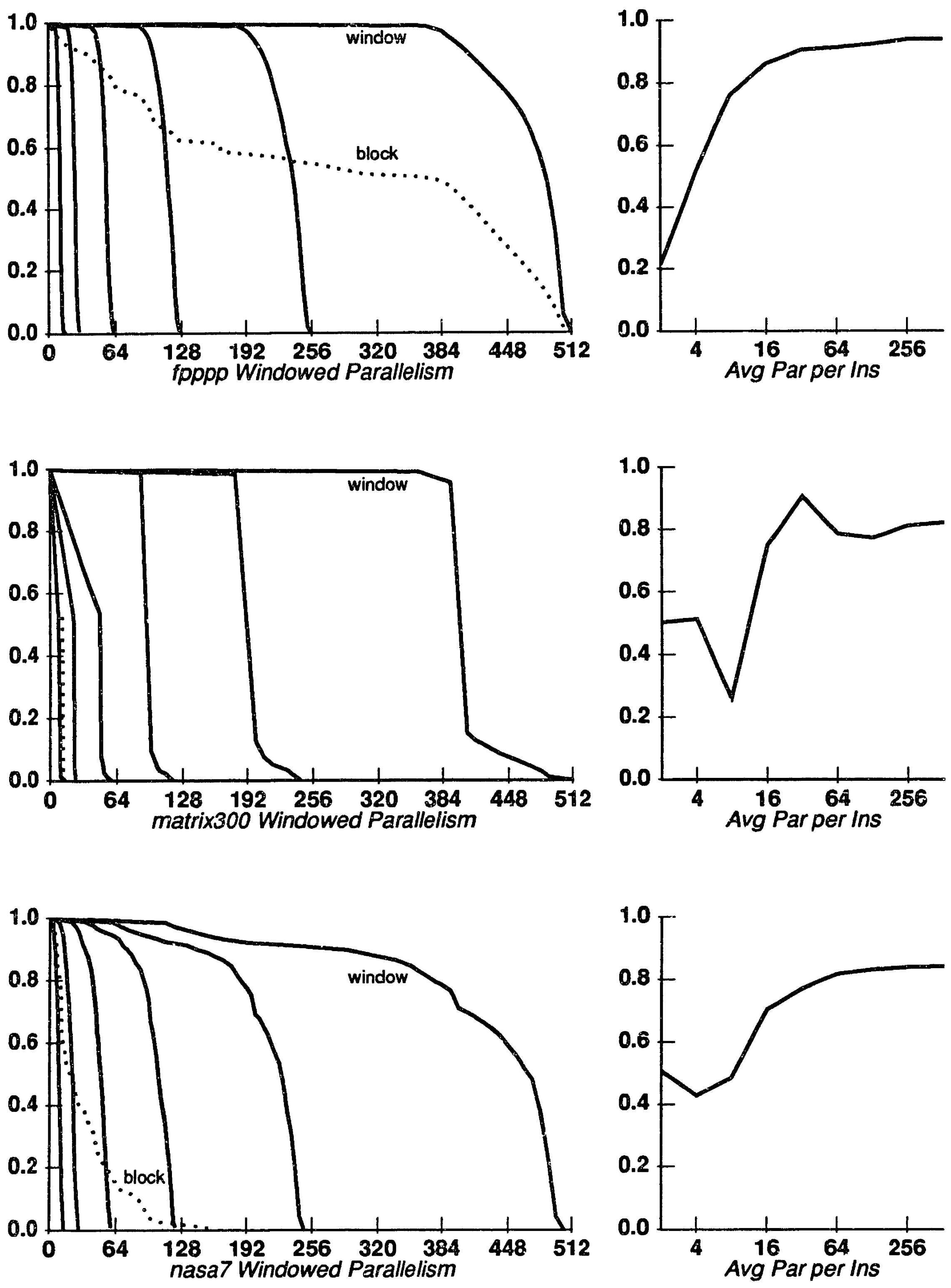
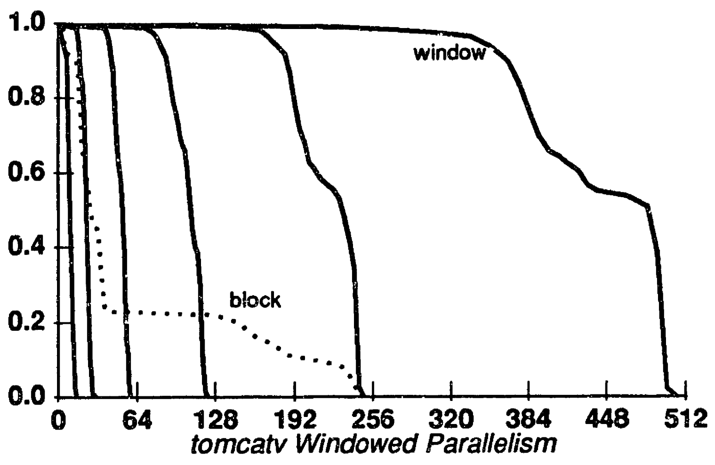


Figure 4-5: Windowed and Basic Block Parallelism for fpppp, matrix300 and nasa7

The graphs in the figures labelled Windowed Parallelism can be interpreted as follows. The x-axis



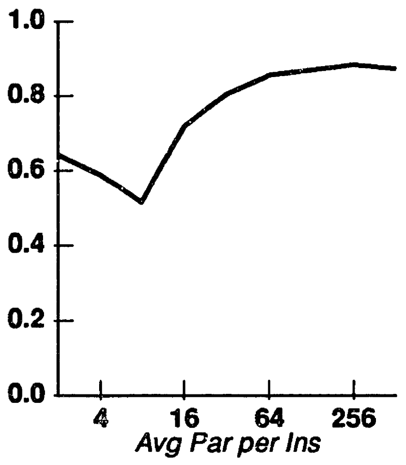


Figure 4-6: Windowed and Basic Block Parallelism for tomcatv

represents a particular level of parallelism. The value plotted at a particular x value corresponds to the fraction of sampled badrefs exhibiting a parallelism score at least as high as the current x value. For windowed parallelism values, plotted as boldface lines, the window size is always the closest power of two to the point at which the line intersects the x-axis. The window sizes plotted are 16, 32, 64, 128, 256 and 512. In each case the window is centered at the badref instruction extending half its distance forward in time with respect to a sequential schedule and half its distance backward in time. As a specific example, consider the curve farthest to the right in tomcatv, the last graph. This curve corresponds to a windowsize of 512. Since the curve is above 0.95 at an x value of 300, this implies that more than 95% of sampled badrefs exhibit in excess of 300 independent instructions in a 512 instruction window extending 256 instructions forward and 256 instructions backward in time. At 450, the curve has dropped to a value closer to 0.5, thus 50% of sampled badrefs exhibit in excess of 450 independent instructions within a window of 512 instructions.

The graphs in the figures labelled Average Parallelism per Instruction show the incremental parallelism per instruction associated with a particular window size. The score at 2 shows the average parallelism in a window of size 2. The score at 4 shows the average parallelism of the two instructions added in changing from window size 2 to 4. The score at 8 shows the average parallelism of the 4 new instructions added to reach a window size of 8.

This data does not leave much room for interpretation. It is clear that there is a vast amount of real parallelism available for the form of processor/memory system concurrency which leads to latency tolerance. One should bear in mind specifically what form of parallelism is measured and the parameters under which this measurement is performed. The data does not show that on average hundreds of instructions can be simultaneously executed. It shows instead that on average any particular badref instruction is unordered with respect to hundreds of its neighbors. It is thus possible to overlap data movement associated with a badref with a sequence of hundreds of other instructions which may or may not be ordered relative to one another. The ordering of this sequence is not relevant with respect to the parallelism sought. In measuring parallelism, neither register nor memory based storage reuse dependencies are considered. Parallelism is tallied without regard to basic block boundaries. In the reverse direction, dependencies are only considered for the addresses of badrefs, not the data. This is legitimate since prefetching can decouple misses from their associated data. In computing transitive dependencies for the addresses, both register and memory carried dependencies are considered. In the forward direction, dependencies are considered only when the memory value accessed by a load is used in a subsequent instruction. Under this model stores have no forward dependencies. As a consequence, they are not measured in this experiment and data applies only to loads.

Parallelism has been measured under a very liberal set of constraints and as a consequence aggressive code transformations may be required to exploit this parallelism. It is undeniable, however, that suitable parallelism exists, requiring only coding techniques to access this parallelism.

In addition to the fact that substantial parallelism exists for processor/memory system concurrency, two more general conclusions can be drawn from this data. The dotted line in the windowed parallelism graphs indicates basic block parallelism. This figure is measured the same way as windowed parallelism but instead of limiting the scope of parallelism measurement to a fixed window of neighboring instructions the window is determined by the boundaries of the basic block containing a sampled badref. In truth, these measurements are also windowed at the maximum window size of 512 instructions if basic block boundaries are not encountered in this window. In many of the benchmarks, restricting ones focus to basic blocks leads to a completely different conclusion with regard to parallelism. It is clear that in such cases, tolerance to latencies on the order of a hundred instructions can only be achieved using code scheduling techniques which are not restricted to operations within a single basic block.

Somewhat surprisingly, several of the benchmarks exhibit fairly substantial parallelism within basic blocks. For these benchmarks, the basic block parallelism is probably underestimated since no parallelism is counted outside the maximum sized 512 instruction window. One might speculate that the prevalence of benchmarks with very large basic blocks is somewhat coincidental. Based on the other benchmarks, one can justifiably claim that a general strategy for software latency tolerance must not rely solely on basic block parallelism.

The final observation one can draw from this data is evidenced in the average parallelism graphs. In general, as the distance between two instructions in a sequential schedule for a program increases it becomes more likely that the two instructions are independent. This observation will be revisited in the coming section.

4.3.2 Forward versus Reverse Parallelism

With respect to a particular badref instruction, non-dependent instructions can be identified both before and after the instruction within a sequential schedule. Parallelism with respect to following instructions will be termed forward parallelism as one must look forward in time starting at the badref in the sequential schedule to identify this parallelism. Parallelism with preceding instructions will be termed reverse parallelism as one identifies this parallelism by looking backward in time. This distinction between forward and reverse parallelism is useful because to some extent, different techniques must be applied to exploit the two forms of parallelism. Forward parallelism is exploited by delaying the use of data accessed by load instructions with respect to other instructions within the sequential schedule. Reverse parallelism is exploited by prefetching, invoking misses associated with badrefs earlier in time than would occur in an unaltered sequential schedule. The primary reason for distinguishing forward and reverse parallelism is that software scheduling techniques can exploit both forward and reverse parallelism while single-threaded hardware dynamic schedulers are primarily restricted to forward parallelism.

Dynamic hardware schedulers consider a windowed buffer of instructions for possible issue. The schedulers keep track of pending dependencies for instructions within the window buffer, thus determining when an instruction can legitimately be issued. Based on availability of hardware resources, schedulable instructions are issued from the buffer. Hardware schedulers have little opportunity to exploit reverse parallelism. An instruction cannot be issued until it reaches the window buffer and all pending dependencies are resolved. Among pending issuable instructions a hardware scheduler is likely to give priority to older instructions in order to free up window buffer space. As a consequence of these factors it is unlikely that initiation of a miss transaction for a badref will ever be reordered substantially from its corresponding position in a sequential schedule.

Exploitation of reverse parallelism fundamentally requires a mechanism for moving miss initiation for badrefs ahead in time with respect to other instructions including goodref memory references.

Furthermore, instructions used to generate badref addresses need to be moved earlier in time as well since a transaction cannot be initiated without its address. Hardware schedulers have no mechanism for prioritizing their scheduling. A short sequence of instructions may compute an address and use this address in a badref memory reference instruction. How can a hardware scheduler conclude that this sequence should be given top scheduling priority over all instructions in the window? One might speculate about instruction sets with two forms of memory reference instructions, one of which received hardware scheduling priority and similarly transferred this priority to all its dependent instructions. This does nct seem even remotely practical and furthermore requires software intervention just like software latency tolerance mechanisms. Dynamic hardware scheduling techniques are fundamentally incapable of exploiting reverse parallelism.

Restricting one's focus to forward parallelism is bad for two reasons. First, parallelism exists in both directions. If it exists evenly, then half is being ignored. As data will show, it is frequently the case that reverse parallelism exceeds forward parallelism, thus more than half the potential parallelism for latency tolerance is ignored by techniques limited to forward parallelism. Of potentially more importance, exploitation of reverse parallelism is in some sense easier than exploitation of forward parallelism. Exploitation of forward parallelism requires the reordering of semantically significant events as it entails the reordering of instructions using loaded valued with other instructions. Exploitation of reverse parallelism involves the reordering of an event which is not semantically significant, namely a memory/cache transaction, with respect to semantically significant instructions. Our compiler implementation in fact restricts its focus solely to reverse parallelism because it is easier to exploit. In general, software latency tolerance has the flexibility to exploit both forward and reverse parallelism, which would be required for optimal performance. Single-threaded hardware schedulers do not have this flexibility.

The figures below illustrate forward and reverse parallelism measurements for the benchmarks. Bold lines indicate reverse parallelism while dotted lines indicate forward parallelism. In each case both windowed paralellism and basic block parallelism are indicated. Windowed parallelism measures non-dependent instructions in forward or reverse windows of sizes 8, 16, 32, 64, 128 and 256, half the sizes of the corresponding combined parallelism measurements in the initial figures. Basic block parallelism measures non-dependent instructions either in the relevant direction until a block boundary is encountered. As before, incremental per instruction parallelism is also indicated as a function of block size for both forward and reverse directions. In the data below, reverse data reflects both badref load and store instructions since each require address computation. Forward data, like the combined data in the previous figures, reflects only loads.

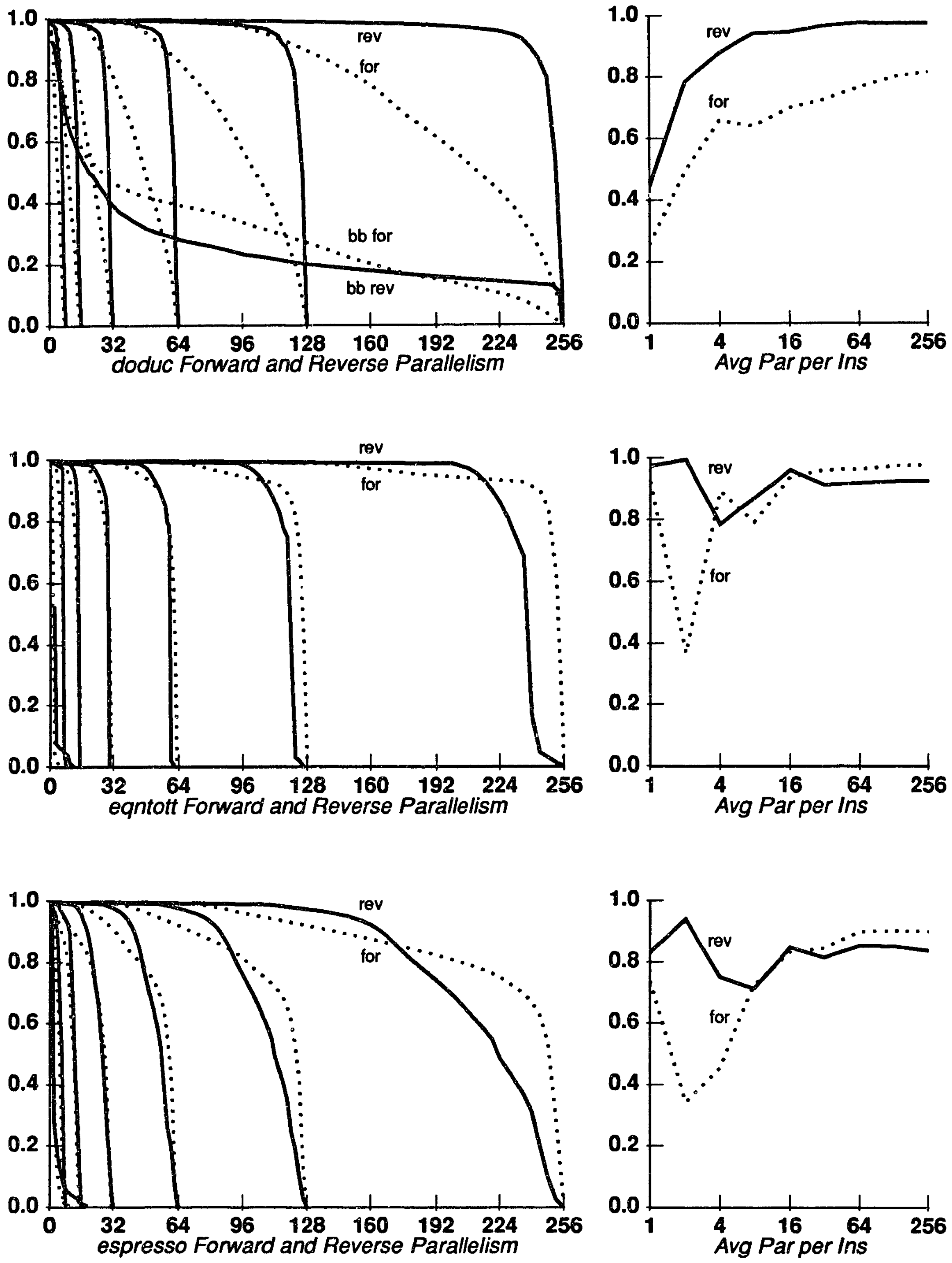


Figure 4-7: Reverse and Forward Parallelism for doduc, espresso and eqntott

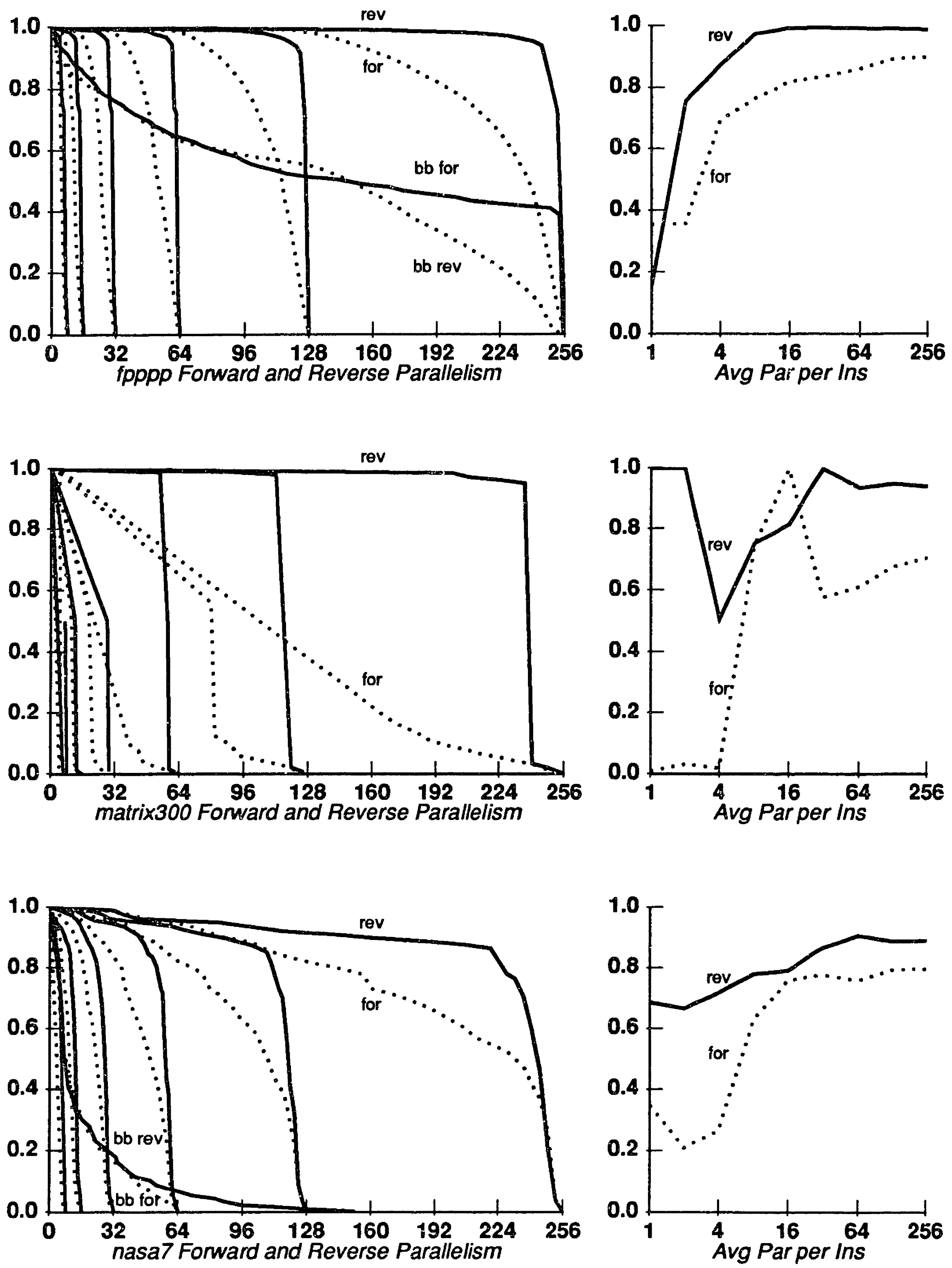


Figure 4-8: Reverse and Forward Parallelism for fpppp, matrix300 and nasa7

Data on forward and reverse parallelism shows several trends. In general, reverse parallelism exceeds

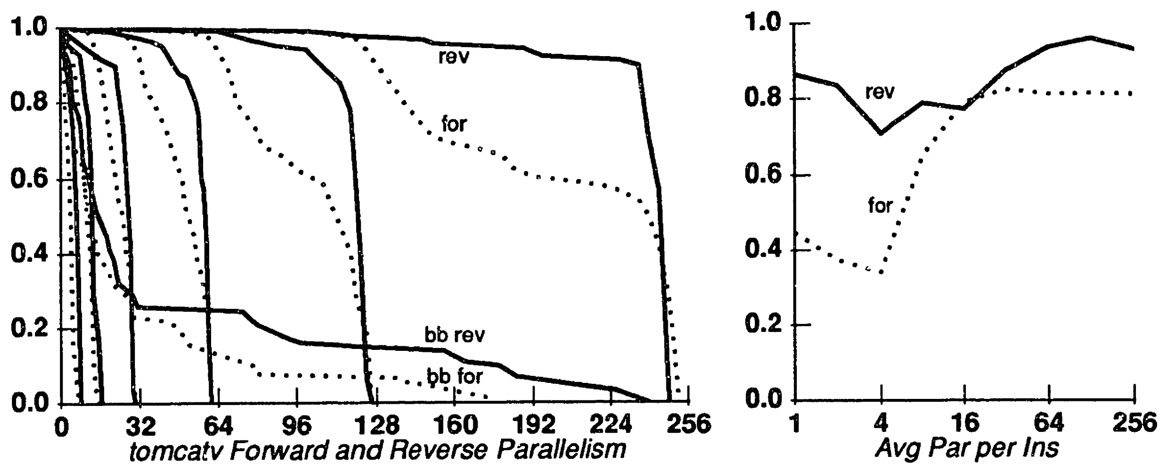


Figure 4-9: Reverse and Forward Parallelism for tomcatv

forward parallelism. This implies that in general the computation required to compute an address is simpler than the computation performed on values once they are loaded from memory. This is not particularly surprising and one might hope that this were the case.

A somewhat more interesting result pertaining to the contrast between software and hardware scheduling techniques for latency tolerance is indicated in the incremental average parallelism data. In the combined parallelism data instructions were more likely to be independent the farther apart they occurred within a sequential schedule. The data in this section indicates that the most likely instruction to be dependent upon a badref load instruction is the very next instruction and other instructions in the very near future. While this is not surprising, it is not good news for dynamic schedulers since this is precisely where they look first for parallelism.

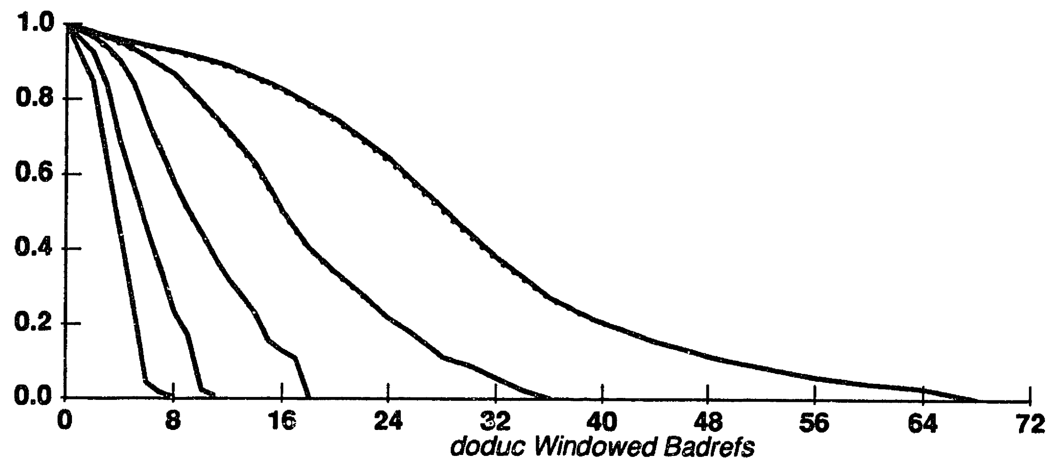
This situation is somewhat reminscent of an anecdote about a man searching for a quarter. He looks for the quarter in the street where the light is good rather than the alley where he has lost the quarter. Hardware dynamic schedulers are forced to look where the light is good even if that is not the most likely place to find the quarter.

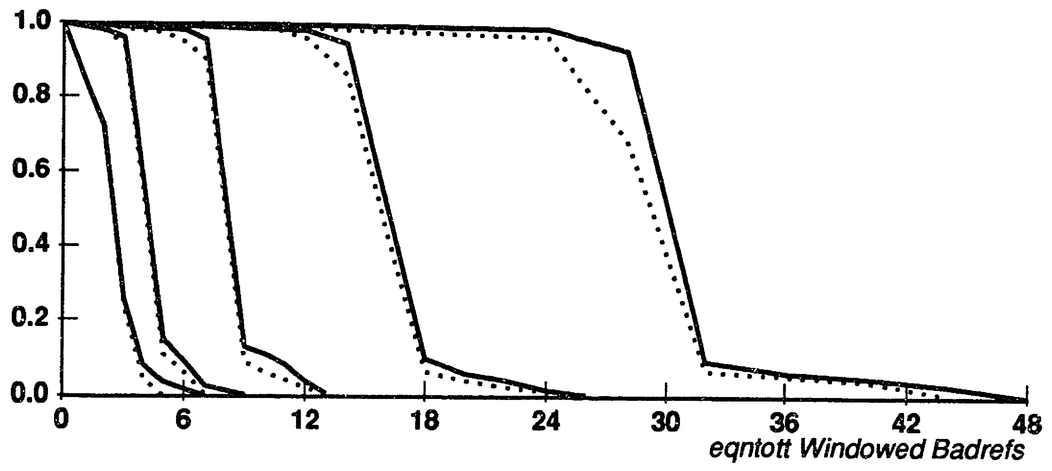
4.3.3 Badref Measurements

In the model developed at the beginning of this section two potential limitations to latency tolerance were discussed. The first was a lack of parallelism between high-latency miss operations and non-miss operations. The second involved interactions between high-latency miss operations. In a system in which several misses can be processed concurrently by the memory system, parallelism is required between high-latency transactions in order to exploit this concurrency. Regardless of the concurrency of the memory, tight clustering of badrefs, in contrast to a relatively uniform distribution, may result in transient imbalances between miss and non-miss processing requirements. This section presents data measuring the distribution and parallelism of badrefs.

As in the prior experiments, measurements are performed in varying sized windows around sampled badrefs. In this experiment other badrefs are tabulated and identified as being either dependent or independent of the target sample badref. The data presented measures badrefs in reverse windows like the reverse parallelism experiments. As in these experiments, dependence implies a transitive flow dependency for some value used in the address computation of the sample badref. The graphs below can be interpreted in the same way as the parallelism graphs. The x-axis corresponds to some tally of badrefs. The graphed value at a particular point on the x-axis indicates the fraction of samples with a tally of badrefs in their window greater than or equal to the current x-axis point. In the graph for benchmark doduc in Figure 4-10 the rightmost curve goes through the point (32, 0.4).

This implies that for the window size corresponding to the curve, 256 in this case, about 40% of samples had more than 32 other badrefs in their window. Each bold line has a corresponding dotted line. The dotted line measures independent badrefs. Thus the space between the bold and dotted lines is an indication of dependent badrefs. Lines in the graph correspond to window sizes of 16, 32, 64, 128 and 256 instructions.





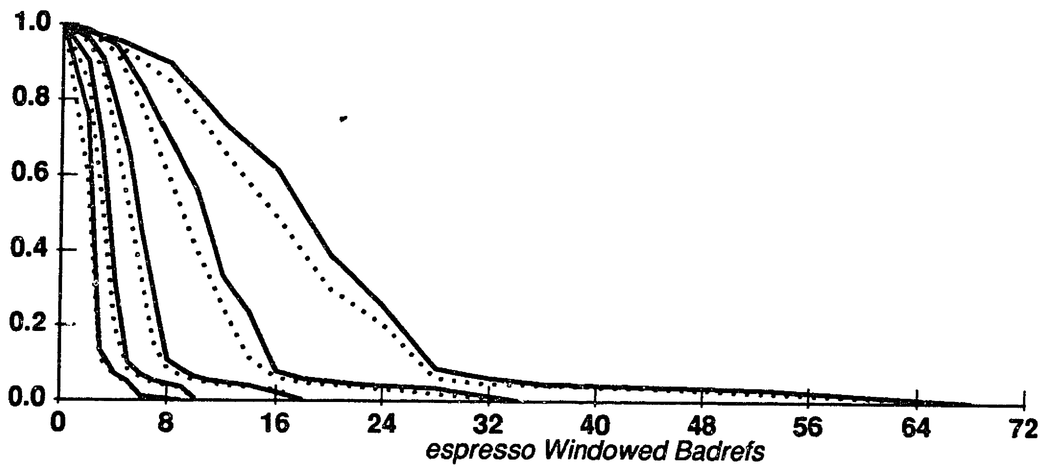
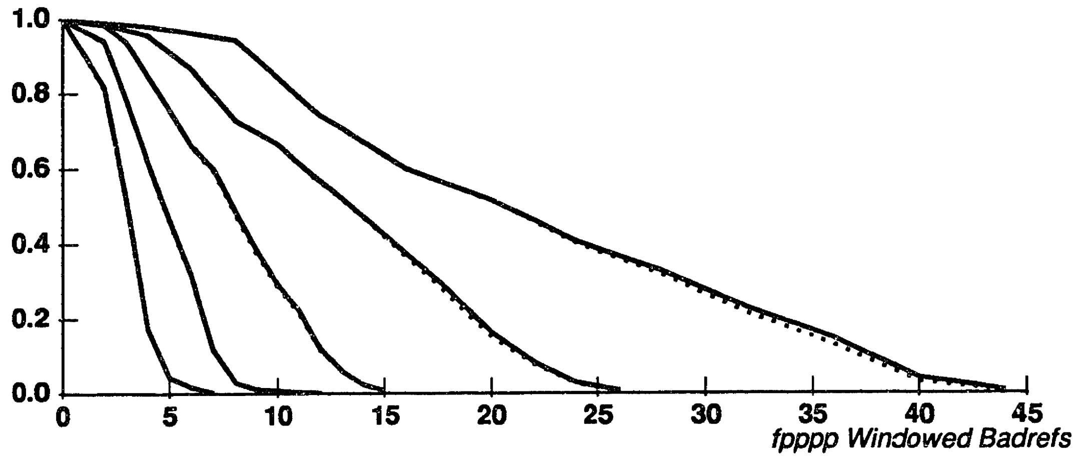
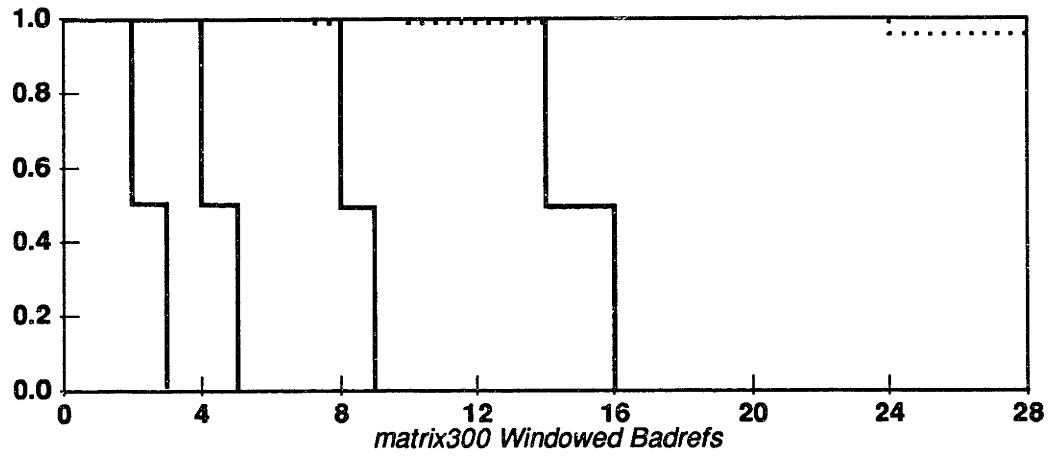


Figure 4-10: Windowed Badref Counts for doduc, espresso and eqntott





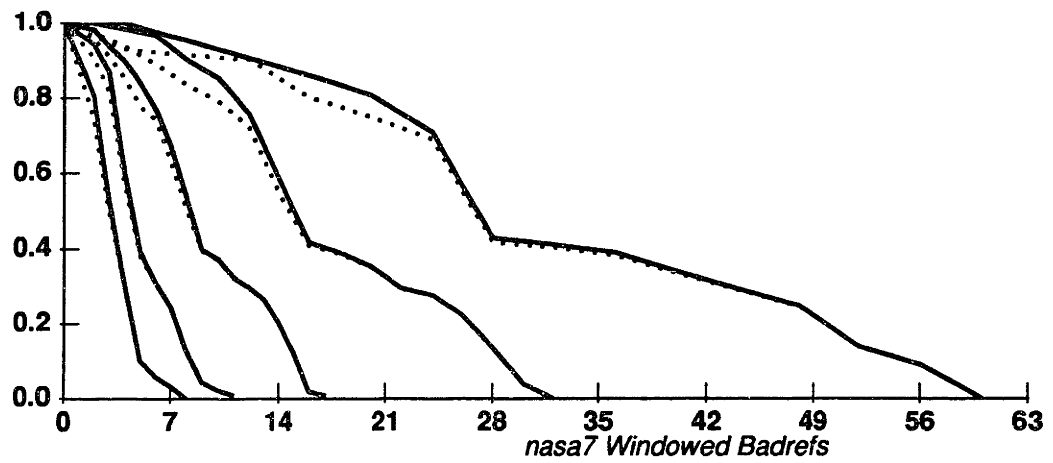


Figure 4-11: Windowed Badref Counts for fpppp, matrix300 and nasa7

Two forms of behavior associated with badref instructions can limit latency tolerance techniques,

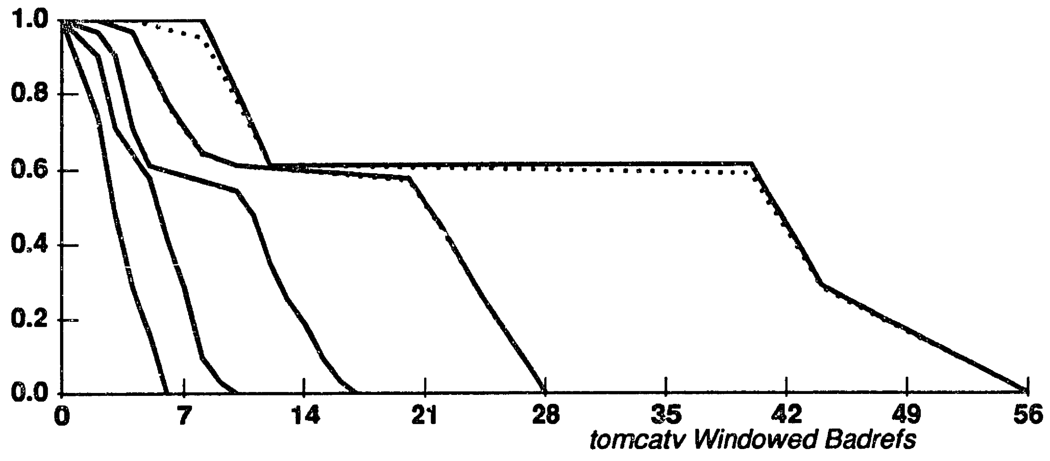


Figure 4-12: Windowed Badref Counts for tomcatv

namely address dependence between badrefs and substantial clustering. The data above indicates fairly clearly that address dependence between badrefs is not very prevalent in the benchmarks studied. To a large extent, almost all badrefs in even relatively large windows are non-dependent with respect to the addresses of sampled badrefs. Two programs illustrate somewhat higher levels of the form of badref dependence measured, namely eqntott and espresso. These programs each use linked-list style data structures.

While clustering behavior of badrefs is not directly measured by the data it can be inferred by comparing the relative shapes of adjacent curves. Curves measure badrefs in windows which increase in size by powers of two. Given uniformly distributed badrefs one would expect the separation between adjacent curves to increase roughly by a factor of two between each curve. Given clustered badrefs the spacing between adjacent curves should not uniformly increase by factors of two. Given the spacing of curves, it would appear that badrefs are relatively uniformly distributed.

Notice that uniform distribution of badrefs does not necessarily imply uniform distribution of misses. Categorization of a particular reference as a badref is performed statically. In the experiments above, any reference with a miss rate in excess of 5% was classified as a badref. Actual misses associated with lower miss rate badrefs may still be clustered even if references arising from badref instructions are uniformly distributed. The absolute spacing of misses cannot be less than the spacing of badrefs since misses are a subset of dynamic badref references. The relative spacing of misses can be more non-uniform than that of badrefs, however.

4.4. Summary

Latency tolerance is achieved through concurrency. Application of concurrency fundamentally requires parallelism. This chapter has developed a model to characterize the forms of parallelism specifically relevant to latency tolerance and an experimental method for evaluating this parallelism in benchmarks.

Latency tolerance principally relies on concurrency between miss processing transactions and non-miss computation. To model constraints on software latency tolerance parallelism is measured in a region around miss-producing memory reference instructions based on a valid sequential scheduling of code. Dependencies modelled include real, runtime dataflow dependencies for the addresses of badref memory references as well as the results produced by badref loads. These are the forms of dependencies which constrain the use of speculative, non-binding prefetching. Benchmark data indicates that in this context large amounts of parallelism exist. This parallelism does not generally exist if the search is confined to basic blocks.

Forward and reverse parallelism are defined and measured. We observe that window-based, dynamic instruction scheduling hardware primarily exploits forward parallelism in a small window ahead of badref memory references. Benchmark data indicates that on average, this particular region has less relevant parallelism than anywhere else one might choose to look. This suggests that software techniques should prove superior to single-threaded hardware techniques for latency tolerance.

Finally, this section presents data to evaluate potential latency tolerance limitations arising from interactions between adjacent high-latency operations. This data indicates that address dependencies which potentially constrain simultaneous miss processing are relatively rare. Additionally it indicates that badref instructions occur at relatively uniform rates within benchmarks and do not exhibit highly clustered behavior.

Chapter 5

Compiler Algorithms, Models and Heuristics

Code optimization is achieved by transforming one piece of code into another semantically equivalent piece of code with better expected performance. Optimizing compilers use algorithms, models and heuristics in order to improve code Analysis algorithms deduce the behavior of code to identify potential transformations and verify their semantic legitimacy. Models characterize the performance of code on an idealized target machine in order to assess the benefit of potential transformations. Heuristics are used to narrow the search space of possible transformations identified by analysis to those estimated to be most fruitful based on the target machine model and an understanding of the specific optimization problem by the compiler writer

A prototype Compiler For LAtency Tolerance, referred to by the acronym c-flat, is implemented. As with other varieties of code optimization, the memory latency tolerance transformations implemented in c-flat require analysis algorithms, machine models and pruning heuristics. The hardware latency tolerance mechanism exploited in our transformations is a prefetch instruction which is assumed to be safe for speculative use. Because adding this type of prefetch instruction to a program typically has few semantic effects, latency tolerance optimization in c-flat tends to emphasize modelling and heuristics, with a lesser amount of analysis. This chapter describes the algorithms, models and heuristics implemented, as well as a few other ideas and observations which were not implemented.

5.1. Memory Reference Behavioral Modelling

Optimizing compilers need machine models to estimate the performance impacts of potential code transformations. Better models facilitate the production of better code. C-flat has a sophisticated model of memory reference behavior in order to address the problem of memory latency.

The simplest machine model which a compiler can use assumes that instructions are issued sequentially and have a uniform execution time. The model assumes that no subsequent instructions are issued until prior instructions complete and thus only a single instruction is active at a time. This simple model provides a reasonable characterization of some RISC processors. Under these assumptions, performance is completely characterized by instruction count and a single instruction represents an unambiguous amount of time. This time is often referred to as a cycle. Where it might be ambiguous, the term cycle will be used to represent an amount of time equal to the inverse of the maximum bandwidth with which instructions can be issued by a processor.

When instructions have variable execution times, particularly when there is an opportunity for several instructions to be active simultaneously, a more sophisticated model can be employed beneficially. Several commercial RISC processors have floating point arithmetic units which are pipelined. Latencies of floating point operations are several cycles and subsequent instructions can be issued during intervening cycles if they do not require results of the floating point operations as their input operands. In order to allow issue of one instruction per cycle on such a machine (or allow correct operation if the machine does not have hardware interlocks), code must be scheduled so that results of high latency floating point instructions are not used before they are available. Code scheduling for these machines requires a model which characterizes the latency of instructions relative to the instruction issue bandwidth or cycle time of the machine. Models used for these machines characterize the performance of code based on instruction type. While latencies vary for different instructions, each particular instruction, such as integer add or floating point multiply, has a fixed associated latency measured in cycles.

Memory reference instructions do not exhibit behavior that can be readily characterized using a fixed latency. Hierarchical memory systems exhibit variable, stochastic latencies, depending on the level within the hierarchy which satisfies a particular reference. This presents a modelling dilemma. References are predominantly satisfied by the cache at the top level of the memory hierarchy, often overwhelmingly so. The natural way to model memory references is to assume a uniform latency equal to the latency of this cache. Such a model is used in virtually all current RISC compilers. In this model, latency associated with references not satisfied by the cache is not modelled. In order to produce code which is tolerant to cache miss latency, this latency must be modelled.

As an alternative, all memory references can be modelled using the latency of some deeper level of the memory hierarchy. While this technique explicitly models miss latency, it ignores the fact that most references are satisfied quickly by the cache. Code scheduled using such a model unduly penalizes cache hits. In fact, if results from memory loads are not used optimistically by assuming memory reference latency is the cache latency then the cache has not served its primary function of decreasing effective memory latency. For the same reason that modelling all references using the cache miss latency is inappropriate, modelling them with any other constant latency, such as the average memory latency based on hit rate and hit and miss times, is also inappropriate. Cache hits are the predominant case so if memory references are to be modelled with a single latency parameter the hit latency should be used.

Static locality correlation and the badref model provide a solution to this conundrum. As evidenced in Chapter 3, misses are highly correlated to specific memory reference instructions in code. Miss rates of different static references often exhibit a bimodal distribution with some static references producing virtually no misses and others producing far more than average. The badref model is a behavioral model for compilers in which references are partitioned into two sets, goodrefs and badrefs. Goodrefs can and should be modelled using cache hit latency. Badrefs, in contrast, can potentially be better modelled by assuming a latency corresponding to the miss latency. Identification of badrefs coupled with differential treatment of goodrefs and badrefs provides a suitable memory behavior model for memory latency tolerant compilation.

The model utilized in c-flat is slightly more complicated than the basic badref model. Rather than immediately and irrevocably partitioning references into goodrefs and badrefs, references are characterized with a miss rate estimate. This estimate is used initially to cull away most goodrefs. The threshold for this initial partitioning is set at the inverse of the miss latency. For references expected to miss less frequently than this threshold even a single cycle of overhead is unjustified. If the miss latency is high, the initial badref set contains some references with relatively low miss rates. References within this set are subsequently evaluated as candidates for latency tolerance optimization by comparing the estimated performance costs and benefits of optimization based on the miss rate estimate and memory hit and miss times.

5.1.1 Badref Identification Techniques

Two contrasting techniques exist for estimating miss rates of individual memory references and partitioning references into badref and goodref sets. The first technique utilizes static dataflow analysis similar to dependence analysis to identify candidate badrefs. The alternative approach relies on dynamic simulation of code to measure the miss performance of memory references. This information can then be exploited in a secondary compilation phase.

As discussed in Chapter 3, static locality correlation arises due to program structure. When several adjacent memory references access the same location, only the first can possibly produce a cache miss. Dependence analysis identifies memory references which access the same memory location and as a result, can be used to estimate memory hit/miss behavior. Classical dependence analysis does not indicate a dependence between pairs of read accesses which refer to the same location. For purposes of cache behavior analysis this interaction should be identified as a "dependence" since cache hit/miss behavior is not sensitive to whether references are reads or writes. The number of distinct memory references interposed between any identified pair of references to the same location can be estimated. Based on this estimate and a model of cache behavior, the latter reference in each dependence pair may be classified as a goodref, *i.e.* a reference which cannot miss by construction. [43] describes an analysis technique which performs goodref/badref partitioning for array references in FORTRAN do loops.

In c-flat, the alternative approach is adopted for badref identification. An initial compilation stage produces an executable program. Miss rates of individual instructions are measured in test executions of this program for one or more input data sets. A subsequent recompilation produces a latency tolerant version of the original program based on gathered miss information.

The choice between an analytic versus an experimental method for determining the memory behavior of programs entails both practical and philosophical considerations. Experimental methods based on mulitphase compilation interspersed with program execution should perform as well or better than analytic methods. Data gathered using such a technique corresponds to some real execution of the program. While experimental data might be inferior to analytically estimated data if the experimental data corresponds to some form of unlikely behavior and the analytic technique correctly deduces that such behavior is unlikely, the latter seems unlikely even when the former occurs. Analytic compiler methods are notoriously ineffective at assessing the relative likelihood of possible events. When goodref/badref behavior depends on dynamic program behavior which is difficult to model statically, such as the frequency of branches or the specific duration of loops, dynamic measurement will likely produce better estimates of typical program behavior.

Based on implementation ease, either method might be deemed more practical depending on the level of preexisting compiler infrastructure. In the absence of sophisticated dataflow infrastructure the experimental approach is relatively easy to implement and as a consequence has been adopted. In the context of simulation-based research an experimental approach dovetails nicely with other simulation required for the research. Given suitable infrastructure in the form of a preexisting compiler with very good dataflow analysis one might view the alternative as a less rocky path.

Finally, the decision involves a philosophical issue. Is it fair, justifiable, ethical for compilers to utilize experimentally gathered information about programs or should compilation always entail a single pass? We take the engineering view on this issue that anything goes. Empirical observation suggests that many real programs are optimized through an iterative process involving compilation, test execution of the program, and finally human intervention at the source level to improve program performance. This form of optimization process based on dynamically measured data is completely mechanized within our implementation with the exception of the choice of input data for experimental program runs.

5.1.1.1 Analysis: The Path Not Taken

While an analytic technique for memory behavior modelling is not implemented in c-flat, research suggests that in some contexts successful analysis is possible. Porterfield, in [43], and Gannon, in [13], each describe analytic memory behavior modelling techniques.

Developing an accurate characterization of memory behavior is generally very hard, if not impossible. It is likely to be most successful in the context of well structured programs manipulating large arrays in loops. This is the context in which the methods of [43] and [13] are appropriate.

Even in a more general context, one may have some success at attacking the problem of estimating memory behavior if the problem is viewed in the right light. The most fruitful approach is probably one which looks for goodrefs rather than badrefs. Thus all references should be considered bad and potentially discovered to be good rather than considered good and discovered to be bad. In analogy to the search for a quarter in the previous chapter, rather than concluding that a quarter is in the dark alley, conclude that it is not in the brightly lit street. Such an approach underestimates cache performance, bounding the set of badrefs from above. It allows latency tolerance to be applied to a set of references likely to contain almost all badrefs while avoiding overhead on those goodrefs which can be easily identified. Based on empirical observation I suspect that such a technique might identify a reasonably large fraction of goodrefs, at least half and perhaps more, in many programs.

This strategy is consistent with that of Porterfield. He perform an initial partitioning between scalar and array references, considering all scalar references to be goodrefs and all array references to be badrefs. He then identifies goodref array references when possible. The heuristic technique of calling all scalar references goodrefs is probably not suitable in a more general context.

We make no further effort to evaluate analytical badref identification techniques. The problem is very hard. Program dynamics such as branch frequencies and loop limits are often very difficult to estimate statically except in very restrictive circumstances. For direct-mapped caches and caches of limited associativity memory behavior is impacted by the placement of different data structures. Alignment of data structures relative to cache boundaries can similarly impact memory behavior. It seems unlikely that an analytical technique can be used to produce a tight estimate of program badrefs in a general context. If somone solves this problem they deserve much credit. Irrespective of the means by which memory system behavior is estimated, however, our other algorithms and results are still applicable.

5.1.1.2 Miss Rate Measurement in C-flat

Measuring miss rates of individual memory references via simulation is primarily a bookkeeping problem. Compilers manipulate programs using a representation which typically differs from the final executable form. A mapping must be established between memory references in the executable code and the constructs within the intermediate representation from which these references have arisen. This mapping must be maintained so that a second compilation can properaly reassociate statistics measured during simulation with the appropriate intermediate structures.

In the intermediate representation in c-flat (inherited from GCC upon which c-flat is retrofitted) most memory references are explicitly represented. References to values which must be stored in memory including arrays, structures and anything accessed through a pointer, have a representation indicating a memory resident value. Values which can potentially be stored in registers have a different representation. At a fixed point during both initial and secondary compilations, the intermediate program representation is labelled with unique memory reference identifiers. To ensure that labelling is consistent between the two compilation phases, identifiers are added before any optimization which acts differently in the two phases is performed.

Latency tolerance optimization is performed fairly late in the compilation process in c-flat. It occurs after the standard compiler optimizations performed by GCC, just prior to register assignment. Positioning latency tolerance optimization late in compilation simplifies bookkeeping and has other benefits. Since labelling can be deferred until after other optimization passes have occurred these optimizations do not need to update identifiers in any way. Several more significant reasons exist for positioning this optimization late. First, putting transformations which result in differences between first and second compilations late in the compilation process tends to minimize the perturbation of the initial code. If differences are produced early on, these differences are magnified by subsequent transformations. Experimental statistics used to characterize memory behavior are based on the initial code and remain most accurate when the code is minimally perturbed. Additionally, latency tolerance optimization is primarily a scheduling process. Insertion of prefetch instructions can be viewed as software scheduling of misses which occur during program execution. This scheduling process can model program behavior most accurately, and thus generate the best code, after other optimizing transformations have occurred.

Register allocation, which occurs after prefetch optimization, creates memory references when registers are spilled. Since these references are not explictily present in the intermediate representation at the time of latency tolerance optimization they are not labelled and are not eligible for optimization. Register spill references could potentially be optimized by moving latency tolerance optimization after register allocation. This is impractical because latency tolerance optimization changes the register requirements of programs, thus changing the spill behavior. Fortunately, references generated by spilling registers are typically goodrefs.

Multiple file compilation adds further bookkeeping chores. In order to support multiple file compilation a separate identifier numbering space is used in executable files from that used in compilation. Mapping information is maintained for each source file, associating intermediate compiler identifiers with new identifiers in the final executable code.

5.1.2 Memory System Modelling

While the badref model and dynamically gathered miss statistics can be used to estimate the hit/miss behavior of memory references, in order to schedule misses a performance model of the underlying memory system which services miss transactions is also required.

Memory systems are modelled by c-flat as asynchronous pipelines. The performance of these pipelines is characterized by two parameters indicating their latency and maximum bandwidth. The parameter *mlat* specifies the latency to service a request which moves one cache block of data from the memory into the cache. This latency is measured in processor cycles. Memory bandwidth is characterized in terms of an intersubmission delay, referred to as *mbwlat*, which is also measured in cycles. C-flat assumes that requests to service misses can be submitted to the memory system no more frequently than once every *mbwlat* cycles. The model assumes that this time interval must occur between any adjacent submissions, rather than just being satisfied in aggregate. The intersubmission delay is assumed to apply at the cache/memory system interface, as opposed to the processor/cache interface.

An additional parameter, *mconc*, which is used in some heuristics can be derived from these parameters. *Mconc* is the product of memory system latency and maximum bandwidth which can be computed as *mlat* divided by *mbwlat*. This parameter characterizes the maximum number of requests which the memory can be processing concurrently. Thus if *mlat* and *mbwlat* are equal then *mconc* is unity and the modelled memory system is assumed to be able to process only a single transaction at a time. If *mlat* is twice *mbwlat* then *mconc* is two and the memory system is modelled as being able to operate on two separate requests simultaneously. As described above, for memory systems with *mconc* greater than unity, request processing is modelled as taking place in a pipelined fashion.

As an example, assume *mlat* is 20 and *mbwlat* is 5. If 4 transactions are submitted in short succession, the first will complete 20 cycles after its submission. The second will be initiated 5 cycles later and thus complete 25 cycles after initiation of the first. The third and fourth requests will complete after 30 and 35 cycles respectively.

An alternative model for a memory system with a concurrency of 4 might assume the existence of four asynchronous memory banks, each with latency 20. Under this model, if all memory banks were idle requests could be submitted to each, and the 4 requests would each finish 20 cycles after their initiation. Implementation of memory systems with large values of *mconc* are likely to be banked, thus the latter model might more accurately characterize their behavior. The pipeline model is simpler, however, and should adequately characterize behavior.

Buffers for unsubmitted prefetch requests are implicitly assumed to exist but not explicitly modelled. The compiler assumes that prefetch requests are flow controlled. Thus if buffers are full and a prefetch request results in a miss the processor is stalled until a buffer is available. Alternative behavior would be to drop prefetch requests. If a memory system dropped prefetch requests it would be important to explicitly model buffers in order to avoid this situation.

One final aspect of memory system modelling regards the processor memory interface behavior in the presence of write requests resulting in misses. Write misses may be buffered externally to the processor in which case these misses only produce stalls when buffers fill up. Alternatively, write misses may stall the processor while the request is processed if no buffering exists. In the former situation it is seldom advantageous to perform prefetching for writes since writes don't result in stalls. The only possible advantage of prefetching writes in this case is to smooth memory bandwidth utilization. In the case of writes which stall it is just as beneficial to perform prefetches for write misses as it is for read misses. This aspect of memory system behavior is specified by enabling or disabling prefetching for badref writes.

5.2. Latency Tolerance Optimization

In order for a particular memory reference instruction to contribute substantially to the total misses of a program it must have a high individual miss rate and must be executed many times. Upon successive executions of the same memory reference instruction, systematic misses can occur for one of several possible reasons.

- The address which is used by the reference changes between successive executions.
- The address does not change but sufficiently many misses intervene between successive executions that most or all of the contents of the cache is purged, including the old value.
- The address used by the reference is involved in a pattern of conflict misses with one or several other memory references.

Repeated execution of a particular instruction primarily results from some form of looping control structure. This looping may occur explicitly within a single procedure. It may alternatively result from explicit looping within a different procedure or some generalization of looping such as recursion.

C-flat uses two distinct heuristic optimization techniques to address badref memory references with different characteristics. The first heuristic technique targets badrefs with changing addresses occurring in explicitly recognizable loops. These badrefs will be referred to as RAADs, Regular Accesses to Aggregate Data Structures. RAADs account for a large fraction of badrefs occurring in many programs. Special purpose heuristics for RAADs can exploit the fact that these references occur in loops. The general strategy of RAAD heuristics is to perform prefetches in a current iteration based on a prediction of addresses to be used by badrefs in some future iteration. Given long loops, sufficient memory bandwidth and good address prediction, prefetches can be performed far enough ahead of references to tolerate arbitrarily high memory latencies.

RAAD prefetching is restricted to those badrefs which are RAADs and occur in a loop which can be explicitly identified. A second general purpose heuristic technique can be utilized for almost any badref. This technique simply inserts prefetches and reschedules a sequence of code in order to provoke misses for badrefs well in advance of the actual occurance of the badref instructions. In order to tolerate latencies which are substantially longer than typical basic block sizes, this scheduling phase operates on sequences of basic blocks in a technique similar to trace scheduling.

Conflict misses which are sufficiently tightly spaced are not addressed by either technique. If conflict misses occur on a time interval comparable to the memory latency then both techniques may even exacerbate this problem. Either might insert a prefetch for a reference which could displace a useful value and then be displaced itself before being used. Conflict misses with spacing comparable to the memory latency are presumably quite rare and can be made even more rare by employing victim caching.

5.3. Loop-based Prefetching Heuristics

A significant fraction of misses incurred by programs occur within explicitly recognizable loops. Memory instructions in loops systematically produce misses when the address to which they refer changes from iteration to iteration. Empirically, we observe that these references with loop varying addresses are primarily associated with RAADs, Regularly Accessed Aggregate Data Structures. Regular aggregate data structures consist of sets of elements with the same storage layout. These sets may be grouped in sequential storage as arrays or alternatively may be grouped through explicitly contained pointer structure. Loops are used to traverse regular data structures, accessing one element and then moving on to another element. This results in the changing addresses.

RAAD traversal in loops leads to goodrefs and badrefs. Consider a loop which moves through an array of data structures, each consisting of several components. In each loop iteration, a small number of memory reference instructions can be identified which access one of the components of the newest array element for the first time. In the absence of conditional behavior this set will contain one memory reference instruction per component. If the aggregate data structure is not contained in the cache at the time the loop starts executing then each time one of the instructions which first references a component executes it produces a miss. This occurs on every iteration. Other memory reference instructions which reaccess a component from the current iteration or a previous iteration always hit.

RAAD addresses change from iteration to iteration. Any loop has some basic set of recurrences from which all loop variant values are derived. The variables in these recurrences will be called induction variables or IVs. The term induction variable is sometimes used to refer specifically to variables involved in linear recurrences. It will be used more generally here to refer to any variable involved in a recurrence, *i.e.* any loop variant variable whose new value in the next loop iteration is a function of its current value, whether or not this function is linear. Loop variant RAAD addresses are computed from the induction variables of the loop.

Since the elements of RAADs have a fixed structure, the process by which addresses are generated to traverse them is usually fairly simple. By examining the recurrences which generate the induction variables used in RAAD addresses it is usually possible to predict the addresses for future iterations. Predicted addresses can be used to perform prefetches and thus initiate memory transactions for the data needed in these future iterations well ahead of its eventual use.

Loop-based prefetching techniques for RAADs are simplest when loops do not contain conditionals, inner loops or procedure calls. In upcoming sections these techniques will first be examined in the simple context of unconditional inner loops and then revisited to address issues associated with conditionals and contained loops or procedure calls.

5.3.1 RAAD Identification Algorithm

RAADs are optimized in c-flat through a procedure involving classification followed by application of a prefetching schema based on the specific form of RAAD involved. While RAADs can arise from a variety of syntactic constructs, in a low-level representation address code for related RAADs has a common structure. RAADs are identified and classified in c-flat by constructing an instruction-level dataflow graph for a loop which includes edges for loop-carried dependencies. In such a graph, induction variable recurrences form a Strongly Connected Component or SCC. Badref RAADs are classified based on the structure of the SCCs used to compute their addresses and the way in which the induction variables based on these SCCs are used. Two distinct forms of induction variables are recognized by c-flat corresponding to linear recurrences and memory indirection recurrences. Figure 5-1 illustrates examples of the SCC graph structure and corresponding recurrence equations for these induction variables. Three distinct varieties of RAADs with addresses built from these IVs are recognized by c-flat. These RAADs correspond to the following data structure access patterns:

- Loop-constant stride array accesses

- Accesses using a loop-constant stride indirection array
- Linked-list dereferencing

Sample dataflow graph structure and C language syntactic constructs producing these RAADs are illustrated in Figure 5-2.

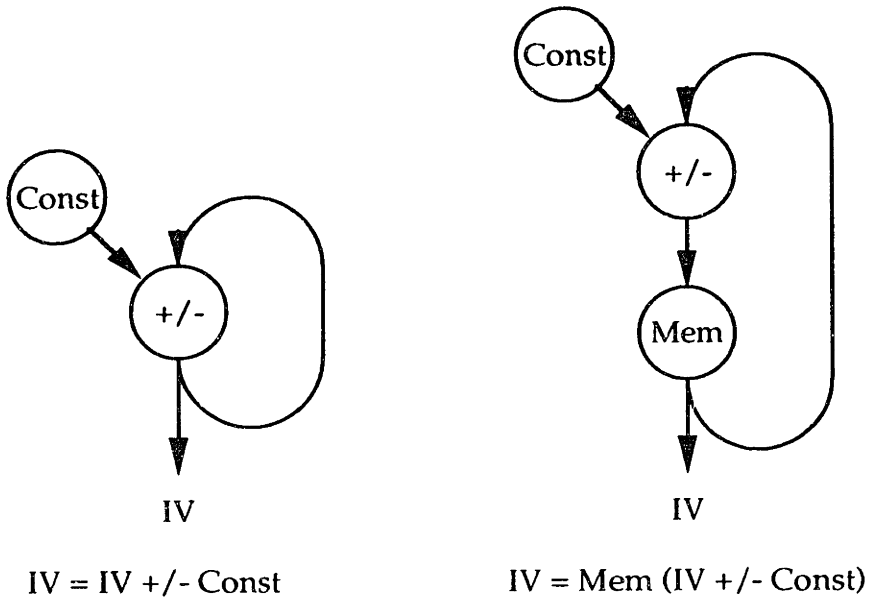


Figure 5-1: Characteristic IVs and SCCs in C-flat

C-flat uses the following algorithm to identify and classify badref RAAD memory references.

```
for each loop in inner to outer order
    if loop contains badrefs
       build an instruction level dataflow graph with loopback edges
       compute SCCs for graph
       classify induction variables based on their associated SCCs
       for each memory reference in the loop
           if reference is a badref
              for each component of the reference's address
                 recursively trace through dataflow graph terminating at SCCs
                 add IVs for these SCCs to an IV set for reference
              examine the set of IVs from which address is constructed
              based on IVs and their uses classify badref
```

In the intermediate structure manipulated by c-flat memory references have an explicit representation. Each such reference is mapped to a specific load or store instruction in generated code from which it receives a memory behavior estimate in the form of a miss rate. Badrefs are identified by thresholding references based on their miss rates.

Each loop in a program is examined for badrefs. In the case of nested loops, inner loops are examined first. In order to classify badrefs the computation producing their addresses must be isolated and

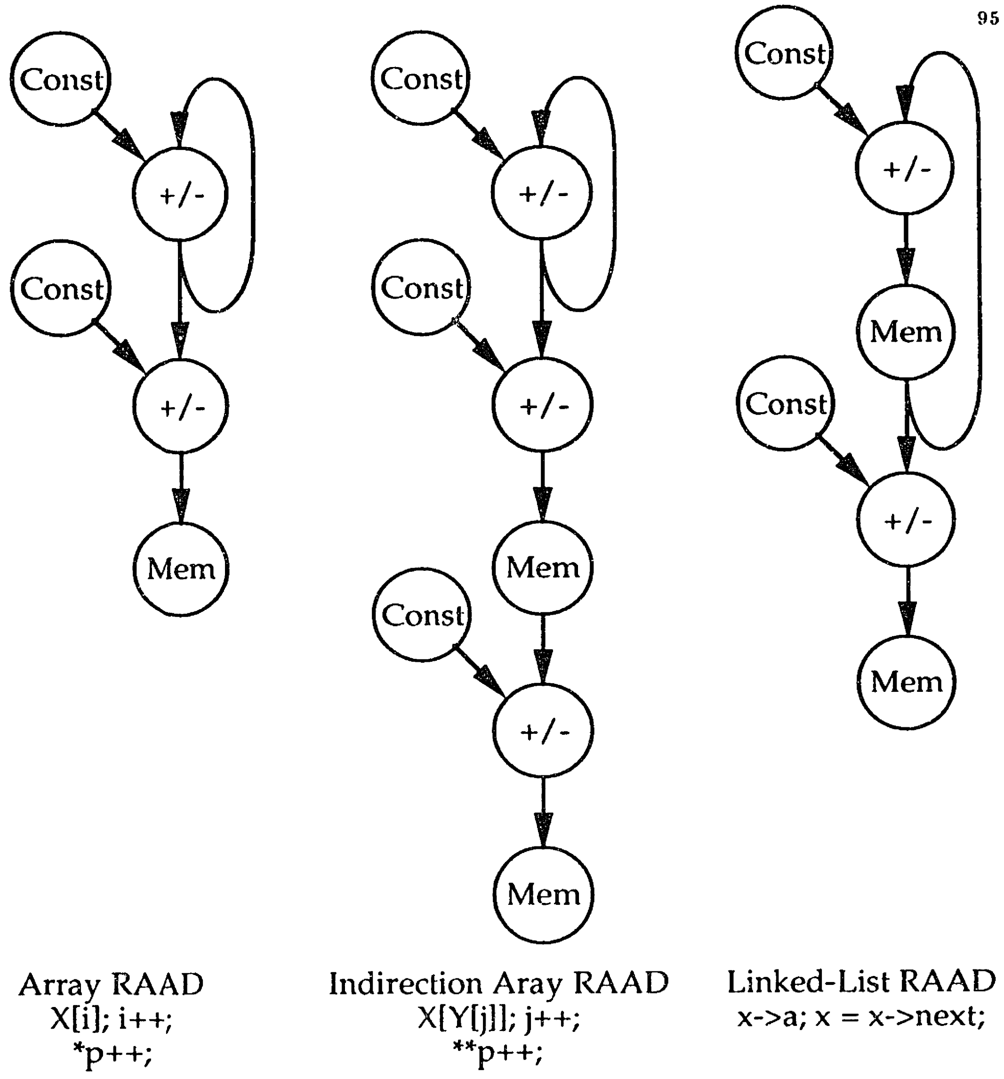


Figure 5-2: RAADs in C flat

examined. A loop dataflow graph is constructed which contains edges for variables live at loopback. SCCs within this graph are computed. These SCCs form a basis set for the address recurrences of potential badrefs.

For each badref in a loop a traversal of the loop dataflow graph starting at the badref identifies all SCCs which contribute to the badref address. By considering the set of SCCs producing a badref address and the way these values are combined, badrefs can either be identified as one of the recognized RAAD structures or identified as an unrecognized construct. Once badrefs are classified, RAAD specific prefetching schemas can be applied. This occurs after all RAADs have been classified because some prefetching schemas optimize multiple badrefs.

5.3.2 Array RAADs

Array RAADs have access patterns which are generalizations of array accesses with a loop constant stride. Array RAADs include as a subset both loop-constant stride array references and pointer dereferences with a fixed pointer update. More generally, an array RAAD is a memory reference instruction whose address is a linear combination of loop constants and linear induction variables. IVs and constants can be multiplied or shifted by a loop constant value and combined using addition or subtraction. For such a reference the difference between addresses used on successive iterations, the stride of the reference, is just the sum of the increments of each induction variable scaled appropriately if the induction variable is scaled in the address computation. An address appropriate for an arbitrary iteration in the future can be computed from an address for the current iteration. To compute the address for a corresponding reference I iterations in the future the current address can be incremented by I times the stride of the reference.

Latency tolerance optimization for array RAADs uses a very simple prefetching schema. Consider the following code fragment which is assumed to come from a loop in which **rp** is not otherwise modified.

```
          ...
ld [rp], rt       (Load 1)
...
add rp,4,rp
          ...
```

Latency tolerance optimization for array RAADs is accomplished using the following four steps.

1. Based on the loop dataflow graph, compute the stride by finding the increment associated with each IV.
2. Choose a prefetch iteration count. The prefetch iteration count is the number of iterations separating RAAD prefetches from their corresponding memory references. This value will be denoted *PIC*. It is the same for every RAAD in a loop and is a small constant. Selection of *PIC* is discussed in more detail in section 5.3.5.
3. Add code to compute the product of the stride and prefetch iteration count. If the stride is a constant, (as opposed to just a loop-constant), then this product will be a constant value and may not explicitly need to be computed.
4. Add code to perform a prefetch reference using an address computed as the sum of the current iteration's address and the value computed in step 3. Since the current iteration's address is used in the prefetech address computation, it is most convenient and often results in lowest overhead to add prefetch code immediately after the associated badref.

If `Load 1` is a badref the steps outlined above are applied. The stride of the reference is identified as 4. A prefetch iteration count is determined. Since both the stride and *PIC* values are constant in this case, no code is required to compute their product. Finally, prefetch code is added using an appropriate address. In code examples, a prefetch instruction will be denoted by the operation **pref**.

```
...
ld [rp], rt     (Load 1)
pref [rp+PIC*4] (Pref 1)
...
add rp,4,rp
...
```

The example above illustrates application of the array RAAD prefetching schema in its simplest form.

The code below illustrates an example in which the reference stride is a loop constant value but not a constant and PIC is chosen as two. Values in registers **ra** and **rs** represent a loop-constant array base and stride.

```
...
ld [ra+ri], rt (Load 1)
...
add ri,rs,ri
...
```

Application of the prefetching schema results in the following code in which **rn1** and **rn2** represent new temporary registers.

```
...
ld [ra+ri], rt  (Load 1)
add ra,ri,rn1
sll ri,2,rn2
pref [rn1+rn2]  (Pref 1)
...
add ri,rs,ri
...
```

Notice that the software overhead associated with prefetching varies in the two examples. In the first example, only a single instruction of overhead is introduced whereas three instructions are introduced in the second example. Given both an estimate of software overhead and an estimate of miss rate a cost-benefit analysis can be performed to determine if adding a prefetch instructions for a particular reference is likely to improve or detract from code performance.

Assume application of a prefetch schema adds O instructions of overhead. Added cycles for these instructions are incurred on every occurance of the badref independently of whether or not it hits or misses. Assuming the badref is executed E times, the total software overhead incurred is just $O * E$ cycles.

A savings of L cycles is realized whenever a miss occurs. If the badref instruction produces M misses then the total savings ignoring overhead is $L * M$. Prefetch optimization improves performance when the net savings, the difference in savings and cost is greater than 0. This occurs when the relation in Equation 5-1 is true.

$$L * M - O * E > 0 \qquad (5-1)$$

M/E is the miss rate of the reference which will be denoted by m. The maximum savings which can be realized for a miss is $mlat$. On machines in which misses produce immediate stalls this will be the actual savings. If misses do not produce immediate stalls this is an upper bound on savings. Rearranging Equation 5-1, a prefetch is likely to improve performance given an overhead of O and a miss rate of m when the relation in Equation 5-2 is true.

$$m > O * 1/mlat \qquad (5-2)$$

Since latency tolerance optimization based on prefetching always has an overhead of at least one cycle, RAAD references with m less than $1/mlat$ cannot be profitably optimized using these techniques. This accounts for the initial goodref/badref partitioning threshold at $m = 1/mlat$.

5.3.3 Indirection Array RAADs

An indirection array RAAD is a reference whose address is a linear combination of constants, linear IVs and one or more values from memory references which are array RAADs. This class includes as a subset accesses produced by the C syntactic constructs `X[J[i]]`, where `i` has a constant increment, and `**p++`. The characteristic SCCs associated with indirection array RAADs are the same as those for array RAADS but for at least one such SCC there is a memory load on the path from the SCC to the indirection array RAAD whereas simple array RAADs have no indirection.

Prefetch optimization for indirection array RAADs has several additional twists not encountered with simple array RAADs, complicating their prefetch schemas. Components of the indirection array RAAD which are themselves array RAADs require the generation of memory reference instructions. These instructions access elements of an array RAAD which will be used in future iterations and thus may themselves be badref memory references.

Consider an example corresponding to the following C source fragment.

```
int a, **p;
while (1)
  a = **p++;
```

If `a` and `p` are allocated to registers, the body of the loop might produce the code fragment below. Variable `a` is in `ra`, variable `p` is in `rp` and `rt` is a new variable which corresponds to values of `*p`.

```
...
ld [rp], rt       (Load 1)
add rp,4,rp
ld [rt], ra       (Load 2)
...
```

`Load 2` in the fragment above is the indirection array RAAD reference. In order to perform an appropriate prefetch for this reference an address based on the value of `rt` for a future iteration is required. This value can be attained by loading a new temporary register using an address generated by the procedure described for array RAADs. An important distinction between the code which loads this temporary and the code generated for an array RAAD prefetch is that in the indirection array case a real memory load is generated as opposed to a prefetch. In the modified fragment `Load 1'` has been added to access the next iteration's value of `rt` and `Pref 2` is a prefetch for `Load 2`. This example assumes the prefetch iteration count, *PIC*, is 1.

```
...
ld [rp], rt        (Load 1)
ld [rp+4], rt'     (Load 1')
add rp,4, rp
ld [rt], ra        (Load 2)
pref [rt']         (Pref 2)
...
```

It is quite possible that in the initial code `Load 1` might have been a badref in addition to `Load 2`. Applying the RAAD prefetch schema to `Load 1` produces the following code.

```
...
ld [rp], rt     (Load 1)
pref [rp+4]     (Pref 1)
ld [rp+4], rt'  (Load 1')
add rp,4, rp
ld [rt], ra     (Load 2)
pref [rt']      (Pref 2)
...
```

Unfortunately, this code is not tolerant to latency from misses associated with dereferencing `rp`. `Load 1'` dereferences values of `rp` before `Load 1`. As a consequence, if `Load 1` is initially a badref it is transformed into a goodref through the addition of `Load 1'`. `Load 1'` is now a badref, however. The array RAAD schema can be applied to newly generated array RAAD `Load 1'` rather than `Load 1`. Notice that since the address used by `Load 1'` is `rp+4`, the address for a generated prefetch is `rp+8`.

```
ld [rp], rt     (Load 1)
ld [rp+4], rt' (Load 1')
pref [rp+8]     (Pref 1')
add rp,4, rp
ld [rt], ra     (Load 2)
pref [rt']     (Pref 2)
```

A somewhat more subtle problem may arise if another reference in the loop uses the same address as a component of an indirection array RAAD. Consider the modified fragment below.

```
int a, *b, **p
 while (1) {
  b = *p;
  ...
  a = **p++;
}
```

Code generated for the assignment to variable a will be similar to the earlier examples. Code generated for the assignment to b has been added.

```
...
ld [rp], rb       (Load 0)
      ...
ld [rp], rt       (Load 1)
add rp,4,rp
ld [rt], ra       (Load 2)
...
```

`Load 1` cannot be a badref since it uses the same address as `Load 0`. If the indirection array prefetching schema described above is applied, despite the fact that `Load 1` is not a badref, the newly generated `Load 1'` may still be a badref. This can occur if `Load 0` is a badref. Just as the addition of `Load 1'` converted `Load 1` from a badref to a goodref in the initial example, it can convert

Load 0 from a badref to a goodref in this example. In both cases, the result is that Load 1′ itself is a badref.

In c-flat these complications are handled by processing all array RAADs with equivalent addresses simultaneously. At most one reference from a set which use identical addresses can be a badref. If one of the references is a badref code is added at this location. A load for the next iteration's value, similar to Load 1′ in the example above, is added. Following this a prefetch like Pref 1′ is added. If no reference in the set is a badref then no prefetch is added. This technique is still vulnerable in the case illustrated in the example below in which some badref instruction in the loop explicitly accesses the next iteration's value of an indirection array, however this appears to happen infrequently. No instance of this situation was detected in any of our benchmarks.

```
...
.. X[J[i]] ..
...
.. J[i+1] ..
...
```

The technique of unifying the treatment of array RAADs with equivalent addresses described above has an additional advantage. It is possible that an array RAAD is used as an address component in more than one indirection array RAAD badref, as occurs in the fragment below if X and Y are badrefs. Processing array RAADs with equivalent addresses as a single entity leads naturally to the creation of only a single new variable with the value J[i+1]. With extra care, any common scaling of the new variable, as would be required if X and Y were arrays of equal sized elements, can also be shared.

```
for (i=0; i<10; i++)  {
    ..X[J[i]]..;
    ...
    ..Y[J[i]]..; }
```

One might ask why, if only a single prefetch access to J[i+1] is required, should there be two accesses to J[i]. This punctuates an issue regarding dependence analysis required for code optimizations like common subexpression elimination in contrast to analysis for prefetch generation. One can easily imagine that a reference is interposed between X and Y in the code which cannot be disambiguated with respect to J[i], such as the reference to *p in the augmented code fragment below.

```
for (i=0; i<10; i++) {
   X[J[i]];
   *p = 1;
   ...
   Y[J[i]]; }
```

Common subexpression optimization cannot be applied by a compiler unless it can prove that *p and J[i] always refer to different locations since improper application of this transformation could produce incorrect program behavior. Prefetch optimization does not semantically alter programs. Even if J[i+1] cannot be proven to be independent of *p, if it is likely that it is generally different from *p then there is no harm in using only a single version of J[i+1]. When prefetching is based on incorrect analysis, the result is that a program incurs some additional or unexpected misses. Prefetch optimization can legitimately be based on approximate or optimistic analysis, as long as this analysis is mostly correct. In contrast, transformations with the potential to semantically alter programs must be based on exact or conservative analysis.

As with array RAAD prefetching, a cost-benefit analysis can be performed for indirection array RAAD prefetching based on miss rate and overhead. The overhead consists of two components, one associated with building a prefetch address and performing the prefetch operation and a second associated with loading future values for required array RAADs. The latter component can potentially be amortized across prefetches for several indirection array RAADs. The heuristic test actually implemented in c-flat is not clever about this amortization. It charges the overhead of loading future array RAAD values to the first indirection array RAAD requiring those values. If prefetch code is generated, no other indirection array RAAD is charged. If overhead is deemed too expensive and prefetch generation is suppressed, subsequent indirection array RAADs are charged with the overhead. Using this technique it is possible to suppress a set of indirection array RAAD prefetches when no single prefetch justifies building shared code but due to amortization, in aggregate, prefetching for each is justifiable. Once overhead from both components has been estimated, the benefit of a particular indirection array RAAD prefetch can be computed using the relation in Equation 5-2.

5.3.4 Linked-List RAADs

Linked-list RAADs are references performing traversal of generalized linked lists. Generalized linked-lists are regular data structures composed of elements with fixed storage formats which contain pointers or array indices identifying other similarly structured elements. The IVs and corresponding dataflow graph SCCs which give rise to linked-list RAADs differ from those associated with array or indirection array RAADs. Linked-list IVs have a memory reference node within their SCC whereas the IV SCCs associated with array RAADs consist solely of addition and subtraction nodes.

Prefetch optimization for linked-list RAADs can follow an optimization schema similar to that used for indirection array RAADs. The memory reference in the induction variable recurrence leads to some added implementation complexity.

Consider the following code fragment.

```
while (p) {
  t = p->a;
  ...
  p = p->next; }
```

If a and `next` have structure offsets of 0 and 4 respectively and p and `t` are allocated to `rp` and `rt` the following fragment of code is a possible translation of the loop body.

```
ld [rp], rt   (Load 1)
...
ld [rp+4], rp (Load 2)
```

As is required for indirection array RAADs, in order to produce prefetch addresses for linked-list RAADs a new variable must be introduced and a memory reference added to give the new variable a value. The new variable needed corresponds to the next iteration's value of `rp`. In the code below `rp`′ contains the new variable and `Load 2`′ assigns values to this variable. `Pref 1` uses `rp`′ in order to prefetch a value for badref `Load 1`.

```
        ld [rp], rt    (Load 1)
        pref [rp']     (Pref 1)
        ...
        ld [rp+4],rp   (Load 2)
        ld [rp+4],rp'  (Load 2')
```

As with indirection arrays, the newly introduced load, **Load 2′**, is likely to be a new badref. This is true irrespective of whether **Load 2** is originally a badref. In order to avoid latency associated with misses for this new badref an additional prefetch can be generated. The final version of the code is illustrated below. This version includes loopback code in addition to the partial loop body.

```
LOOP:
        ld [rp], rt    (Load 1)
        pref [rp']     (Pref 1)
        ...
        ld [rp+4],rp   (Load 2)
        ld [rp+4],rp'  (Load 2')
        pref [rp'+4]   (Pref 2')
        tst rp
        jne LOOP
```

In the code above, the first non-prefetch reference to a new structure element, **Load 2′**, accesses the link pointer. This is immediately dereferenced by **Pref 2′** in order to initiate a memory transaction for the cache block containing the next needed link pointer. As badref components of the element are accessed prefetches are issued for the corresponding component of the next element with an address constructed by replacing **rp** with **rp′**.

Notice in the code above that the newly introduced variable **rp′** is used as a prefetch address before it receives an initial value. This results in prefetches through spurious addresses. Since prefetch instructions are assumed to be safe for speculative use, suppressing any associated faults, this does not result in program errors. Performance can be improved, however, by initializing **rp′**. In prologue code prior to beginning the loop **rp′** can be initialized with a value by appropriately dereferencing the initial value of **rp**. Initializing **rp′** converts the first occurance of **Pref 1** from a wasted instruction to an effective prefetch. This in turn avoids latency associated with a subsequent miss by **Load 1**. Under the indirection array RAAD prefetching schema this same form of behavior can occur, in which a newly created variable is used in a prefetch computation before receiving its first value within the loop. In each case c-flat initializes these variables in a loop prologue.

It was stated above that memory references in the induction variable recurrences of linked-list RAADs lead to additional complexity. Upon close examination or execution of the code above a problem manifests itself. This code dereferences a null pointer, potentially leading to memory faults. What has happened?

At the end of each iteration when **rp** is updated a new value of **rp′** is loaded by dereferencing **rp**. On the final iteration when **rp** is updated it becomes a null pointer. The corresponding update of **rp′** dereferences the null **rp** producing an invalid memory reference.

Use of a fault-safe prefetch instruction generally allows speculative prefetching to be performed without the need for significant verification of the validity of resulting code. This fault-free safety net does not apply to memory references using normal load and store instructions. The term **required reference** will be used to refer to semantically significant memory references, *i.e.* loads and stores. If address generation code used for prefetching necessitates the addition of required references, as opposed to prefetch references, then compiler analysis is required to ensure that added references do not result in faults.

The problem with the above code can be addressed using one of two different software techniques or alternatively solved with additional hardware support. The software techniques modify the code to avoid dereferencing the null pointer. The hardware technique provides a legal way to dereference such a pointer.

The problem can be addressed in hardware by adding another new instruction in addition to the prefetch instruction. The new instruction is a speculative load instruction which will be given the code syntax ldsp. Speculative load acts like an ordinary load with the following caveat. Under normal circumstances, speculative load accesses and returns valid data. At any time, including any fault conditions and any other time deemed convenient, speculative load is free to return an arbitrary value. Speculative load is not permitted to produce any faults, however. Since it does not produce semantically meaningful data speculative load is not useful under normal circumstances. The time when it can be beneficially employed is when a memory value is required speculatively for a prefetch computation.

In the likely absence of a speculative load instruction the problem with the linked-list prefetching schema must be addressed in software. Two techniques can be employed to ensure that spurious faults are not introduced by prefetch address generation. Each avoids loading **rp′** once **rp** has become null.

The first technique, implemented in c-flat, avoids the spurious load by interchanging the order of prefetch address generation code and the loop test. Once the new **rp** value has been tested a new iteration is guaranteed to occur and it is safe to dereference **rp**. Moving the prefetch code to a position logically following the loopback tests requires moving it into the next iteration at the beginning of the loop. In this revised schema the new variable **rp′** is no longer used before it is set so prologue initialization is also no longer performed. The resulting code is illustrated below.

```
LOOP:
        ld [rp+4],rp' (Load 2')
        pref [rp'+4]   (Pref 2')
        ...
        ld [rp], rt    (Load 1)
        pref [rp']    (Pref 1)
        ...
        ld [rp+4],rp  (Load 2)
        ...
        tst rp
        jne LOOP
```

Intuitively, the code above is valid. It has avoided potential faults associated with **Load 2′** by exiting the loop before these faults arise. One can more formally argue the validity of this technique as follows.

1. A memory reference added to a program cannot introduce faults associated with invalid addresses if it is dominated or post-dominated by another reference or set of references which use the same address.

   By the definition of dominance/post-dominance, execution of the added reference implies prior/eventual execution of another reference using the same address. The added reference can only fault if the (post)dominating reference faults, thus no new faults are produced.

2. If all loop-exit conditions are dominated by memory references using some particular address, instructions in the first basic block of the loop are post-dominated by these references.

   In a program which terminates, executing the first block of some loop must eventually be followed by a loop exit. Thus the set of loop exits post-dominate the first block in a loop. If A is post-dominated by B which is dominated by C, then C either dominates or post-dominates A.

Thus a load such as **Load 2′** can safely be added at the top of a loop if each loop exit occurs in conjunction with a memory reference using the same address. If some paths can exit the loop without dereferencing the required pointer, prefetch code cannot be inserted until after the branches leading to these exits. C-flat uses a slightly less restrictive test, ensuring that each exit is dominated by a memory reference which accesses a component of the structure pointed to by **rp** but not necessarily the same component used in the newly added load.

The technique of moving the prefetch code to the beginning of loops can almost always be validly applied. In a situation in which the addresses associated with **Load 2** and **Load 2′** were somewhat more complicated, lower software overhead might result if **Load 2′** could be placed adjacent to **Load 2**. Loop peeling, described in [40] as a technique for dealing with vagaries in initial loop iterations, can be used to eliminate the spurious faults in the original code by peeling off the final iteration. Loop peeling is applicable when the loop termination condition involves **rp** and when **rp** cannot be changed by any instruction in the loop except the update. (It is also applicable for do loops with loop constant upper limits, in which case a different procedure from that outlined below is appropriate.) Loop peeling is accomplished by modifying the loop termination condition to use **rp′** rather than **rp**. Since **rp′** is a copy of **rp** advanced by one iteration the peeled loop terminates one iteration sooner than it should. A copy of the loop body must be placed after the loop to perform the final iteration. Within this copy, prefetching code can be omitted, avoiding the invalid load. This transformation has the potential to change program semantics so dependence analysis to ensure that **rp** is not changed by unexpected instructions within the loop must be conservative.

While not discussed in the context of indirection array RAADs, a potential problem with faults exists in this context as well. Array elements are accessed with required references beyond the bounds imposed by loops. (Addresses beyond the end of arrays are even accessed in array RAAD prefetching but only with prefetch instructions.) Techniques like those described above could be applied to solve this problem for indirection array RAADs. Alternatively, faults associated with the validity of addresses can be avoided by padding data segments with the maximum distance by which indirection array prefetching might overstep the top of an array with a load reference. c-flat assumes data segments are suitably padded and ignores this issue for indirection array RAAD prefetching.

Padding of data segments is perhaps a dubious mechanism for dealing with this problem although it is the simplest. First, if an array with non-constant stride is overstepped it may be difficult to quantify by what amount its corresponding data segment needs to be padded. Additionally, even though padding eliminates faults associated with referencing illegal addresses, in the context of virtual memory it may produce extraneous page faults, causing unncessary disk traffic. Despite these issues, since c-flat is a research compiler, the simplest strategy has been adopted.

5.3.5 Prefetch Iteration Count Selection

Prefetches must precede their associated badrefs by an adequate distance in order to avoid memory latency induced stalls. If a memory latency parameter, *mlat* in c-flat, is provided to a compiler in units of cycles, (inverse instruction issue bandwidth units), adequate separation can be attained independent of superscalar or superpipelined behavior by scheduling *mlat* instructions between prefetches and badrefs. In the loop-based techniques in c-flat prefetch references primarily occur adjacent to their associated badrefs. The effective separation in instructions is an integral multiple of the number of instructions in a loop. This multiple is the prefetch iteration count.

Assume a loop has I instructions. In order to completely mask latency, the prefetch iteration count should be at least $\lceil \frac{mlat}{I} \rceil$. It is not advantageous to set the count any higher than this value. If data arrives in the cache 1 cycle or 1000 cycles prior to when it is needed it can be accessed without delay in either case. There are several performance factors which favor "just in time" scheduling. The loop prefetching techniques described above implement a software pipeline to overlap miss latency and computation. This pipeline has fill and drain costs proportional to the depth of the pipeline. In order to minimize these costs the depth should be kept to a minimum. Additionally, when prefetching into a cache, the cache is being used for two distinct purposes. It serves its normal caching function and additionally provides buffering space for prefetched values between the time at which they are prefetched and eventually used. If the number of buffered values is very small in comparison to the size of the cache then the buffering function results in minimal interference with the caching function. In contrast, if sufficiently many prefetches are performed far in advance of their eventual use that the number of buffered prefetches is not negligible in comparison to cache size, then the buffering function is likely to negatively impact normal cache performance. Prefetching should be performed far enough ahead of use to mask latency but, if possible, no further. This suggests setting PIC, the prefetch iteration count, at exactly the value prescribed above.

$$PIC = \lceil \frac{mlat}{I} \rceil \qquad (5-3)$$

Irrespective of the number of iterations computed using the formula above, memory latency and bandwidth considerations sometimes suggest further limiting the prefetch iteration count. If memory performance considerations predictably limit the latency of loops to a value higher than that estimated based on instruction count this revised latency estimate should be used in computing the prefetch iteration count.

An event triggering the initiation of a slow memory transaction followed by an event which potentially stalls if the transaction has not completed will be referred to as a miss-use pair. If both components of a miss-use pair occur in the same loop iteration the latency of the loop for a single iteration will exceed *mlat* and thus the prefetch count should be set to 1. Miss-use pairs can occur within a single loop iteration for a variety of reasons, several of which are listed below.

Unrecognized Badrefs

Not all badrefs within loops can be recognized as RAADs and prefetched. One example of such a badref arises sometimes in computations performing floating point histogramming. In such a computation a floating point value is normalized and converted to an integer which is used to choose an element of a histogram array. Any address computation involving floating point arithmetic is unrecognized within c-flat. Unrecognized prefetches also result if addresses are derived from non-linear arithmetic recurrences or memory-based recurrences not matching the linked-list RAAD pattern recognized.

Linked-list RAADs

The existence of linked-list RAADs act as a miss-use pair if the memory reference in the SCC is a badref. Each iteration dereferences a pointer. The result of this memory access is required before the instruction can be repeated in the next iteration. Viewed another way, linked-list RAADs produce a sequence of dependent badrefs in which the dependence arises through an address computation. This form of dependence cannot be skirted using speculative computation without guessing future addresses by some means. Since there is one such reference per iteration and these references cannot execute in parallel, the time for a loop iteration is bounded below by the memory latency.

Inner Loops

Computation associated with execution of a set of loop iterations almost always produces miss-use pairs. As a consequence, outer loops almost always have miss-use pairs within each iteration associated with their inner loops.

Procedure Calls

Similar to inner loops, almost all procedure calls have miss-use pairs.

Many Intermediate Miss Rate References

While the badref model downplays memory references with low but non-negligible miss rates such references sometimes occur. A loop containing a large number of references with miss rates too low to warrant prefetching may be likely to encounter miss-use pairs within many iterations. c-flat computes the sum of the miss rates of references which are not prefetched. If the sum exceeds one, this is considered to be equivalent to an unrecognized badref. This is an overly pessimistic strategy when misses occur in correlated clusters. Correlation may be likely, particularly in conjunction with initial loop iterations. Nonetheless, this is the heuristic used in c-flat.

Unrecognized badrefs or linked-list RAADs can occur in loops of any size. The latter three sources of miss-use pairs do not typically occur in loops with small expected instruction counts. By virtue of containing either inner loops, procedure calls or a large number of memory references, loops must be reasonably large, resulting in a natural tendency for the associated PIC value to be one. Given very large memory latencies it is possible that these issues might result in decreasing a PIC value which is not one.

In addition to memory latency based performance limits, memory bandwidth constraints can also limit loop iteration latencies. If the number of misses expected on each loop iteration exceeds the maximum memory system concurrency, denoted *mconc* in c-flat, then bandwidth related stalls will occur. More generally, the memory bandwidth becomes saturated when the product of the prefetch iteration count and the number of misses expected on an iteration, computed by summing the expected miss rates of all memory references, exceeds the maximum concurrency of the memory system. An additional component of the bandwidth requirements of a loop arises from writebacks of dirty cache blocks. Given an estimate of the writeback rate associated with miss references, this can be included in bandwidth estimates as well to yield the formula below.

$$PIC = \lceil \sum_{references} m_i * (1 + wb_{avg})/mconc \rceil \qquad (5-4)$$

The final prefetch iteration count should be chosen as the minimum valued determined based on the criteria above.

Klaiber performs an analysis similar to that above in [25]. He assumes an interface without flow control for prefetches and thus develops an additional constraint to ensure that buffers for uninitiated prefetch references are not overflowed. His analysis appears to ignore the situations described above in which c-flat sets PIC to one due to miss-use pairs within an iteration.

5.3.6 Cache-block Based Optimizations

When cache misses are serviced a fixed-sized block of data which is typically larger than the data size of a single memory reference is transferred to the cache. If a set of memory references occur with a spatially dense address pattern, transfer of data in large blocks exploits this spatial locality to improve performance by lowering the miss rate in comparison to that incurred if data for each individual reference were transferred separately. Use of large cache blocks can be viewed as a form of hardware prefetching. This hardware prefetching has the effect of lowering the miss rates of badref memory references, thus decreasing the relative benefits of software prefetching. Under some circumstances prefetching associated with cache blocks can be systematically exploited to decrease the overhead of software prefetching. Two distinct optimization techniques can be applied corresponding to situations in which the dynamic references which benefit from cache block transactions arise from a set of distinct memory reference instructions or from the reexecution of the same memory reference instruction.

An example of the first form of optimization arises in linked-list RAAD prefetching. On successive loop iterations, new multicomponent data structures are accessed. Components of a particular structure comprise a dense set of memory locations. In order to prefetch all components it is only necessary to perform prefetches for the set of cache blocks containing the components, as opposed to prefetches for each individual component. Consider a data structure consisting of 5 4-byte components. Assume a cache block size of 8 bytes. As illustrated in Figure 5-3 this structure has two possible alignments with respect to cache block boundaries and in each case spans 3 cache blocks. It is quite possible that different memory references miss depending on the two possible alignments of the structure, producing 5 badref memory references with average miss rates of about 0.6. All misses associated with a single data structure can always be eliminated by adding only 3 prefetches, however, one for each cache block in which the structure is contained.

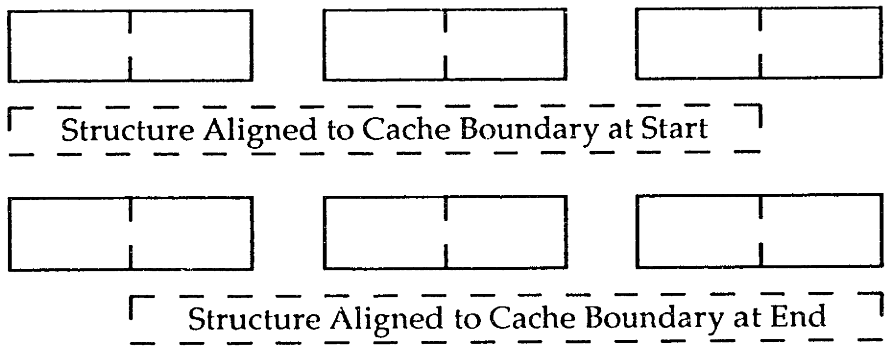

Structure Aligned to Cache Boundary at Start
Structure Aligned to Cache Boundary at End

Figure 5-3: Block Alignments of 5 word Data Structure

The algorithm below produces a minimum set of prefetch addresses to cover a dense interval of addresses characterized by a lower bound address, LB, an upper bound address UB and given a cache block size of b. It uses a series of addresses starting at LB and spaced by b, terminating when a component of the series exceeds UB. If the series does not include UB then an additional prefetch for UB is added. This algorithm makes no assumptions about the alignment of LB with respect to cache block boundaries. If LB is known to be cache block aligned then the final prefetch to UB in the algorithm below can be eliminated if the loop terminates with address not equal to UB.

```
address = LB
while (address < UB)
  prefetch address
  address = address + b
prefetch UB
```

Optimization of prefetches for dense address regions is employed by c-flat in linked-list RAAD prefetching. Whenever the number of distinct badref instructions associated with a data structure exceeds the number of prefetches required to cover the address interval associated with those references prefetches are generated based on the algorithm above. Array and indirection array RAADs could also benefit from this technique when array elements are aggregate data structures rather than scalar values but c-flat does not currently attempt to apply this optimization to these cases.

The second form of cache block optimization can be applied when references satisfied by a single cache block arise from successive executions of a single badref instruction in a loop. In order to avoid overhead, it would be desirable to omit prefetch instructions from some loop iterations. This can be accomplished if the loop is unrolled. If the stride of a reference is less than half the cache block size, overhead can be avoided by unrolling. Compute an integer i as the cache block size divided by the reference stride, rounded down. After i-way unrolling is applied to the loop only a single prefetch is required per iteration to cover all misses.

One must be careful in choosing the address for the remaining prefetch, however. Consider the example below. The stride is 4, the cache block size is 8 and the original prefetch iteration count is 2. The first code fragment shows an array reference, assumed to be a badref. Code resulting from application of the normal array RAAD schema is adjacent to the original.

```
...                              ...
ld [rp], rt   (Load 1)           ld [rp],rt       (Load 1)
add rp,4,rp                      pref [rp+8],rt   (Pref 1)
...                              add rp,4,rp
...                              ...
```

Since the block size is at least half the stride the loop unrolling technique can be applied, resulting in the following code.

```
...                              ...
ld [rp], rt   (Load 1a)          ld [rp],rt       (Load 1a)
add rp,4,rp                      pref [rp+8],rt   (Pref 1a)
...                              add rp,4,rp
...                              ...
ld [rp], rt   (Load 1b)          ld [rp],rt       (Load 1b)
add rp,4,rp                              (Pref 1b omitted)
...                              add rp,4,rp
```

Under ideal circumstances, specifically when the address used by `Load 1` is aligned to a cache block boundary in the initial loop iteration, the code works as desired. In this case in the unrolled code the reference `Load 1a` is a badref and `Load 1b` is a goodref. `Pref 1a` accesses addresses needed by successive occurances of badref `Load 1a`.

Under improper alignment the code does not achieve the desired performance result. If the address used by `Load 1` in the initial loop iteration is not cache block aligned then in the unrolled code `Load 1b` rather than `Load 1a` is the resulting badref. `Pref 1a` still serves to initiate cache misses but data from these blocks is now needed by `Load 1b` rather than `Load 1a`. As a consequence, the original degree of latency tolerance desired is not achieved.

This problem can be avoided by situating prefetch instructions adjacent to the first corresponding memory reference in the unrolled loop but increasing the PIC value used for the prefetch. If the PIC value is increased by $i-1$, one less than the unrolling factor, then the originally desired level of latency tolerance is maintained in spite of potential cache block misalignment. This can be viewed equivalently as using a prefetch address corresponding to the final unrolled prefetch, `Pref 1b` in the example above, but situating the prefetch at the site of the first unrolled prefetch, `Pref 1a` above.

While the technique above can be applied regardless of whether the stride of the badref is a factor of the cache block size, a more complicated technique can be applied to further eliminate overhead if the stride is not a factor of the block size and can be used whenever the stride is smaller than the block size.

Identify the maximum common factor of the stride and block size. Denote this by f. Compute a value i by dividing the block size by f. Thus for instance, with a stride of 6 and a block size of 8, i would be 4 since 6 and 8 have a common factor of 2. Unroll the loop i ways. The number of distinct cache blocks accessed during an iteration of the unrolled loop can be computed as the stride divided by f. This is the number of prefetches which should be inserted into an iteration of the unrolled loop.

Notice that in unrolling a loop a single memory reference instruction generating a dense set of addresses is transformed into an ensemble of distinct instructions generating these addresses. Treatment of groups of references in unrolled loops is similar to the treatment of groups of references associated with structures described previously. One relevant difference is that if unrolling as described above is applied, references maintain a fixed relationship with respect to cache block boundaries. As a consequence, the final upper bound prefetch used in the structure algorithm can be omitted. As with the first loop unrolling technique the prefetch iteration count must be conservatively adjusted to account for initial misalignment with respect to cache block boundaries.

Neither of these unrolling optimizations is implemented in c-flat.

5.3.7 Prologue Prefetching

RAAD prefetching provides latency tolerance for misses occuring during the steady state execution of loops. Additional transient misses are incurred during the first iteration or iterations. Two main sources of transient misses exist. Instructions which access values with loop constant addresses may produce a miss in the first loop iteration (or the first iteration in which the instruction is executed if the loop has conditional behavior). After the initial miss, subsequent references result in hits. The second form of transient misses occur because prefetch instructions for RAADs use addresses for future iterations with addresses advanced by the prefetch iteration count. Occurances of badrefs in the first iteration have no corresponding prefetch. In fact no prefetches occur for any badref instances in the first PIC iterations. For both of these classes of misses latencies can be overlapped by adding loop prologue code with extra prefetches.

References with loop constant addresses can be identified relatively easily using dataflow analysis. Within the framework of c-flat these references have an address computation which does not have any SCCs. Based on measured reference miss rates and average loop iteration counts it is possible to determine fairly accurately whether a loop constant address experiences initial iteration misses. If the miss rate is approximately equal to the inverse of the average traversal count of the corresponding loop a miss occuring on each initial iteration is the likely cause of this behavior.

Prologue prefetching for references with loop-constant addresses can be beneficially applied even if the reference has a relatively low miss rate. The cost of prefetching is not incurred on each execution of the reference. It is incurred only once each time the loop is executed which is equivalent to the expected frequency of the corresponding miss. Software overhead instructions associated with prefetching have been localized to a region of code statically correlated with the occurance of misses. In this case the correlation is with a region of code not actually including the memory reference producing the miss.

Loop-based prefetching for RAADs can be viewed as constructing a software pipeline for memory miss transactions. Software pipelining is a technique which exploits parallelism between successive loop iterations to produce code which is tolerant to high latencies of individual instructions. Operations logically associated with a single loop iteration are spread across several iterations. This decouples the bandwidth of initiating loop iterations from the latency of the loop. Normally, when applying software pipelining to a loop, prologue and epilogue code are required to fill and drain the software pipeline. Prologue and epilogue refer to added code produced by peeling loop iterations from the beginning and end of the loop. In the software pipeline created by RAAD prefetching the only code which has migrated across loop boundaries is prefetch code. Since this code is not semantically signficant and is safe for speculative execution neither prologue nor epilogue code are strictly necessary. Transient misses associated with filling the software pipeline result when prologue code is omitted. Generating appropriate prologue code avoids these misses. Prologue code consists of one or more prefetch instructions for each RAAD badref. The number required is equal to the prefetch iteration count. For array RAADs with small strides these addresses form a dense address region which may be amenable to the optimization technique described earlier. Epilogue code would consist of one or more loop iterations with no prefetch instructions.

C-flat can generate prologue code for both constant address references and software-pipelined RAAD references. An additional form of prologue code is sometimes added for indirection-array RAADs and linked-list RAADs. In the prefetch schemas associated with each of these reference types new variables are introduced into loops. Sometimes the use of these variables within a loop precedes the spot at which they are set. When this happens initialization code for these variables is also included in the prologue. C-flat does not produce epilogue code. The consequence of this is that the final iterations of the loop perform prefetch operations for potentially unused cache blocks.

5.4. RAAD Prefetching in Conditional Loops

Loops with conditionals may exhibit more complex memory reference behavior than simple loops and as a consequence can often benefit from more sophisticated heuristic techniques. Conditional behavior complicates RAAD prefetching strategy when either badref RAADs or the induction variables used to compute their addresses are treated asymmetrically on different paths through the loop. This section describes modifications to the basic loop prefetching schemas introduced above which are applicable under a variety of circumstances which may arise in loops with conditionals.

Consider the following loop.

```
for (i=0; i<N; i++) {
  if (x[i] > xmid)
     y[i] = x[i];
  }
```

Assuming that **x** and **y** are not initially cache resident, the reference to **x** in the conditional predicate and the reference to **y** are both badref array RAADs. Direct application of the array RAAD schema described above to each badref produces the following code.

```
for (i=0; i<N; i++) {
  t = x[i];
  prefetch(x[i+PIC]);
  if (t < xmid) {
     y[i] = x[i];
     prefetch(y[i+PIC]); }
  }
```

The syntax `prefetch(variable reference)` is intended to indicate code producing a prefetch reference using the address of the argument suppied. `PIC` is the prefetch iteration count chosen for the loop.

Suppose that the conditional is true half of the time and that the chosen value of `PIC` is 1. The prefetch for `x[i+PIC]` behaves as desired because the badref reference to `x[i]` appears symmetrically on all paths through the loop. Two extremes for the behavior of the prefetch for `y[i+PIC]` can be identified. It is possible that the iterations in which the predicate is true occur as a dense set, for instance the first `N/2` iterations. If this is true then each reference to `y[i]` has a corresponding prefetch. Furthermore, the software overhead of prefetching is only incurred on the first `N/2` iterations. Alternatively, it is possible that the predicate is true in alternating iterations, only the even numbered iterations for instance. In this case every single reference to `y[i]` may still result in a miss despite the addition of prefetches. Prefetches only occur in even iterations using prefetch addresses targetting references in the odd iterations. Prefetches are never performed targetting the y values which are actually referenced since these prefetches would need to occur in odd iterations and are suppressed due to the conditional behavior. Choosing `PIC` to be 2 rather than 1 has optimal performance in the case where the predicate is true on alternating iterations. No `PIC` value solves the problem in general, however. For each value, the pattern consisting of alternating sequences of `PIC` true values followed by PIC false values incurs the cost of prefetching while deriving no performance benefit from the added instructions.

Consider another example.

```
while (i<N) {
     z[i] = x[i]+y[i];
     if (z[i] > zmid)
       i += 1;
     else
       i += 2;
  }
```

References in the first line of the loop body are badref array RAADs and they occur symmetrically on all paths through the loop. In this case the stride of these references is not a loop constant. Their induction variable `i` and its corresponding SCC involve conditional behavior.

The result of applying the array RAAD prefetching schema is indicated below.

```
while (i<N) {
  t1 = x[i];
  prefetch(x[i+offset]);
  t2 = y[i];
  prefetch(y[i+offset]);
  z[i] = t1 + t2;
  prefetch(z[i+offset]);
  if (z[i] > zmid)
    i += 1;
  else
    i += 2;
  }
```

In this situation, the problem associated with applying prefetching is how to choose a value for the prefetch offsets. Consider an alternating pattern of updates. The indices of array values referenced are a sequence with every third value missing, such as 0,1, 3,4, 6,7, ... Consider the result of prefetching using an offset chosen assuming a constant stride of 1 with a prefetch iteration count of 1. Prefetches are generated for references 1 greater than the numbers in the sequence above, *i.e.* 1,2, 4,5, 7,8, ... Half of the prefetches generated occur for unused index values while half of the used indices go unprefetched. The successful prefetches occur when the stride has been correctly predicted. Consider the alternative of assuming a stride of 2. One might hope for better performance in this case since given a string of references with stride 1 using a prefetch stride of 2 is equivalent to using a prefetch iteration count of 2. Analysis of the sequence of addresses produced again shows that half of the prefetches target unused addresses while half of the addresses go unprefetched. One might instead try an adaptive strategy, in which the offset used corresponds to the most recent stride. This strategy turns out to have even worse performance, targetting unused addresses with all prefetches. For the particular pattern analyzed an adaptive strategy which chooses as an offset the alternative to the current update turns out to be optimal. This strategy works because it always predicts the next update correctly. In general, without some knowledge of the pattern in which updates will occur an optimal pattern of prefetches cannot be determined.

The two examples above illustrate potential problems associated with direct application of simple prefetch schemas to loops with conditionally executed badrefs or conditionally updated induction variables. Situations involving more conditional paths can result in even more complicated behavior arising from a mixture of the two cases or the repeated occurance of one or the other. The next section describes a unifying observation which can help to unravel conditional behavior and produce heuristic prefetch schemas with reasonable efficiency and likelihood of success. Prefetch schemas for a variety of conditional patterns are presented, along with heuristic evaluation criteria to determine whether prefetch code, if added, would help or hurt performance.

5.4.1 Conditional Strategy

Misses for RAAD badrefs within loops occur as a result of the intersection of two distinct events. One of these events is the badref memory reference itself. The second is the modification of one or more induction variables used to calculate the badref address. Without the first, no reference occurs and thus no miss. Without the second an old address is reused. Strategies for RAAD prefetching in conditional loops can exploit this observation by focusing on misses and the situations in which both events necessary to produce misses can occur in conjunction.

A second relevant observation to conditional strategy is that prefetching overhead can be minimized by adding prefetch code in the place where it is most highly correlated to the specific misses for which it is targetted. It was observed in section 5.3.7 that constant address references within loops which only miss on the first loop iteration could be prefetched with acceptable overhead, even if the references have low miss rates, by situating prefetch code in a loop prologue. This prologue is correlated on a one-to-one basis with execution of the first iteration of the loop and thus the occurance of the misses. For badref RAADs in loops without conditionals all loop iterations contain both badref instructions and the induction variable updates which change their addresses. Every loop iteration is correlated to the intersection of events necessary to produce a miss so prefetch code can be added anywhere within the loop. In the non-conditional schemas it is added adjacent to the badrefs since this often results in the minimum amount of overhead. In conditional loops it is often possible to identify regions of code containing either a badref memory reference or more commonly an induction variable update which are more highly correlated to the future misses of a particular badref than other regions of code. When possible, prefetch code should be added in these regions. When choosing between two regions which are equally correlated, prefetches should be added to the code which is executed more frequently since this results in more successful prefetches.

5.4.2 Badref and IV Update on a Common Conditional Path

The simplest conditional construct to handle occurs when a badref and all of its associated induction variable updates occur on a single common path or set of paths through a loop. The example below illustrates this case.

```
while (1) {
  if (pred)
     a = *p++;
  else
     a = b;
  ...
}
```

Assume that dereferencing p produces a badref and that pred represents some loop-varying predicate. The two events required for a RAAD miss, a memory reference and an induction variable update, occur in the same path. Notice that these events do not occur in the same loop iteration. The induction variable is modified in an iteration when pred is true and the memory reference incurring the associated miss occurs in the next iteration in which pred is true again. Despite the fact that these events do not occur in the same iteration, since they occur in the same region of code this region is correlated to the potential misses produced by each IV update. As a consequence, application of the prefetch schema for unconditional behavior produces appropriate code, illustrated below.

```
while (1) {
  if (pred) {
    a = *p;
    prefetch(*(p+PIC));
    p++;
    }
  else
    a = b;
         ...
}
```

Prefetch code has been added in a location which is perfectly correlated with potential future misses. As in the unconditional case, overhead is incurred for each execution of the badref instruction while benefit accrues for each miss. The test determining whether such a prefetch is beneficial is equivalent to that for the unconditional case. If O is the number of overhead instructions which must be added to produce a prefetch, *mlat* is the memory latency parameter and m is the miss rate of the badref, the prefetch improves performance when the following relation is true.

$$O * 1/mlat < m \qquad (5-5)$$

5.4.3 Conditionally Executed Badref with Unconditional IV Update

This case corresponds to the situation illustrated in the first example in the section. A badref is situated on a conditional path but induction variables used in its address computation are updated every iteration of the loop. Another example of this case is repeated below.

```
while (1) {
  if (pred)
     a = *p;
  p++;
}
```

Assume that the dereference of p is a badref. Induction variable updates incrementing p occur on every iteration. The intersection of an update and reference occurs whenever the predicate is true. The miss associated with this intersection is the one for the current execution of the badref. RAAD prefetching is concerned with misses for future iterations, not current iterations. Without knowledge of the behavior of successive predicate values one might choose to model successive predicate values as being independent of each other. Assuming independence of predicate values across iterations all regions of code are equally correlated to future misses.

Assume that the probability of the predicate being true on any iteration is P_{pred} and that successive predicate values are indeed independent. If a prefetch is added for some future iteration it will be beneficial when the predicate is true in the target iteration, occurring with probability P_{pred}, and the badref instance in that iteration produces a miss, occuring with probability m. The expected benefit of such a prefetch is

$$\mathrm{E}[Benefit] = mlat * P_{pred} * m \qquad (5-6)$$

For a prefetch requiring O cycles of software overhead, benefit outweighs cost when the following relationship holds.

$$O * 1/mlat * 1/P_{pred} < m \qquad (5-7)$$

Prefetch instructions can be added in one of three code regions. They can be added to one of the arms of the conditional or to the code outside the conditional. A prefetch is only added when it is expected to be beneficial based on the formula above. The aggregate benefit of a prefetch for all iterations depends on the product of the average benefit per execution and the total number of times the prefetch is executed. If overheads are equal under the three possible placements the prefetch should be placed where it is executed most frequently, *i.e.* in an uncondtionally executed region of the code. C-flat achieves this placement by adding the prefetch after the induction variable update. If overhead varies with different prefetch placement choices, an aggregate cost benefit analysis can be used to determine where to place prefetches.

```
while (1) {
  if (pred)
     a = *p;
  p++;
  prefetch(*(p+PIC-1));
}
```

Code with an unconditionally executed prefetch appears above. Since the prefetch occurs after update of the corresponding induction variable the desired prefetch iteration count is modified to account for this.

As indicated by Equation 5-7, under various forms of conditional behavior, the benefit of prefetching may be dependent on the frequency of execution of various code blocks. C-flat uses basic block execution profiles, gathered simultaneously with memory reference behavioral profiles, to estimate the probabilities in heuristic tests such as this.

5.4.4 Conditionally Updated IV: Single Update

A conditionally updated IV with a single update refers to an induction variable which is updated identically on one or more loop paths and not updated on others. The code below illustrates a badref with an address computed from a single update conditional IV.

```
while (1) {
  a = *p;
  if (pred)
    p++;
}
```

Induction variable updates occur less frequently than memory references. Each IV update results in a potential miss. Memory references in iterations not preceded by an update cannot produce misses. Thus future misses are more highly correlated with induction variable updates than with the memory references themselves. As a consequence it is advantageous to add prefetch code within the conditional in which the induction variable is updated.

```
while (1) {
  a = *p;
  if (pred) {
    p++;
    prefetch(*(p+PIC-1));
  }
}
```

In the code above, each miss associated with dereferencing p has a corresponding prefetch. The aggregate benefit of the transformation is the product of the memory latency and total number of misses incurred by the badref. The cost of prefetching is only incurred on iterations in which an update occurs. Assume this represents a fraction of iterations P_{pred}. Prefetching is benefical when the benefit exceeds the cost or

$$M * mlat > E * O * P_{pred} \qquad (5-8)$$

where M is the aggregate number of misses associated with the badref and E is the aggregate number of loop iterations. Since M divided by E is the miss rate of the badref the relation above is equivalent to the following relation.

$$O * 1/mlat * P_{pred} < m \qquad (5-9)$$

Notice that this relation, corresponding to unconditional badref execution and conditional induction variable update, allows beneficial prefetching for references with lower miss rates than its unconditional equivalent. In contrast, when a badref reference occurs conditionally and its induction variable update occurs unconditionally a higher miss rate is required than the unconditional case.

5.4.5 Conditionally Updated IV: Multiple Updates

Conditionally updated induction variables with multiple updates are IVs with more than one distinct non-trivial update, *i.e.* more than one update in which the next IV value differs from the current value. The second example analyzed at the beginning of this section contains a conditional IV with multiple updates, as does the example shown below.

```
while (1) {
  a = *p;
  if (pred)
    p++;
  else
    p += 2;
}
```

In code with multiple IV updates a prefetch address calculated using some assumed stride may not correspond to an address which is ever used in a memory reference. This can occur when the actual update differs from the assumed update. The expected benefit of prefetches can be discounted to reflect the possibility that an incorrect assumption has been used.

Assume in the above code that the predicate is true with probability P_{pred}. If a prefetch with an iteration count of 1 is added using an offset computed based on the assumption that the predicate is true this prefetch has a probability of at least P_{pred} of targetting an address which is used. Similarly, if a prefetch is generated based on the assumption that **pred** is false, this prefetch has a probability of at least $1 - P_{pred}$ of targetting an address which is used. The benefit of adding each of these potential prefetches can be calculated independently and either one, none or both can be added to the code.

Unconditionally executed prefetches for unconditionally executed memory references with conditional IV behavior can be evaluated using the following heuristic test.

$$O * 1/mlat < m * P_{upd} \qquad (5-10)$$

Here P_{upd} refers to the probability that the update assumed in the prefetch offset will be used in any given loop iteration.

For a suitable value of P_{pred}, it might be determined that prefetches based on both offsets could profitably be added to the code in the example above. This would produce the resulting code.

```
while (1) {
  a = *p;
  prefetch(*(p+1));
  prefetch(*(p+2));
  if (pred)
    p++;
  else
    p += 2;
}
```

The analysis described above is not exact. Cache blocks larger than the unit of data accessed by memory references may allow prefetches to be beneficial even if they do not directly target an accessed memory location. A prefetch is beneficial whenever it targets a non-resident cache block which contains an accessed memory location. As a consequence of this phenomenon, if strides based on all possible updates are less than or equal to the cache block size, use of a single uniform stride equal to the maximum stride is a suitable strategy. If strides based on some IV updates exceed the cache block size while strides based on other updates are smaller, ignoring the effect of cache block size leads to a conservative estimate of the benefit of any individual prefetch. Additionally, given multiple updates, if a smaller update stride is a factor of a larger update stride, prefetches based on the larger update may eliminate misses for addresses arising from a sequence of the smaller updates. In this case the individual benefit of the larger update is underestimated. In the formula above the aggregate benefit of avoiding latency for all misses is apportioned among the individual prefetch options. If prefetches for all updates are added the aggregate benefit is accurately characterized even though the benefit of any single prefetch is underestimated.

The simplified model also misrepresents prefetching cost when strides exceed the cache block size and loops are bandwidth limited. In such cases, adding multiple prefetches increases the bandwidth requirements of loops, impacting performance.

In c-flat these perturbations are ignored. In the rare case of IVs with multiple updates prefetches for each offset are individually evaluated. Based on the test above they are either added to the code or ignored.

Equation 5-10 above applies when the prefetch iteration count is 1. If a prefetch iteration count greater than 1 is desired, assumptions must be made regarding the likelihood of a sequence of updates. Rather than deal with the complexity associated with prefetch iteration counts exceeding one, when multiple update IVs exist c-flat limits any such loop to have a prefetch iteration count of 1.

As a final note, the formulas above can be modified slightly when loop paths exist on which a multiply updated IV is not changed. Adding prefetches at update sites avoids overhead on non-update paths. This scales the cost of prefetching down by the probability that an iteration updates the IV, which is the sum of the probabilities of all update paths.

$$O * 1/mlat * (\sum_{allupdates} P_{upd\ i}) < m * P_{upd} \qquad (5-11)$$

In the formula above, P_{upd} is the probability of occurance of some particular update being considered as a prefetch offset. $\sum P_{upd\ i}$ represents the sum of the probabilities of each distinct, non-trivial update.

5.5. Loop Dependence Analysis

Addition of prefetch code based on a fault-safe prefetch instruction is not a semantically significant program transformation. It does not have the potential to change the results of programs, only the performance. Any compiler optimization worth its salt requires some form of dependence analysis and latency tolerance optimization using explicit prefetch instructions is no exception. The goal of dependence analysis for prefetch generation is to produce a good estimate of the values of addresses for badref memory references. Because prefetch transformations are semantically safe, this estimate does not need to be exact or even conservative. In fact, an approximate analysis algorithm which ignores unlikely behavior is better than a conservative algorithm which models all possible behavior. Unlikely behavior does not significantly impact average performance.

Prefetch dependence analysis must characterize the likely values of badref addresses. Dependencies should be modelled in the address estimate when they are likely or certain to exist. They should be ignored when they are possible but unlikely. In c-flat this value-based characterization takes the form of an instruction level dataflow graph. The graph primarily reflects scalar flow dependencies.

It does not include information about non-value dependencies such as antidependencies or output dependencies. Additionally, loop carried flow dependencies for non-scalars are not modelled. Based on an empirical analysis of code one observes that scalar dependencies largely account for the address values of most memory references. In the case of indirection array RAADs it is possible that ignoring loop-carried non-scalar dependencies might sometimes lead to inaccurate characterization of addresses. This would occur when a loop carried flow dependence with a non-zero dependence distance for the target loop existed for an array reference used in an indirection array address. This dependence distance would also need to be less than or equal to the prefetch iteration count for the loop. Under these circumstances, an indirection array element used in a prefetch address computation could change between the prefetch and eventual reference. Were this to happen, an unneeded block would be moved into the cache and a miss would go unprefetched. This combination of circumstances seems relatively unlikely. (Terms used in this paragraph are defined in references on non-scalar dependence analysis such as [61].)

In GCC and thus c-flat, programs are represented in a sequential format in which scalar dependencies are carried through explicit temporary names. For prefetch generation, this format is transformed into a dataflow graph in which dependencies are indicated with graph edges. The transformation is accomplished by performing a sequential scan through the code maintaining a table of the most recent graph node to have changed a named value. Any use of a named value produces a graph edge from the node in the appropriate table location to the node using the value. Any internal loops are modelled within this graph as single nodes which modify all scalar values written in the internal loop in an unpredictable way. A two-pass process is used in order to capture loop-carried scalar dependencies leading to address SCCs. Traversal procedures which identify the addresses of badrefs start at the badref and walk backwards through the dataflow graph identifying nodes participating in the address computation. As a consequence, the actual graph generated corresponds to the transpose of the dataflow graph, permitting easy traversal in this backward direction. Such a graph is generated for each loop in a program.

When a loop contains conditionals which rejoin the loop more than one alternative path exists through the loop. These alternate paths are modelled in a technique somewhat akin to trace scheduling. Based on basic block profiling statistics a principle trace through the loop is identified. Graph structure is built for this trace. At each control fork which rejoins the loop the tables used in graph generation are saved. The saved tables are used to generate graph structure for the divergent traces. When these traces rejoin the principle path no further structure is generated and in contrast to trace based analysis for semantically significant transformations, dependencies are not recorded across the rejoins.

Since dependencies are not explicitly represented across rejoins induction variable updates which are not on the principle trace are not reflected in the resulting dataflow graph. In a subsequent pass, the graph is modified to reflect this information when deemed useful. Strongly connected components are identified on the principle trace and on other traces. These SCCs are used to construct a table of induction variables. When SCCs on different traces update the same induction variable, either resulting in equivalent or different updates, this information is maintained in the table. Finally, the graph is modified when the principle trace has a reference to an induction variable which does not have an SCC on the principle trace but does have an SCC on some side trace.

Heuristic prefetching techniques described above are applied based on the resulting trace-based dataflow graph and induction variable tables. Address estimates based on this information are not exact. Approximations result from inexact modelling of non-scalar memory references and ignoring dependencies at join points between side traces and the principle trace. Despite these approximations, the analysis provides an adequate basis for loop-based prefetch optimization.

5.6. Sequential Miss Scheduling

Loop-based RAAD prefetching is the frontline technique in c-flat used for latency tolerance transformation. For a variety of reasons, loop-based prefetching may be inapplicable to the badrefs in some region of code. When loop-based prefetching cannot be beneficially applied, an alternative prefetching approach which attempts to schedule prefetch instructions for misses in sequences of code is adopted. This technique will be referred to as sequential miss scheduling or simply miss scheduling.

The most common reason that loop-based RAAD prefetching cannot be applied to a region of code is that no loop can be identified. While most badrefs which can be profitably optimized with prefetching are revisited through some form of looping behavior, not all control constructs which lead to looping behavior are easily identifiable. Code revisitation may arise through procedure calls inside of loops in different procedures, through various forms of self or mutually recursive procedure call patterns or through the use of goto-type control constructs in various Byzantine patterns. c-flat does no interprocedural analysis for loop identification. If the process by which code is revisited is not reflected within the procedure containing the code then loop heuristics cannot be applied. Loops arising from complicated patterns of gotos may not be recognized, and again, loop heuristics cannot be applied.

Sequential miss scheduling provides a fallback prefetching technique. In sequential miss scheduling prefetch instructions are added to a sequence of code for each contained badref. If miss-invoking prefetches can be scheduled in the code to precede their corresponding badrefs by a distance exceeding the memory latency then latency tolerance is achieved. RAAD prefetching can be viewed as a variant of sequential miss scheduling specialized to be able to move the misses for occurances of badrefs across loop iteration boundaries. When loops are not identified instead of performing miss scheduling across loop iterations it is performed on linear sequences of code.

The simplest scheduling techniques do not move instructions across basic block boundaries. If sequential miss scheduling were required to schedule prefetch code in the same block as a corresponding badref instruction maximum latency tolerance would be limited by basic block sizes. As seen in Chapter 4, it is frequently the case that insufficient parallelism exists in basic blocks for tolerance to substantial miss latencies. Cross block scheduling of prefetch code is essential for tolerance to high memory latencies. Cross block miss scheduling is accomplished in c-flat using a variant of trace scheduling. In trace scheduling, non-repeating sequences of basic blocks which are frequently executed as a unit are identifed. Trace scheduling takes its name from these sequences which are referred to as traces. Prefetch instructions are added into a trace for each badref encountered. Prefetches can be added anywhere on the trace, subject to a few constraints, in order to separate misses adequately from their badrefs. Latency tolerance can be achieved when traces can be constructed which are long in comparison to the memory latency.

Trace scheduling compilers such as that described in [6] utilize sophisticated trace selection heuristics. Since trace-based miss scheduling is not the primary latency tolerance technique in c-flat a simple trace selection method is chosen. A set of basic blocks is identified for miss scheduling. Such a set might include an entire procedure containing no explicit loops or non-loop regions of code surrounding outer loops. Blocks from the set are ranked based on basic block profiling statistics. Traces are generated by choosing a starting point and adding one block at a time to the trace by choosing the highest ranked successor to the last block which is in the set of candidate blocks. Blocks are removed from the set as they are added to a trace. Traces are started for blocks with no predecessors in the initial set or blocks which are not added to the traces containing their predecessors. If loops exist the blocks in these loops are not included in traces with non-loop blocks. Resulting traces can be viewed as large basic blocks, long sets of instructions likely to occur in sequence. Traces are processed with the dataflow graph generation algorithm used in RAAD prefetching. Trace dataflow graphs are used to determine how address values are computed.

In conjunction with trace generation, dominator information as defined in [3], is computed for the basic blocks submitted for miss scheduling.

5.6.1 Sequential Miss Scheduling Algorithm

Miss scheduling for sequential code can be modelled as a process of merging two streams of instructions. One stream contains the original instructions. The second stream contains a prefetch instruction for each badref in the original sequence in the order of appearance of the badrefs. If one assumes that prefetches occur sufficiently soon in the final schedule with respect to their corresponding badref instructions then each instruction in the original sequence can be modelled as consuming a single cycle. Each prefetch instruction in the prefetch stream can be modelled as producing a miss. Misses can be processed by the memory system with some fixed bandwidth. In c-flat this bandwidth is parameterized with an intermiss submission latency, *mbwlat*. Each miss produced by the prefetch stream consumes *mbwlat* cycles. If these two streams were independent an optimal schedule could be produced by merging the streams in there original order using the following algorithm. Form the output stream by adding one prefetch from the prefetch stream to produce a miss followed by $mbwlat - 1$ instructions from the original stream to delay until the next miss can be processed by the memory.

If the assumptions in the model above were always true the scheduling technique described would be optimal. Several deviations from these assumptions which occur in practice are listed below.

1. The prefetch stream and original instruction stream are not independent. They interact through the addresses required for prefetches and the need for prefetches to precede their badrefs.
2. The original instruction stream is a trace, as opposed to a single sequence of code. Thus the execution frequencies of instructions may vary within the original instruction stream.
3. All prefetches do not produce misses.
4. Bandwidth of real memory systems may not be accurately described by intersubmission latencies.

The scheduling technique used in c-flat is based roughly on the merging algorithm described above. It takes a stream of prefetches, one for each badref in a trace, and merges them into the original instruction stream of the trace with a spacing chosen based on memory bandwidth. It obeys constraints imposed by the interaction of prefetch and instruction streams while permitting only a limited reordering in each stream. It is by no means optimal in general. In practice, due to the fact that address computations tend to be fairly simple, interactions between the streams are limited. As a consequence, the algorithm seems to perform reasonably well.

5.6.1.1 Address Interactions

Departures from the basic merge strategy occur due to interactions between the prefetch and instruction streams. The first source of interaction arises because prefetches need addresses which are computed based on values formed by the instruction stream. C-flat uses a demand-driven scheduling approach which moves computation required for address generation to the location chosen for a prefetch. Instructions can be moved across block boundaries using two different techniques which will be referred to as scheduling and copying. When instructions are moved across a basic block boundary using scheduling the instructions are eliminated from their original basic block and added without modification to the new basic block. When they are moved by copying, new temporary storage is allocated for all temporary values produced. Copies of the original instructions which have been modified to use the new temporary locations are added to the target block. The code in the original basic block is left unchanged. In order to schedule a prefetch instruction, all computation required for its address must be moved to the target site. Address computation is identified by a traversal of the trace dataflow graph, starting at the badref. This traversal identifies needed instructions which are moved either by scheduling or copying.

Moving code by scheduling is a semantically significant operation. Copying, since it avoids modification of original program state, is not semantically significant. While copying can be applied whenever it is deemed beneficial, scheduling can only be applied based on conservative dependence analysis. Within a basic block the dataflow graph generated by c-flat for miss scheduling reflects all necessary dependencies. Dependencies across blocks are not suitably modelled. Additional constraints are imposed by the miss scheduler in order to ensure that scheduling is applied only when it is semantically legitimate to do so. Non-scalar or non-local data values are distinguished from scalar local variables, arguments and compiler generated temporary values within GCC. In the constraints below the term memory reference refers to values which are non-scalar or not procedure local. The following constraints are applied in order to ensure that scheduling does not change program semantics.

1. Memory store operations are never moved, either by scheduling or copying.
2. Memory loads are not moved across memory stores by scheduling. C-flat has no mechanism to disambiguate these operations to verify their legitimacy.
3. Memory loads are only moved across memory stores by copying when a dependence is deemed unlikely. Heuristics based on address computations determine when a dependence is likely or unlikely.
4. Instructions are only scheduled into blocks which dominate their original block.
5. Instructions are only scheduled if all instructions upon which they transitively depend are scheduled, as opposed to copied.
6. Instructions are only scheduled if they are the only instruction which produces a particular named value.
7. Instructions are only scheduled if the value they produce is not live at the beginning of their basic block.

The first three rules constrain the motion of undisambiguated values. The latter four insure that flow dependencies, antidependencies and output dependencies are satisfied for code motions involving disambiguated values. Given the unsophisticated dependence analysis employed in generating dataflow graphs, these added constraints guarantee that code motion using scheduling rather than copying is valid.

Since memory loads are not moved across some memory stores and memory stores are never moved it may be impossible to generate the address needed for some prefetch at its target location. The prefetch, then, cannot be scheduled at this target. When this occurs the prefetch is added to a queue and reconsidered at a later time when it may be eligible for scheduling. The queue is associated with the memory store preventing scheduling. After the store is scheduled elements of the queue are reconsidered.

Much as in RAAD prefetching, the benefit and cost of a particular prefetch can be assessed. The cost can be assessed at 1 cycle for the prefetch instruction and additional cycles for any instructions which must be copied, as opposed to being scheduled. It is possible that prefetch and address code may be moved into a block executed more frequently than the block containing the corresponding badref. In this case, the cost can be scaled appropriately based on the relative execution frequencies of the blocks. Consider moving prefetch code consisting of S scheduled address instructions, C copied address instructions and 1 prefetch instruction into a block executed with frequency f_{targ} from a block with frequency f_{src}. Assume the badref in the source block has a miss rate of m. If f_{targ} is at least as large as f_{src}, the prefetch is beneficial when the relation below is true.

$$(f_{targ} * (1 + C) + (f_{targ} - f_{src}) * S) * 1/mlat < m \qquad (5-12)$$

If a prefetch fails the heuristic test above it is queued for reconsideration at a point in the trace corresponding to a different basic block. In a different block the prefetching cost may be lessened since C, S and f_{targ} may all change.

5.6.1.2 Miss Latency Interactions

The second form of interaction between the prefetch and instruction streams will be termed a miss latency interaction. Prefetches are coupled to their corresponding badrefs by a constraint on the minimum separation which should occur between the two instructions. Badrefs are ideally scheduled at least *mlat* cycles after prefetches. The merge scheduling procedure described above exploits reverse parallelism as defined in Chapter 4 to produce this separation. If insufficient latency tolerance is achieved, indicated by separation of less than *mlat* between a prefetch and badref, instruction stream reordering can be used to delay the badref. This reordering exploits forward parallelism.

In c-flat the original instruction stream is not reordered. When a badref occurs too soon after a prefetch a series of additional prefetches are scheduled to fill available memory bandwidth and then the badref is scheduled, incurring any remaining latency penalty. Since instruction rescheduling to exploit forward parallelism is not performed the code generated by c-flat does not achieve maximal latency tolerance. The scheduling algorithm does achieve good memory bandwidth utilization, however, since it fills any potential bandwidth usage gaps which might occur in conjunction with latency stalls.

5.7. Choice of RAAD Prefetching or Miss Scheduling for Loops

Under several circumstances, even if an explicit loop can be identified, application of miss scheduling rather than RAAD prefetching may result in better performance. These circumstances involve loops with a large number of badrefs.

In section 5.3.5 it was observed that under the prefetching model assumed in c-flat the cache performs two functions, caching and buffering. For loops in which a single iteration produces a large number of misses these functions may interfere with one another. When this occurs, RAAD prefetching is likely to be ineffective at best and detrimental at worst.

Consider a loop which contains an inner loop nest which produces more misses than the total cache size. Applying RAAD prefetching to badrefs in the outer loop will be ineffective since the prefetched values will be replaced as a result of misses in the inner loops before they can be beneficially used. In this case, prefetching has added software overhead but produced no latency tolerance benefits. Potentially even more harmful, it has increased the net memory bandwidth requirements of the program by moving unused information into the cache. In the situation described the negative impact of outer loop prefetching is likely to be small since the inner loops that replace prefetched data probably account for the bulk of computation time. Nonetheless, there is no reason to apply "optimizations" to programs that detract from their behavior if this can be avoided. RAAD prefetching can be disabled with a heuristic test to avoid this behavior.

The loop described above can potentially benefit from miss scheduling. Given a sequential trace of basic blocks starting at the end of the inner loop nest, proceeding across the loopback to the top of the outer loop and terminating at the beginning of the inner loop nest, any miss scheduling on this trace would be unaffected by the inner loop. In practice c-flat produces two separate traces, split at loopback, although the single trace alternative would likely be more effective since miss scheduling has the greatest opportunity for latency tolerance in long traces.

Just as inner loops can potentially result in replacement of buffered prefetch values if they occur between prefetch and corresponding badref, so can procedure calls. Judging the cache performance of procedure calls requires some amount of interprocedural analysis. If the only interprocedural information desired is an estimate of the average number of misses associated with a call, this can be produced with minimal effort in a system which gathers memory performance statistics at instruction level granularity. C-flat uses a preprocessor to coalesce gathered memory statistics into an average miss tally for each procedure.

The number of misses associated with a loop including any misses resulting from inner loops and procedure calls can be compared against a threshold based on cache size and associativity. When this threshold is exceeded, miss scheduling should be applied to the loop rather than RAAD prefetching. For caches with large associtivity, buffered prefetch data is primarily susceptible to capacity related misses. Thus the threshold should be set at some large fraction of the cache size. For direct-mapped caches collision misses will interfere with buffered prefetches so the threshold should be set at a smaller fraction of the cache size.

The situations above could be characterized as caching interfering with buffering. Cache misses for references not directly in the outer loop flush buffered data. For a loop with a large number of badrefs buffering can interfere with caching. Under RAAD prefetching the cache buffers a block of data for each prefetch added to a loop. If this buffer space represents a substantial fraction of the cache space then the buffered data may replace useful data in the cache. Unfortunately, it is much more difficult to detect this situation with a heuristic test. Buffering interferes with caching when reused data which could be maintained within the cache across multiple loop iterations is replaced to make room for prefetched data. The amount of this interference depends on the amount of successfully cached data associated with a loop. This cannot be accurately estimated from the dynamic miss statistics used in c-flat although it could be estimated using static analysis techniques.

In c-flat a single heuristic test is employed for loops with large numbers of badrefs and/or internal misses. For high associativity caches the RAAD prefetching threshold is set at half the cache size while for direct-mapped caches it is set at 10% of the cache size. These numbers have both been chosen somewhat arbitrarily.

Another situation in which miss scheduling is likely to outperform RAAD prefetching occurs when a loop contains a large number of badrefs relative to the average number of iterations associated with a loop execution. For loops, the two forms of scheduling exhibit two different forms of pipeline fill and drain overheads. RAAD prefetching incurs pipeline overhead on initial and final iterations of the loop. The initial iteration incurs a miss for each badref. The final iteration contains prefetches for potentially unused memory addresses. Prologue prefetching reduces the cost of initial iteration misses if enabled. Unnecessary misses in the final iteration could be eliminated by performing loop peeling to produce an epilogue with no prefetches, however, c-flat does not perform this transformation. As a result of these added misses in the final iteration, RAAD prefetching has a pipeline drain cost proportional to the number of badrefs. The benefit associated with these costs is that the memory system can be actively processing misses during loopback.

As a result of scheduling misses for a sequential trace, miss scheduling produces code in which the memory system is likely to be idle at the beginning and end of traces. In the context of loops the trace is broken at the loopback edge. Due to this inability to overlap prefetching activity across loopback edges, loops scheduled using miss scheduling have memory system pipeline fill costs in each loop iteration proportional to the memory latency. Consider a memory bandwidth limited loop. Since the memory system is idle under miss scheduling during loopback the latency of each loop is likely to be extended by at least the difference between the memory latency and intersubmission latency, *mlat* - *mbwlat*.

For memory bandwidth limited loops the cost of each strategy can be compared. Consider a loop with I iterations. The RAAD version incurs a number of misses on each iteration approximately equal to the sum of the miss rates of its badref memory references. Because of software pipelining, these misses are incurred $I + 1$ times. In a bandwidth limited loop, the time to execute the loop will be approximately the product of the total number of misses and *mbwlat*. The miss scheduled version incurs the same number of misses each iteration but only I iterations worth. It has an added cost of approximately *mlat* − *mbwlat* in each iteration due to memory pipeline end effects. The two costs are indicated below.

$$Cost_{RAAD} = (\sum_{badrefs} m_i) * mbwlat * (I + 1) \qquad (5-13)$$

$$Cost_{miss} = ((\sum_{badrefs} m_i) * mbwlat + mlat - mbwlat) * I \qquad (5-14)$$

Dividing each equation by *mbwlat* and simplifying leads to the following condition for using RAAD prefetching.

$$\sum_{badref} m_i < I * (mconc - 1) \qquad (5-15)$$

Notice that if *mconc* is 1, the right hand side is zero, suggesting that RAAD prefetching should never be used. From a practical standpoint, the performance of miss scheduling has been overestimated. It may well incur a per iteration cost differential of more than *mlat* - *mbwlat*. In c-flat, the differential is assumed to be at least *mlat*, resulting in the modified formula below.

$$\sum_{badref} m_i < I * mconc \qquad (5-16)$$

5.7.1 Optimal Loop Scheduling

For loops in which the number of badrefs exceeds the maximum memory system concurrency, *mconc*, a prefetching technique which blends the techniques used in RAAD prefetching and miss scheduling could be applied in order to avoid the pipeline costs associated with each. *Mconc* characterizes the maximum number of miss transactions which can be pending at loop back. There is minimal advantage to producing prefetch code which shifts more than *mconc* prefetches across loop iterations. The only potential advantage is reduced software overhead when prefetches are directly adjacent to their badrefs. Rather than RAAD prefetching, miss scheduling could be performed. The prefetch stream merged into the loop code would consist of two sets of prefetches. Prefetches for all but the first *mconc* badrefs in the loop produced using the standard miss scheduling technique would be first in the prefetch stream. Following these prefetches would be *mconc* RAAD-based prefetches for the first *mconc* badrefs in the loop produced using a *PIC* value of 1. In this scheme, a minimal number of prefetches are shifted across loop boundaries, minimizing software pipelining overhead. This technique has not been implemented in c-flat.

5.8. Summary

This chapter describes modelling, analysis and heuristic techniques applicable to memory latency tolerant compilation primarily in the context of c-flat, the prototype latency tolerant compiler implementation developed for this research. The first major problem in memory latency tolerant compilation is modelling the behavior of static memory references. Based on the observation of static locality correlation and badref behavior, the badref model, a compiler model in which static references are partitioned into a goodref set modelled as hits and a badref set modelled as misses is described.

One finds empirically that explicit or implicit loops provide a major source of frequently executed badrefs. Tolerance to very high latencies is faciliated by a technique which allows miss latency to be overlapped with several loop iterations. RAAD prefetching, a technique related to software pipelining, is developed as a means to achieve this overlap in scheduled code. Prefetching schemas for a variety of RAAD data structures are described as well as heuristics for dealing with loops involving conditional behavior.

A second scheduling technique termed miss scheduling is also described. This technique is applicable to code not containing identifiable loops. The technique merges prefetch instructions into a long sequential trace of instructions, exploiting reverse parallelism. Heuristics are developed to govern the choice between RAAD prefetching and miss scheduling when both are potentially applicable.

Chapter 6

Compiler Results and Evaluation

Several types of experiments can be performed using c-flat. The process of compilation produces structural information about programs and the static costs of latency tolerance optimizations. C-flat must classify the misses in programs in order to identify an appropriate prefetching strategy. When summarized in tabular form with dynamic memory behavior statistics this information serves to identify the prevalence of various forms of badrefs within programs, both in terms of their static frequency and their dynamic contributions to program references and misses. In addition to compiler gathered information, code produced by c-flat can be run in conjunction with a memory timing simulator in order to evaluate the real runtime performance of latency tolerant code. Benchmark codes are tested using a variety of memory system parameters and processor models. This chapter presents results gathered using c-flat.

6.1. Characterization of Badrefs

Each memory reference within a program is analyzed by c-flat. Based on a miss rate threshold, references are classified as goodrefs and badrefs. Badrefs are further considered for prefetch optimization. Information about badrefs is stored into tables at two points during the compilation process. Information is first stored when badrefs are recognized. At this point a high level categorization is performed, *i.e.* the reference is identified as an array RAAD, a linked-list RAAD, a badref to be scheduled by miss scheduling, etc. Subsequently, upon actually performing prefetch optimization, additional information is recorded. This additional information identifies the specific prefetching strategy adopted for the reference and the estimated overhead measured in terms of added instructions.

Badref thresholding and subsequent optimization strategies are dependent upon the memory parameters assumed during a compilation. As a consequence, the statistics gathered during a compilation are also dependent upon this parameterization. The tables below categorize badrefs for each benchmark based on memory statistics gathered using the **FA** cache model and assuming a miss processing latency of 20 cycles and a memory concurrency of 1.

For each category of badref four statistics are presented. The first column, labelled **Stat**, is the number of static references in the given category. The second and third columns, labelled **Dyn** and **Miss**, are the number of dynamic references and misses arising from static references in the category based on the memory performance data used during compilation. The final column, **Ovhd**, is the dynamic weighted average prefetch overhead for the category. The dynamic weighted average overhead is the sum of overheads for each individual reference in the category weighted by the fractional contribution of the particular static reference towards the total dynamic references of the category. This is an estimate of the expected average dynamic cost of prefetches in the category.

Rows in the tables are broken into 5 major categories. The first four categories are associated with those badrefs arising in explicit loops for which loop prefetching techniques are considered. The categories include Array RAADs, Indirection Array RAADs, Linked-list RAADs and an additional category for unrecognized references. Within each major category references are further partitioned depending on whether they occur in loops without conditional behavior, labelled **Norm**, or loops with conditional behavior, labelled **Cond**. Prologue prefetches for loop-constant address badrefs are included as a special class of Array RAADs, labelled **Prologue**. Prefetches which are eliminated by cost-benefit heuristics due to excessive overhead are tabulated in rows marked **Unpref'd**.

The final major category of rows includes all badrefs handled by miss scheduling. The row labelled **Short** tallies those references for which the desired miss-use spacing based on memory latency is not achieved. The following line, labelled **Distance**, indicates the miss weighted average distance by which references fell short of the desired spacing. This number can be less than one. When the spacing is not achieved a prefetch for another reference is moved to fill the gap if possible. If overhead instructions for this jockeyed prefetch instruction exceed the shortfall the badref is still counted in the short category but a 0 is averaged into the shortfall distance statistic. The weighted averaging technique used to compute the shortfall distance statistic is based on dynamic misses rather than dynamic references since shortfalls only effect performance in conjunction with misses. The shortfall statistic only reflects those references tabulated in the **Short** column.

After rows associated with RAAD prefetching and miss scheduling there is a row labelled **Total**. The total row indicates the total number of references for which some form of latency tolerance optimization is actually performed and the fraction of dynamic references and misses impacted by these optimizations.

	doduc				eqntott			
	Stat	Dyn	Miss	Ovhd	Stat	Dyn	Miss	Ovhd
Array RAADs	207	0.0125	0.0534		36	0.7911	0.8178	
Norm	172	0.0093	0.0323	1.19	17	0.0574	0.0516	1.02
Cond	22	0.0023	0.0201	1.00	13	0.7335	0.7660	1.00
Prologue	10	0.0007	0.0007	2.20	4	0.0001	0.0002	3.00
Unpref'd	0	0	0		1	0.0001	0.0000	
Ind Arr RAAD	0	0	0		4	0.0002	0.0017	
Norm	0	0	0	0.00	3	0.0001	0.0007	3.00
Cond	0	0	0	0.00	1	0.0001	0.0010	3.00
Unpref'd	0	0	0		0	0	0	
Lnk-Lst RAAD	0	0	0		8	0.0005	0.0030	
Norm	0	0	0	0.00	5	0.0003	0.0033	1.27
Cond	0	0	0	0.00	3	0.0001	0.0009	1.67
Unpref'd	0	0	0		0	0	0	
Unrecognized	0	0	0		1	0.0000	0.0000	
Miss Scheduled	668	0.0591	0.7325	2.44	42	0.0447	0.1526	1.75
Short	389	0.0328	0.4098		14	0.0290	0.0733	
Distance	1.4931				0.0146			
Unpref'd	0	0	0		0	0	0	
Total	872	0.0714	0.7856	2.22	88	0.8363	0.9764	1.04

Table 6-1: Badref Categorization for Doduc and Eqntott

The data in Tables 6-1 to 6-4 can be used to evaluate the relevance and effectiveness of the RAAD prefetching and miss scheduling techniques. Data in the **Miss** column indicates the number of misses potentially avoided with a particular technique, indicating its benefit. Data in the **Dyn** and **Ovhd** columns indicates the costs of a technique.

Based on miss coverage data, in three of the benchmarks, matrix300, nasa7 and tomcatv, array RAAD prefetching, even if applied only to non-conditional loops, would be an adequate latency

	espresso				fpppp			
	Stat	Dyn	Miss	Ovhd	Stat	Dyn	Miss	Ovhd
Array RAADs	161	0.0874	0.3845		22	0.0007	0.0053	
Norm	91	0.0729	0.2564	1.09	12	0.0000	0.0001	1.83
Cond	40	0.0130	0.1269	1.02	6	0.0004	0.0049	1.99
Prologue	2	0.0000	0.0000	1.00	0	0	0	0.00
Unpref'd	6	0.0005	0.0005		1	0.0003	0.0003	
Ind Arr RAAD	8	0.0001	0.0004		0	0	0	
Norm	2	0.0000	0.0002	3.00	0	0	0	0.00
Cond	2	0.0000	0.0001	4.00	0	0	0	0.00
Unpref'd	4	0.0001	0.0001		0	0	0	
Lnk-Lst RAAD	37	0.0026	0.0103		0	0	0	
Norm	13	0.0002	0.0067	2.19	0	0	0	0.00
Cond	24	0.0008	0.0222	1.37	0	0	0	0.00
Unpref'd	0	0	0		0	0	0	
Unrecognized	45	0.0030	0.0128		3	0.0012	0.0031	
Miss Scheduled	250	0.0532	0.4470	1.87	510	0.0660	0.6223	1.39
Short	84	0.0158	0.1541		144	0.0145	0.1203	
Distance	0.0148				1.5666			
Unpref'd	0	0	0		0	0	0	
Total	424	0.1400	0.8596	1.38	528	0.0664	0.6272	1.40

Table 6-2: Badref Categorization for Espresso and Fpppp

	matrix300				nasa7			
	Stat	Dyn	Miss	Ovhd	Stat	Dyn	Miss	Ovhd
Array RAADs	7	0.2832	0.9868		235	0.2122	0.9145	
Norm	7	0.2832	0.9868	2.50	232	0.2119	0.9132	1.81
Cond	0	0	0	0.00	2	0.0003	0.0013	1.00
Prologue	0	0	0	0.00	0	0	0	0.00
Unpref'd	0	0	0		0	0	0	
Ind Arr RAAD	0	0	0		0	0	0	
Norm	0	0	0	0.00	0	0	0	0.00
Cond	0	0	0	0.00	0	0	0	0.00
Unpref'd	0	0	0		0	0	0	
Lnk-Lst RAAD	0	0	0		0	0	0	
Norm	0	0	0	0.00	0	0	0	0.00
Cond	0	0	0	0.00	0	0	0	0.00
Unpref'd	0	0	0		0	0	0	
Unrecognized	0	0	0		6	0.0000	0.0001	
Miss Scheduled	10	0.0014	0.0049	1.33	23	0.0000	0.0002	2.65
Short	7	0.0009	0.0033		8	0.0000	0.0000	
Distance	2.0017				4.8335			
Unpref'd	0	0	0		0	0	0	
Total	17	0.2846	0.9917	2.49	257	0.2122	0.9147	1.81

Table 6-3: Badref Categorization for Matrix300 and Nasa7

tolerance technique. In these benchmarks array RAADs in non-conditional loops account for over 90% of program misses. In another benchmark, eqntott, array RAADs account for over 80% of misses. Although these RAADs occur in a conditional loop, they do not exhibit conditional behavior and are handled by a straigtforward application of the array RAAD prefetech schema. Overall, these

	spice				tomcatv			
	Stat	Dyn	Miss	Ovhd	Stat	Dyn	Miss	Ovhd
Array RAADs	44	0.0165	0.0167		51	0.2144	0.9971	
Norm	15	0.0055	0.0166	1.12	47	0.2005	0.9327	1.52
Cond	8	0.0000	0.0000	1.03	2	0.0138	0.0643	1.50
Prologue	5	0.0000	0.0000	2.00	2	0.0001	0.0001	2.00
Unpref'd	1	0.0000	0.0000		0	0	0	
Ind Arr RAAD	2	0.0000	0.0000		0	0	0	
Norm	0	0	0	0.00	0	0	0	0.00
Cond	2	0.0000	0.0000	3.50	0	0	0	0.00
Unpref'd	0	0	0		0	0	0	
Lnk-Lst RAAD	0	0	0		0	0	0	
Norm	0	0	0	0.00	0	0	0	0.00
Cond	0	0	0	0.00	0	0	0	0.00
Unpref'd	0	0	0		0	0	0	
Unrecognized	24	0.0136	0.0564		0	0	0	
Miss Scheduled	759	0.2575	0.7831	6.09	15	0.0000	0.0000	1.86
Short	249	0.1114	0.3158		12	0.0000	0.0000	
Distance	0.0204				0.0127			
Unpref'd	0	0	0		0	0	0	
Total	789	0.2629	0.7997	5.99	66	0.2144	0.9971	1.52

Table 6-4: Badref Categorization for Spice and Tomcatv

benchmarks achieved the highest level of miss coverage. The prevalance of misses in non-conditional loops indicates very structured behavior. Structured behavior is most amenable to scheduling-based latency tolerance techniques of which both RAAD prefetching and miss scheduling are examples.

Examining the categorization of individual RAAD references, it is clear that in the benchmarks analyzed, indirection array RAADs and linked-list RAADs play a minimal role.

In another set of benchmarks, consisting of fpppp, spice and doduc, RAAD prefetching is of very limited benefit. In these programs badrefs which are RAADs account for between 1 and 5% of misses. Miss scheduling is able to cover 60 to 80% of misses. In the final benchmark, espresso, roughly equal fractions of misses are covered by RAAD prefetching and miss scheduling. Based on the fact that about half the benchmarks are handled by each technique one might argue that latency tolerant compilers should support both techniques.

The data warrants some clarification on two points. First, this data probably underestimates the fraction of references which might be identified as RAADs in a more aggressive compilation environment. No form of interprocedural analysis is performed in c-flat. As a consequence, if a RAAD is isolated from its loop by a procedure call it is not recognized. In a compiler which performed interprocedural flow analysis one might apply loop-based prefetching across procedure boundaries to generate prefetches for these hidden RAADs. Upon examination of fpppp and doduc, two of the benchmarks for which miss scheduling is primarily utilized, one observes that many badrefs are indeed array references in loops in which the array references are separated from the loops by procedure boundaries.

Second, the number of references which can be optimized using miss scheduling is probably underestimated by the data. In c-flat a relatively stringent initial thresholding is applied to references in order to qualify for miss scheduling. For the data presented, no reference with a miss rate less than 20% is miss scheduled. This thresholding is applied in c-flat because of the way miss scheduling models memory bandwidth consumption. In particular it assumes that each badref consumes bandwidth for a miss transaction, irrespective of the miss rate of the badref. Separation between prefetches is determined based on bandwidth utilization so performance can be adversely impacted by grossly overestimating the bandwidth requirements of badref references. By relaxing the miss rate threshold, perhaps in conjunction with the use of a more sophisticated bandwidth consumption model for badrefs with relatively low miss rates, miss scheduling could achieve better miss coverage.

While miss data measures potential prefetching benefit, dynamic reference data and overhead estimates can be used to assess the costs of prefetching. In seven of the benchmarks prefetched references account for between about 7 and 30% of dynamic references. Differentiated treatment of badrefs and goodrefs has eliminated a substantial amount of unnecessary overhead by ignoring from 70 to 90% of references. In one case, eqntott, prefetched references account for over 80% of dynamic references. In this benchmark, two particular static references in a loop produce about 70% of the dynamic references and 75% of the misses. Given the miss threshold chosen for badref partitioning these references are both identified as badrefs. As a consequence, in this benchmark the badref set accounts for over 80% of dynamic references. Only 20% of references are considered goodrefs. One of the two primary static references just barely exceeds the badref miss rate threshold. For a latency of 20 the threshold is 5%. The two references exhibit miss rates of about 6 and 20%. If the threshold were moved slightly the lower miss rate reference would be classified as a goodref, decreasing the miss coverage by 18% but excluding an additional 36% of references from overhead. This marginal prefetch is only added because it has an overhead of one instruction and thus is deemed slightly beneficial. At an overhead of two it would be rejected.

Estimated overhead costs for prefetching are also included in the tables. Overheads for both array RAAD prefetching and miss scheduling typically fall between about 1.5 and 2.5 instructions, including the prefetch instruction. In general RAAD prefetching involves slightly lower overhead than miss scheduling although there a few exceptions to this rule. The highest array RAAD overhead ocurs for matrix300. Matrix300 has loops with non-constant, loop-constant strides. Code generated by c-flat in this case could be improved by one instruction per prefetch by additional loop-constant optimization.

An additional noteworthy statistic is the overhead due to miss scheduling in spice. In this benchmark the average miss scheduling overhead is about six instructions. Costs of moving instructions across basic block boundaries exceed those for intrablock scheduling due to increased copying for address computation instructions. Two of the benchmarks which extensively used miss scheduling, doduc and fpppp, had a prevalence of very large basic blocks. These benchmarks exhibit the lowest average miss scheduling overheads. In the case of spice substantial code motion across blocks has lead to a very large average overhead for miss scheduling. C-flat has fairly primitive dataflow analysis and must be conservative in copying code for cross block miss scheduling. It is quite likely that overhead in spice could be lessened given better dataflow analysis.

The final statistics of interest in the tables are the fractions of miss scheduled references for which the desired miss-use gap is not achieved and the average shortfall distances for these references. In the four benchmarks in which miss scheduling is significant, scheduling exceeds the desired miss-use gap for the majority of badrefs. The shortfall distance, which is computed only for badrefs in the **Short** row, is also relatively small. The worst case for miss-use shortfall is doduc in which about 60% of misses experience shortfalls averaging about 1.5 instructions.

6.2. Optimized Code Size

An additional static statistic which can be measured is the fractional increase in code size associated with prefetching. This information is presented in tabular form below for compilations based on the **FA** cache model for three different memory performance parameterizations.

Relative Code Size			
Benchmark	20 10	20 20	160 20
doduc	1.06	1.06	1.15
eqntott	1.05	1.04	1.07
espresso	1.03	1.03	1.05
fpppp	1.07	1.06	1.09
matrix300	1.09	1.09	1.10
nasa7	1.06	1.06	1.06
spice		1.03	
tomcatv	1.30	1.30	1.33

Table 6-5: Fractional Increase in Optimized Code Size

Code size variation is relatively undramatic in most cases, typically measuring between 5 and 10%.

In one benchmark, tomcatv, code expansion is somewht more pronounced. This can be traced primarily to poor generation of prologue prefetch code. In order to avoid misses in initial loop iterations c-flat can emit a prologue which performs these prefetches. It generates poor code for loops with a large number of badref array references in which the initial index variable value is not zero. It duplicates the scaling of this index variable in each individual prologue prefetch computation. In the case of tomcatv this results in a substantial amount of code. The unusually large increase in code size for tomcatv can be avoided by either suppressing prologue code generation, (an option available under c-flat), or subjecting this code to a common subexpression elimination pass. By suppressing prologue code generation the optimized code is only 8% larger than the unmodified code, in comparison to the 30% increase reported in the table. The number in the table is an artifact of the prototype compiler.

6.3. Dynamic Performance

Code produced by c-flat is executed in conjunction with a timing simulator in order to measure the dynamic performance of the compiled code. The performance of latency tolerance techniques is impacted by more than just cache hit rate. Prefetched blocks may be used by a required reference before they arrive in the cache, resulting in a stall. Similarly, miss processing traffic generated by prefetches may result in delays for miss processing of non-prefetched references. A detailed timing model for processor/memory interface behavior is used in order to assess the performance of software latency tolerance through prefetching in the presence of these effects.

6.3.1 Processor and Interface Models

Two distinct processor models are simulated. The models are referred to as the **Stall** model and the **Interlock** model. The primary difference between these two models from the standpoint of memory latency tolerance is in the behavior of the processor/memory system interfaces.

The **Stall** model assumes all forms of processor/memory synchronization and flow control are achieved through immediate stalls. If a load or store operation cannot be serviced by a cache hit a stall occurs for the duration of miss processing. This effectively ensures that all memory references have an apparent latency with respect to the processor of a single cycle, avoiding the need for further synchronization. No addtional form of flow control is required between the processor and memory system for loads or stores under the **Stall** model because a new transaction cannot be initiated while an old transaction is pending.

Under the **Stall** model prefetch operations are assumed to be buffered within the cache. The size of this buffer is parameterized in the timing simulations. In the data presented the size is set to one more than the memory concurrency. Thus at any time one uninitiated prefetch request can be buffered by the cache. Since prefetch operations do not return data to the processor no form of data synchronization is required for prefetches. Prefetch operations are flow controlled. If a prefetch is attempted by the processor when the prefetch buffer is full a stall results until a buffer slot becomes available.

The **Interlock** model uses explicit mechanisms for all processor/memory synchronization. The model assumes that registers within the processor have scoreboard information or full/empty bits specifying their availability. When a load operation is initiated the target register becomes unavailable. Upon receipt of the appropriate data from the memory the register becomes available. Synchronization related stalls are not incurred until a loaded value is actually used in a subsequent instruction. Since write operations do not return values to the processor they are not subject to these forms of stalls. Both load and store operations are buffered by the cache just as prefetch operations are buffered in the **Stall** model. Flow control stalls occur if a memory request would result in buffer overflow. Prefetch references are similarly buffered in the **Interlock** model

In order to simulate load synchronization stalls in the **Interlock** model each miss reference is tagged with timestamps identifying the time at which a memory transaction for the miss is initiated and the time at which the value is ultimately used by the processor. The use time is updated to reflect any stalls which intervene between instruction issue and use. After all intervening stalls have occurred the separation between transaction initiation and use can be compared with the memory latency to determine if use of the data produces a stall. This same timestamp mechanism is used to determine whether prefetched data arrives in the cache soon enough to avoid stalls for subsequent hits on the data.

The **Interlock** model assumes that the processor/memory interface supports multiple transactions through an explicit tagging scheme. An arbitrary number of load and store operations can be simultaneously active at the interface. This is subject only to buffer size limitations for uninitiated transactions within the cache and also to the constraint that only one load per register may be active simultaneously.

The **Interlock** model provides a suitable mechanism for memory latency tolerance without a prefetch instruction since all load operations are non-blocking and the interface supports multiple outstanding requests. C-flat does not directly target this mechanism so an explicit prefetch instruction is included as part of the model as well. The prefetch instruction is buffered and flow-controlled. Two separate buffers are modelled in the **Interlock** memory system, one for uninitiated prefetch operations and one for uninitiated required references, *i.e.* loads and stores. Required references receive priority for miss processing bandwidth. If a buffered, uninitiated prefetch is the target of a required reference the status of the reference is changed from prefetch to non-prefetch and it is moved between buffers. Cache write-back operations arising from either prefetch or non-prefetch references are treated as required references and added to the required reference buffer, consuming a buffer slot. In the data presented in this section the **Interlock** model uses buffer capacities allowing one uninitiated reference each of required and prefetch references to be buffered.

The **Stall** and **Interlock** models were designed to reflect two extremes in terms of the flexibility with which a processor and memory system might interact. The processor/memory interface in the stall model limits the number of outstanding required transactions to one and has no explicit means of synchronization other than stalls. The interface in the **Interlock** model provides explicit synchronization mechanisms and allows an arbitrary number of outstanding required transactions.

6.3.2 Memory System Modelling

The memory system underlying either interface uses a fairly simple model. This model corresponds exactly with the model assumed within c-flat, namely an asynchronous pipeline with a processing latency, *mlat*, and a minimum intersubmission latency, *mbwlat*. Transactions are completed a fixed time after they are initiated, characterized by *mlat*. After transaction initiation, no further transactions can be started for at least the intersubmission latency, *mbwlat*. After this time has elapsed a transaction can be started at any time.

6.3.3 Time Accounting

Runtime in the simulation environment is partitioned into three distinct categories: **instructions**, **latency stalls** and **bandwidth stalls**. Each simulated cycle is charged to one of these three categories.

Instructions, as implied by the name, correspond to cycles on which new instructions are issued. Based on the definition of a cycle as the inverse of the maximum instruction issue bandwidth at most one instruction can issue in a cycle. The total number of instruction cycles is exactly equal to the total number of instructions executed and represents a lower bound on the total execution time.

Cycles on which instructions are not issued are stalls. Stalls can arise in the timing model for two different types of reasons and are subclassified as latency stalls and bandwidth stalls. **Latency stalls** arise when a processor is forced to wait for the completion of a required memory transaction. **Bandwidth stalls** occur when a processor must wait to initiate a transaction due to some previously initiated transaction.

In the **Stall** model there are two sources of latency stalls and in the **Interlock** model there is one source. In the stall model a latency stall occurs for any required memory reference, either a load or store, which cannot be immediately serviced with a cache hit. In this model, required misses result in a stall at least as long as the memory latency. The stall can exceed the memory latency if transactions due to prefetches or writebacks are pending. Stall cycles beyond the miss latency occur when the intersubmission interval has not passed since initiation of the most recent transaction. Latency stalls can similarly be incurred for required references which experience hits on blocks for which a prefetch transaction is pending but not completed. In this case the stall is equal to the time remaining before the completion of the prefetch-initiated miss transaction. Latency stalls in the **Interlock** model are only incurred when loaded data is accessed. A stall occurs when the time at which a value is accessed precedes its arrival time in the processor. The stall experienced upon using a value in the **Interlock** model can vary from a a single cycle to a number in excess of the memory latency. (For both the **Stall** and **Interlock** models the maximum stall for any instruction is bounded by the sum of *mlat* and the product of *mbwlat* and the number of previous transactions which may be buffered externally to the processor.) Under either model if a stall classified as a latency stall exceeds *mlat*, the miss processing latency, that component of the stall beyond *mlat* is counted as a bandwidth stall rather than a latency stall.

Bandwidth stalls occur when one reference is delayed by another due to memory bandwidth limitations. When a latency stall exceeds *mlat* that component above *mlat* has occurred because of bandwidth limitations. Flow control stalls are also counted as bandwidth stalls. Flow control stalls arise when a reference cannot be issued because its associated buffer is full.

6.3.4 Performance Results

Simulated runtime experiments have been performed for a variety of memory system configurations. The experiments investigate three combinations of latency and bandwidth parameters. Memory systems will be identified using a combination of their values for the parameters *mlat* and *mbwlat* with *mlat* appearing first. The baseline system is 20/20, *i.e.* it has a value of 20 for each parameter. Also measured are 20/10 and 160/20. The first alternative configuration halves the value of *mbwlat*, thus doubling the memory system concurrency and bandwidth. This bandwidth increase occurs at latency equivalent to the baseline. The second alternative increases *mlat* by a factor of 8 at a constant value of *mbwlat*. The 160/20 model thus has a memory concurrency of 8. Each benchmark is simulated using both the **FA** and **DM** cache models. Since memory system performance is characterized in terms of the latency for transactions moving a block of data and the block size differs between the **FA** and **DM** cache models, the underlying memory bandwidth also differs between the models for the same simulation parameters. The **DM** model, with a blocksize twice that of the **FA** model, has twice the bandwidth for a given memory parameterization.

The process by which simulated runtime data is produced warrants review. Benchmarks are initially compiled and run for sample input data. This run is coupled with a cache simulation using the eventual target cache model. The simulation produces the dynamic memory performance statistics used by c-flat for latency tolerance optimization. A second compilation is performed. This compilation uses the simulated memory statistics and also receives a parameterized estimate of the memory system corresponding to the eventual parameters. Thus for each benchmark six different optimized versions are produced, corresponding to the two different cache models and the three different memory system configurations. The optimized versions of the code are then run using the timing simulation. The data used for the eventual timing is the same as the sample data. The benchmark espresso has four different input data sets. Memory statistics for the second compilation pass are attained by averaging the four cases and the reported runtime performance is the sum of the four cases.

The figures below show graphs of performance data for the benchmarks. Color-coded columns indicate the runtime of various versions of the program decomposed into the three components: instructions, bandwidth stalls and latency stalls. Columns are marked with the following labels.

B represents a baseline column. A basline column indicates the performance of unmodified code for some processor/cache/memory system configuration.

P represents a prefetch column. These columns show the performance of optimized code generated by c-flat for a given configuration.

Min is a statistic derived from baseline data. It represents a lower limit on the runtime achievable through the latency tolerance optimizations employed by c-flat. Prefetching optimizations in c-flat do not decrease either the total instruction count of the benchmark or the aggregate traffic between memory and cache. Two lower bounds on time can be computed. The first is the total number of instructions in the unoptimized code representing a lower bound on processor time. The second is a consequence of memory bandwidth. The product of the memory intersubmission delay for the memory system, *mbwlat*, and the total number of cache misses and writebacks in the unoptimized case represents a lower bound on memory time. **Min** represents the maximum of these two performance lower bounds and is color-coded to indicate whether it is an instruction or memory bandwidth bound.

6.3.4.1 FA Data

Data in the following figures is based on the **FA** model. All columns are scaled relative to the **Min** value for the 20/20 memory system.

Speedups computed as the ratio of basline performance to prefetch optimized performance are summarized in Table 6-6.

Under the **Stall** model using the baseline 20/20 memory system, benchmarks exhibited between 10 and 30% speedups relative to unoptimized code. Due to the small 8-byte cache blocks of the **FA** model the memory system has a relatively small bandwidth for processing cache misses and writebacks. Half of the benchmarks are memory bandwidth limited and the optimized code for these benchmarks exhibits the highest speedups. With the exception of spice, the optimized code is able to mask away all latency stalls, essentially reaching the performance limits imposed by memory bandwidth. Three benchmarks, matrix300, nasa7 and tomcatv, each exhibiting a speedup in excess of 20%, are numerical codes for which Array RAAD prefetching successfully eliminates essentially all unscheduled misses. Spice, the remaining memory bandwidth limited benchmark, is optimized primarily through sequential miss scheduling. It does not reach the memory bandwidth limit like

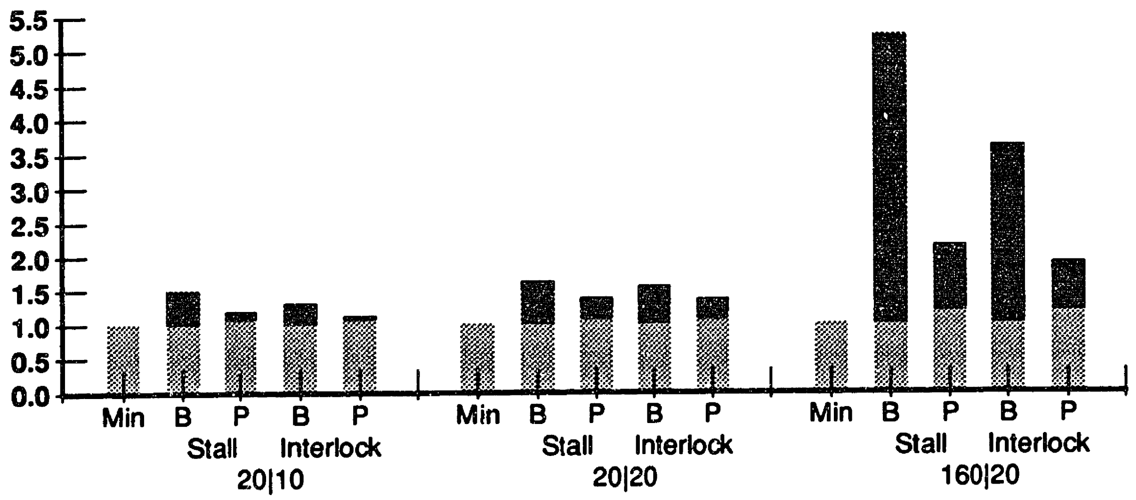


Figure 6-1: Doduc **FA** Performance

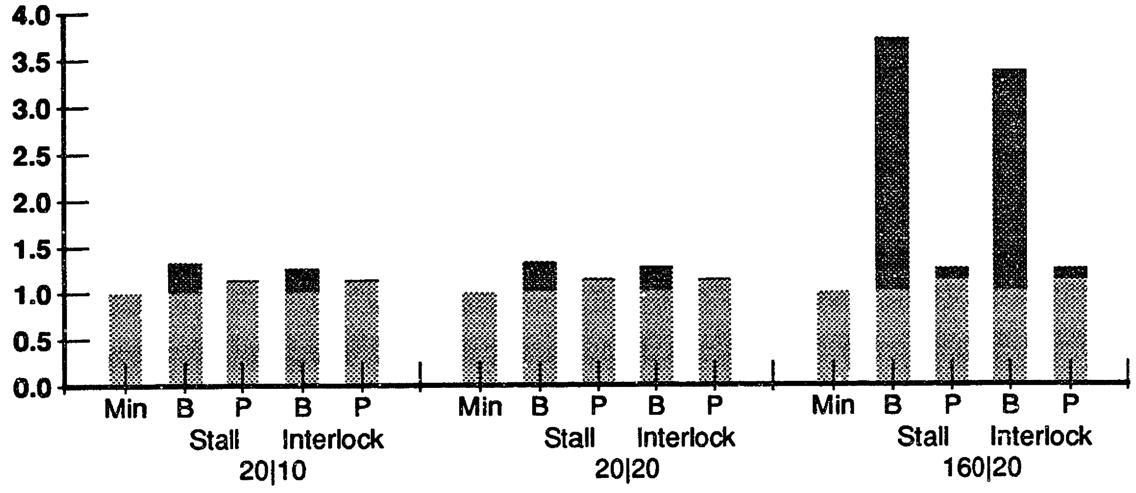


Figure 6-2: Eqntott **FA** Performance

the other three because about 20% of its misses are not optimized. The reason these misses are not optimized is discussed above in section 6.1.

The set of benchmarks which are not memory bandwidth limited includes three programs primarily optimized using miss scheduling. These benchmarks, doduc, fpppp and espresso, exhibit speedups of between 10 and 17% under the **Stall** model. In each case latency stalls are decreased by over 50%. Latency stalls are not decreased by the fraction one might anticipate based on relative miss coverage, however.

Looking specifically at doduc, based on information in Table 6-1, the compiler estimates a coverage of 78.5% of misses. For somewhat more than half of these misses, (41% of the total misses) a latency tolerance shortfall is anticipated. The miss weighted average of this shortfall is predicted to be about 1.5 cycles. Factoring in the shortfall, one might predict a relative decrease in total latency stall cycles of $0.375 + 0.41 * (1 - 1.5/20)$ or a little better than 75%. In actuality, although only 21.5% of required misses remain, (exactly matching the compiler estimate), measured latency stall cycles decrease by only about 55%.

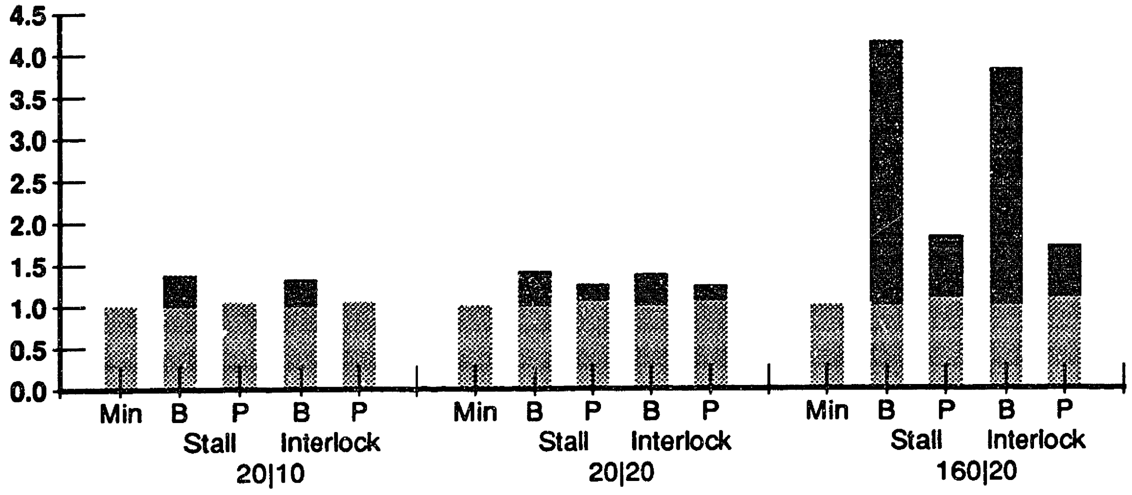


Figure 6-3: Espresso **FA** Performance

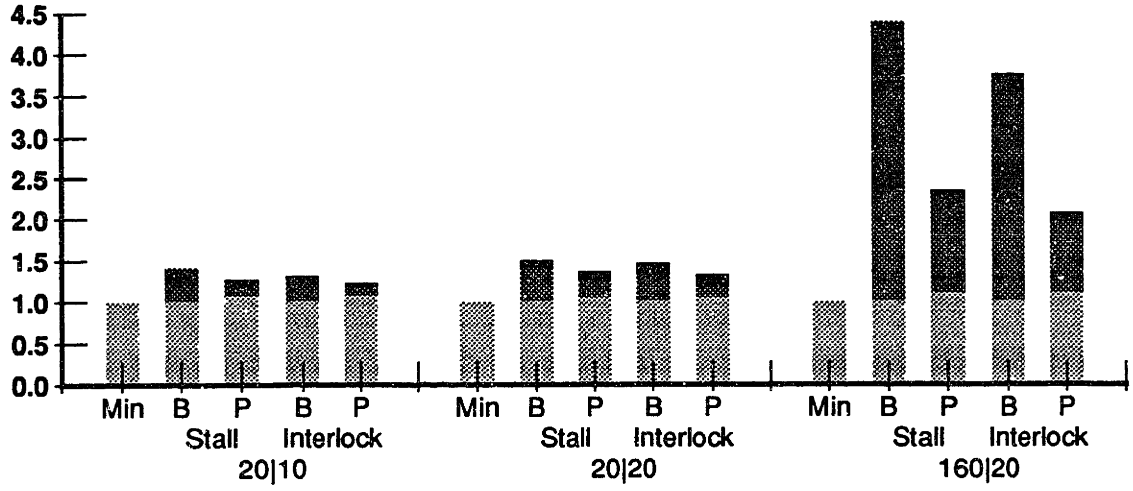


Figure 6-4: Fpppp **FA** Performance

The discrepancy arises from a higher level of latency tolerance shortfall than anticipated. In the simulated data only about 30% of prefetched values experienced a latency tolerance shortfall but the miss weighted average of this shortfall is about 13 cycles. The explanation for this discrepancy is cache writeback operations. About 40% of cache misses incurred by the benchmark resulted in writebacks. For each writeback, memory bandwidth consumption which is not modelled by the compiler occurs. This bandwidth consumption is 20 cycles. Given a latency shortfall substantially higher than expected in a fraction of misses corresponding roughly to the fraction of misses experiencing writebacks, it appears certain that writebacks account for the performance discrepancy experienced.

Accounting for writeback bandwidth during miss scheduling would certainly be feasible in a compiler using simulated or measured memory behavior information like c-flat. In fact, writeback information for each reference is already available in c-flat although it is not incorporated into the miss scheduling algorithm. The performance of doduc and other scheduled benchmarks suggests that modification of the miss scheduling algorithm to account for writebacks is probably warranted.

Turning to performance under the **Interlock** model, one finds that speedups are a little less dra-

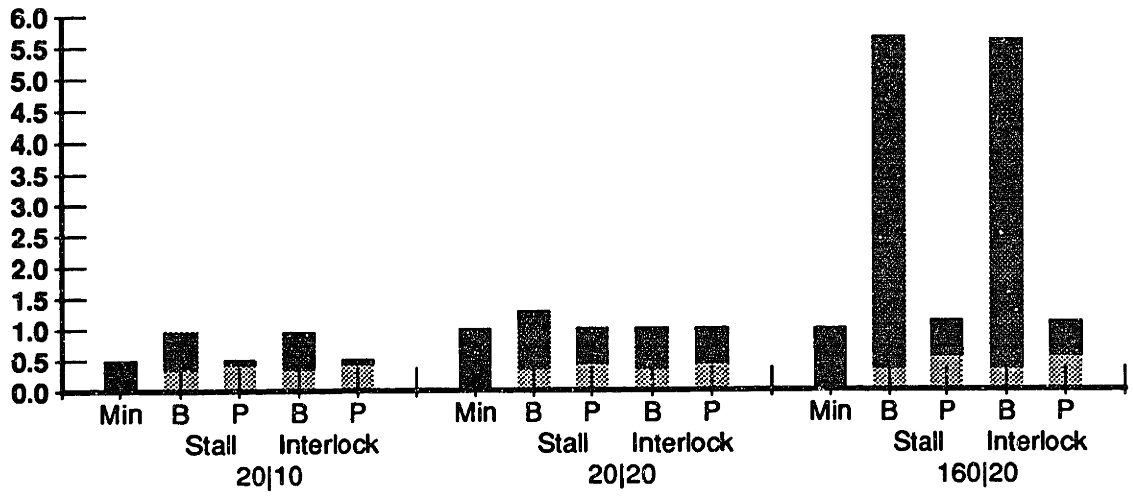


Figure 6-5: Matrix300 **FA** Performance

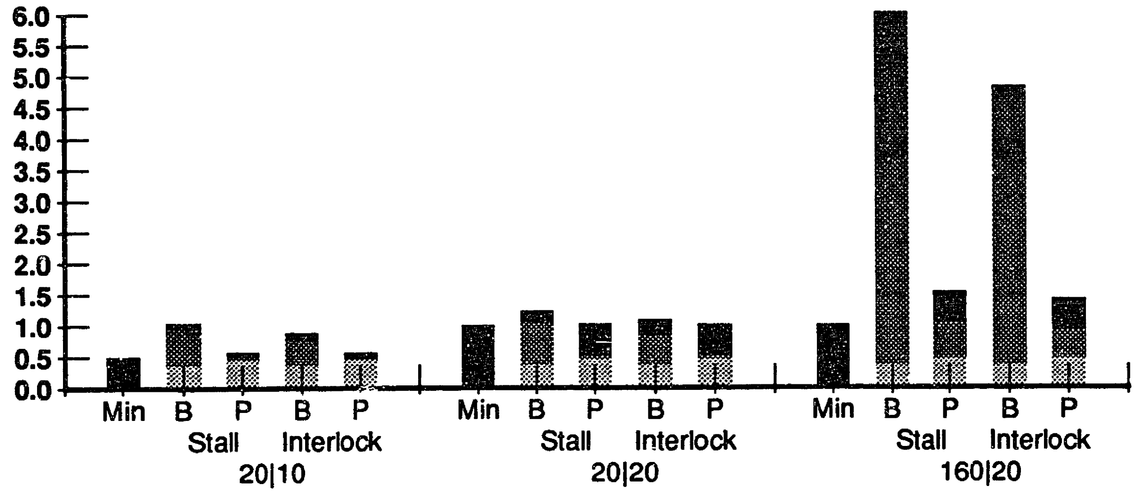


Figure 6-6: Nasa7 **FA** Performance

matic for both the memory bandwidth and instruction limited benchmarks than speedups under the **Stall** model. The memory interface of the **Interlock** model is already able to exploit some processor/memory concurrency which is only available through prefetching under the **Stall** model. In particular, the **Interlock** model provides write buffering and does not stall on loads until data is used. Speedup is still exhibited by most benchmarks. Excluding matrix300, speedups range from 8 to 10% at the bottom end to 25% at the top end. The performance difference between the **Stall** and **Interlock** models for prefetched code is generally in the neighborhood of 2 to 3%, whereas the difference in performance of the unoptimized code on the two models is more like 5 to 10%. Code compiled without any particular attention to latency tolerance thus has some intrinsic level of tolerance given a very unrestrictive memory interface. Even under such an interface, scheduling specifically addressing latency tolerance has the ability to improve performance by an appreciable fraction.

In the case of matrix300 no speedup is achieved for optimized code under the **Interlock** model. The unoptimized code for matrix300 runs 27% faster under the **Interlock** model than the **Stall**

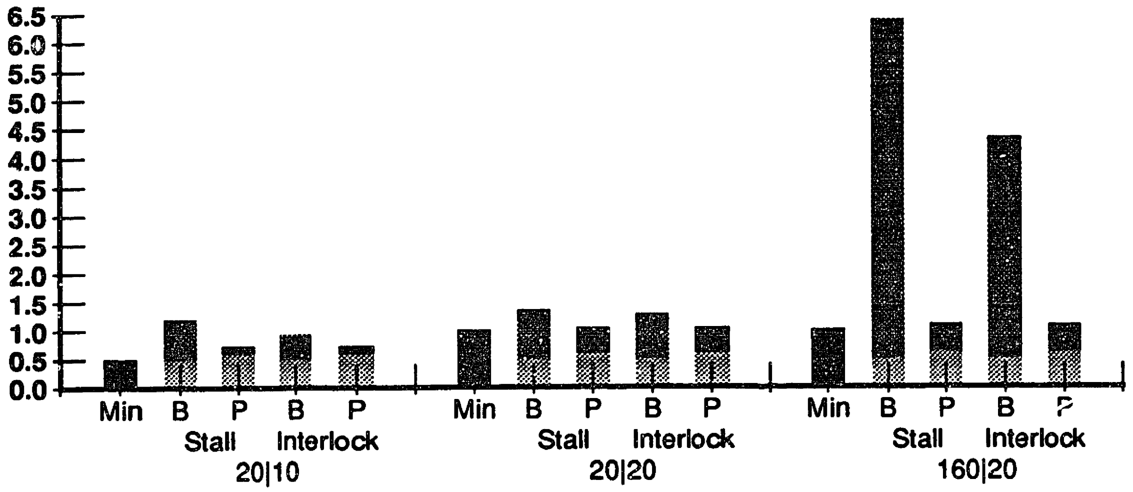


Figure 6-7: Tomcatv **FA** Performance

Speedups for Benchmarks Under FA Model						
Benchmark	20 10	20 10	20 20	20 20	20 160	20 160
	Stall	Intlck	Stall	Intlck	Stall	Intlck
doduc	1.27	1.16	1.17	1.13	2.43	1.90
eqntott	1.17	1.12	1.16	1.11	2.96	2.71
espresso	1.31	1.25	1.12	1.11	2.28	2.24
fpppp	1.11	1.07	1.10	1.10	1.88	1.82
matrix300	1.92	1.88	1.27	1.00	5.06	5.06
nasa7	1.82	1.59	1.21	1.08	3.92	3.43
spice			1.29	1.25		
tomcatv	1.65	1.28	1.30	1.23	5.85	4.02

Table 6-6: Ratios of Benchmark Speedups

model and reaches the memory bandwidth imposed performance limit. Thus there is no chance for prefetching to improve the code.

The data presented for the **Interlock** model may slightly underestimate the potential performance improvements available through prefetching. Since the model includes write buffering there is little potential for performance improvement associated with prefetching for writes. The only performance gain can arise from achieving more uniform bandwidth utilization. C-flat has an option to suppress all prefetching for writes, however, this is not enabled for the data above. The two interface models can be simulated simultaneously when they use the same code. In order to avoid multiple simulations, code specifically optimized for the **Stall** model by performing prefetches for writes is used to gather the data for both the **Stall** and **Interlock** models.

While speedup for optimized code for the 20/20 memory system is noticable, the speedups associated with the other memory configurations are very dramatic. In each of these models, the memory system concurrency exceeds one. Code optimized using latency tolerance transformations is much more capable of utilizing this added bandwidth.

Under the **Stall** model using the 20/10 model, the memory bandwidth limited benchmarks show speedups of between 65 and 90%. The only mechanism by which the **Stall** model can exploit memory concurrency in the absence of prefetching is by overlapping writeback transactions with computation. Using prefetching, all available memory bandwidth can potentially be utilized. This leads to a speedup potential for the optimized code of as much as a factor of two. This speedup is nearly achieved for several of the memory bandwidth limited benchmarks. For the instruction limited benchmarks the speedups under the 20/10 model exceed their corresponding 20/20 values. Just as with the bandwidth limited benchmarks, the added bandwidth can be better exploited by the optimized code.

The data gathered using the 20/10 model can be used to test the hypothesis above regarding lower than anticipated performance improvements for miss scheduled benchmarks like doduc, fpppp and espresso. It is theorized that writeback bandwidth is responsible for a large component of latency stalls in optimized code by producing latency tolerance shortfalls, *i.e.* prefetched references used before they arrive. Under the 20/10 memory model, additional bandwidth exists which can be used to perform these writebacks. In the data for doduc under the **Stall** model, latency stall cycles for the optimized code measure 24% of those for the unoptimized code, precisely matching the fraction computed above based on miss coverage and expected latency tolerance shortfall.

The real test of latency tolerance in code comes from the 160/20 model. Latency is increased by a factor of eight at constant bandwidth, leading to a memory concurrency of eight. Data for this memory system serves both to better differentiate the performance of the **Stall** and **Interlock** models for unmodified code and to highlight the benefits of latency tolerance optimizations.

The performance difference for the **Stall** and **Interlock** models using unmodified code is much more pronounced for the 160/20 memory system. For doduc, latency stalls under the **Stall** model exceed those for the **Interlock** model by over 60%. For other benchmarks the difference is typically close to 20% and is 0 for matrix300. Performance improvements for this memory system with high latency and high concurrency result when multiple misses can be satisfied concurrently. Intrinsic ability to exploit miss processing concurrency arises from two sources. Buffered writes which produce misses can potentially be overlapped with following transactions. Additionally, load operations sometimes occur in pairs followed by a binary operation combining their data. In this case, if both loads produce misses, they can be concurrently serviced. Based on empirical observations of code, one finds that it is relatively rare for many more than two load references to occur in series without the data for some load being used in a computation. This would suggest a limit on the maximum intrinisic capability to exploit memory system concurrency in codes at a level of about two. Comparison of **Stall** and **Interlock** data for the 160/20 memory system show that in general even a memory concurrency of two cannot be effectively exploited by unmodified code, even with a highly flexible interface. Intrinsic latency tolerance of unmodified code is thus quite limited in the context of very high latencies.

In striking contrast, code which is appropriately transformed exhibits substantial latency tolerance. Speedups between unoptimized and optimized code for the 160/20 memory system range from about a factor of two to almost a factor of six. The smallest speedup is exhibited by fpppp. This is attributable to the relatively large number of unoptimized misses which occur in this benchmark, about 30%. Speedups in the range of 5 to 6 are exhibited by several benchmarks.

Even more telling, as far as latency tolerance is concerned, is the relative performance of optimized code under the 160/20 memory model and unmodified code under the 20/20 model. For several of the benchmarks including eqntott, matrix300 and tomcatv, optimized code run using the 160 cycle memory actually outperforms unmodified code for the 20 cycle memory. Doduc and espresso require only about 25 to 33% more time for optimized code to run using the high latency memory system. The worst performer, fpppp, which leaves about 30% of misses unscheduled requires 40 to 50% more runtime. This data clearly shows that for many programs, compilation-based tolerance to even very high memory latencies is feasible and highly effective, given that memory concurrency is used to provide adequate memory bandwidth.

6.3.5 DM Data

Data gathered using the **DM** cache model is presented in figures below.

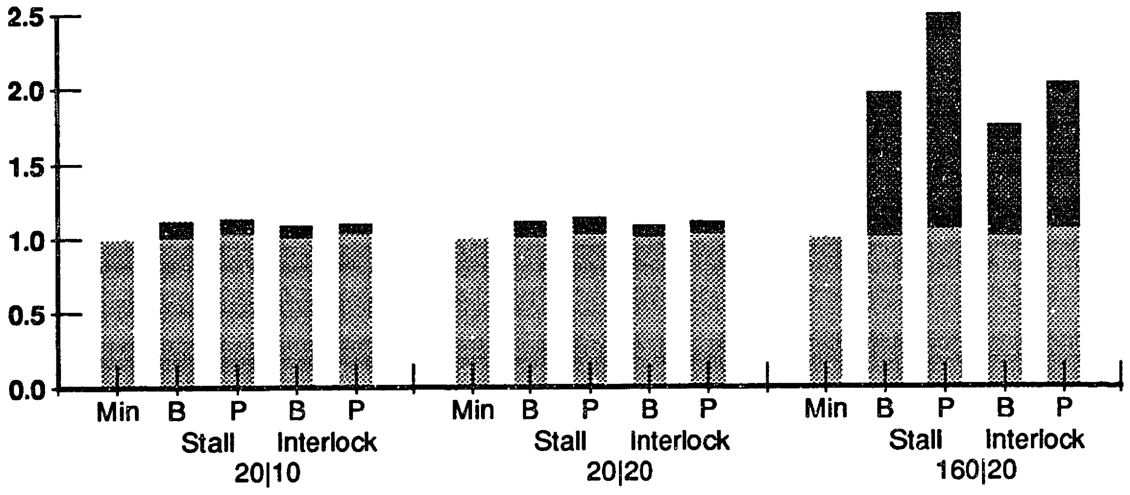


Figure 6-8: Doduc **DM** Performance

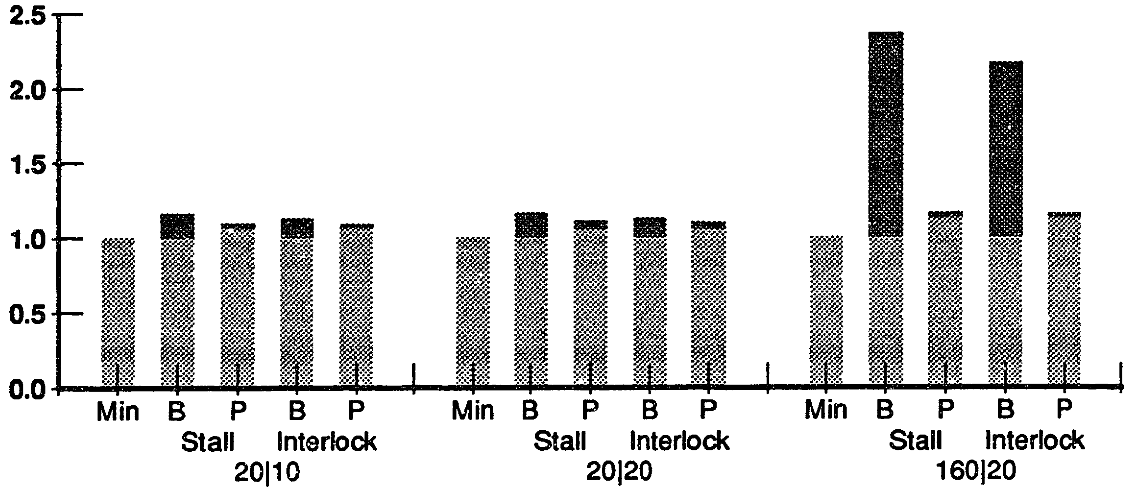


Figure 6-9: Eqntott **DM** Performance

Performance data gathered for the **DM** cache model does not show the same improvements as that of the **FA** cache model. Several factors contribute to this behavior. The cache in the **DM** model is substantially larger than that in the **FA** model and has a larger block size. As a consequence of these changes, most benchmarks exhibit higher hit rates, some dramatically so. Higher hit rates decrease the relative component of performance associated with miss processing latency. Also, since the memory system is parameterized in terms of the time to service a miss transaction and miss transactions move cache blocks, the increase in block size results in increased memory bandwidth. Decreasing the pressure on memory bandwidth also serves to decrease the potential benefits of prefetching since prefetching allows more efficient utilization of memory bandwidth.

These two factors account for some performance difference but they do not explain the behavior of several benchmarks whose performance is actually degraded by prefetching in the data above. In these benchmarks a large number of required misses remain after prefetch optimization. These unoptimized misses are investigated further in coming sections.

Despite the poor behavior of some benchmarks, other benchmarks exhibit speedup under the **DM**

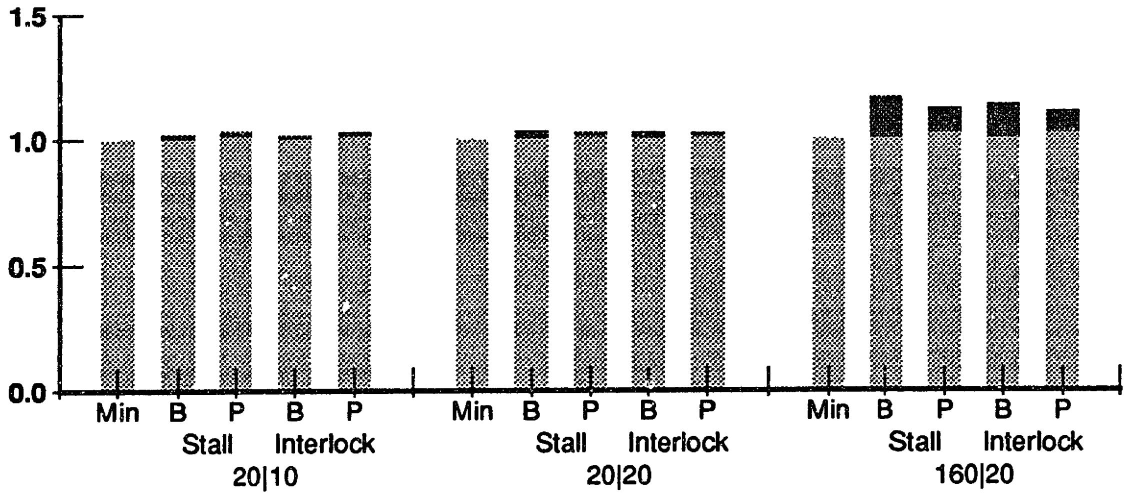


Figure 6-10: Espresso DM Performance

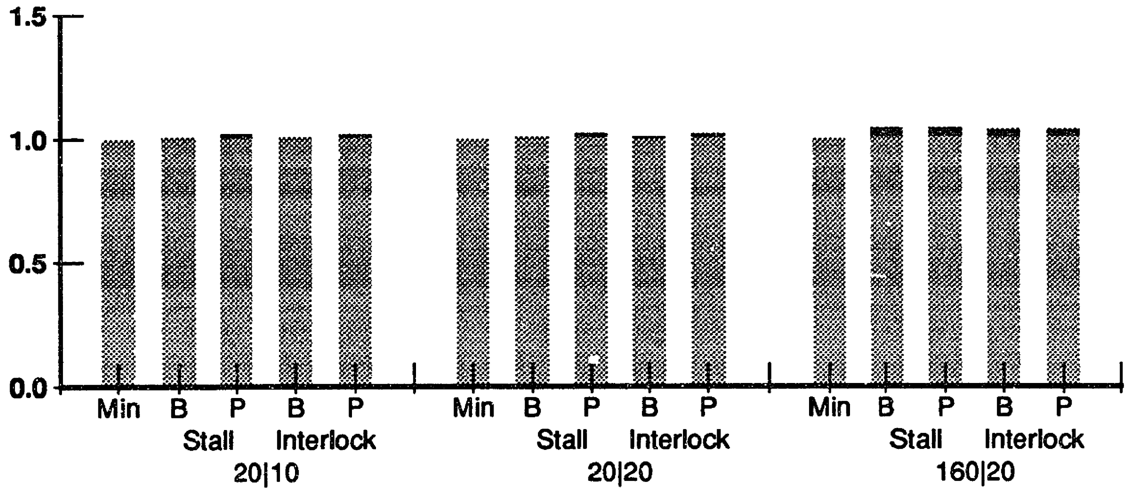


Figure 6-11: Fpppp DM Performance

model. Matrix300 exhibits speedups of 28% for models with a memory latency of 20 cycles and speedups of a factor of 3 for 160 cycle latency models. Tomcatv exhibits speedups of 10% to 17% for latency 20 models and 1.5 to 1.8 for latency 160 models.

6.3.5.1 Cache Pollution

Under the DM model, modified code produced by c-flat is slower than unmodified code for several benchmarks. The modified code for these benchmarks exhibits a large number of required misses, in some cases only slightly less than the number exhibited by unmodified code. This code also generates substantially more memory traffic than the unmodified code. One might first be tempted to point the finger of blame for these required misses at cache pollution. In the DM tests, data is prefetched directly into the cache. As soon as a prefetch is initiated, some block of data is removed from the cache. Once prefetched data arrives it is forwarded to the emptied cache block. One can certainly imagine hardware schemes in which old data is not replaced until new data arrives, but this is not the scheme simulated in gathering the DM data. Cache pollution signifies added misses

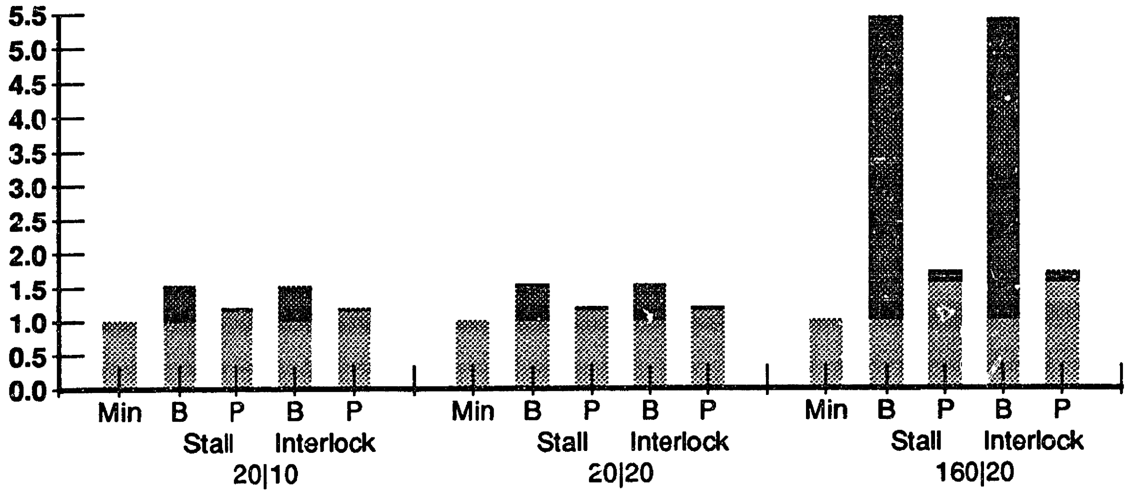


Figure 6-12: Matrix300 DM Performance

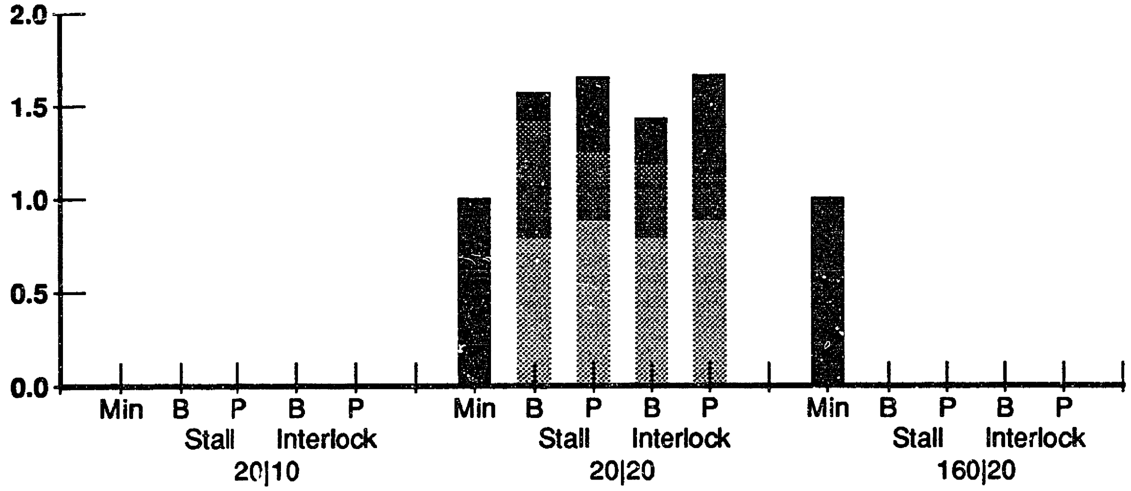


Figure 6-13: Nasa7 DM Performance

incurred due to the fact that prefetch references replace data blocks within the cache sooner than they would normally be replaced in unmodified code.

Consider the set of prefetch operations which trigger miss transactions. In analyzing cache pollution, one only needs to consider prefetches which actually produce misses since others do not replace cache blocks. One can subdivide those prefetches moving data in two different ways. First, divide the prefetches into two sets, which will be termed matched and unmatched prefetches. Each prefetch operation is associated with some potential future instance of a badref. Matched prefetches describe those prefetches for which the associated badref reference actually occurs at runtime. Unmatched prefetches are speculative references for which the associated required reference does not occur. Second, characterize prefetches as beneficial and detrimental. A beneficial prefetch occurs when a prefetched value is used by a required reference before being displaced from the cache. A detrimental prefetch occurs when a prefetched block is displaced prior to being used by a required reference.

Armed with this terminology, consider the circumstances under which a matched prefetch is rendered either detrimental or beneficial. Prior to the prefetch and subsequent to the matching reference,

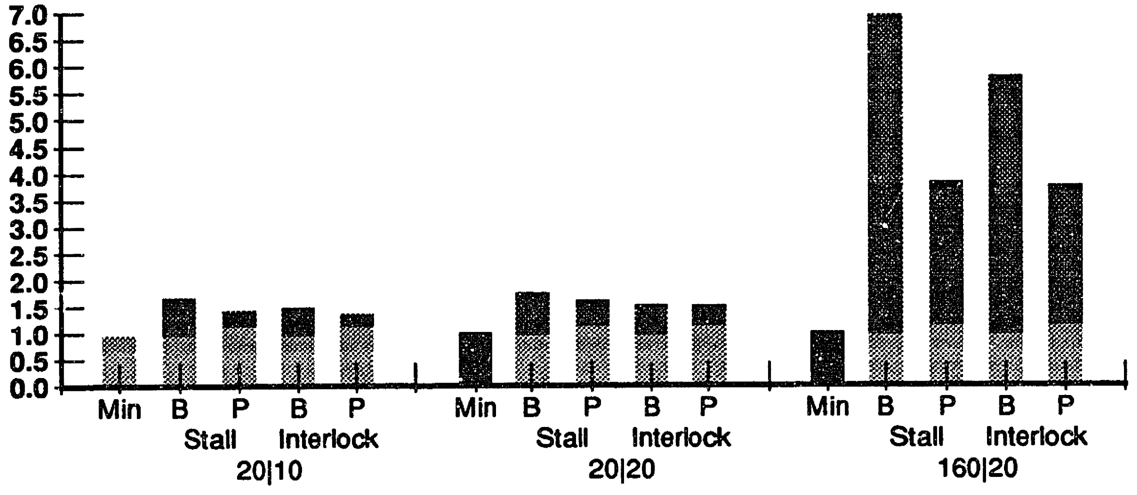


Figure 6-14: Tomcatv DM Performance

the cache contents are the same in the presence or absence of the prefetch. Thus the window of vulnerability for a matched prefetch to be detrimental is the period between the prefetch and matching reference. The duration of this window in time is determined by scheduling heuristics. These heuristics attempt to make this window as close to the memory latency as possible, while erring on the high side. For purposes of cache pollution, this window must be measured in terms of memory references, rather than cycles. One might suspect that the number of memory references in the window would be somewhere between 1/5 and 1/3 of the window size in cycles. Using a window size of 160 cycles and a ratio of 1/4 for the fraction of memory reference instructions in the window, one might expect somewhere in the neighborhood of 40 memory references. At a window size of 20 cycles, this number is only 5 or 6. The DM cache model holds 16K blocks, thus one might expect about $40/16K$ or one in four hundred matched prefetches which move data to be detrimental in the case of the 160/20 memory system. In the event of a detrimental prefetch, one miss may be turned into three, one for the prefetch, one for the intervening reference which might have been a hit otherwise and one for the matching reference. Tight interference patterns can produce badrefs, leading to the behavior above. Nonetheless, one would expect relatively few matched prefetches to be detrimental.

Next consider unmatched prefetches. The window of vulnerability for an unmatched prefetch to be detrimental is not bounded at a number close to the memory latency since there is no matching required reference to close this window. It is fairly likely that a block fetched by an unmatched prefetch is displaced before it is used, resulting in a detrimental prefetch. Furthermore, if it is displaced by the block that it originally displaced, this adds an extra miss for a required reference. One might expect that most detrimental prefetches are associated with unmatched rather than matched prefetches.

Detrimental prefetches can be easily measured in our simulation technique by marking each cache block brought into the cache by a prefetch instruction, unmarking any block servicing a hit by a required reference and monitoring the status of these marks in blocks replaced in the cache. Furthermore, in the context of direct-mapped caches without victim caches, detrimental prefetches represent an upper bound on cache pollution due to prefetching. If a prefetch is not replaced prior to use of the prefetched block by some required reference it is guaranteed that the block replaced by the prefetch is not used during this time either, in which case prefetching has not produced cache pollution.

Detrimental prefetches are measured for a variety of cache configurations in two of the benchmarks exhibiting poor performance under the DM model, doduc and tomcatv. These configurations include both the **DM** model and two variants of the **DMV** model. In the **DMV** model, since the cache system consists of two components, a main cache and a victim cache, one has the option of using either component as the buffering storage for prefetches. In these tests, for configurations with victim caches all detrimental prefetches are eliminated, independent of whether prefetching uses the main or victim cache for buffering storage. For the standard **DM** cache without a victim cache, detrimental prefetches occurred over 1000 times less often than required misses. Although this data is limited to two benchmarks, based on the data one would conclude, particularly in the context of victim caching, that cache pollution does not represent a serious limitation for prefetching.

This conclusion, however, leaves the large quantity of required misses in the **DM** data presented above unexplained.

6.3.5.2 Memory Behavior Perturbation

An alternative explanation for the behavior exhibited in the **DM** data above is that the code transformations applied in latency tolerance optimization perturb memory behavior sufficiently to change the identity of goodrefs and badrefs. Examples of the potential for such behavior were described in Chapter 5 in the context of the prefetch schemas for indirection array and linked-list RAADs. In the case of indirection arrays such as `X[Y[i]]` the prefetching schema produces a new reference to `Y[i+1]` which can be a badref even if the original reference to `Y[i]` is not. A simple, systematic modification to memory reference behavior such as this is inconsistent with the fact that poor performance is only associated with the **DM** model and not the **FA** model. However, one can identify other forms of memory behavioral modification which would be much more prone to occur in the **DM** model.

Before discussing these problems, it is worthwhile to note that memory behavior perturbation is indeed the explanation for the poor performance of several of the benchmarks in the **DM** tests, exemplified by doduc. Badref references exist in modified code for these benchmarks whose corresponding references in unmodified code are not badrefs. Data is not presented to support this claim because it was verified by hand. Performance diagnostic tools developed in conjunction with the compiler prototype and simulation environment provide the ability to trace the identity of particular badref memory references both to their intermediate structure in the compiler and to their orignating constructs in the source code. Using these tools, the performance of remaining badrefs in modified code was compared with the performance of corresponding references in the unmodified code. To a large degree, badrefs in the optimized code were not present in the original code.

Latency tolerance transformations can perturb the memory behavior of individual static memory references through a number of different mechanisms. In the case of multiword data structures with some particular alignment to cache block boundaries, if these data structures are shifted with respect to block boundaries, this could lead to the swapping of identities between goodref and badref accesses to structure components. This problem would primarily be associated with dynamically allocated data on the stack or allocated using memory allocation routines. By restricting allocation routines to supply cache block aligned data and by adding alignment restrictions on stack frames, memory behavior perturbation arising from block alignment issues can be minimized.

New badrefs observed in the experimental data are primarily associated with statically allocated data structures which are not moved by latency tolerance optimization. Thus although memory behavior can be changed by shifts in data structure alignment, this is not the relevant explanation for the benchmark data.

Having ruled out detrimental prefetches and block alignment shifts, there is at least one more way in which the latency tolerance optimizations in c-flat can impact memory performance. Miss scheduling, and to a lesser extent RAAD prefetching, can change memory behavior by changing register allocation and memory references associated with register spills. Miss scheduling tends to increase the lifetimes of register variables. Additionally, when transformations are performed using copying, new temporary variables are creating, increasing the need for storage. One might hope that these impacts would be relatively minimal but it appears that they are the source of the misses in our test data.

Several factors combine to allow this spill phenomenon to be significant. Take the doduc benchmark as an example. First, for the given cache configuration, the program demonstrates a fairly high aggregate hit rate, 98.5%. As a consequence of the high hit rate and direct-mapped cache, new references targetting formerly unaccessed memory addresses are reasonably likely to replace useful data in the cache. When a newly referenced address results in such a replacement, this really produces two new misses since the new reference misses itself and turns some future reference into a miss. This miss magnification occurs because the cache is direct-mapped. An added consequence of the high hit rate is that the addition of a small number of misses relative to the total number of references potentially represents a large fractional increase in the number of misses.

Required misses in the **DM** test may in large part be an artifact of the prototype compiler. For large procedures, particularly those with very large basic blocks, the compiler generates a significant amount of register spill traffic. Furthermore, it is inefficient in the use of spill slots allocated on the stack. New spill slots are allocated for most spills produced. Very large procedures with large basic blocks produce many register spills and more importantly a correspondingly large number of spill slots. Miss scheduling, by creating new temporary variables and increasing register lifetimes, increases the number of spill slots. The new spill slots cause interference misses with other cached values, producing the observed behavior.

Owing to their large basic block sizes and large procedure sizes, doduc, fpppp, and tomcatv are particularly vulnerable to this spill slot induced cache pollution. The problem might well be substantially lessened in a compiler which more efficiently used stack slots. As an alternative, if a direct-mapped cache is targetted and some amount of spill-slot inflation is anticipated, extra spill slots could be generated and accessed in the code used to calibrate memory performance of static instructions in order to partially compensate for this effect.

6.4. Summary

This chapter presents data gathered using the prototype compiler c-flat and a timing simulation environment. This data takes two main forms, statically produced data from the compiler and dynamically produced data from the simulation environment.

The static data identifies the form of badrefs identified in benchmarks. It also shows the level of miss coverage attainable using either or both of the scheduling techniques implemented and the average associated overheads. This data indicates that in about half the benchmarks it is sufficient to focus solely on scheduling array RAADs. Furthermore, in no benchmarks examined are indirection array RAADs or linked-list RAADs particularly relevant. One might expect to find more frequent occurances of these non-array data structures in other kinds of programs. The overhead costs associated with prefetching for RAADs, even in our fairly primitive compiler implementation, is only in the range of 1.5 to 2.5 on average.

RAAD prefetching is not applicable in about half the benchmarks. The typical reason for this is that looping constructs are separated from RAAD references by procedure boundaries. In these cases miss scheduling is applicable. Overheads are a little higher and miss coverage is not quite as good for miss scheduling.

Dynamic data based on the **FA** cache model exhibits very encouraging performance. Speedups vary from 10 to 30% under the moderate memory latency in the baseline model. Under a high latency memory model, corresponding to a very fast processor, speedups as high as a factor of 6 are exhibited. Compiler optimization can yield latency tolerant code.

Data based on the **DM** cache model shows more mixed results. While some benchmarks exhibit improved performance, others are actually degraded by compiler transformation. Several possible sources of this performance degradation are examined. Empirical data indicates that cache pollution occuring specifically as a result of prefetch operations does not significantly impact memory behavior. The source of performance degradation in the data is isolated to cache interference resulting from increased spill code which perturbs goodref/badref behavior for some references. We speculate that this problem could be somewhat lessened in a compiler which reused stack spill slots in order to minimize the number required. Thus the **DM** data may be a somewhat anomolous result. Nonetheless, it indicates an issue for concern in further implementations of latency tolerant compilers.

Chapter 7

Conclusions

Latencies associated with data cache misses are a potential performance limitation in computer systems. Technological trends indicate that the ratio between memory latency and insruction cycle time is increasing with time, leading to an increasing memory latency problem in the future. The problem of efficient computation in the face of high memory latency can be addressed through compiler-managed concurrency, overlapping one or more miss transactions with non-miss processing. Compilation-based prefetching using explicit prefetch instructions is a means for achieving this concurrency.

7.1. Summary

Study of the locality and parallelism behavior exhibited by programs leads to insights which can be exploited for memory latency tolerance. Three related but distinct aspects of the locality behavior of programs are highly relevant to latency tolerance.

Locality behavior is statically correlated. Individual static memory references within programs each display their own characteristic hit/miss behavior. Similar behavior is displayed by dynamic references produced by the same static reference both within the same program run and across multiple program runs.

The locality behavior exhibited by static references is polarized. Program structure dictates that many static references produce essentially no misses. Other static references exhibit characteristic miss rates greatly in excess of the program average. Static references exhibiting miss rates comparable to the program average are rare in comparison to high and low miss rate references.

While static references fall into two sets, goodrefs, with low miss rates and badrefs, with high miss rates, these sets do not account for equal fractions of either dynamic references or dynamic misses exhibited by programs. Dynamic references occur predominantly in conjunction with goodrefs and dynamic misses occur predominantly in conjunction with badrefs.

Based on these observations regarding locality one can develop an appropriate metric for characterizing the form of program parallelism which can be exploited for latency tolerance. Parallelism suitable for the concurrency which leads to latency tolerance takes the form of sets of instructions in the neighborhood of badref instances which are independent of the badref, although potentially dependent upon one another. These instructions can and will include both arithmetic instructions and goodrefs. Programs exhibit sufficient parallelism of this form to tolerate memory latencies of several hundred processor cycles.

Compilation algorithms exploiting static locality behavior can access the inherent parallelism in programs for latency tolerance. Software pipelining can be applied to badrefs with predictable addresses which occur in loops by using prefetch instructions to initiate miss transactions for cache blocks needed in future iterations. Miss scheduling can insert prefetches for badrefs in sequential code. In conjunction, these two techniques allow latency tolerance transformations to be applied to most badrefs in programs.

Mechanicially generated code produced by c-flat, our prototype compiler implementation, demonstrates that compilation-based prefetching is both feasible and effective. Using a fully-associative cache model, code exhibits speedups of 10 to 30% for moderate memory latency assumptions and factors of 2 to 6 for very high levels of relative latency which would occur in conjunction with a very high-performance processor.

7.2. Program Behavior

The key to successful implementation of software latency tolerance is an appreciation of significant, relevant aspects of the locality and parallelism behavior of programs. Static locality phenomena serve to make software latency tolerance feasible by limiting the scope of the problem to badrefs. Analysis of potential limitations on parallelism leads to techniques, such as speculative prefetch initiation, which can relax away unnecessary constraints.

7.2.1 Static Locality Correlation

While the behavior of hierarchical memory systems is typically considered at a very coarse granularity, namely aggregate system performance, we study this behavior at a finer granularity. In particular, we examine the cache hit/miss performance of sets of dynamic memory references arising from the same static memory reference instructions in programs. This study leads to three related but distinct observations about memory performance.

The first behavioral phenomenon is captured in the **Static Locality Correlation Hypothesis.** Dynamic memory references arising from the same static reference exhibit correlated hit/miss behavior. Thus the set of dynamic references produced by a static memory reference may show a predisposition to miss more or less frequently than the program average. Since this predispositionn arises as a result of program structure, it occurs consistently across different program runs in which input variations are not dramatic.

The second observation is that the miss rates exhibited by different static instructions tend to be highly polarized with respect to the average miss rate of a program. This behavior is extremely pronounced in some benchmarks in which essentially all static references exhibit miss rates of either 0 or 1. While not all benchmark data exhibits this ideal polarization, all data displays substantial polarization. In no case are miss rates for static reference clustered near the program average.

The final observation regarding static hit/miss behavior concerns the actual distributions of dynamic references and dynamic misses with respect to high and low miss rate, static references. All benchmarks exhibit static references with miss rates at or very nearly 0. These zero miss rate goodrefs account for at least half and sometimes more than 90% of total dynamic memory references in all the benchmarks except one. These references cannot produce any misses, however, since they have zero miss rates. Misses are predominantly associated with high miss rate static references, *i.e.* badrefs.

The behavior which results from these observations is summarized in the **Badref Hypothesis.** Badrefs, a subset of static references exhibiting miss rates much higher than the program average, produce the majority of program misses and relatively few dynamic references. Goodrefs, references with low miss rates, account for the predominance of dynamic memory references but few misses.

These memory behavioral phenomena are real and they are important. We provide empirical evidence supporting this behavior across a range of benchmarks under a variety of memory configurations. Programs exhibit static locality correlation and badref behavior.

7.2.2 Implications of Static Locality Correlation and Badref Behavior

Static locality correlation and badref behavior have important implications for software latency tolerance. Misses produced by programs can largely be isolated to an identifiable subset of static memory references. They do not occur randomly in conjunction with all static memory references. They occur primarily as a result of execution of badrefs, a subset of static references, and occur in this context with a frequency much higher than the average program miss rate. Using this insight, the problem of latency tolerance is shifted from providing tolerance to unlikely events in a wide context to tolerating likely events in a narrow context. The latter form of problem is by far the one to be preferred.

Badref behavior is exploited within a compiler by partitioning static references into goodref and badref sets. Parallelism, a limited resource for latency tolerance, can then be focused specifically on badrefs. Furthermore, goodrefs now become a component of this parallelism. They become part of the solution rather than part of the problem. Realistically, software, latency tolerance is impractical without partitioning static memory references based on static locality correlation and badref behavior. Owing to the fortunate existence of this behavior, software latency tolerance becomes practical.

Given suitable hardware and software mechanisms, the same form of simplification which can be applied to software, memory latency tolerance might also be applied in the context of some forms of hardware, latency-tolerance techniques. One might envision using two different forms of memory reference instructions which carry non-semantic software hints about the expected cache performance of the reference. Such a scheme could convey goodref/badref information to the hardware. Through such a mechanism, hardware scheduling techniques involving either single or multiple threads might be able to reap some benefit from static locality correlation. In the context of single-threaded, hardware schedulers, this information might be used to elevate the priority of badref operations and their address dependent predecessors in order to exploit some level of reverse parallelism. In the context of multiple-threaded processors, this information might be used to start fetching and executing instructions in a new thread upon fetching a badref memory reference. Such a mechanism could decrease pipeline fill and drain costs associated with changing threads.

We believe that the principle means by which static locality and badref behavior can be exploited is through code explicitly scheduled by compilers for latency tolerance. This code can use either prefetch instructions or non-blocking loads to initiate cache misses allowing the explicit scheduling of processor/memory and memory/memory concurrency. Nonetheless, dissemenation of the insights above regarding memory behavior may well lead to architectural innovations beyond those currently anticipated.

7.2.3 Parallelism

With respect to software latency tolerance, reference locality behavior is only half the battle. Sufficient parallelism must exist and be identifiable to a compiler to allow the scheduling of concurrency resulting in latency tolerance. As with locality behavior, a focused study of the problem results in insights.

One finds that average parallelism is not a relevant metric for assessing the form of parallelism needed for latency tolerance. Software latency tolerance should not be dismissed on the basis of relatively low levels of instruction-level parallelism cited in the literature. Latency tolerance is not achieved by emitting a large number of instructions in parallel. Instead, it is achieved by emitting a long series of instructions in conjunction with servicing one or more slow miss transactions.

One can and should apply the results above concerning static locality and badref behavior to the search for parallelism. In order to achieve latency tolerance, it is not necessary to identify concurrency suporting parallelism for all memory references, just those which are badrefs. In point of fact, parallelism is only needed for those instances of badrefs actually resulting in cache misses at runtime, but software scheduling is incapable of exploiting this. Focusing then on all dynamic badref occurances, in order to support the desired concurrency, a set of non-dependent instructions must exist which can be grouped with each dynamically occurring badref reference. These grouped instructions must be independent of the badref but need not be independent of one another since they can execute as a sequential stream concurrently with the badref miss transaction. Finally, this set can and will include goodref memory references. If such a set can be identified for each badref and static code can be devised in which these instructions can overlap badref miss transactions, then latency tolerance can be achieved.

While reference partitioning based on badref behavior provides essential simplifications with respect to finding parallelism, additional aspects of the problem provide further simplification. In particular, the fact that transactions between a cache and the underlying memory system are not semantically significant with respect to the processor can be beneficially exploited. The initiation of a cache/memory transaction is not meaningful to the processor and as a consequence does not have dependencies with any instructions. Only the termination of such a transaction, in which transferred data is used or a transferred block is modified has semantic meaning. With an oracle to predict addresses, blocks could be transferred at will. Hardware prefetchers attempt to act as such an oracle. Using a software latency technique the addresses for transferred blocks are provided by the software, thus producing some dependencies on the initiation side of miss transactions. These dependencies are restricted solely to instructions required for address generation, however.

Although few actual dependencies constrain miss transaction initiation at runtime, some amount of effort is required to relax away non-essential dependencies during code scheduling. This process is greatly facilitated through a safe mechanism for speculatively initiating miss transactions. Safe, in this case, specifically refers to a miss initiation mechanism which does not produce faults or similar behavior in conjunction with the use of invalid or unmapped addresses. A safe, speculative, prefetching mechanism, in conjunction with code copying and storage replication when necessary, allows code motion of miss initiation across both basic-block boundaries and also memory stores which cannot be disambiguated with respect to loads used in the prefetch address computation.

7.2.4 Parallelism Measurement

We measure potential parallelism for latency tolerance using a technique which counts independent instructions in the neighborhood of badrefs. Dependencies which do not arise directly from dataflow constraints for the transaction address or the use of transferred data are relaxed away. Relaxation of these constraints is justified based on the fact that code techniques including speculative prefetching, storage replication and code copying provide a way to relax away unnecessary constraints in scheduling algorithms. Using this measurement scheme, substantial parallelism is indicated. Also measured is potential parallelism for latency tolerance within the same basic blocks as badrefs. In many benchmarks this is very small as might be expected. Somewhat surprisingly, several benchmarks exhibit very large basic blocks and thus a reasonable fraction of badrefs with high, basic-block parallelism measurements. Even in these benchmarks, a large fraction of badrefs still occur in small basic blocks. Thus interblock scheduling techniques are necessary for tolerance of significant memory latency.

On balance, parallelism data collected leads primarily to one main conclusion. Lack of parallelism should not be a substantial limitation for tolerance to even relatively large memory latencies, assuming sufficient miss processing bandwidth is provided. Programs exhibit sufficient, inherent parallelism. Whether this parallelism can effectively be exploited through code scheduling is a different question, but the parallelism does exist.

7.2.5 Implications of Forward vs Reverse Parallelism

Measured parallelism is classified as forward and reverse parallelism. The distinction between these two forms of parallelism lies in the relative position of independent instructions with respect to the original badref reference in a sequential program schedule. Reverse parallelism describes independent instructions identified by moving backward in time, *i.e.* instructions preceding the badref in a sequential schedule. Forward parallelism describes independent instructions identified by moving forward in time, instructions following the badref in a sequential schedule. The principle reason for distinguishing these two forms of parallelism is that single-threaded hardware scheduling techniques are more able to exploit forward parallelism than reverse parallelism. Software scheduling techniques, particularly those employing speculative transaction initiation, can more easily exploit reverse parallelism. Data indicates that reverse parallelism exceeds forward parallelism. This is particularly true for instructions directly adjacent to badref loads. These instructions frequently use the result of the load as an operand and are themselves used by successive instructions. As a consequence, the least fruitful place to look for instructions which are independent of a badref load is in a short window directly following the operation in a sequential schedule. Unfortunately, this is the principle area in which single-threaded, hardware, dynamic schedulers are forced to seek parallelism.

7.2.6 Badref Distribution

Additional issues potentially limiting successful tolerance of badref miss latency concern the placement and interaction of adjacent dynamic badref instances. Memory bandwidth and concurrency limits the number of miss transactions which can be simultaneously active. If misses due to badrefs are highly clustered in sequential schedules then, in order to achieve latency tolerance, code transformations must be applied to spread these references out in time. Address dependencies between adjacent badrefs prevent the exploitation of concurrency between resulting misses.

Address dependencies between badrefs occur when the address computation for one badref uses a value loaded by a prior badref. Benchmark data indicates that address dependencies between badrefs are very rare. Dependencies between badrefs will not significantly limit overlap of miss transactions in memory systems offering miss processing concurrency.

Benchmark data also indicates that badrefs are distributed relatively smoothly in time. The data regarding badref occurances does not necessarily show that misses are not clustered. If a series of several adjacent badrefs whch all produce misses were followed by a series which all produce hits, one would find that misses would be clustered relative to the average intermiss spacing. Since not all badrefs produce misses, the average spacing between badrefs is smaller than the average spacing between misses. Data indicates that badrefs, representing potential misses, occur with a relatively even distribution in time.

7.3. Compiler Issues

Identifying and publicizing the locality and parallelism behavioral phenomena above represent a contribution with some potential impact. These insights are a fortunate derivative of the study of compilation for latency tolerance. While it may be of less general significance, with respect to the goal of compilation-based latency tolerance, the proof is in the implementation. C-flat is a real compiler exploiting badref behavior and prefetch mechanisms to mechanically generate latency-tolerant code.

7.3.1 Compiler Memory Modelling

The first challenge in a compiler for latency-tolerant code is a technique for modelling memory latency. This modelling dilemma is solved by the **Badref Model**, in which memory references are partitioned based on their individual locality behavior. In order to perform badref/goodref partitioning, miss behavior estimates are required for all static references. These estimates can be produced through program analysis or through simulation and measurement. C-flat adopts the latter approach. While somewhat less elegant, this approach is easily implemented and can be applied in a general context. Precise static analysis would appear to be an intractable problem except in the case of highly structured loops with known limits. We do observe that many goodrefs may be identified relatively easily as they occur in close conjunction to other references using the same address. Coarse partitioning, separating obvious goodrefs from other static references, may represent a good start in memory behavior modelling.

7.3.2 Scheduling Algorithms

In many programs most badrefs occur in conjunction with data structure traversal, particularly array traversal, within explicit loops. RAAD prefetching applies the technique of software pipelining to miss latency for badrefs in loops. RAAD prefetching uses explicit prefetching to schedule miss transactions in the current iteration for badref instances in future iterations, thus avoiding any latency penalty when the badrefs are eventually executed. By appropriate choice of the prefetch iteration count, the number of iterations separating prefetches from their target badrefs, the technique can be adapted to tolerate arbitrary latencies subject only to bandwidth limitations.

Application of RAAD prefetching has two requirements. The loop containing a badref reference must be identifiable and the address computation for the badref must match one of a set of compiler-recognized schemas associated with common access patterns. Benchmark data shows that the set of recognized RAAD patterns covers almost all badrefs which occur in loops. Array RAADs alone account for essentially all such badrefs. The data also shows that it is relatively common for procedure boundaries to separate badrefs from their corresponding loops.

Miss scheduling, a prefetching technique applicable to sequential code, is applied when loops cannot be identified for references. Miss scheduling generates prefetch operations for each badref in a sequence, in order, and merges these prefetches into the original sequence. Limited rescheduling of the original sequence allows address dependencies for prefetch operations to be satisfied.

7.3.3 Characterization of Badrefs

Empirical data collected using c-flat provides a characterization of the badrefs exhibited by the benchmark set. This data indicates that in many programs the principle contributors towards badrefs are RAAD references in loops and that these references are overwhelmingly dominated by array RAADs with loop-constant strides. Compiler data also indicates that it is not uncommon for this RAAD within loop structure to be split by procedure boundaries, potentially hiding it from unsophisticated compilers. RAAD prefetching is applicable to badrefs in loops. Miss scheduling is applied to other badrefs. This combination enables optimization of a large fraction of badrefs, typically covering over 80% of dynamic misses and sometimes coming close to 100%.

7.3.4 Variations of Static Locality Correlation

The process of developing a strategy for loops with conditional behavior leads to an additional observation about static locality correlation. Sometimes misses caused by a badref may be more highly correlated with some other program event than the execution of the badref instruction. This phenomenon is exhibited in two notable instances which are exploited by c-flat. First, in loops in which induction variables used in address computations are conditionally updated, the conditional induction variable update may be the program event most strongly correlated to misses. Similarly, loop prologue code is highly correlated to misses which occur systematically on the first iterations of loops. This correlation phenomenon is exploited by positioning prefetch code at the correlated site.

7.4. Compiler Performance

Runtime data collected for code generated using c-flat allows the assessment of the latency tolerance of modified and unmodified code. Data for two interface models, two cache configurations and a variety of memory system latency parameters is presented.

By comparing the performance of the two processor models examined, the **Stall** and **Interlock** models, one can assess the intrinsic latency tolerance of unmodified code. In general, the increased flexibility of the **Interlock** model leads to performance improvements of 5 to 10% for moderate latency parameters and about 20% for the high latency memory system measured. These numbers show significant variation.

At both moderate and high memory latency, modified code outperforms unmodified code for a fully-associative cache model. A memory system with 20 cycle latency and bandwidth sufficient to process a single miss transaction at a time exhibits between 10 and 30% speedups. Speedups between factors of 2 and 6 are demonstrated in conjunction with a 160 cycle memory system with bandwidth sufficient to service 8 concurrent miss transactions. These performance improvements both at moderate and high latencies demonstrate that compilers can indeed generate code that is latency tolerant.

Data collected for some benchmarks using a direct-mapped cache model demonstrates a potential pitfall. The code is degraded rather than being improved. Simulation shows that this degradation does not occur as a result of cache pollution from prefetch operations. Detrimental prefetches, which occur when prefetched blocks are purged from the cache before they are used, represent only a very small fraction of misses when prefetching data directly into direct-mapped cache and are essentially eliminated in systems employing victim caching. The performance degradation actually results from increased spill activity in some large procedures after the application of miss scheduling. This result may primarily be an artifact of inefficient use of stack locations by spill code generation in the prototype compiler. It illustrates that caution is required in implementing latency-tolerant compilers, particularly if the target memory system employs a direct-mapped cache.

7.5. Epilogue

Until some time in the future when revolutionary technology eliminates the problem of memory latency, what should be remembered?

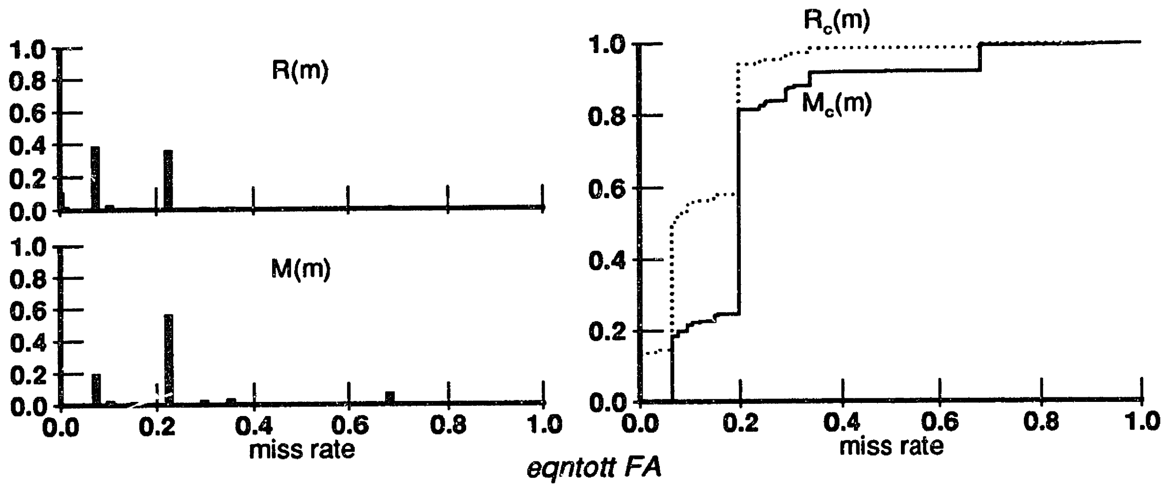
Static Locality Correlation, Badref Behavior, Compilation-based Memory Latency Tolerance, it's all real.

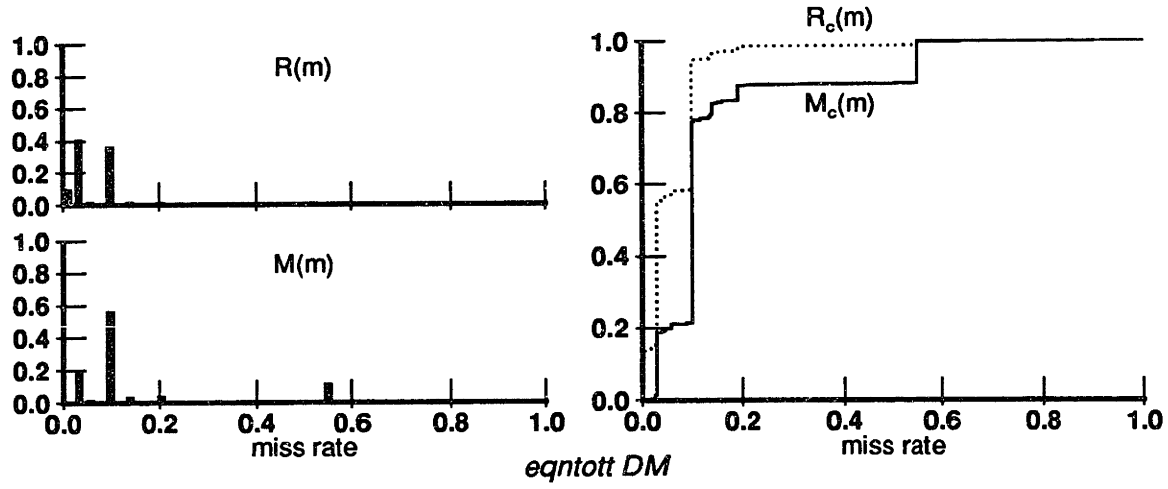
Appendix A

Additional Static Locality Correlation Data

A.1. Baseline Locality Correlation Data

These figures present additional benchmark data for Section 2.2. As in section Chapter 2 these figures indicate $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ data for benchmarks for the **FA**, **DM** and **DMV** cache models.





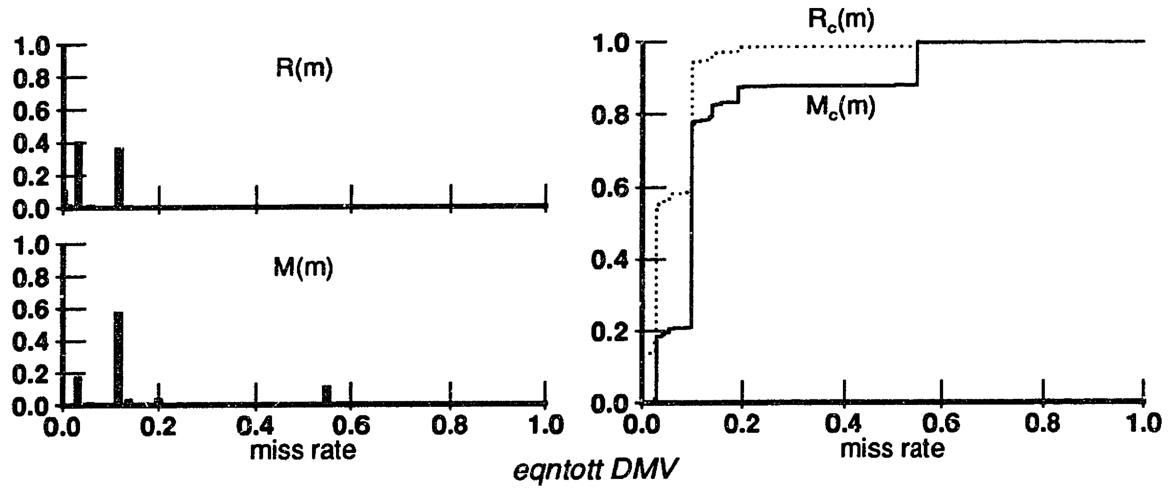


Figure A-1: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Eqntott **FA**, **DM** and **DMV**

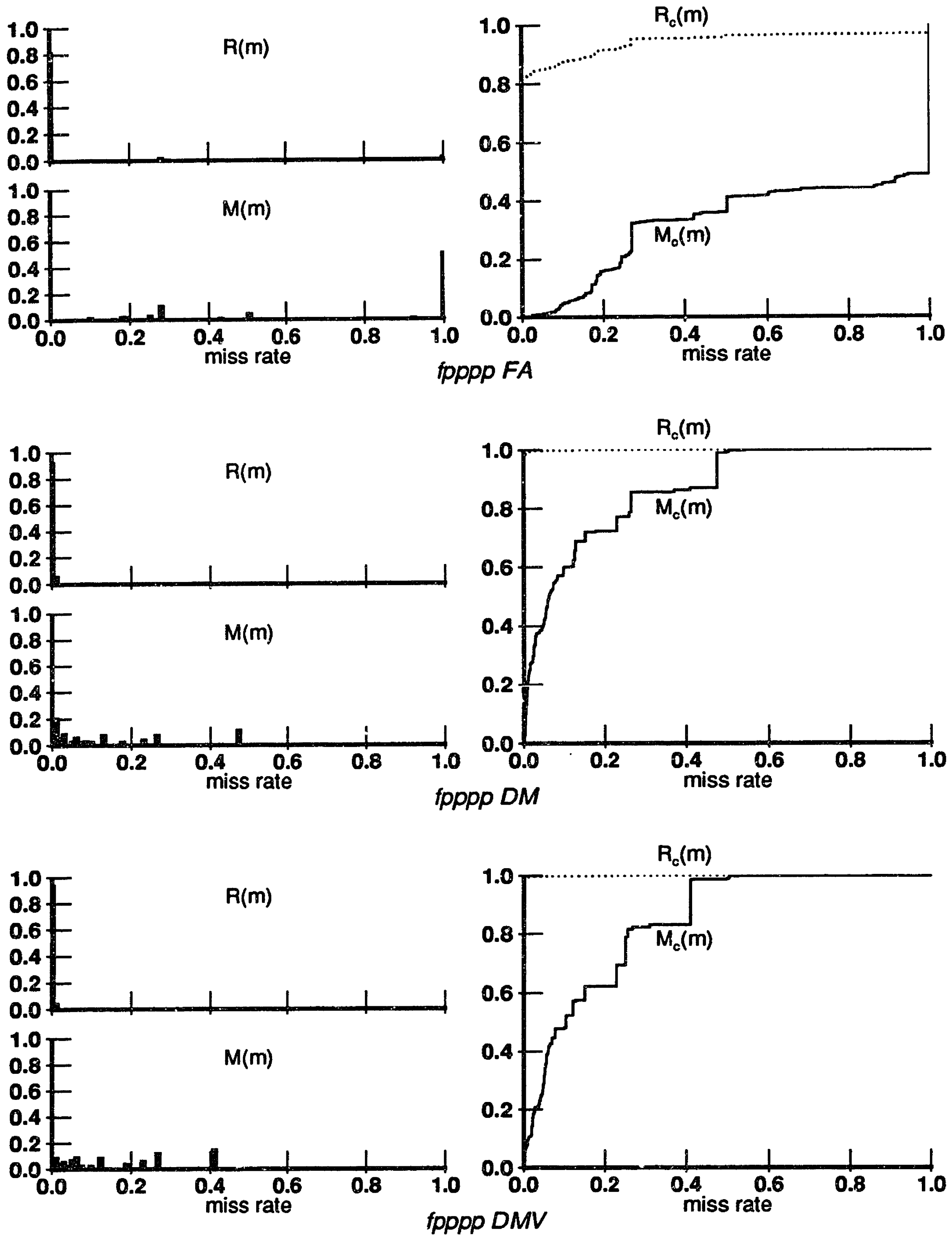


Figure A-2: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Fppp **FA**, **DM** and **DMV**

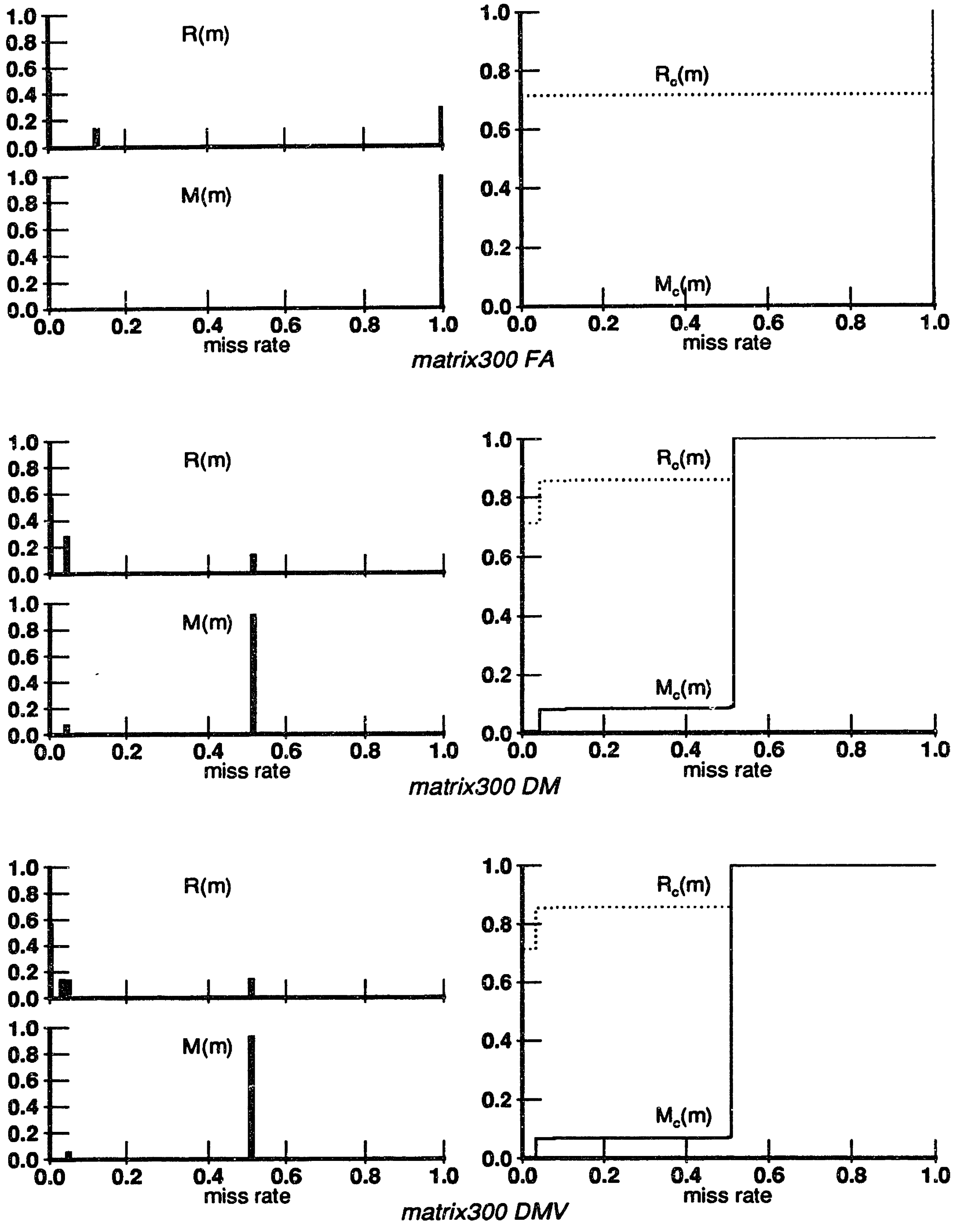
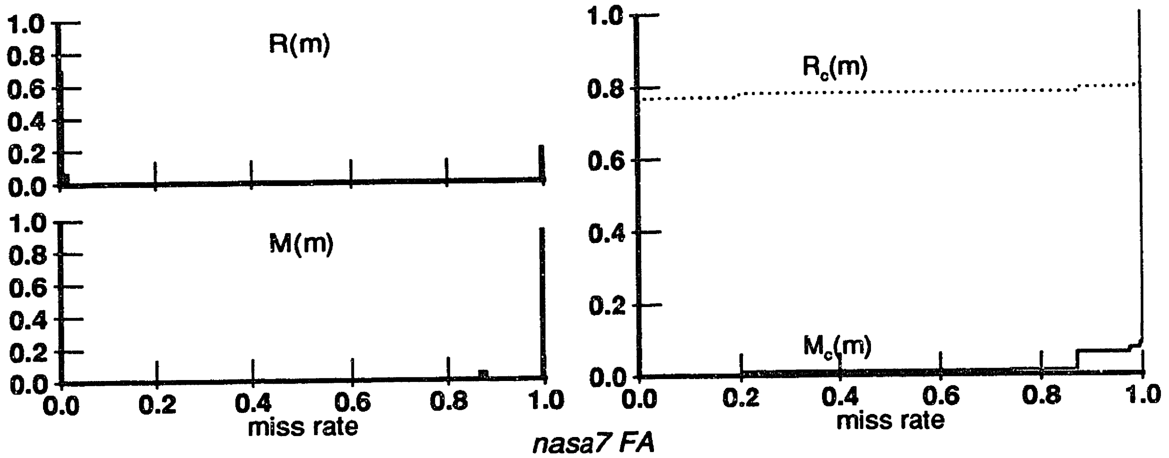
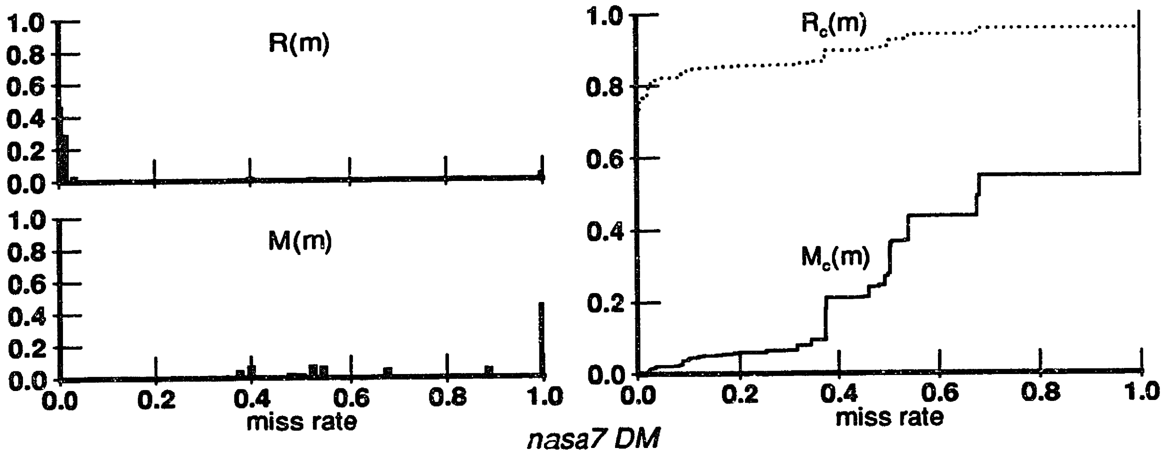


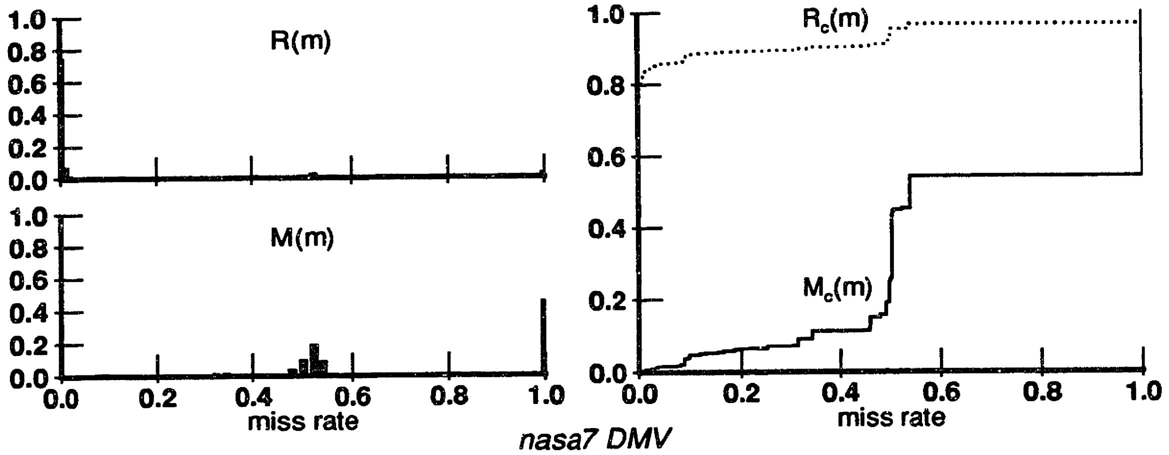
Figure A-3: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Matrix300 **FA**, **DM** and **DMV**



nasa7 FA

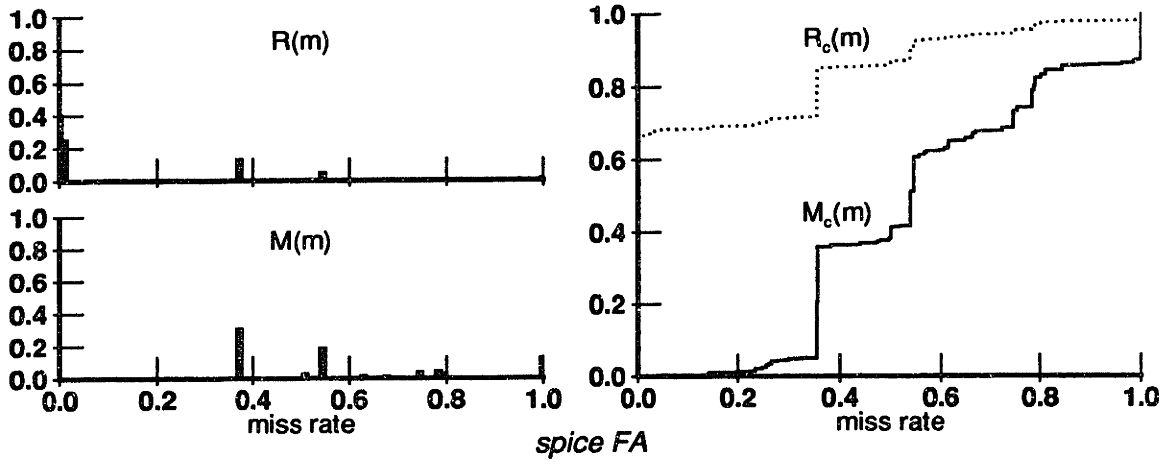


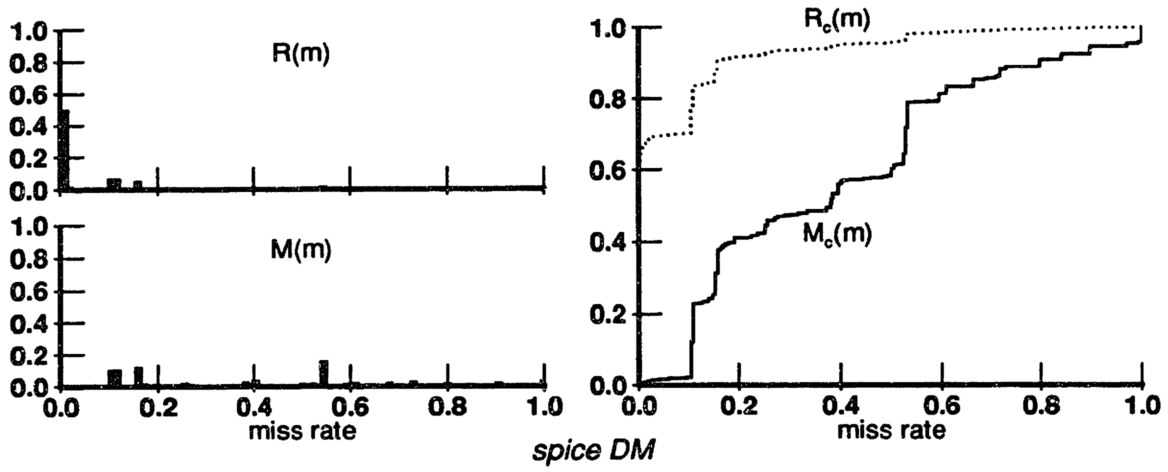
nasa7 DM



nasa7 DMV

Figure A-4: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Nasa7 **FA**, **DM** and **DMV**





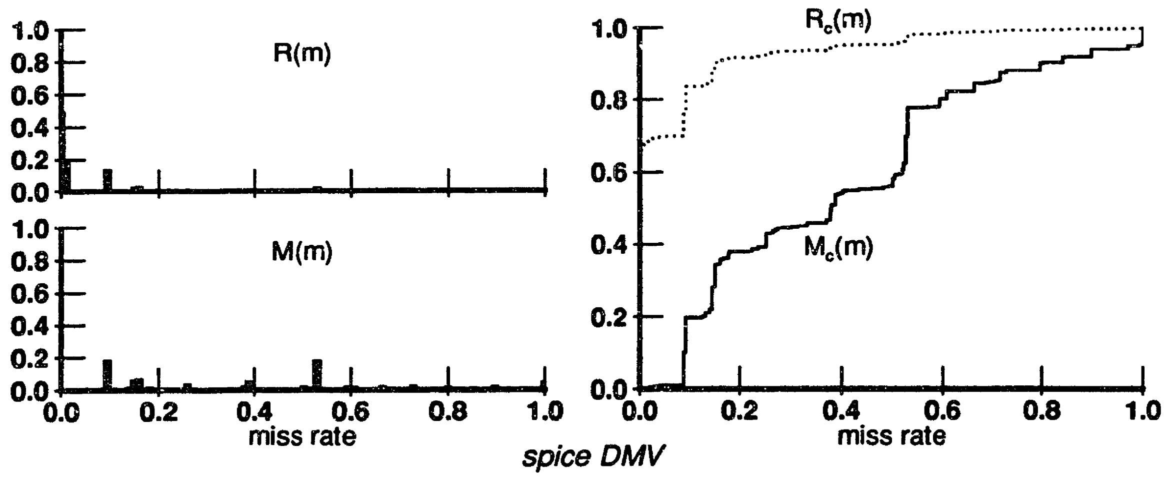


Figure A-5: $R(m)$, $M(m)$, $R_c(m)$ and $M_c(m)$ for Spice **FA**, **DM** and **DMV**

Bibliography

[1] Anant Agarwal, John Hennessy, and Mark Horowitz. Cache performance of operating system and multiprogramming workloads. *ACM Transactions on Computer Systems*, 6(4):393–431, Nov 1988.

[2] Anant Agarwal, Mark Horowitz, and John Hennessy. An analytical cache model. Technical report, Stanford University Computer Systems Laboratory, August 1987.

[3] Alfred Aho, Ravi Sethi, and Jeffrey Ullman. *Compilers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1986.

[4] Glen A. Brent. *Using Program Structure to Achieve Prefetching in Cache Memories.* PhD thesis, University of Illinois, 1987.

[5] David Callahan, Ken Kennedy, and Allan Porterfield. Software prefetching. In *ASPLOS-IV Proceedings*, pages 40–52, Santa Clara, April 1991. ACM.

[6] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, Cambridge, MA, May 1986.

[7] Anant Agarwal et al. The mit alewife machine: A large-scale distributed-memory multiprocessor. Technical Report MIT/LCS/TM-454, Massachusetts Institute of Technology, 1991.

[8] D. Kuck et al. Measurements of parallelism in ordinary fortran programs. *IEEE Transactions on Computers*, 23(1):37–46, 1974.

[9] Dan Dobberpuhl *et. al.* A 200 mhz 64b dual-issue cmos microprocessor. In *Proceedings of the International Solid State Circuits Conference*, pages 104–105, 1992.

[10] Joseph Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[11] Joseph Fisher, John Ellis, John Ruttenberg, and Alexandru Nicalau. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the Symposium on Compiler Construction*, volume 19, pages 37–47, June 1984.

[12] John Fu and Janak Patel. Data prefetching in multiprocessor vector cache memories. In *The 18th Annual International Symposium on Computer Architecture*, pages 54–63, Toronto, Canada, May 1991.

[13] Dennis Gannon, William Jalby, and Kyle Gallivan. Strategies for cache and local memory management by global program transformation. In *Proceedings of the International Conference on Supercomputing*, pages 229–254, 1987.

[14] Edward Gornish, Elana Granston, and Alexander Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *International Conference on Supercomputing*,

volume 18, pages 354–368. ACM, June 1990.

[15] Anoop Gupta, John Hennessy, Kourosh Gharachorloo, Todd Mowry, and Wolf-Dietrich Weber. Comparative evaluation of latency reducing and tolerating techniques. In *The 18th Annual International Symposium on Computer Architecture*, pages 254–263, Toronto, Canada, May 1991.

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

[17] John L. Hennessy. Vlsi processor architecture. *IEEE Transactions on Computers*, C-33(12):1221–1246, December 1984.

[18] Hewlett Packard. *PA-RISC 1.1 Architecture and Instruction Set Manual*, November 1990.

[19] Mark D. Hill and Alan J. Smith. Evaluating associativity in cpu caches. *IEEE Transactions on Computers*, 38(12):1612–1630, December 1989.

[20] Mark Donald Hill. *Aspects of Cache Memory and Instruction Buffer Performance.* PhD thesis, University of California, Berkeley, 1987.

[21] Harlan Husman. *Compiler Memory Management and Compound Function Definition for Mulitiprocessors.* PhD thesis, University of Illinois, 1986.

[22] *Proceedings of the International Solid State Circuits Conference*, 1980–1992.

[23] Norman P. Jouppi. The nonuniform distribution of instruction-level and machine parallelism and its effect on performance. *IEEE Transactions on Computers*, 38(12):1645–1658, December 1989.

[24] Norman P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[25] Alexander Klaiber and Henry Levy. An architecture for software controlled prefetching. In *The 18th Annual International Symposium on Computer Architecture*, pages 43–53, Toronto, Canada, May 1991.

[26] David Kranz, David Chaiken, and Anant Agarwal. Multiprocessor address tracing and performance analysis. Technical report, MIT Laboratory for Computer Science, 1989.

[27] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *The 8th Annual Symposium on Computer Architecture*, pages 81–87, 1981.

[28] D. Kuck, Y. Muraoka, and S. Chen. On the number of operations simultaneously executable in fortran-like programs and their resulting speedup. *IEEE Transactions on Computers*, 21(12):1293–1310, December 1972.

[29] Manoj Kumar. Effect of storage allocation/reclamation methods on parallelism and storage requirements. In *Computer Architecture Conference Proceedings*, pages 197–205. ACM Sigarch, 1987.

[30] Roland Lun Lee. *The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Mulitprocessors.* PhD thesis, University of Illinois, 1987.

[31] Monica Sing lin Lam. *A Systolic Array Optimizing Compiler.* PhD thesis, Carnegie-Mellon University, 1987.

[32] Scott McFarling. Program optimization for instruction caches. In *ASPLOS-III Proceedings*, pages 183–191, 1989.

[33] Scott McFarling and John Hennessy. Reducing the cost of branches. In *The 13th Annual International Symposium on Computer Architecture*, pages 396–403, Tokyo, Japan, June 1986.

[34] S. McGeady. A programmer's view of the 80960 architecture. In *Proceedings of the 34th CompCon*, pages 4–9, San Francisco, February 1989. IEEE.

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

[36] MIPS Computer Systems, Inc. *Mips Language Programmer's Guide*, 1986.

[37] Motorola, Inc. *MC88100 Risc Microprocessor User's Manual*, 1988.

[38] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, June 1991.

[39] Peter R. Nuth and William J. Dally. A mechanism for efficient context switching. In *Proceedings of the International Conference on Computer Design.* IEEE, October 1991.

[40] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[41] Gregory Papadapoulos and David Culler. Monsoon: An explicit token store architecture. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 82–91, Seattle, WA, June 1990.

[42] David Patterson and Carlo Sequin. A vlsi risc. *Computer*, pages 8–21, 1982.

[43] Allan K Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, 1989.

[44] Steven Przybylski. The performance impace of block sizes and fetch strategies. In *The 17th Annual International Symposium on Computer Architecture*, volume 18, pages 160–169. ACM

Sigarch, June 1990.

[45] Michael Slater. Rambus unveils revolutionary memory interface. *Microprocessor Report*, pages 15–21, March 1992.

[46] Alan Smith. Cache evaluation and the impact of workload choice. In *Proceedings of the 12th Annual Symposium on Computer Architecture*, pages 64–73, New York, June 1985. IEEE.

[47] Alan J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, September 1982.

[48] Alan Jay Smith. Sequential program prefetching in memory hierarchies. *Computer*, pages 7–21, December 1978.

[49] Alan Jay Smith. Line (block) size choice for cpu cache memories. *IEEE Transactions on Computers*, C-36(9):1063–1075, September 1987.

[50] Burton Smith. A pipelined, shared resource mimd computer. In *Proceedings of the 1978 International Conference on Parallel Processing*, pages 6–8, August 1978.

[51] Michael Smith, Mike Johnson, and Mark Horowitz. Limits on multiple instruction issue. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 290–302, Boston, MA, April 1989.

[52] Michael Smith, Monica Lam, and Mark Horowitz. Boosting beyond static scheduling in a superscalar processor. In *The 17th Annual International Symposium on Computer Archiiecture*, pages 344–354, Seattle, WA, May 1990.

[53] Chriss Stephens, Bryce Cogswell, John Heinlein, Gregory Palmer, and John Shen. Instruction level profiling and evaluation of the ibm rs/6000. In *The 18th Annual International Symposium on Computer Architecture*, pages 180–189, Toronto, Canada, May 1991.

[54] John Swensen. *High-Bandwidth/Low Latency Temporary Storage for Supercomputers.* PhD thesis, University of California, Berkeley, 1987.

[55] System Performance Evaluation Cooperative, Fremont, CA. *The SPEC Benchmark Report*, January 1990.

[56] Dominique Thiebaut. From the fractal dimension of the intermiss gaps to the cache-miss ratio. *IBM Journal of Research and Development*, 32(6):796–803, 1988.

[57] Dominque Thiebaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 38(7):1012–1026, 1989.

[58] James G. Thompson. *Efficient Analysis of Caching Systems.* PhD thesis, University of California, Berkeley, 1987.

[59] David W. Wall. Limits of instruction-level parallelism. In *ASPLOS-IV Proceedings*, pages

176–186, Santa Clara, April 1991. ACM.

[60] Steve Ward and Robert Halstead. *Computation Structures*, chapter 16, page 475. McGraw-Hill, 1990.

[61] Michael Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. The MIT Press, 1989.