CRITICAL LOADS - CLASSIFICATION, CHARACTERIZATION, AND EXPLOITATION

by

SRIKANTH T. SRINIVASAN

Department of Computer Science
Duke University

Date: ______________________

Approved:

____
Alvin R. Lebeck, Supervisor

____
Thomas M. Conte

____
Gershon Kedem

____
John Board

____

Dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2001

ABSTRACT

(Computer Architecture)

CRITICAL LOADS - CLASSIFICATION, CHARACTERIZATION, AND EXPLOITATION

by

Srikanth T. Srinivasan

Department of Computer Science
Duke University

Date: ______________________________

Approved:

____
Alvin R. Lebeck, Supervisor

____
Thomas M. Conte

____
Gershon Kedem

____
John Board

____

An abstract of a dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
Duke University

2001

Abstract

Limitations to Instruction Level Parallelism (ILP) due to data dependencies, finite resources, and branch mis-predictions introduce a variation in the latency requirements of load instructions – some loads need to complete early while others can wait for a long time to complete without adversely affecting processor performance. Based on their latency requirements, I classify loads as critical and non-critical. This thesis investigates exploiting critical loads to improve processor performance using a subset of the SpecINT2000 and the Olden benchmark suites.

This thesis has three parts. The first part describes my technique for quantifying the criticality of loads using a theoretical processor with look ahead/roll back capability. My simulations reveal that there is a variation in load criticality and that if all loads are completed based on their criticality, up to 101% improvements in performance over a traditional memory system are possible.

The second part of this thesis deals with devising a practical scheme to classify loads as critical and non-critical. My hardware scheme uses a structure called the Criticality Table to track load dependencies of instructions. It classifies a load as critical if it feeds into a mis-predicted branch, or if it feeds another load that misses in the L1 cache, or if the number of independent instructions issued following a loads is below a threshold of 2 instructions per cycle. On average, this scheme classifies 30% of all dynamic loads as critical.

The third part evaluates practical techniques for exploiting critical loads. First, I focus on the memory hierarchy and investigate if criticality is a strong enough program property to warrant changes in cache management. My criticality based cache organization experiments reveal that it is very difficult to retain critical data for long enough periods of time to translate into significant performance gains. This is

mainly due to large critical data working sets. Criticality based selective prefetching shows only small improvements for benchmarks that are resource constrained. Thus I find that, in practice, it is not worth violating locality to exploit criticality.

Next, I examine several load criticality based schemes that do not violate locality, such as priority scheduling, and Load-based Branch Prediction (LBP). Priority Scheduling queues critical loads earlier than other instructions and produces up to 9% gains in performance compared to my default scheduling policy. LBP focuses on one subset of critical loads: loads that feed branches. LBP exploits the correlation between load values and dependent branch outcomes for better branch prediction. A practical implementation of LBP achieves a reduction in branch mis-predictions of up to 19%, which translates into improvements in performance of up to 11% compared to an aggressive default branch predictor.

Acknowledgements

First and foremost, I thank God for bringing me close to a lot of nice people. Their support, encouragement, friendship and guidance has not only been a huge positive factor during my graduate career but am sure will continue to enrich my after-school-life.

Special thanks to my advisor, Prof. Alvin R. Lebeck for his leadership and guidance at each step of my PhD. He has been a big source of inspiration throughout.

I would like to thank Professor Gershon Kedem for his advice on implementation details of various hardware structures. I also thank him as well as Professor Thomas Conte and Professor John Board for taking the time to serve on my committee.

My colleagues during my internship at Intel, Chris Wilkerson and Roy Ju were fun to work with and a lot of bright ideas came up while interacting with them. They played an important role in making my internship a very productive one. I thank Steve Reinhardt for his valuable comments on our "Locality vs. Criticality" paper and Todd Austin for his SimpleScalar toolset. Almost all experiments in my thesis were performed using it.

Academics is only a part of a successful PhD. Family and friends constitute the other important part. My wife, Priya's never ending enthusiasm, support and encouragement have been something that I could always fall back on. Right from preparing Bhel puri to constructively criticizing aspects of my talk, she was there to fill in all the missing gaps.

All that I am is what my parents have taught me to be and I thank them for that. My brothers and my network of cousins (headquartered in NJ) have provided much-needed respite from the daily grind. My friends, both at work and off of work have helped me make progress in a variety of fields ranging from volleyball to spirituality.

Contents

Abstract iii

Acknowledgements v

List of Tables x

List of Figures xi

1 Introduction 1

1.1 Motivation 2

1.2 Methodology 4

1.3 Contributions 6

1.4 Thesis Organization 8

2 Load Criticality Characterization 9

2.1 Background 9

2.2 Measuring Load Criticality 11

2.3 Load Completion Policies 12

2.3.1 Determining If Loads Should Complete 12

2.3.2 Determining Which Loads to Complete 14

2.3.3 Determining When Loads Should Complete 15

2.3.4 Determining How Many Loads to Complete 15

2.4 Rollback Processor 16

2.5 Experimental Setup 17

2.6 Load Latency Tolerance Measurements 19

2.6.1 Constant Latency Memory Systems 19

2.6.2 Detecting that Loads Must Complete 21
2.6.3 Determining Which Loads to Complete 27
2.6.4 Determining When to Complete Loads 29
2.6.5 Limiting the Number of Completed Loads 31
2.6.6 Effects of Processor Micro-architecture 32
2.7 Load Latency: Expected vs. Realized 34
2.8 Conclusions 38
3 Critical Load Classification 39
3.1 Critical Load Criteria 39
3.2 Criticality Table 41
3.3 Load Criticality Predictor 46
3.4 Criticality Computation Validation 47
3.5 Classification Criteria Comparison 51
3.6 Related Work 52
3.7 Conclusions 54
4 Locality vs. Criticality 55
4.1 Introduction 55
4.2 Criticality-based Cache Organization 56
4.2.1 Locality vs. Criticality at the L1 Cache Level 58
4.2.2 Locality vs. Criticality at the L2 cache Level 59
4.2.3 Critical Load Working Set 62
4.2.4 Summary 68
4.3 Criticality-based Prefetching 69

4.3.1 Prefetching Schemes 69
4.3.2 Prefetching Into the L1 Cache 71
4.3.3 Prefetching Into the L2 Cache 74
4.3.4 Summary 78
4.4 Related Work 78
4.5 Conclusions 80

5 Priority Scheduling **82**

5.1 Introduction 82
5.2 Queue Scheduling 82
5.2.1 Queues on a Load Path 83
5.2.2 Default Scheduling 85
5.2.3 Priority Scheduling 86
5.3 Results 86
5.4 Conclusions 89

6 Load-based Branch Prediction **90**

6.1 Introduction 90
6.2 Background 92
6.2.1 Related Work 93
6.3 Experimental Setup 96
6.4 Exploiting Data Set–Branch Correlation 97
6.5 Implementation 104
6.5.1 Establishing Data Set–Branch Correlations 104
6.5.2 Utilizing Data Set–Branch Correlations 106

6.6 Results 107

6.6.1 LBP: Ideal vs. Practical 108

6.6.2 LBP: Effect of Pipeline Depth 109

6.6.3 Obtaining Load Values Early 111

6.7 Conclusions and Future Work 113

7 Conclusions 115

7.1 Thesis Summary 115

7.2 Future Work 117

Appendix A: Critical Load Distributions 118

Bibliography 120

Biography 128

List of Tables

2.1 Benchmark Description 19

2.2 Effectiveness of Traditional Memory Systems at Capturing Critical Loads 36

3.1 Criticality Predictor accuracy 46

4.1 Working Set: Cache size in KB required to achieve 95% cache hit ratio 65

6.1 LBP: Benchmark Characteristics 97

6.2 Contents of ABT Entry 105

List of Figures

1.1 Sample Dependence Graph . . . 4

2.1 Computing Dependence Graph Depth . . . 14

2.2 Need for Rollback – Example . . . 16

2.3 Memory System Configuration and IPC . . . 20

2.4 Branch Prediction and IPC . . . 22

2.5 Branch Prediction and Latency Tolerance . . . 23

2.6 Performance-based Load completion and IPC . . . 25

2.7 Performance-based Load completion and Latency Tolerance . . . 26

2.8 Load Selection and IPC . . . 28

2.9 Load Selection and Latency Tolerance . . . 29

2.10 Completion Time and IPC . . . 30

2.11 Completion Time and Latency Tolerance . . . 31

2.12 Load completion limit and IPC . . . 32

2.13 Load completion limit and Latency Tolerance . . . 33

2.14 Processor Architecture (Issue-width / RUU-size / LSQ-size) and IPC 34

2.15 Processor Architecture (Issue-width / RUU-size / LSQ-size) and Latency Tolerance . . . 35

3.1 Flowchart to compute Load criticality . . . 41

3.2 Criticality Table . . . 42

3.3 Criticality Table 43

3.4 Issue Counter Update 44

3.5 Potential Criticality-based Performance Improvement at L1 cache level 48

3.6 Potential Criticality-based Performance Improvement at L2 cache level 49

3.7 Critical Load Classification Criteria: Performance Benefits 51

3.8 Critical Load Classification Criteria: Benefits per Critical Load 52

4.1 Critical Cache Illustration 57

4.2 L1 Content Management: IPC 58

4.3 L1 Content Management: Miss Ratio 60

4.4 L2 Content Management: IPC 61

4.5 L2 Content Management: Miss Ratio 62

4.6 Working Set 64

4.7 Critical Load Distribution 66

4.8 Load Criticality: Unstable, yet predictable 67

4.9 Prefetching Example 70

4.10 L1 Locality vs. Criticality Prefetching: IPC 71

4.11 L1 Locality vs. Criticality Prefetching: Bandwidth 72

4.12 L1 Locality vs. Criticality Prefetching: Miss Ratio 74

4.13 L2 Prefetching: IPC 75

4.14 L2 Prefetching: Miss Ratio 76

4.15 L2 Prefetching: Bandwidth 77

5.1 Queues on Load Path 83

5.2 Priority Scheduling Benefits 88

5.3 Priority Scheduling Benefits: Increasing L1 Cache Latencies 88

6.1 Data Set Correlation Example 92

6.2 Path Based Correlation Example 94

6.3 Data Set–Branch correlation: Effect of Pipeline Depth 99

6.4 Data Set–Branch Correlation: Branch Coverage 101

6.5 Data Set–Branch correlation: Bigger and Better Base Predictors . . . 103

6.6 LBP: "Ideal vs. Practical" Comparison 108

6.7 LBP: Effect of Pipeline Depth 110

6.8 LBP variations 112

Chapter 1: Introduction

Solutions to problems in today's computing world involve three distinct components – hardware, user software (end-user application) and system software (compilers and operating systems). Performance increases can occur due to improvements in any of the above three components. Each of these components are becoming increasingly sophisticated, and they often interact with each other in complex ways. Researchers are being forced to understand the interaction between these three components and come up with techniques that understand and/or exploit this interaction. For example, the processor has been shown to do better if it understands application behavior (predictability of branches [10], data addresses [14], data values [29]), the compiler can do better if it knows the hardware it is generating code for (vendor compilers produce better executables than generic compilers), and programs can perform better if they are written with the memory system in mind [55]. This thesis examines the usefulness of one such characteristic that depends on the interaction between the hardware, the compiler and the user program – load criticality – to increase performance.

Load criticality is a measure of the latency requirements of load instructions. The latency requirement of a load is the maximum amount of time a load can wait to complete without adversely affecting processor performance. It is determined by the inherent instruction level parallelism (ILP) that is available in programs, the compiler's ability to expose that ILP and the processor's ability to exploit that ILP. In the presence of a memory hierarchy, the actual latency incurred by a load is the difference in time between when the load request is sent to the memory system and when the data referenced by the load is received. Even with today's state of the art

processors and memory systems, there is a gap between the latency requirements of loads and actual load latency. Superior performance can be achieved by bridging this gap between the latency requirements of loads (load criticality) and their actual latencies [51]. This thesis aims to do so by conveying information about critical loads to the processor and the memory system.

1.1 Motivation

Many of today's microprocessors use dynamic scheduling [54, 58] to maximize the number of instructions executed per cycle. Dynamic scheduling involves buffering instructions that are waiting for their operands and looking ahead to execute other independent instructions out of program order. To find enough independent instructions, most processors employ sophisticated branch prediction mechanisms [43, 46, 67] and allow speculative execution [9, 27, 59], committing results only when the true outcome of a branch is known. These mechanisms provide the processor with the capability to tolerate the latencies of some operations.

Unfortunately, long latency operations can reduce the effectiveness of these mechanisms because of the following limitations.

- When a dynamically scheduled processor looks ahead, if most of the newly fetched instructions are dependent on the results of an earlier instruction waiting to complete, then the processor could stall since there are no ready instructions to execute.

- The dependent instructions waiting for their operands to become available, and independent instructions that have completed but are waiting to commit their results occupy entries in the buffer. Eventually, the processor could run out of buffer space and will not be able to look ahead any further.

- When a branch is mis-predicted, most of the work done by the processor executing on the wrong path is useless. If the outcome of the branch is dependent on the results a long latency instruction, it could be a while before the branch is resolved and the mis-prediction is detected.

Thus, even though today's processors have some amount of latency tolerance built in them, long latency operations can decrease their effectiveness. In this thesis, I concentrate on one such potential long latency operation – load instructions.

Unlike most other instructions, load instructions have a variable latency. The latency of a load is determined to a large extent by the level in the memory hierarchy where the data accessed by the load is found. By changing the processor scheduling, and/or memory management policies, it is possible to change the latencies incurred by loads. The limitations discussed earlier, introduce a disparity in the latency requirements of loads – some loads need to complete early, while other loads can wait for a long amount of time without adversely affecting processor performance.

Consider the two load instructions, *ld1* and *ld2*, shown in the dependence graph in Figure 1.1. *ld1* has many instructions dependent on it. It feeds into another load that misses in the Level 1 cache. It also has a mis-predicted branch dependent on it. If *ld1* takes a long time to complete, it is very likely to decrease processor performance. In this example, *ld1* needs to complete early.

On the other hand, *ld2* has only one instruction dependent on it. Even if it takes a long time to complete, the processor might be able to tolerate its latency by looking ahead and executing other independent instructions. Hence, *ld2* can wait for a long time to complete.

Based on their latency requirements, loads can be classified as critical and non-critical. Loads that must complete early to avoid performance degradation are critical and those that can tolerate long latencies are non-critical. In the example in Fig-

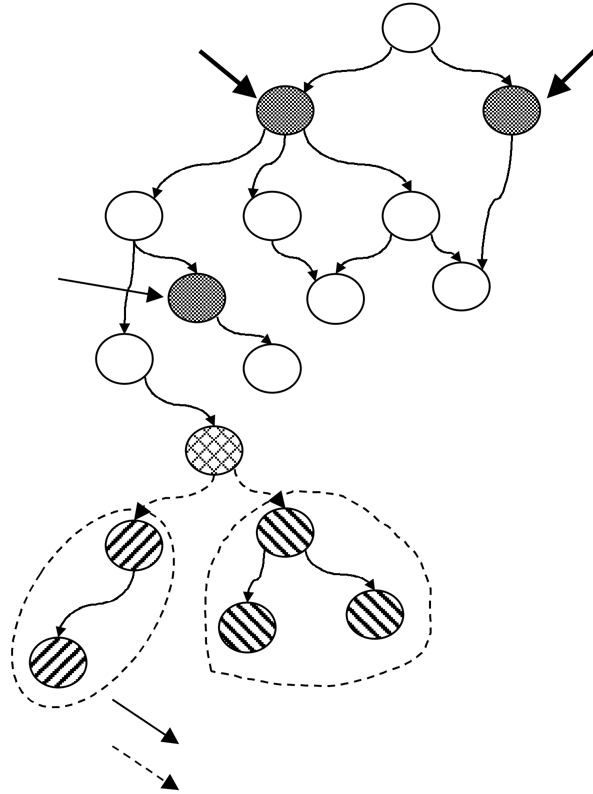


Figure 1.1: Sample Dependence Graph

ure 1.1, *ld1* is a critical load while *ld2* is a non-critical load. This example shows that there could be variation in the criticality of loads.

One of the goals of this thesis is to quantify this variation in load criticality. Current processors and memory systems are incognizant of the difference in criticality of loads and treat all loads alike. Performance could improve if the latencies incurred by loads match their latency requirements [51]. Hence, my second goal is to investigate practical schemes to exploit the variation in criticality of loads.

1.2 Methodology

To quantify load criticality, I measure the latency requirements of loads. For this purpose, I use a processor simulator that does not complete loads based on a memory

hierarchy. Instead, it schedules load completions such that performance stays close to that of a processor with an ideal memory system, where all loads complete in one cycle. This theoretical processor has the capability to look at future instructions and rollback its state so as to decide when to complete loads.

By studying the effect of different load completion policies on processor performance, I identify the set of features that make a load critical. My experiments reveal that a load is critical if any of the following conditions are true.

1. The load feeds into a mis-predicted branch,
2. The load feeds into another load that misses in the L1 cache, or
3. The processor is unable to find enough independent instructions in cycles immediately following the load and hence is unable to tolerate the load's latency

I gauge the potential for exploiting load criticality information by comparing my criticality based load completion scheme with a scheme that completes loads based on a traditional memory system. I find that average improvements in committed Instructions Per Cycle (IPC) of 101% can be achieved, if the latencies incurred by loads are in tune with their criticality. Since the processor used for these experiments is a theoretical one that has look ahead/roll back capability, the above improvements are upper bounds on the benefits that can be achieved by completing loads based on their criticality. Therefore, the question this thesis addresses next is "Can practical criticality based schemes lead us towards these potential performance improvements?"

To answer this question, I first focus on cache management. I begin by answering the question, "In practice, is criticality a strong enough program property to warrant changes in the way we currently manage caches?" Current caches are managed based on locality of references [7] both spatial and temporal locality. Hence, I compare practical criticality based optimization schemes with equivalent locality based

schemes. My criticality based cache organization results indicate that it is difficult to retain critical data in the cache for long enough periods of time to produce significant performance gains. This is primarily because the amount of data referenced by critical loads (the critical load working set) is large for most benchmarks. Selectively prefetching for critical loads is beneficial for the benchmarks for which contention for the bus to the next level of memory and competition for MSHR entries [24] are a problem. However, the performance gains over existing locality based prefetching schemes are small. Thus I find that it is very difficult to build memory hierarchies that violate locality to exploit criticality.

Hence, next I investigate criticality based schemes that do not violate locality. I examine two such schemes, Priority Scheduling and Load-based Branch Prediction. Priority Scheduling queues critical loads earlier than other instructions so as to enable critical loads to acquire the buses and functional units earlier than other instructions and is beneficial for some of the benchmarks. Load-based Branch prediction deals with one subset of critical loads – loads that feed mis-predicted branches. It reduces the number of branch mis-predictions by using the correlation between the value of a load and the outcome of a branch that is dependent on the load value for better branch prediction.

1.3 Contributions

This thesis makes contributions in the following areas:

Critical Load Characterization

Identifying the set of features that distinguish a critical load from a non-critical load is an important first step for exploiting load criticality. This thesis establishes the critical load classification criteria by studying the effect of different load completion policies on processor performance. My experiments indicate that loads that feed

into mis-predicted branches, loads that feed into other loads that miss in the L1 cache, and loads with few independent instructions need to complete early in order to achieve superior processor performance. Hence, such loads are classified as critical. If loads are completed based on their criticality rather than based on a traditional memory system, this thesis shows that on average across all the benchmarks I use, 101% improvements in performance are possible.

Critical Load Classification

This thesis presents a dynamic scheme using a hardware structure called the Criticality Table to classify loads as critical and non-critical. The Criticality Table uses both the number and type of instructions dependent on a load to do the classification. It also uses a load criticality predictor, which is similar to a two-level branch predictor [66] to make the criticality information available in time for use by criticality exploitation schemes.

Critical Load Exploitation

This thesis investigates several criticality based optimization schemes to translate the potential performance gains based on criticality into actual benefits. First, this thesis compares two of the properties of loads – locality and criticality – in the context of cache organization and prefetching schemes to answer the question, "Can practical criticality based schemes beat existing locality based schemes?". The primary finding from this comparison is that, due to large working sets of critical loads it is very difficult to achieve performance benefits by designing criticality based caching schemes that violate locality.

Next, this thesis examines criticality based schemes that do not violate locality. Priority Scheduling gives priority to critical loads in the Ready Queue and the Miss Status Handling Registers (MSHRs) so as to enable critical loads to acquire the buses and functional units earlier than other instructions. Priority scheduling produces

modest gains of up to 9% for a few benchmarks. Load-based Branch Prediction (LBP) supplements a traditional branch predictor and is targeted towards capturing the difficult to predict branches. It reduces the number of branch mis-predictions by as much as 19% which results in improvements in IPC of up to 11%.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 presents my load criticality measurement scheme using a processor with lookahead/rollback capability. Chapter 3 describes a practical online scheme for computing the criticality of loads. Chapter 4 compares two properties of loads – Locality and Criticality – to find out if in practice it is worth violating Locality to exploit Criticality. Chapter 5 presents Priority Scheduling and Chapter 6 discusses Load-based branch prediction, two criticality exploitation schemes that do not violate locality. Chapter 7 concludes this thesis. Relevant related work is discussed in each Chapter.

Chapter 2:
Load Criticality Characterization

2.1 Background

Superscalar processors maximize serial program performance by issuing multiple instructions per cycle. Each cycle the processor attempts to issue up to issue-width instructions. One of the most important aspects of these systems is identifying independent instructions that can execute in parallel.

In order to identify and exploit instruction level parallelism, most of today's processors employ techniques such as dynamic scheduling, branch prediction, and speculative execution. Dynamic scheduling is an all hardware technique for identifying and issuing multiple independent instructions in a single cycle. The hardware looks ahead by fetching instructions into a buffer – called the reorder buffer – from which it selects instructions to issue to the functional units. Instructions are issued only when all their operands are available, and independent instructions can execute out-of-order.

The reorder buffer is filled with instructions from several basic blocks by predicting the direction of conditional branches [8, 33, 42, 46]. The instructions from the predicted path are speculatively executed under the assumption that branches on the path are correctly predicted. Furthermore, to implement precise exceptions, instructions can not update the architectural state of the processor until they reach the head of the the reorder buffer and are known to be non-speculative.

The above techniques are very effective for well-behaved programs with short-latency operations. However, long latency operations such as load instructions, can reduce their effectiveness for the following reasons:

1. **Data Dependencies**: If one or more operands of instruction i are produced by a previous instruction j, then i is dependent on j, and i can begin execution only after j has completed. Clearly, instruction i can not be issued in the same cycle as j. While looking ahead, if most of the newly dispatched instructions are dependent on earlier issued instructions waiting to complete, even a dynamically scheduled processor will be unable to identify ready instructions to execute, and the processor utilization will decrease.

2. **Finite Resources**: The number of instructions the processor can look ahead to identify independent instructions is limited by the number of available entries in the reorder buffer. Dependent instructions waiting for the result of a load and independent instructions waiting to commit their results occupy buffer entries. For long latency operations, the buffer could become full and stall the processor.

3. **Branch Mis-predictions**: When the predicted outcome of a branch is incorrect, most of the work performed by the processor while executing along the mis-speculated path is useless. If computation of the true branch outcome is delayed because of a long latency operation, program completion could be delayed many cycles because of the time lost due to mis-prediction detection and correction.

The remainder of this Chapter examines load criticality in the context of the above potential limitations. First, I present a scheme to measure the criticality of each instance of each load instruction. By doing so, I

1. Identify features of a load that make it critical,

2. Determine if there is a variation in load criticality, and if so, quantify this variation, and

3. Establish the potential performance gains that are possible by exploiting the variation in criticality among loads.

2.2 Measuring Load Criticality

Developing a formal and rigorous definition of load criticality is difficult mainly because there can be many definitions of criticality. In this thesis, I use an operational definition of load criticality: loads that must complete early to avoid performance degradation are critical, while those that can tolerate long latencies are non-critical.

To determine the criticality of a load, I must find out how long that load can wait to complete without causing degradation in processor performance. Hence, in my simulator, I get rid of the memory hierarchy, and do not complete loads based on any fixed memory hierarchy. Instead, I delay load completion as long as the processor is able to do useful work by looking ahead and executing independent instructions. I complete loads whenever there is a threat to processor performance. The goal of my load completion policies is to delay load completion as much as possible and still achieve IPCs close to that of a processor with an ideal memory system, where all loads complete in one cycle.

In my simulated processor, every cycle, I ask the following questions:

1. Should one or more loads complete?
2. If so, which loads should complete?
3. How many loads should complete?
4. When should they complete?

All these load completion questions are parameters to my simulator and I find suitable answers by varying the parameter values.

Once I determine that a certain load needs to complete, I compute the number of cycles that elapse between the time the load was issued and the time it completes.

This gives the latency tolerance of that load. If the load has a lot of latency tolerance and can wait for a long period of time, then it is non-critical. On the other hand, if the load does not have any latency tolerance and needs to complete early, then it is critical. Thus, the criticality of a load is inversely proportional to its latency tolerance.

2.3 Load Completion Policies

My goal in choosing the different load completion parameters is to achieve IPCs close to that of a processor with an ideal memory system, where all references complete in one cycle. At the same time, I want to maximize load latency tolerance as much as possible. Note that I evaluate load criticality in the context of a processor with constrained resources. Extending this study beyond the limits of real hardware may provide interesting insights, but is beyond the scope of this thesis. The following sub-sections examine the individual load completion parameters in detail.

2.3.1 Determining If Loads Should Complete

In my simulator, I let loads remain outstanding as long as they are not adversely affecting processor performance. Therefore, to determine if any load(s) should be forced to complete, I must first determine if the processor performance is degrading. Recall that the performance of dynamically scheduled processors can degrade because it is unable to execute independent instructions due to limited buffer space, data dependencies, or it executes useless instructions due to incorrect branch prediction. Therefore, I force loads to complete if their results are required to ensure that the processor executes useful instructions (branch based load completion), or enable the processor to sustain execution rates close to those achievable with a processor with an ideal memory system (performance based load completion).

Branch-based Load Completion

Most modern processors predict the outcome of branches and speculatively execute instructions on the predicted path. On a mis-prediction, most of the work done by the processor executing along the wrong path is useless.[1] Delaying completion of a load on which a branch instruction is dependent can increase the number of mis-speculated instructions executed, and can therefore degrade performance. Hence, loads on which branches are dependent need to be given priority for early completion. Moreover, it is only mis-predicted branches that cause the processor to execute useless instructions. Therefore, it may be sufficient to complete only those loads early that feed into a mis-predicted branch.

Performance-based Load Completion

Using branch prediction information to force completion of certain loads ensures that loads do not cause the processor to execute useless instructions. However, that alone is not enough. Arbitrarily delaying completion of the rest of the loads will aggravate the data dependencies problem and the finite resources problem. This could prevent the processor from sustaining a reasonable level of performance.

Hence, I constantly monitor processor performance using one of two standard performance metrics: instruction issue rate or functional unit utilization. The instruction issue rate metric takes into account the number of instructions issued each cycle, while the functional unit utilization metric tracks the total number of integer and floating point computational units that are busy during each cycle.

When the processor performance drops, I complete loads freeing up dependent instructions as well as buffer space. In order to attain high IPCs I do not delay load completion until the processor actually comes to a stand still. Rather, I complete

[1]Some of the results obtained during mis-speculated execution can be reused [48]

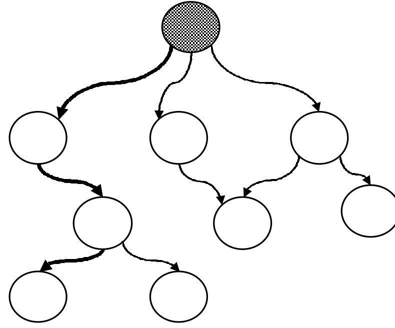


Figure 2.1: Computing Dependence Graph Depth

loads as soon as the number of instructions issued or the number of functional units that are busy drops below a tunable threshold.

Loads can also be forced to complete when there is a system call. However, for the benchmarks I study, there are very few system calls, therefore I do not discuss this case further.

2.3.2 Determining Which Loads to Complete

Once it is determined that some load(s) must complete, I need to decide which specific load(s) to complete. In the case of mis-predicted branches, clearly all loads on which the branch is dependent must be completed to decrease the execution of useless instructions. In contrast, I have the freedom to choose any load for completion when the issue rate or functional unit utilization decreases.

I investigate two policies: *fifo* and *dg*. The *fifo* policy simply forces the longest outstanding load to complete. The *dg* policy tracks the depth of a load's dependence graph in cycles and chooses the load with the largest dependence graph depth for completion, since delaying it can occupy resources for an extensive period of time. Thus the *fifo* policy completes loads in program order, while the *dg* policy completes loads in decreasing order of dependence graph depth.

Figure 2.1 shows a sample dependence graph depth computation. The length of the longest chain of instructions (shown by the highlighted path) for the load in Figure 2.1 is 7 cycles and hence its dependence graph depth is 7 cycles. The dependence graph of a load might contain other loads. Since loads feeding other loads are important in order to expose memory level parallelism, I give such loads priority for early completion in the *dg* scheme, by assigning all loads a high latency. The highest latency for any instruction in the instruction set I use is 24 cycles and hence, I assign all loads a uniform latency of 24 cycles while computing the dependence graph depth.

2.3.3 Determining When Loads Should Complete

After establishing which loads to complete, the next step is to determine when (i.e. in which cycle) the loads must complete. To minimize execution of useless instructions due to a mis-predicted branch, the appropriate load must be completed such that the entire dependence chain between the load and the branch finishes execution before the branch is reached. Section 2.4 describes how my simulations accomplish this.

If, on the other hand, a load is forced to complete because of a decrease in the instruction issue rate or functional unit utilization, it could be completed in the same cycle that the degradation in performance is detected. However, this may not provide enough time for the pipeline to fill up with ready instructions, and hence it may be beneficial to complete the load earlier. Therefore, I use a tunable threshold to study the effect of pipeline fill-up time on load latency tolerance.

2.3.4 Determining How Many Loads to Complete

Finally, in order to obtain an instruction issue rate or functional unit utilization above the set threshold, I may need to complete more than one load in a given

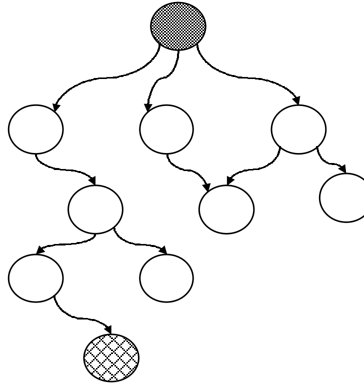


Figure 2.2: Need for Rollback – Example

cycle. Similarly, branch based load completion may require more than one load to be completed in a given cycle. An important parameter in this scenario is the limit on the number of loads that may complete. I study this by limiting the number of loads that can complete in a single cycle to one, two, or four.

2.4 Rollback Processor

Many of the load completion policies outlined in the previous Section decouple detecting that a load must complete from determining when the load should complete. It is possible for my scheme to determine that a load must have completed at an earlier time, even before I detect that it should complete. Consider the scenario shown in Figure 2.2 where I detect that a load should complete when a branch is dispatched and attaches itself to the dependence chain of a load. Assume this detection occurs at cycle t and there are 7 cycles worth of instructions in the dependence chain from the load to the branch. To minimize execution of useless instructions, the load should complete at cycle (t-8), 8 cycles before it is even established that the load should complete.

To enable this type of analysis, I add rollback capabilities to my simulator. This allows me to look ahead to compute load completion time, rollback the processor,

complete the load using the predetermined latency, and then restart execution. The replayed execution may itself incur roll-backs, but the completion of new loads and the execution of their dependent instructions guarantees that there is always forward progress made. The rollback technique ensures the processor instruction schedule is determined by the computed load latency values.

Supporting rollback requires logging all processor state during each simulated cycle when there is a load outstanding. To log smaller, finite structures like the register file or the RUU/LSQ unit, I create per cycle instances of them and perform a block copy whenever their states need to be logged/restored. However, huge structures like main memory, caches, branch predictors etc. are not amenable to replication. Instead, the updates to such structures can be kept track of, so as to be able to revert back to an earlier state. Hence I log the individual update events and play them back in the reverse order whenever a rollback is required. I do not log read accesses to any of the structures since they do not alter the state of the structures.

Since the amount of state to be saved per cycle is very large, I limit the maximum number of cycles that need to be logged to 32 cycles. This restricts the maximum amount of time a load can be outstanding to 32 cycles. However, this does not limit the scope of my study because: (i) 32 cycles is a large enough window to distinguish a critical load from a non-critical one, and (ii) in most cases, almost all loads get completed within 32 cycles, as can be seen form the results in Section 2.6.

2.5 Experimental Setup

Most of the simulators used for experiments in this thesis have been written using the SimpleScalar tool set [3], which models a dynamically scheduled processor using a Register Update Unit (RUU) and a Load/Store Queue (LSQ) [49]. The processor pipeline stages are:

Fetch: Fetch instructions from the program instruction stream.

Dispatch: Decode instructions, allocate RUU, LSQ entries.

Issue/Execute: Execute ready instructions if the required functional units are available.

Writeback: Supply the results of the operation to dependent instructions.

Commit: Commit results to the register file in program order, free RUU and LSQ entries.

My baseline processor consists of an 8-issue out-of-order processor with 256 RUU entries and 128 LSQ entries. I use a functional unit configuration consisting of 8 integer adders, 4 integer multiply/ divide units, 8 floating point adders, 4 floating point multiply/divide units, and 2 cache ports. My default branch predictor is a 2-level 16,384 entry gshare scheme that uses 12 bits of global branch history. I use a dynamic memory disambiguation scheme that prevents loads from issuing when any previous store's address is not known, or when the data to be written by a previous store to the same address as the load is not ready.

The rollback scheme does not simulate a fixed memory hierarchy. However, for other experiments in this thesis, I use a traditional memory system consisting of an 8KB, 2-way set associative Level 1 cache with 16-byte blocks and a one cycle latency, a 256-KB, 2-way set associative Level 2 cache with 64-byte blocks and a 10 cycle latency, and ideal main memory (all accesses hit in main memory) with a minimum latency of 300 cycles. I model contention (Miss Status Handling Registers (MSHRs), buses etc.) at both cache levels.

I use a subset of 9 benchmarks from the SPEC2000 integer benchmark suite and the Olden benchmark suite. Table 2.1 lists the benchmarks I use along with a brief description. For the SPEC benchmarks, I use the reference data set. I fast forward the first 4 billions instructions and simulate in detail the next 100 million instructions.

Benchmark	Description
bh	Hierarchical Force-calculation Algorithm
bzip2	Data Compression Utility
gcc	C Programming Language Compiler
gzip	Data Compression Utility
parser	Natural Language Processing
perimeter	Computing Perimeters of Regions in Images represented by Quad-Trees
perlbmk	Perl Programming Language
twolf	Place and Route Simulator
vpr	FPGA Circuit Placement and Routing

Table 2.1: Benchmark Description

Whenever caches are involved, I warm-up the caches while fast forwarding, to avoid cold start effects.

2.6 Load Latency Tolerance Measurements

This Section presents my load latency tolerance measurement results. I begin by examining the performance of the benchmarks for different memory systems. Next, I study the effects of branch prediction on load latency tolerance. This is followed by an analysis of the effects of varying the different load completion parameters on load latency tolerance and performance. By means of this analysis, I choose appropriate load completion parameter values. Finally, I examine various processor configurations and investigate the match between the latencies incurred by loads in conventional multi-level memory hierarchies and their computed latency tolerance.

2.6.1 Constant Latency Memory Systems

One approach to obtain information on the amount of latency tolerance in a system is to evaluate its performance for various memory system delays. Previous studies, that examined load latency tolerance in decoupled architectures [26], analyzed the

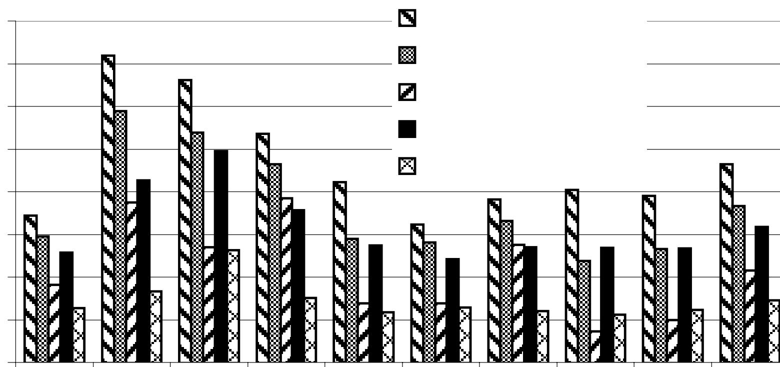


Figure 2.3: Memory System Configuration and IPC

effects of increasing memory latency for systems both with and without caches. In the systems without caches, this produces a uniform increase in latency for all memory accesses. This type of analysis can provide some insight into the latency tolerance of entire programs.

Similarly, increasing the memory latency in cache based systems does not accurately measure individual load latency tolerance, and the results may be highly dependent on the cache organization. The latency tolerance of references that hit in the cache cannot be easily measured with such a scheme, even though it may be quite large.

I perform a similar experiment by examining memory systems ranging from simple fixed cost memory accesses with no contention to detailed memory hierarchies with contention accurately modeled at all levels. The fixed cost memory systems assume all loads take the same amount of time; I examine 1, 8, and 32 cycle memory accesses. The detailed two-level memory hierarchies assume the base 256KB second level cache, but vary the second-level cache miss penalty. Specifically, I simulate L2 cache miss penalties of 60 cycles (*memlat60-2way8k*) and 300 cycles (*memlat300-2way8k*).

Figure 2.3 shows the performance of the benchmarks in terms of committed instructions per cycle (IPC) for the above memory system configurations. I make several observations from these results. First, all of the benchmarks are very sensitive to load latency. Compared to an ideal memory system, on average, performance degrades by 54% for the *memlat300-2way8k* configuration and by 21% for the *memlat60-2way8k* configuration. Second, gcc and bzip2 are better at tolerating load latency compared to the rest of the benchmarks. Even when all loads take 8 cycles to complete, these two benchmarks have instruction level parallelism enough to sustain over two instructions per cycle.

The above analysis of fixed cost memory accesses provides some insight into latency tolerance. However, it is an all or nothing approach where every load has the same cost, and is suitable only for studying entire applications. The uniform cost model doesn't exist in multi-level memory hierarchies where some loads can be satisfied faster than others. Therefore, as described in Section 2.2, my methodology is targeted at measuring the latency tolerance of individual load instructions.

2.6.2 Detecting that Loads Must Complete

This Section investigates the policies for determining when loads must complete in order to sustain performance comparable to that of a processor with an ideal memory system. I begin by examining how branch prediction affects load latency tolerance. This is followed by analysis of the performance based load completion parameters.

Branch Prediction and Load Latency Tolerance

I compare several branch based load completion schemes using my base two-level predictor and a perfect branch predictor. For the two-level branch predictor, the first policy always forces loads to complete if any branch attaches itself to the load's

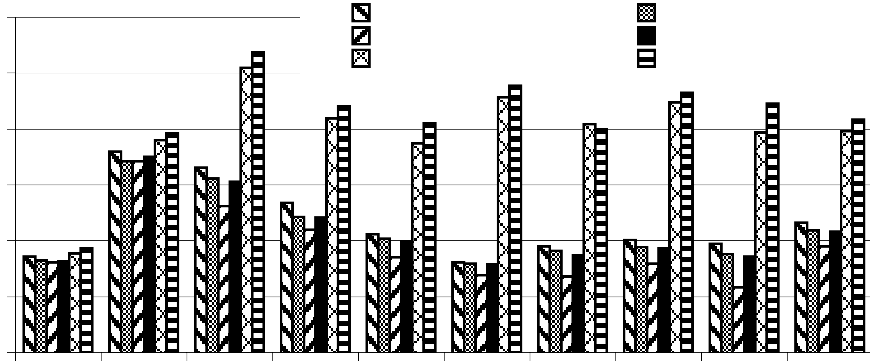


Figure 2.4: Branch Prediction and IPC

dependence chain (*2-lev, all*). The next policy is similar, except it forces a load to complete only if the branch that attaches to the load is mis-predicted (*2-lev, mispred*). Finally, for both the two-level and perfect branch predictor, I evaluate a policy that does not use any branch information to force completion of loads (*2-lev, none* and *perfect, none*). For comparison, I also simulate an ideal memory system for both the two-level predictor (*2-lev, ideal*) and perfect prediction (*perfect, ideal*).

In all the above branch based load completion runs, I make the following assumptions: loads not forced to complete by a branch are completed according to an instruction issue threshold of four instructions per cycle; which load to complete is determined by the dependence graph depth; the pipeline fill-up time is two cycles; and up to four loads can complete per cycle; I vary and study these parameters in the following subsections.

Throughout this Section I present the results in two parts: IPC and latency tolerance. I present IPC results using bar graphs like those in Figure 2.4. I present latency tolerance results in terms of the fraction of loads that must complete in a specific number of cycles as shown in Figure 2.5. For example, for the *2-lev,*

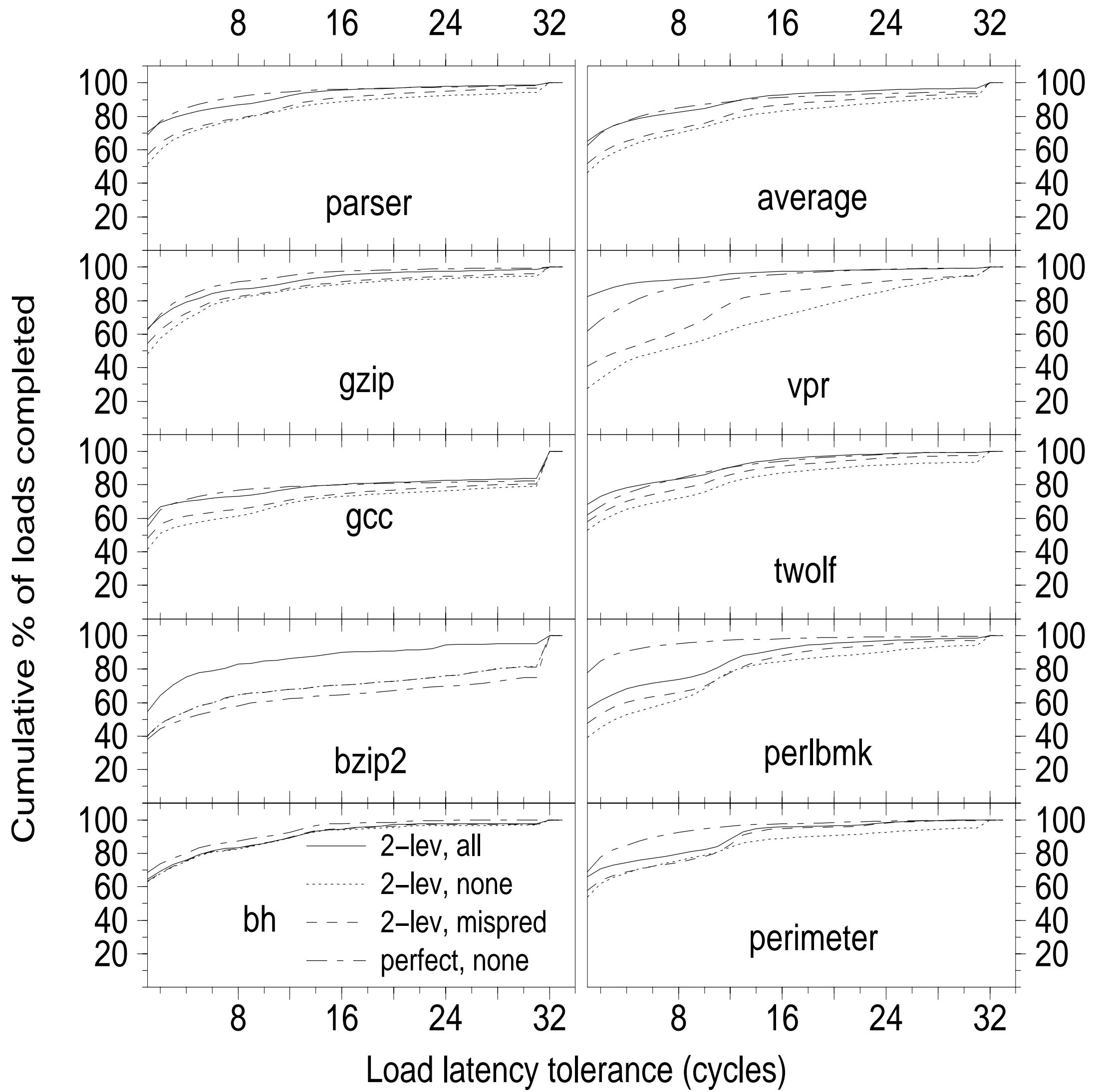


Figure 2.5: Branch Prediction and Latency Tolerance

---

mispred configuration of the benchmark vpr, 35% of loads need to complete in 1 cycle, while 58% of loads need to complete in 8 cycles or less. When comparing two different configurations, a lower curve indicates more latency tolerance and hence, fewer number of critical loads for that configuration.

From Figure 2.4, we can see that the 2-level predictor policies that exploit branch information to force load completion(*2-lev, all* and *2-lev, mispred*) meet my goal of

IPC close to a processor with an ideal memory system. Furthermore, we can see that using mis-predicted branches alone produces similar IPC values as forcing loads to complete for all branches, and that ignoring branch information entirely (*2-lev, none*) dramatically reduces the IPC values of most programs. We can also see the expected result that perfect branch prediction dramatically increases performance for all the benchmarks.

From Figure 2.5 we can see that loads exhibit significant variation in completion delays. Across benchmarks, between 19% and 79% of loads need to complete in one cycle, while 50% to 94% of the loads need to complete in eight cycles or less.

Furthermore, the latency tolerance of the benchmarks are very sensitive to the various branch-based load completion schemes. In particular, differentiating mis-predicted branches from accurately predicted branches produces noticeable improvements in load latency tolerance without significant changes in IPC. Ignoring branches altogether yields high latency tolerance, but the IPC values are too low. Since my goal is to maximize both IPC and load latency tolerance, for the rest of my branch-based load completion experiments I complete loads that feed into mis-predicted branches alone.

For the benchmarks bh and bzip2, less than 2% of the loads are completed based on mis-predicted branch information. For the rest of the benchmarks, between 11% and 22% (average 12%) of loads are completed because they feed into mis-predicted branches. With load completion based on mis-predicted branches enabled, there is a considerable reduction in the average dispatch-issue delay for branches (time for the operands of the branch to become available) compared to a traditional memory system. Also, the number of mis-speculated instructions executed drops by up to 36% (average 23%) for the *2-lev, mispred* configuration compared to tmem.

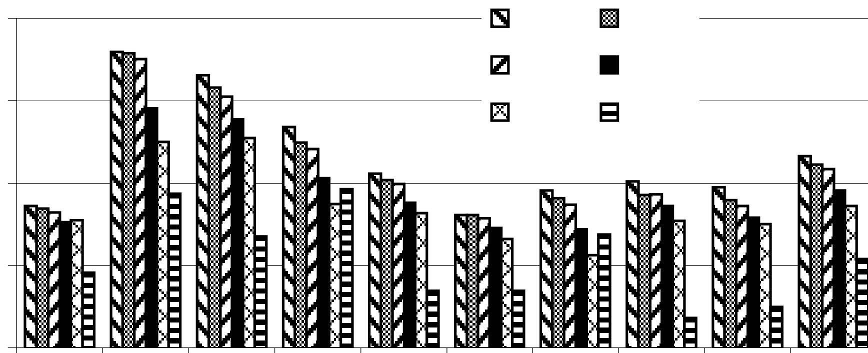


Figure 2.6: Performance-based Load completion and IPC

Processor Performance and Load Latency Tolerance

The second source of information for determining if loads should complete is processor performance. This involves monitoring a suitable performance metric and completing loads if performance falls below an appropriate threshold. For this purpose, I evaluate using either the instruction issue rate or the functional unit utilization as a performance metric and vary the threshold as 4, 2, and 1. To isolate the effects of the performance metric alone, I fix the other load completion parameters: mis-predicted branches force completion of loads; which load to complete is determined by the dependence graph depth; the pipeline fill-up time is 2 cycles; and up to four loads can complete per cycle.

Figure 2.6 shows the effect of issue rate thresholds of 1 (*nis1*), 2 (*nis2*) and 4 (*nis4*), and a functional unit utilization threshold of 4 (*fuu4*), on instructions per cycle. For each configuration, whenever the processor issue rate (or number of busy functional units in the case of *fuu4*) drops below the respective threshold, I force loads to complete. For comparison, I include the IPC values for the traditional memory system (tmem).

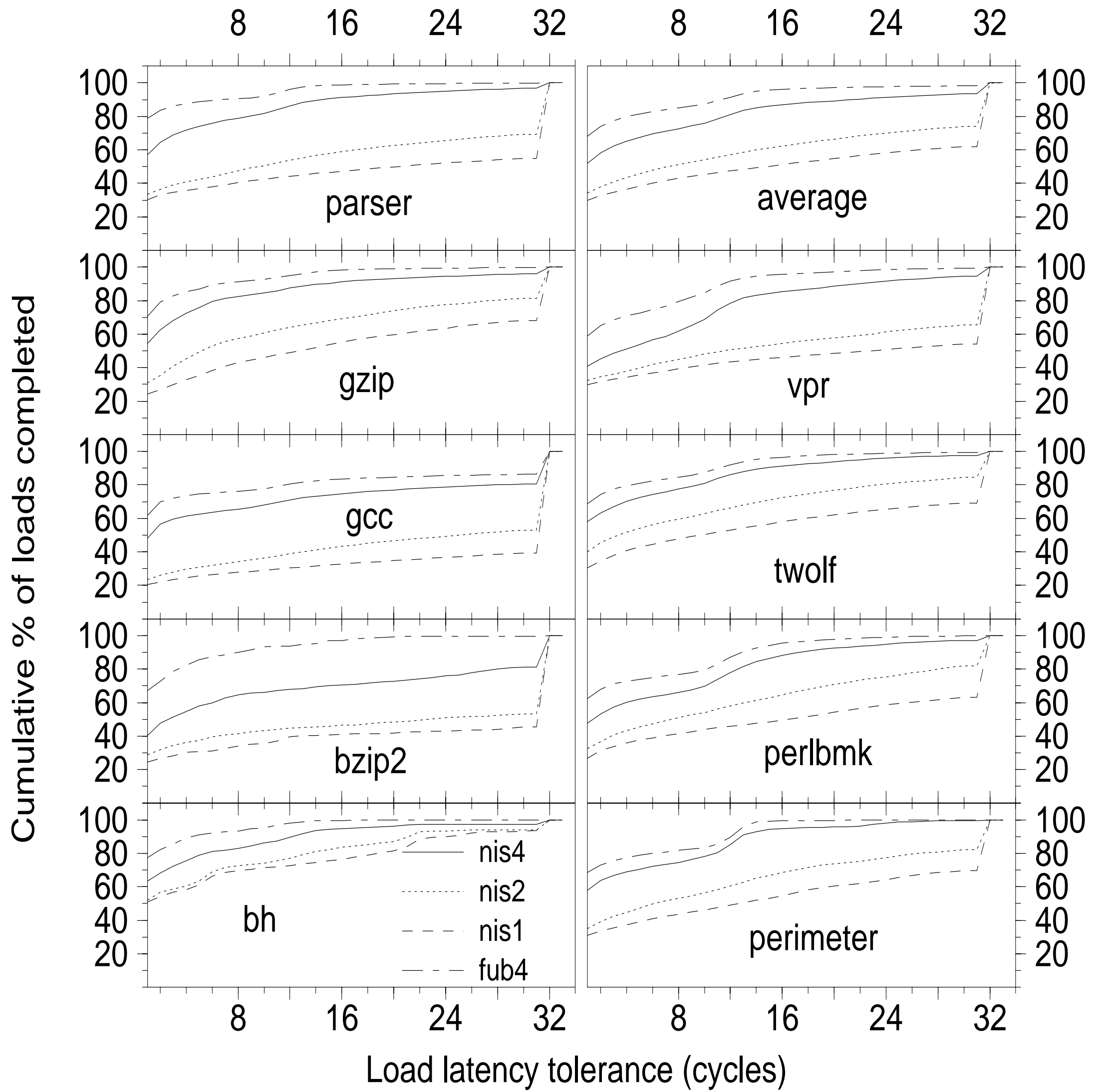


Figure 2.7: Performance-based Load completion and Latency Tolerance

From this data we can see that the metric functional unit utilization produces slightly higher IPC values than instruction issue rate (*fuu4* vs. *nis4*).

The simulations also reveal that decreasing the instruction issue rate threshold produces a commensurate decrease in IPC. This is true for lower functional unit utilization thresholds (*fuu2*, *fuu1* – not shown in the Figure) as well. Note that for all but two of the benchmarks (gzip and perlbmk), IPC values are still higher than

the traditional two-level memory system even when the threshold is one instruction per cycle.

Figure 2.7 shows the corresponding latency tolerance numbers. The first observation is that functional unit utilization (*fuu4*) produces much lower latency tolerance than the instruction issue rate metric (*nis4*). The functional unit utilization metric counts only the number of integer and floating point units that are busy each cycle. It does not include memory operations (loads and stores) or instructions that do not require computational units (such as TRAPs and PAL system calls). Thus, it is a stricter metric than the instruction issue rate metric and causes more loads to complete early. This leads to higher IPCs and lower load latency tolerance compared to the instruction issue rate metric.

We can also observe that decreasing the issue rate threshold can dramatically increase the latency tolerance. This matches our intuition that if the processor is consuming data at a lower rate, it can take longer for the data to arrive. However, the cost of this increased latency tolerance is reduction in IPC. A four instruction per cycle threshold produces IPC values within 12% of that of a processor with an ideal memory system, whereas a threshold of one instruction per cycle produces IPC values up to 41% lower than ideal. Therefore, in this Chapter, I do not consider thresholds of one or two any further. Similarly, I omit further discussion of the functional unit utilization metric since the decreased latency tolerance more than offsets the marginal increase in IPC.

2.6.3 Determining Which Loads to Complete

Now that it has been determined that either mis-predicted branches attaching to a load's dependence graph or the instruction issue rate falling below four should force load completion, I focus on identifying which outstanding load to complete.

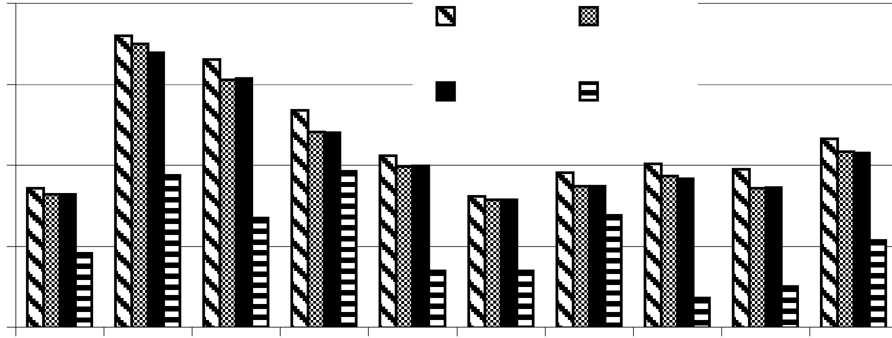


Figure 2.8: Load Selection and IPC

For branch based load completion, I need to complete all loads that feed the mispredicted branch. Hence, the choice of which load to complete arises only for the processor performance load completion case.

Completing the load at the head of the LSQ (*fifo*) can prevent processor stalls due to the RUU/LSQ being full and thus help alleviate the finite resource problem. On the other hand, completing the load with the maximum depth (in cycles) of dependent instructions (*dg*) will help tackle the data dependency problem by freeing up the most dependent instructions and thereby keep the processor maximally utilized.

Figure 2.8 shows the effect of the load selection policy on IPC and Figure 2.9 shows the corresponding latency tolerance values. The Figures show that both the *fifo* and *dg* load selection policies produce similar IPC numbers. However, completing loads based on the dependence graph depth increases the latency tolerance of loads for many benchmarks. Therefore, *dg* is a better choice since it meets my goal of maximizing the time a load can remain outstanding while maintaining high performance. These results indicate that completing loads in program order is not necessarily the "best" schedule for exploiting load latency tolerance.

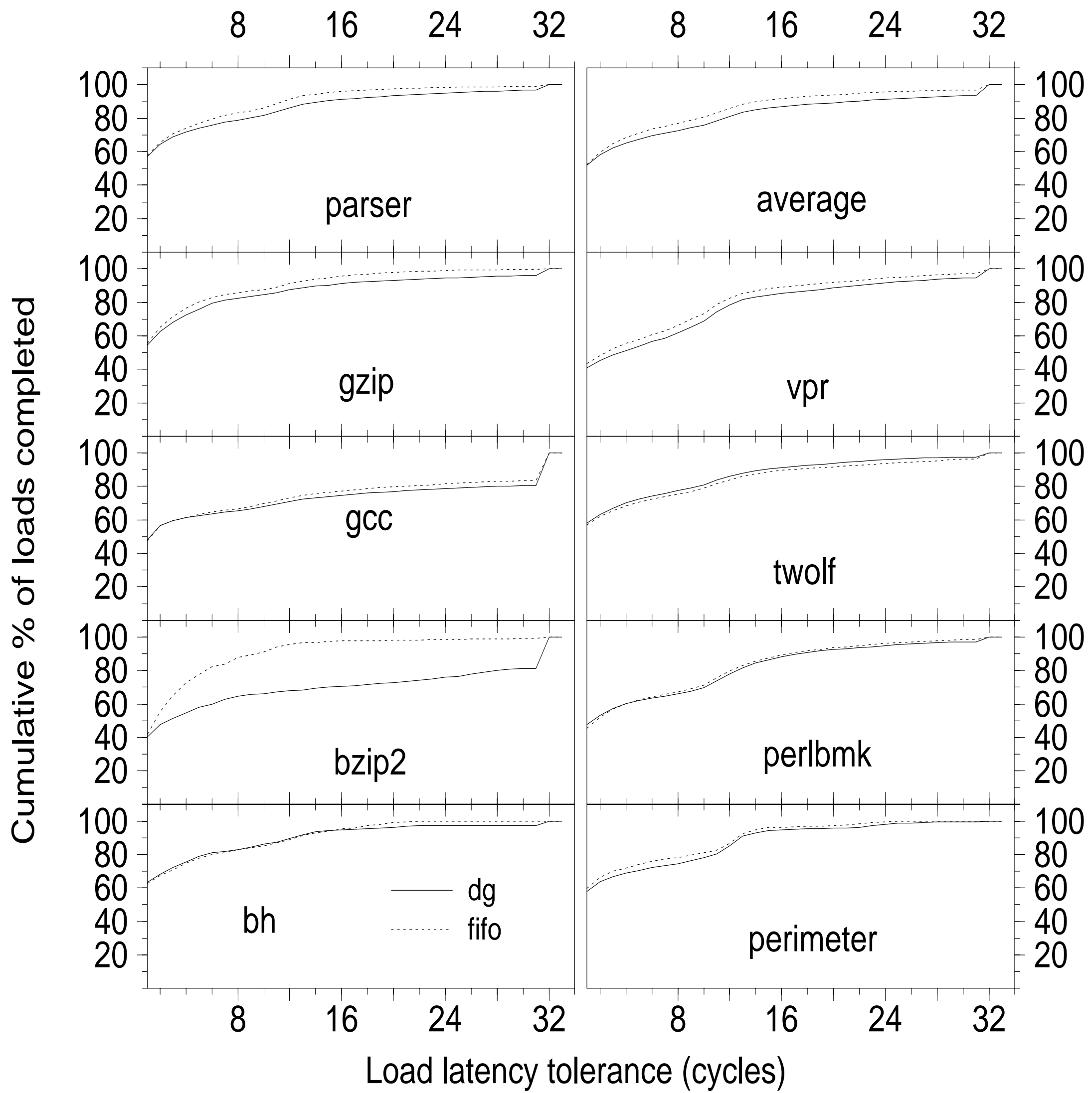


Figure 2.9: Load Selection and Latency Tolerance

2.6.4 Determining When to Complete Loads

Having decided to complete the loads with the maximum depth of dependent instructions, I proceed to investigate when such loads should be completed. The load completion time controls the duration available for the pipeline to fill up with ready instructions. I study the effect of completing loads the same cycle as detecting per-

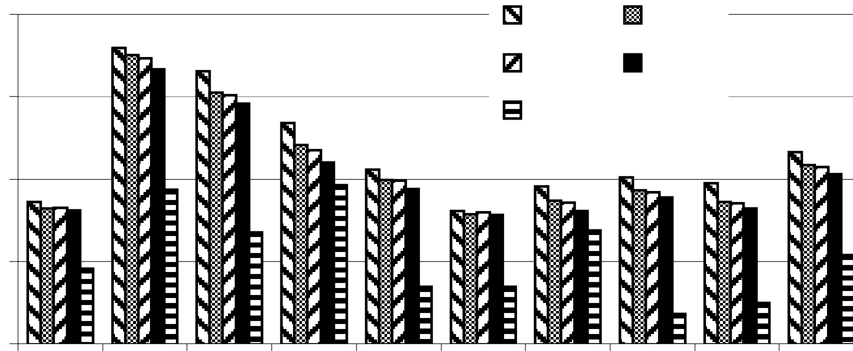


Figure 2.10: Completion Time and IPC

formance degradation (pipeline fill-up time of zero − *fut0*), one cycle earlier (*fut1*) and two cycles earlier (*fut2*) on load latency tolerance. Note this only applies to loads that are completed by the performance criteria and does not apply to loads that are forced to complete due to a mis-predicted branch.

From Figure 2.10 we can see that as the fill-up time decreases, IPC also decreases. On average, the performance degradation compared to ideal for *fut2* is 7% while it goes up to 11% for *fut0*. The maximum performance degradation in IPC over ideal is 12% for *fut2* while it goes up to 18% for *fut0*. Looking at the corresponding latency tolerance graphs in Figure 2.11, latency tolerance generally decreases as we increase the fill up time.

However, the processor requires some recovery time as results propagate down the dependence graph and a sufficient number of instructions become ready to execute. Hence, I use a pipeline fill-up time of 2 cycles (*fut2*), which produces a good combination of IPC and latency tolerance numbers.

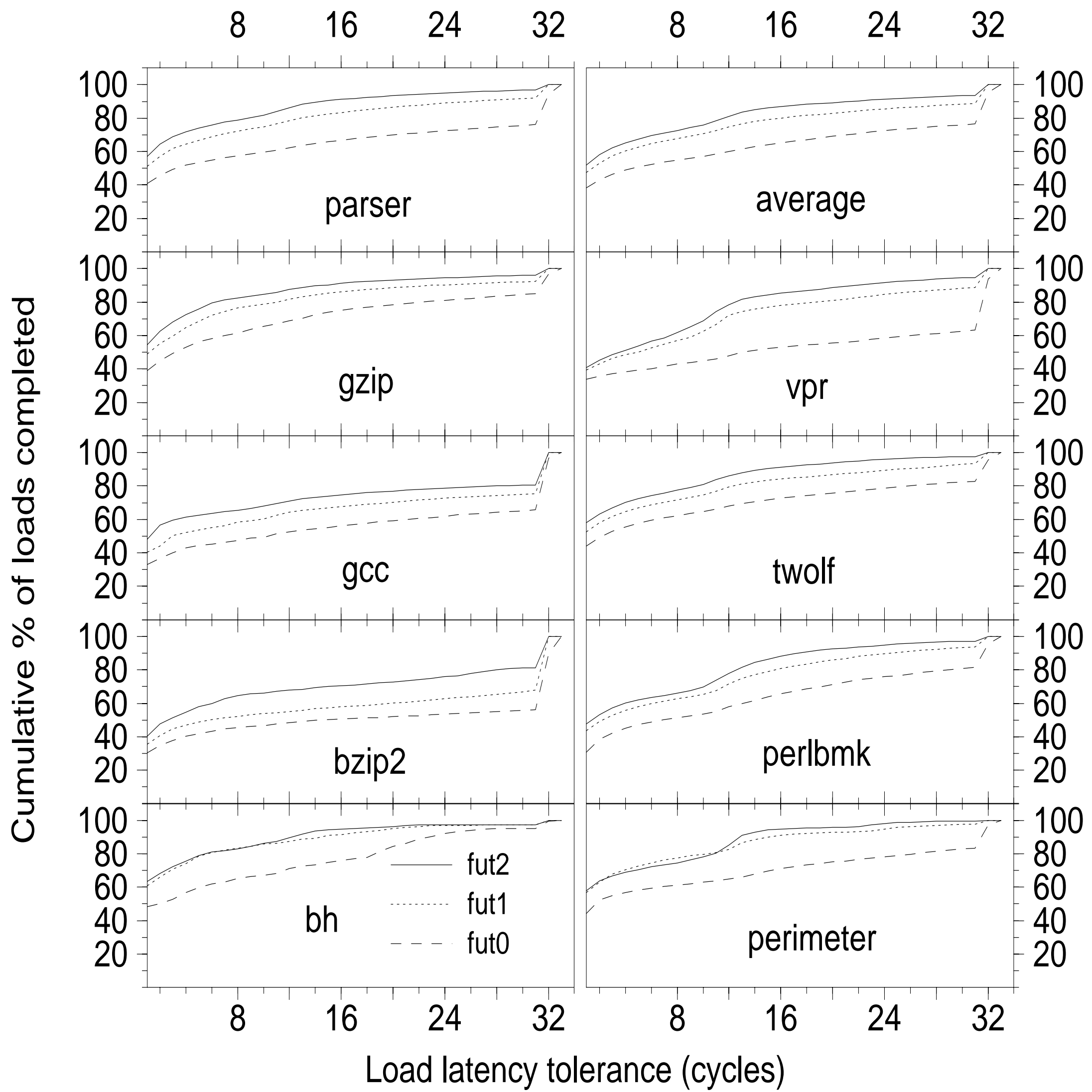


Figure 2.11: Completion Time and Latency Tolerance

2.6.5 Limiting the Number of Completed Loads

Finally, achieving IPCs close to that of a processor with an ideal memory system will likely require completing more than one load per cycle. Keeping all other parameters fixed, I examine limits of one (*nl1*), two (*nl2*) and four (*nl4*) on the number of loads that can complete in a single cycle.

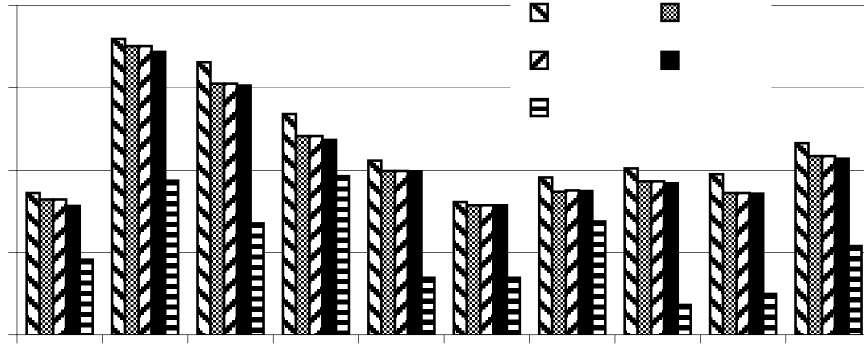


Figure 2.12: Load completion limit and IPC

Figure 2.12 shows the impact of these limits on IPC and Figure 2.13 shows the corresponding latency tolerance numbers. The overall trend we can observe from this data is that, as we increase the limit on the number of loads that can complete in a cycle from 1 to 2, IPC increases and the latency tolerance decreases. This is in line with expectations, since a lower limit causes some loads to complete later than they should according to the rest of my load completion policies, which causes a decrease in IPC. Both *nl2* and *nl4* produce almost identical IPC and latency tolerance values for these set of benchmarks. However, for some of the floating point benchmarks that I use earlier [51], a load completion limit of four loads per cycle produces IPC numbers closest to ideal. Hence, for the rest of my experiments, I use a load completion limit of four loads per cycle.

2.6.6 Effects of Processor Micro-architecture

To evaluate the impact of various micro-architectural changes on load latency tolerance, I use an 8 issue processor with 128 RUU entries and 64 LSQ entries, and 4 issue processors with both the 128/64 and 256/128 RUU/LSQ configurations. In the case

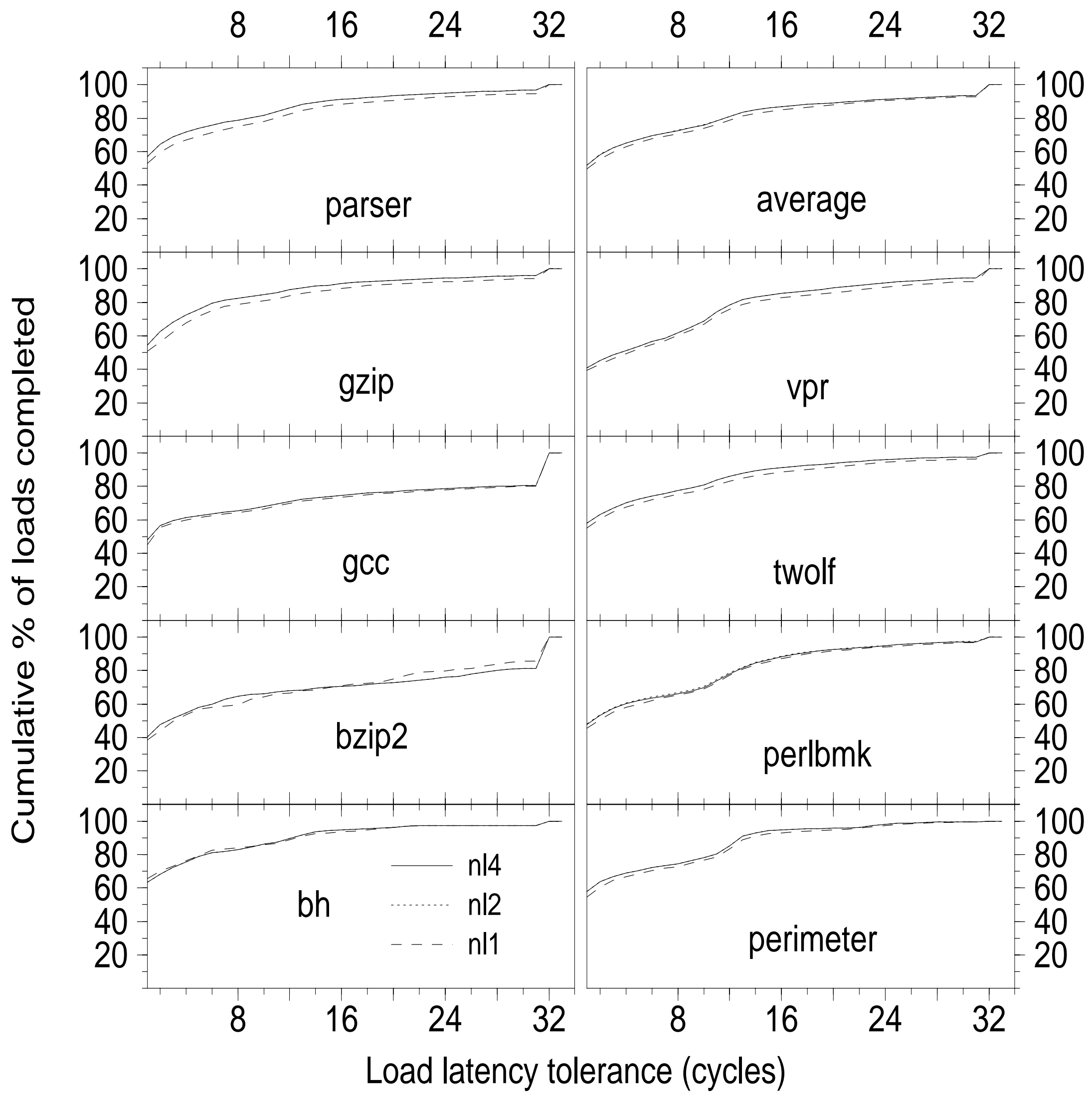


Figure 2.13: Load completion limit and Latency Tolerance

---

of the four issue processor, I use an issue rate threshold of three instructions per cycle and up to three loads can complete in a single cycle. Figure 2.14 shows the effect of various processor configurations on IPC and Figure 2.15 shows the corresponding latency tolerance numbers. The graphs are labeled according to issue-width/RUU entries/LSQ entries.

Note that all the IPC values shown are within 18% of the corresponding processors

Figure 2.14: Processor Architecture (Issue-width / RUU-size / LSQ-size) and IPC

with an ideal memory system. The first observation from this data is that the IPC values are mostly independent of the number of RUU/LSQ entries and that the issue width has a much larger impact on IPC than buffer space does. However, the situation is very different with respect to the amount of latency tolerance. From Figure 2.14, we can see that both issue width and buffer space have significant impact on load latency tolerance. In general, latency tolerance increases when either the issue width decreases or the RUU/LSQ entries increases.

2.7 Load Latency: Expected vs. Realized

The results presented thus far indicate that loads do exhibit a significant variation in latency tolerance. Next, I evaluate the effectiveness of traditional memory systems in capturing this variation. To determine this, in my rollback simulator, along with the computed latency tolerance for each load I record which level in a traditional memory hierarchy satisfies that load. If the computed latency tolerance for a load is less than the L2 cache access latency, then the load needs to be satisfied in the L1 cache. Otherwise, it can be satisfied by the L2 cache or by main memory. Based on

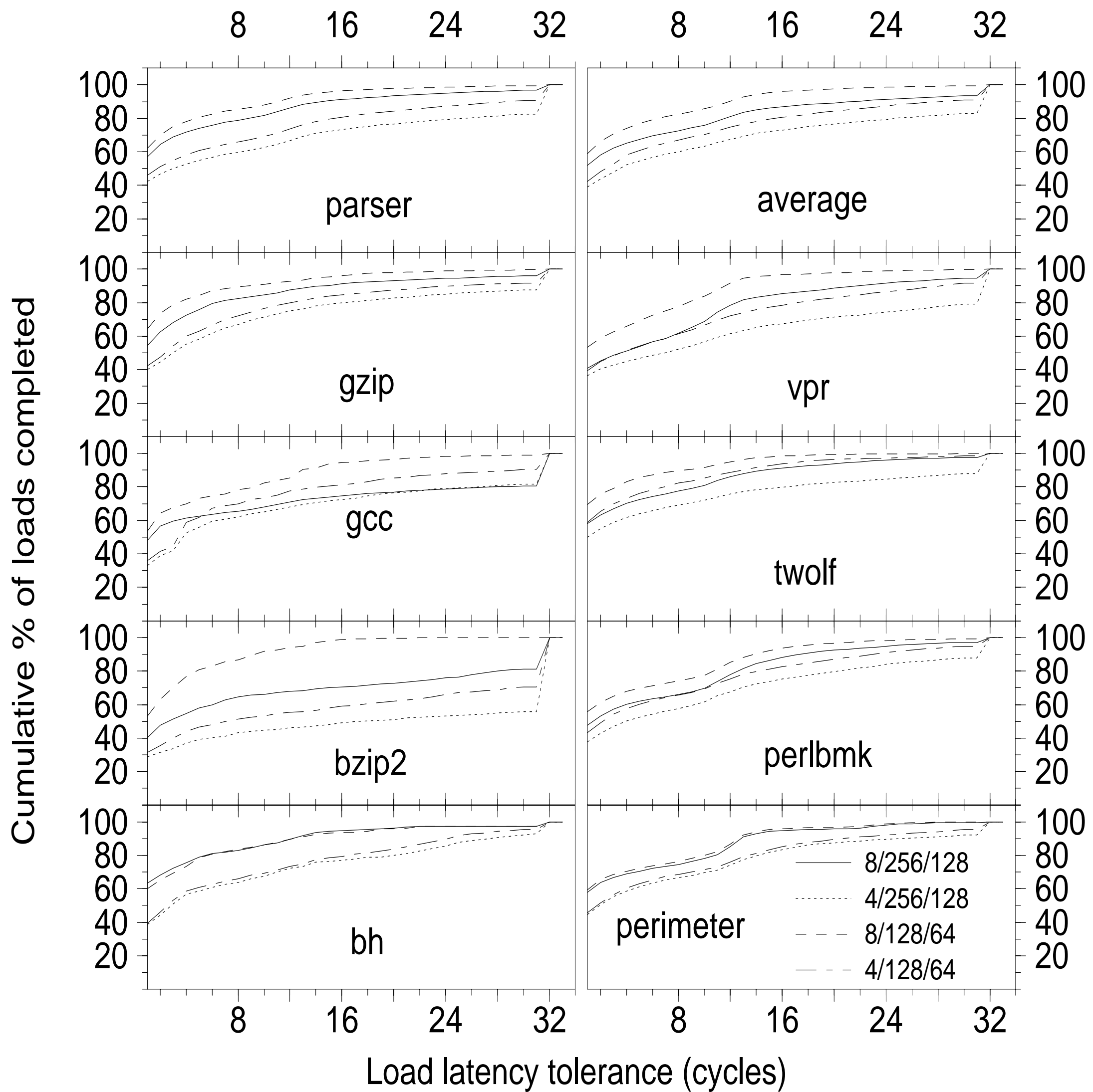


Figure 2.15: Processor Architecture (Issue-width / RUU-size / LSQ-size) and Latency Tolerance

---

which level in the memory hierarchy the load is actually satisfied, I determine if the realized latency of the load matches its expected latency.

I perform this analysis on an L1 cache size of 8KB, and vary the L1 cache associativity as direct-mapped, 2-way set-associative, and 4-way set-associative. I also conduct this study for different L2 cache latencies of 6, 8 and 10 cycles using the base

Table 2.2: Effectiveness of Traditional Memory Systems at Capturing Critical Loads

L2 cache configuration (256KB, 64-byte blocks). I present results only for an 8KB L1 cache with varying associativity and an L2 cache latency of 8 cycles because the results using other L2 cache latencies are very similar.

Table 2.2 shows the effectiveness of the traditional two-level memory hierarchies at capturing latency tolerance. The top row for each benchmark in the Table is for loads with computed latency tolerance less than 8 cycles (critical loads). Similarly, the

bottom row for each benchmark corresponds to loads with computed latency greater than or equal to 8 cycles. The columns indicate which level in the memory hierarchy the loads are satisfied. The levels of the memory hierarchy are the load/store queue (LSQ), L1 cache, L2 cache, and main memory.

I focus my discussion on the L1 and L2 caches. In particular, the number of low latency tolerance loads (critical loads) not satisfied by the L1 cache and the number of high latency tolerance loads (non-critical loads) satisfied by the L1 cache indicates the mismatch between the applications latency demands and the latency incurred in the memory hierarchy.

From Table 2.2 we can see that for all benchmarks there is some discrepancy between the latency demands of loads and the latencies incurred in a real memory hierarchy. Consider the 2-way set-associative cache for twolf, 8% of loads have low latency tolerance but are satisfied by the L2 cache, whereas 22% of loads have high latency tolerance and are L1 cache hits.

For all benchmarks, the percentage of loads that have high latency tolerance but are satisfied by the L1 cache (benign mis-match) is higher than the percentage of loads that have low latency tolerance but get satisfied in the L2 cache (malign mis-match). If somehow, the set of malign mis-match loads can be swapped with the set of benign mis-match loads and be satisfied by the L1 cache with a low latency, performance could improve. Ideally, this performance improvement could be between 25% and 412% (average 101%) if all loads are completed based on their computed latency tolerance as opposed to a traditional memory system. The rest of this thesis investigates if these potential performance benefits can be realized using realistic load criticality exploitation schemes.

2.8 Conclusions

The results presented in this Chapter indicate that there is a considerable variation in the latency tolerance of loads. Across the different benchmarks, I find that between 19% and 79% of loads need to complete in one cycle, while between 6% and 50% of loads can wait for more than 8 cycles to complete and still achieve IPCs within 12% of an ideal memory system.

The load completion studies reveal that loads that feed mis-predicted branches need to complete early to reduce mis-speculation, and loads with few independent instructions need to complete early to minimize performance degradation due to the data dependencies and the finite resources problems. I also find that completing loads in decreasing order of dependence graph depth produces more latency tolerance while still achieving similar IPCs as completing loads in program order.

I also observe that there is a mis-match between load criticality and load latencies realized using a traditional memory system. If this gap between load criticality and load latency can be bridged, average performance improvements of 101% can potentially be obtained. To achieve this in a realistic processor, I must (i) design a practical scheme to classify loads as critical or non-critical, and (ii) provide a processor/memory system that exploits critical loads to improve performance. Chapter 3 focuses on identifying critical loads in practice. The subsequent Chapters deal with criticality based optimization design.

Chapter 3:
Critical Load Classification

Many of the insights from the studies in the previous Chapter can be used to develop a practical implementation for determining load criticality. My experiments using the rollback processor indicate that load criticality is determined to a large extent by the chain of instructions dependent on the load. In particular,

- The type of dependent instructions (e.g., mis-predicted branch) indicates if the load is likely to degrade processor performance, and
- The number of instructions in its dependence chain indicates if a long latency load will stress the processor's finite resources. If most instructions issued after a load are dependent on the load, the processor may not be able to find independent instructions to execute and hence could stall.

From these observations, an online criticality computation scheme can be devised using the following critical load classification criteria.

3.1 Critical Load Criteria

Branch Criteria

The branch based load completion results in Section 2.6.2 indicate that loads feeding mis-predicted branches are critical. Hence, my online load criticality computation scheme must determine if a load feeds into a mis-predicted branch or not (parameter B).

Cache Criteria

In the rollback scheme, all loads that feed into other loads are given priority for early completion in order to unravel memory level parallelism (MLP). With a

traditional memory hierarchy, MLP depends on the number of cache misses that are simultaneously outstanding. To expose MLP, we need to get to the cache misses as early as possible. Hence in my practical criticality computation scheme, it may be sufficient to complete early only those loads that feed into other loads that miss in the L1 (L1) cache. Therefore, another criteria for classifying a load as critical is whether it feeds into a load that misses in the L1 cache or not (parameter C).

Issue Criteria

Corresponding to the performance criteria in the rollback scheme, my online scheme uses an Issue criteria that takes into account the processor's ability to tolerate a load's latency. This criteria keeps track of the number of independent instructions issued (parameter P) in a certain window (called the issue window) following the load's issue. If there are a lot of independent instructions issued in a load's issue window, it indicates that the processor is able to tolerate the load's latency, and hence the load should be classified as non-critical. If the number of independent instructions issued in cycles immediately following the load is less than an issue threshold, then the processor is unable to tolerate the load's latency, and so the load should be marked critical.

Thus the criticality of a load is a function of these three parameters B, C, and P. My criticality computation scheme uses a simple binary function that is represented by the flowchart in Figure 3.1 to compute the criticality of loads. If a load feeds into a mis-predicted branch, or if it feeds into another load that misses in the L1 cache, or if the number of independent instructions issued in the issue window of the load is less than an issue threshold, then the load is critical. If all of these three conditions are false, then the load is non-critical.

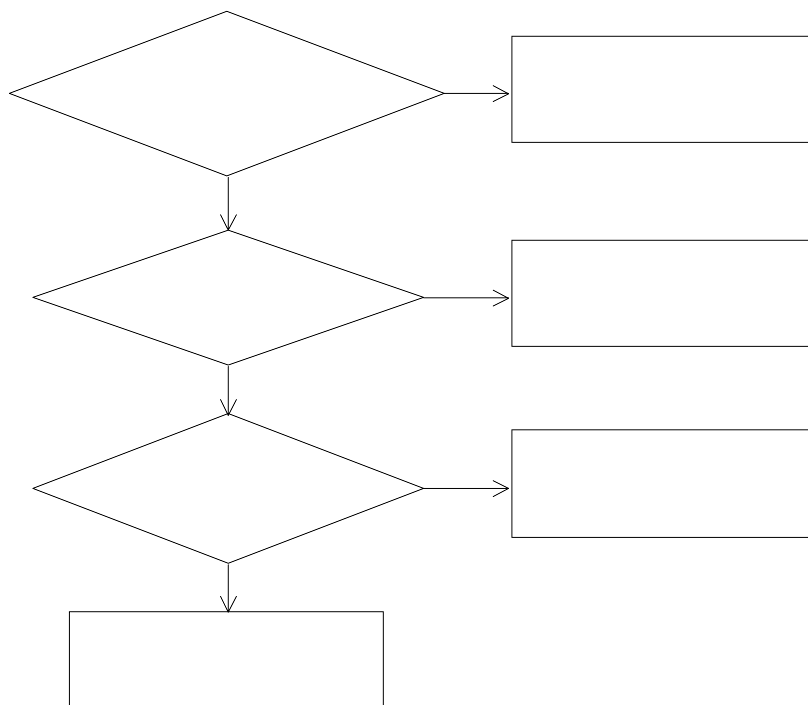


Figure 3.1: Flowchart to compute Load criticality

3.2 Criticality Table

Each of the critical load classification criteria require information about the chain of instructions dependent on a load. Such a chain of dependent instructions is called the dependence subgraph or the forward slice of the load. The Branch criteria and the Cache criteria require monitoring the type of instructions (mis-predicted branch or L1 cache miss) in a load's dependence subgraph, while the Issue criteria requires counting the number of independent instructions (instructions that are not in the load's dependence subgraph) issued in cycles immediately after the load's issue. Thus, one of the main things needed to determine the criticality of a load is the ability to keep track of the dependence subgraph of loads.

To track a load's dependence subgraph, I add a hardware structure called the Criticality Table (CT). Figure 3.2 shows a sample CT. The CT has one row for each

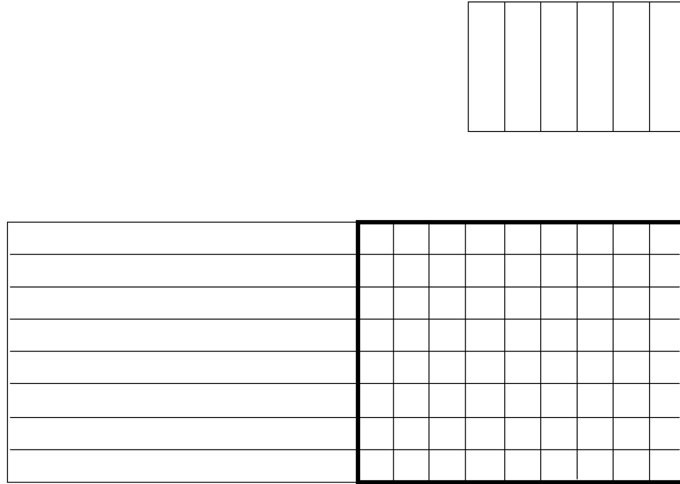


Figure 3.2: Criticality Table

entry in the RUU. Each row in the Criticality Table has a Branch flag, a Cache flag, an Issue counter, and a Dependence vector. The Branch flag and Cache flag are used to indicate the type of dependent instruction, the Issue counter holds the number of independent instructions following a load and the Dependence vector tracks load dependencies of instructions.

The Branch flag is used for branch instructions alone and a set flag indicates that the instruction occupying the corresponding RUU entry is a mis-predicted branch. Similarly, the Cache flag is used only for load instructions, and indicates if the instruction is a load that missed in the L1 cache or not.

The Dependence vector is a bit-vector and has one bit per LSQ entry. If bit j of the Dependence vector is set for RUU entry i, then instruction i is dependent on the jth entry in the LSQ. The LSQ holds both loads and stores, but since only loads have instructions dependent on them, bits that are set in the dependence vectors indicate load dependencies alone.

To understand how the Dependence vector bits are updated, consider the example dependence graph shown in Figure 3.3. Using the register→source mapping from the

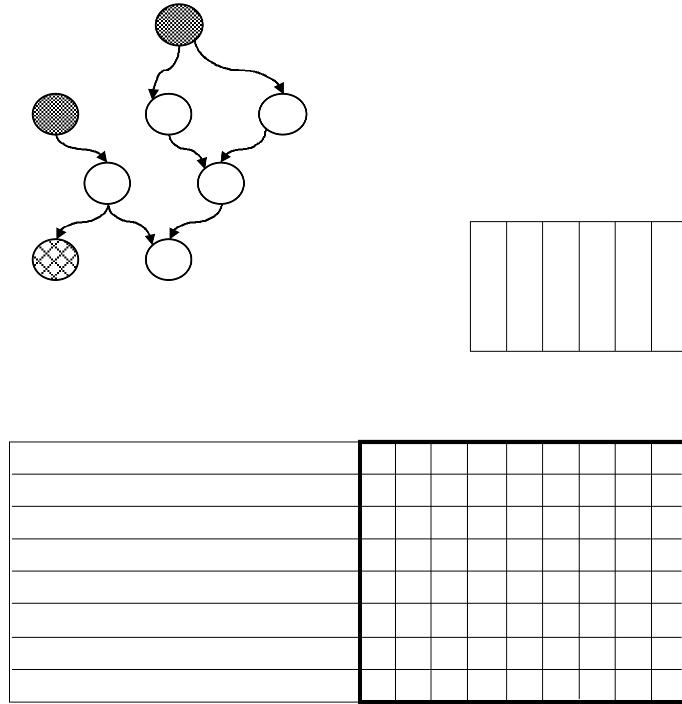


Figure 3.3: Criticality Table

dependence logic, it can be determined that the operands of I7 are produced by instructions I4 and I5. I4 depends on load I1, while I5 depends on load I0. I7 inherits the dependencies of both its source instructions I4 and I5 and hence it depends on both loads, I0 and I1. In order for the Dependence vector to reflect this, the dependence information of both instructions I4 and I5 needs to be propagated down to I7. This is done by performing a bitwise OR of the Dependence vectors of I4 and I5 to arrive at the Dependence vector for I7.

At any instance of time, dependence bits along a row in the CT indicate which currently active loads a particular instruction depends on, while the bits along a column correspond to the instructions dependent on a specific load. The number of instructions dependent on a load is given by the sum of the column in the CT corresponding to the load. The column sum of LSQ entry 0, that corresponds to load

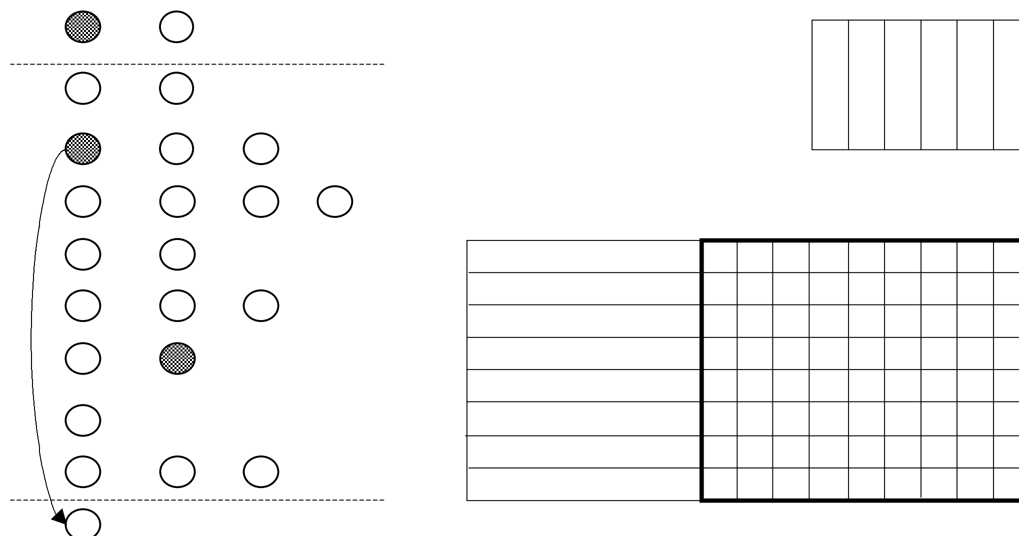


Figure 3.4: Issue Counter Update

I0 is five, which is the same as the number of instructions dependent on I0 (including itself) in the dependence graph.

To determine if a mis-predicted branch is dependent on a load (B) I compute a bit-wise AND of the column in the CT corresponding to the load with the Branch flag column. If the result is non-zero, then the load has a mis-predicted branch dependent on it. In the above example, the Branch flag column bits are 00000010 and the bits in the column corresponding to load I1 are 01001011. The result of a bitwise AND of these two columns is 00000010, which indicates that load I1 has a mis-predicted branch in its dependence chain. Similarly, to determine if a load feeds into another load that misses in the cache or not (L) I perform a bitwise AND of the Cache flag column with the load's column in the CT.

The functioning of the Issue counter can be understood by means of the example in Figure 3.4 which shows a stream of instructions that are issued and the time they are issued. Suppose that instruction 22 is issued in time t and that the issue window

size is 8 cycles (i.e. the number of independent instructions issued in the 8 cycles following each load's issue needs to be computed).

The Dependence vector for instruction 22 will indicate that it is dependent on load 4 and is independent of loads 0 and 17. Since instruction 22 is dependent on load 4, the Issue counter for load 4 is not incremented when instruction 22 is issued. Instruction 22 falls outside the 8 cycle window for load 0 and so the Issue counter for load 0 is also not incremented. Instruction 22 is both within the 8 cycle window and is independent of load 17. Hence, the Issue counter for load 17 is incremented.

By this process, at the end of 8 cycles after each load is issued, its Issue counter will have the number of independent instructions issued during its issue window. Choosing appropriate values for the issue window size and the issue threshold are addressed in Section 3.5. Once the values for the parameters B, C and P for a particular load are known, its criticality can be determined using the flowchart in Figure 3.1.

The CT is updated at different stages of the processor pipeline. The Branch flag in the CT can be set after the corresponding branch is resolved, in the Writeback stage. Similarly, as soon as the L1 cache access for a load is completed, the Cache flag of the load's CT entry can be updated using the hit/miss information. The processor's dependence check logic maintains information about which currently active RUU/LSQ entries are going to produce each register value in order to rename registers. I modify the dependence check logic to retain the register→source mapping until the source RUU/LSQ is re-used or until another source RUU/LSQ writes to that register. Using this information, the Dependence vector bits and the Issue counters can be updated in the Issue stage of the pipeline.

Benchmark	Overall Criticality Prediction Accuracy	Critical Load Prediction Accuracy	Non-critical Load Prediction Accuracy
	in percentage (%)		
bh	93.0	89.6	94.8
bzip2	94.6	73.5	98.0
gcc	86.0	61.3	92.5
gzip	88.9	66.9	95.2
parser	77.6	65.4	84.3
perimeter	72.9	70.9	74.4
perlbmk	86.4	75.0	91.3
twolf	77.8	71.5	82.1
vpr	72.2	53.3	80.9
average	83.2	69.7	88.2

Table 3.1: Criticality Predictor accuracy

3.3 Load Criticality Predictor

The above scheme can determine load criticality only after a certain window after a load issues. The Issue criteria requires an issue window of 8 cycles. Even after these 8 cycles, a mis-predicted branch or a load that misses in the L1 cache could attach itself to the load's dependence chain. Hence, I monitor each load's dependence subgraph until its RUU/LSQ entry is reused and compute its criticality only when its RUU/LSQ entry is reused. However, many schemes that exploit load criticality require the classification information much earlier, even before a load is issued. For this purpose, I use a load criticality predictor.

The criticality predictor is similar to a two level branch predictor. It uses two bits of local criticality history per load-PC and two bits of global branch history. Using global branch history serves to correlate the path information leading to a load with its criticality. I find that having more than two global branch history bits does not improve the criticality prediction accuracies by much.

Table 3.1 shows the load criticality prediction rates using a 4096 entry load criti-

cality history table. The average load criticality prediction rate across all the benchmarks is 83%. Among the critical loads, 70% are correctly predicted as critical, while among the non-critical loads, 88% are correctly predicted as non-critical, on average.

The following Section shows that if these loads that are predicted to be critical are captured in the L1 or L2 cache, there is substantial opportunity for performance improvement.

3.4 Criticality Computation Validation

Now I have a hardware scheme to classify loads as critical and non-critical. Next, I evaluate how good my scheme is in identifying critical loads. To evaluate my criticality computation scheme, I force all loads that are predicted to be critical to hit in the cache. If a load that is predicted to be critical misses in the cache, I choose a victim block and instantaneously update its tag and data with those of the critical load. Thus, I fake a cache hit if a predicted critical load misses in the cache. This scheme is denoted as *crl*.

I compare the crl scheme with two other schemes, *rand-crl* and *rand-miss*. The *rand-crl* scheme randomly classifies loads as critical such that the percentage of loads classified as critical matches that of the *crl* scheme and forces them to hit in the cache. Even if the *crl* scheme performs better than the *rand-crl* scheme, one could argue that *crl* classifies more cache misses as critical and thereby forces more cache misses to hits than *rand-crl*. To determine if *crl* is capable of achieving superior performance by correctly identifying critical loads and not just by forcing more cache misses to hits, I compare it with another scheme, *rand-miss*. The *rand-miss* scheme randomly converts the same proportion of load misses to hits as the *crl* scheme. I conduct these experiments at both the L1 and L2 cache levels.

Figure 3.5 and Figure 3.6 show the percentage improvement in Instructions Per

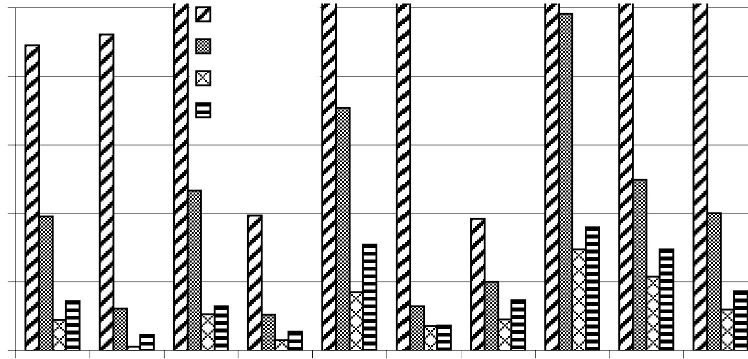


Figure 3.5: Potential Criticality-based Performance Improvement at L1 cache level

Cycle (IPC) over a traditional memory system when loads are forced to hit in the L1 and L2 cache respectively. The case where all loads hit in the cache (shown as ideal L1/L2 in Figure 3.5 and Figure 3.6) represents an upper bound on the performance improvements possible.

At the L1 cache level, the *crl* scheme achieves 40% improvement in performance on average, whereas the average performance benefits for *rand-crl* and *rand-miss* are only 12% and 17%, respectively. The average performance improvements at the L2 cache level are 43%, 20%, and 31% for the *crl*, *rand-crl*, and *rand-miss* schemes respectively. These results suggest that my classification scheme does a good job of identifying the set of performance critical loads.

In general, forcing loads to hit in the L2 cache achieves higher performance than forcing loads to hit in the L1 cache. This is because the L2 block-size is larger than the L1 block-size, and forcing loads to hit in the L2 cache creates additional L2 hits due to spatial locality. Also, the gap between the performance improvements produced by *crl* and *rand-miss* is narrower at the L2 cache level because the number of unique cache blocks referenced by the misses is much smaller at the L2 cache level

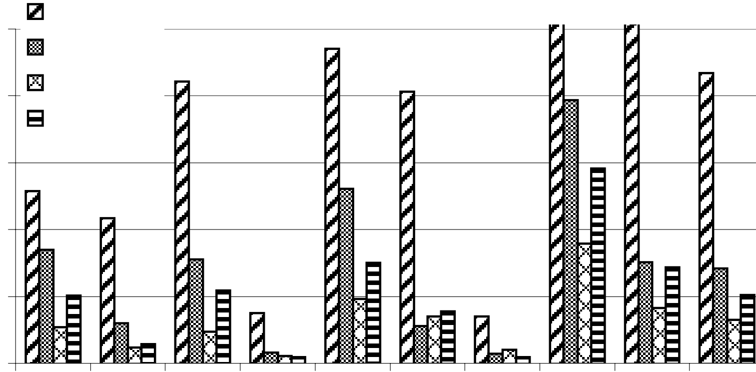


Figure 3.6: Potential Criticality-based Performance Improvement at L2 cache level

due to factors such as smaller number of cache misses, larger cache block size and spatial locality between cache misses. Hence at the L2 cache level, the chances that the same cache blocks are converted from a miss to a hit by both *crl* and *rand-miss* are greater.

The average performance improvements were shown to be 101% using the rollback scheme. However, using the practical on-line scheme we see that the average performance improvements when all loads classified as critical are guaranteed to hit in the L1 cache is only 40%.

The main reason for this drop in potential performance improvement is that the parameters in the rollback scheme were aimed at achieving performance close to an ideal memory system. For example, an instruction issue rate threshold of four instructions per cycle was chosen to achieve IPCs within 12% of the ideal memory system. This would produce an average performance improvement of 101% over a traditional memory system, but would require nearly 70% of loads to be classified as critical and be satisfied in the L1 cache.

Reducing the instruction issue rate threshold to 2 instructions per cycle would

decrease the average performance improvements for the rollback scheme over the traditional memory system to 78% from 101%, but will also correspondingly decrease the average percentage of critical loads to 48% from 70%. I trade-off a lower number of critical loads for higher performance benefits in order to facilitate criticality based optimization schemes. Hence I choose an issue threshold of 2 instructions per cycle for the online scheme that is comparable to the instruction issue rate threshold of 2 instructions per cycle in the rollback scheme.

Using an issue threshold of 2 instructions per cycle in the online scheme, about 30% of all dynamic loads are classified as critical on average. Out of this 30%, 12% are classified as critical based on the Branch criteria, 5% based on the Cache criteria and 13% based on the Issue criteria. I find that the average percentage of loads classified as critical based on the Branch criteria (12%) matches that of the rollback scheme. However, the Performance criteria of the rollback scheme is more aggressive than the Issue criteria of the online scheme (the rollback scheme requires 2 instructions to be issued each cycle, while the online scheme requires an average of 2 independent instructions to be issued over an 8 cycle window for each load). Hence, the rollback scheme classifies more loads as critical based on the Performance criteria than the online scheme's Issue criteria.

Another reason why the online scheme achieves lower performance improvements compared to the rollback scheme is that the rollback scheme has perfect future knowledge which enables it to schedule load completions appropriately. The online scheme however is an approximation of the rollback scheme: it predicts that a load is critical by implicitly forecasting a future event such as a branch mis-prediction, a cache miss, or very few independent instructions for the load. This prediction mechanism introduces some inaccuracy (83% criticality prediction accuracy on average) in the online critical load classification process, thereby decreasing the potential benefits. In spite

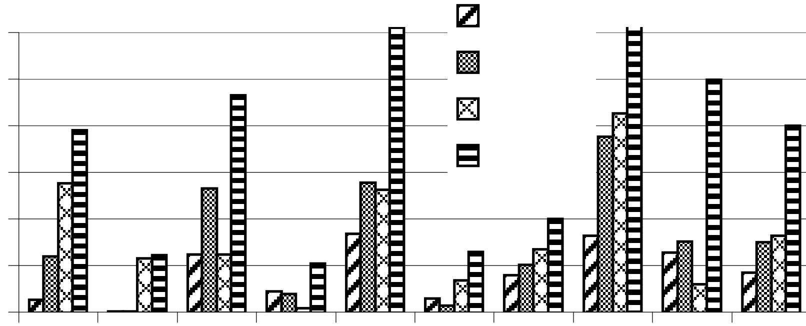


Figure 3.7: Critical Load Classification Criteria: Performance Benefits

of this limitation, the average 40% performance gains obtainable by capturing loads classified as critical in the the L1/L2 cache indicates that there is considerable scope for improvement.

3.5 Classification Criteria Comparison

To understand the performance contributions of the individual classification criteria (Branch, Cache, and Issue criteria), I compare three separate runs, each with just one of the classification criteria enabled and force loads classified as critical during each run to hit in the cache. Figure 3.7 compares the performance improvements of the individual classification criteria over a traditional memory system at the L1 cache level. The performance benefits when all three criteria are enabled is shown by the bar labeled *all*.

From Figure 3.7, we can see that the average performance gain is 9% for the Branch criteria, 15% for the Cache criteria, and 16% for the Issue criteria. This suggests that using the individual criteria alone for classifying loads as critical will drastically reduce the performance improvements (maximum 16% compared to a total

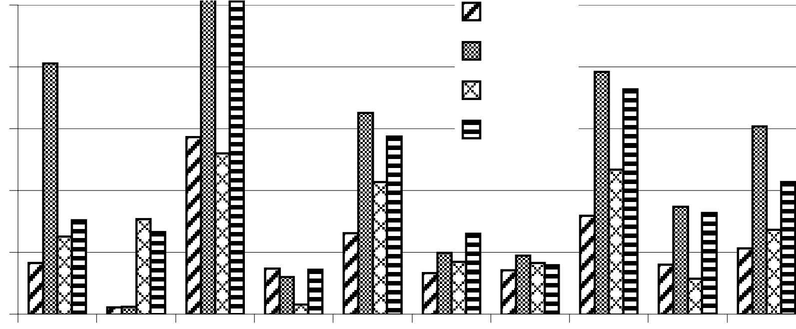


Figure 3.8: Critical Load Classification Criteria: Benefits per Critical Load

of 40%) possible. Therefore, I use all three criteria for classifying loads as critical.

The Issue criteria gives the most benefits, but it also classifies the most number of loads as critical, on average. Figure 3.8 compares the three classification criteria using a normalized metric: performance improvements per million loads classified as critical. For most of the benchmarks, the Cache criteria gives the most performance per critical load classified. This suggests that for these benchmarks, exposing MLP by completing loads feeding L1 cache misses early has a larger impact on performance than increasing ILP or minimizing mis-speculative execution.

3.6 Related Work

There are several definitions of criticality in previous literature. Calder et al. [4] define instructions on the longest path at any point of time as critical. Fisk and Bahar [13] consider counting the number of dependent instructions that attach to the load, or monitoring the processor issue rate during the time a cache miss is being serviced to classify loads as critical.

These schemes look at only the number and not the type of dependent instructions

to classify loads as critical. Secondly, Fisk and Bahar determine the criticality of loads only when they miss in the L1 cache and assume that its criticality will remain the same during its lifetime in the cache, which is not often the case. Also, the criticality information for a load is available only after the cache miss is satisfied. Most criticality exploitation schemes need the criticality information as soon as a load is issued. My dynamic critical load classification mechanism addresses all the above issues.

Recently, Fields et al. [12] proposed a dependence graph model of the critical path and devised a predictor that follows the model to identify instructions on the critical path at run-time. Their dependence graph model captures information about both the number and type of dependent instructions similar to my scheme. However, their criticality predictor relies on hysteresis and uses per-instruction criticality history alone. I find that utilizing the correlation between the criticality of loads and the path that leads to the loads (global branch history) improves criticality prediction accuracy and hence use it for my criticality classification scheme.

Dependence graph based analysis has also been used in the past [2, 27, 36, 47, 57, 64] to study the amount of parallelism available in programs. Such studies use a static/dynamic execution trace with idealistic assumptions such as unconstrained resources, single cycle operational latencies for functional units, perfect branch prediction and alias analysis. These studies typically use the dependence information prior to an instruction to find the optimal schedule for a program.

In contrast, this thesis uses information about instruction dependencies that follow a load as well as processor utilization information to compute the criticality of loads. Also, in this thesis the criticality of loads are computed on-line on a realistic processor with constrained resources.

This thesis uses a hardware structure called the Criticality Table to track the

dependence subgraph of loads. Similar structures are used by Calder et al. [4] and Fields et al. [12] to determine the set of instructions on the critical path.

3.7 Conclusions

This Chapter presents a practical online scheme for computing the criticality of loads and shows that if the loads classified as critical by this scheme are captured in the L1 or the L2 cache, significant performance improvements can be achieved. The challenge is to deliver this potential with realistic criticality exploitation schemes. The proposed hardware technique for determining load criticality is just one component of the solution. The following Chapters investigate the other component: criticality-based optimization schemes.

Chapter 4: Locality vs. Criticality

4.1 Introduction

To exploit the variation in load criticality, I first focus on caches [45] and the memory hierarchy. Caches work by exploiting spatial and temporal locality [7] in a program's access pattern. Spatial locality is exploited by fetching a region of memory – called a cache block – rather than just the accessed data. Caches exploit temporal locality by retaining recently accessed cache blocks. The goal of most cache management schemes in exploiting locality is to increase the fraction of memory accesses satisfied by the cache (i.e., cache hit ratio).

Although increasing the overall number of cache hits is usually desirable, my load criticality measurement results presented in Chapter 2 show that not all memory accesses are equal. The latency of some memory load operations (critical loads) can have a much larger influence on overall performance than other non-critical loads. Therefore, it may be possible to improve overall performance by decreasing the latency of critical loads at the expense of increased latency for non-critical loads.

The goal of this Chapter is to determine if criticality is a strong enough program property to warrant a change in memory hierarchy management techniques for practical implementations. Specifically, I investigate if practical criticality-based approaches can equal or surpass the performance of existing locality-based techniques.

For this purpose, I compare criticality-based load latency reduction techniques, specifically, criticality-based cache organization and prefetching schemes with corresponding locality-based schemes. My criticality based cache organization scheme uses a critical cache that is functionally similar to a conventional victim cache [22],

but holds only blocks that were touched by a critical load. I also examine multi-line prefetching [21, 25] based on both locality and criticality.

The primary result from my simulations is that load criticality is not a sufficiently strong property to warrant cache management changes. Although, criticality can be used as a filter for making prefetching decisions, it is not worth the added complexity. In particular, my results reveal the following:

- Managing caches based on locality outperforms criticality-based techniques. The working set of critical loads is large and because of spatial locality between non-critical and critical loads, a locality-based cache provides lower critical load miss ratios than a criticality-based cache.

- Criticality-based prefetching into the L2 cache achieves an average speedup of 4%, compared to 2% for locality-based prefetching, across several benchmarks with resource constraint problems. However, if resource constraints are not a problem, the most aggressive locality-based prefetching scheme performs best.

The remainder of this Chapter is organized as follows. I evaluate cache organization schemes in Section 4.2. Section 4.3 investigates prefetching. Section 4.4 discusses related work and Section 4.5 concludes this Chapter.

4.2 Criticality-based Cache Organization

The most significant component of a load's latency is the level in the memory hierarchy where the accessed data resides. Performance may improve if critical loads can be satisfied by caches close to the processor even if non-critical loads suffer increased misses. This Section examines criticality-based cache organization.

The central idea behind criticality-based caching is to keep critical data[1] close

[1] I define critical data as data touched at least once by a critical load while resident in the cache.

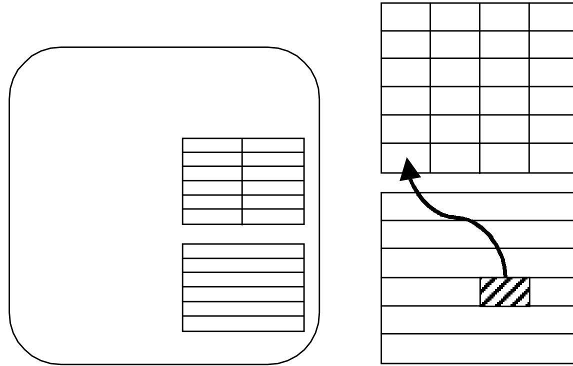


Figure 4.1: Critical Cache Illustration

to the processor at the potential expense of non-critical data. To achieve this, I investigate both a new cache replacement policy and a different cache organization. In this thesis, I focus on an alternative cache organization called a critical cache. However, as I discuss later in this Section, I achieve similar results even with modified replacement techniques.

A critical cache serves as a victim cache [22] for critical data. When a primary cache sub-block, equal to the critical cache block size, is touched by a critical load, I set a corresponding critical bit. When the block is replaced from the primary cache, the sub-blocks with the critical bit set are copied to the critical cache. My design uses a smaller block size for the critical cache than the primary cache, to store only the critical portions of a cache block.

Figure 4.1 shows an illustration of the critical cache next to the L1 and L2 caches. On every reference, both the primary cache and the critical cache are accessed in parallel. If the requested data is found in either of the cache structures, it is treated like a cache hit. When a reference hits in the critical cache, all sub-blocks that are part of the primary cache block are transferred to the primary cache and a request is sent to the next level in the memory hierarchy to fetch any missing sub-blocks.

The equivalent locality based caching scheme consists of a locality cache alongside

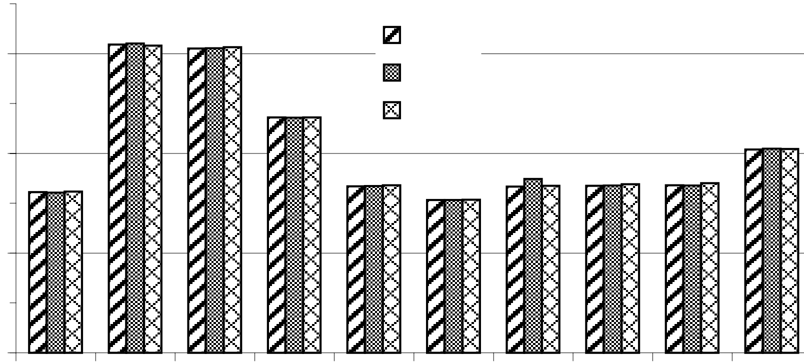


Figure 4.2: L1 Content Management: IPC

the primary cache. The capacity, block-size and associativity of the locality cache match those of the critical cache. Any block that is evicted from the primary cache is transferred to the locality cache, which helps retain both critical and non-critical data longer than just the primary cache. The function of the locality cache is similar to a conventional fully-associative victim cache. However, I want a sufficiently large capacity, and hence use lower associativity to keep access time low. The remainder of this Section compares a critical cache with an equal sized locality cache at both the L1 and L2 cache levels.

4.2.1 Locality vs. Criticality at the L1 Cache Level

For experiments at the L1 cache level, I model an ideal L2 cache (all L2 cache accesses are hits) to eliminate the L2 cache performance effects. The critical/locality cache configuration consists of a 4KB 2-way set associative L1 cache with 16B lines and a 4KB 2-way set associative critical/locality cache with 8B lines. Even though the sum of the cache sizes in the critical cache configuration is equal to the primary cache size in a traditional memory system (*tmem*), it is different from splitting *tmem* into a critical half and a non-critical half, because of the smaller block-size of the

critical cache. Also, in general, the critical/locality cache could have a higher associativity than the primary cache. However, simulations of up to 8-way associative critical/locality caches reveal no significant changes in my results.

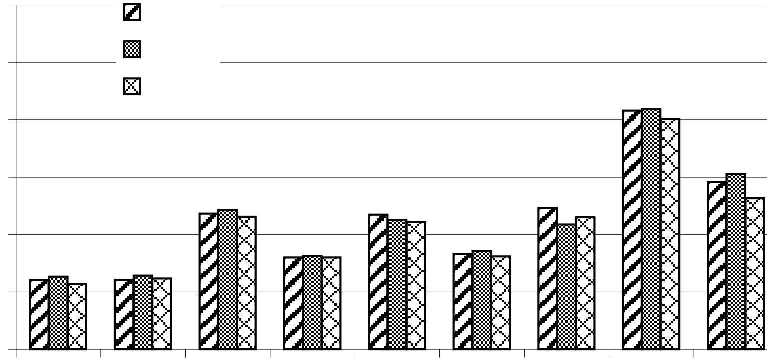
Figure 4.2 compares the IPC numbers of a critical cache (*cc*) and a locality cache (*loc*) with those of *tmem*. These results show there is little change in the IPC numbers compared to *tmem* for most benchmarks. The exception is perlbmk, where a critical cache improves performance by 5%.

Figure 4.3 shows the overall L1 cache miss ratio and the critical load miss ratio for the three configurations. The critical cache achieves the lowest critical load miss ratios among the three configurations, in-line with expectations. The critical cache however, increases the overall miss ratio for all benchmarks except parser and perlbmk, while the locality cache decreases the overall miss ratio for all benchmarks. The reductions in the critical load miss ratios compared to *tmem* are not significant enough to translate into noticeable performance gains, except for perlbmk.

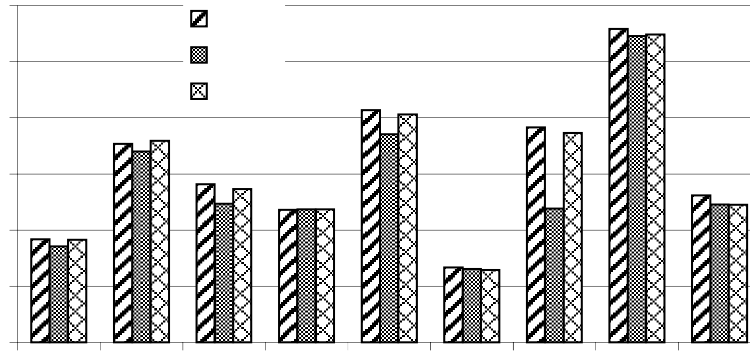
4.2.2 Locality vs. Criticality at the L2 cache Level

To compare criticality-based caching with equivalent locality based caching at the L2 cache level, I use two different configurations. One configuration has a 128KB 2-way set associative critical/locality cache with 16B lines alongside a 128KB 2-way set associative L2 cache with 64B lines. The second configuration has an 8KB 2-way set associative critical/locality cache with 16B lines along with a 256KB 2-way set associative L2 cache with 64B lines.

Figure 4.4 shows the IPC values for each L2 cache organization. From this data we can see that neither critical cache configuration achieves performance benefits over a traditional memory system. Moreover, when half of the available cache space is used as a critical cache (*cc:128k+128k*), performance drops by 21% for bzip2 and



(a) Overall Miss Ratio



(b) Critical Load Miss Ratio

Figure 4.3: L1 Content Management: Miss Ratio

by 31% for gzip. gzip with the *loc:128k+128k* configuration, is the only case that achieves noticeable performance improvement (8%) over a traditional cache.

By examining the overall miss ratio in Figure 4.5(a), we can see that *cc:128k+128k* increases the overall miss-ratio for all benchmarks except bh, twolf and vpr. We can also observe that the locality-based cache performs comparably to the criticality-based scheme for most benchmarks. From the critical load miss ratios, shown in Figure 4.5(b), we can see there is no significant reduction in the critical load miss

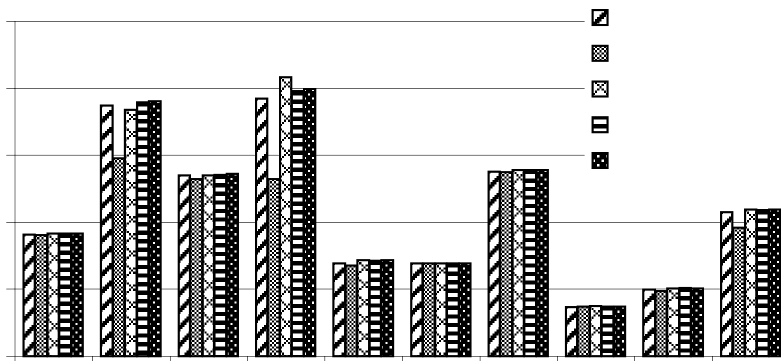
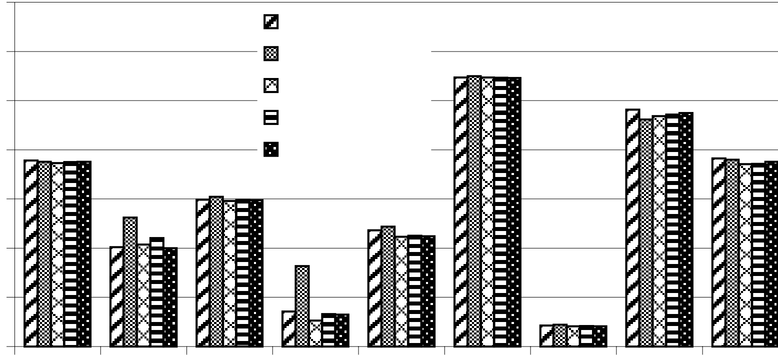


Figure 4.4: L2 Content Management: IPC

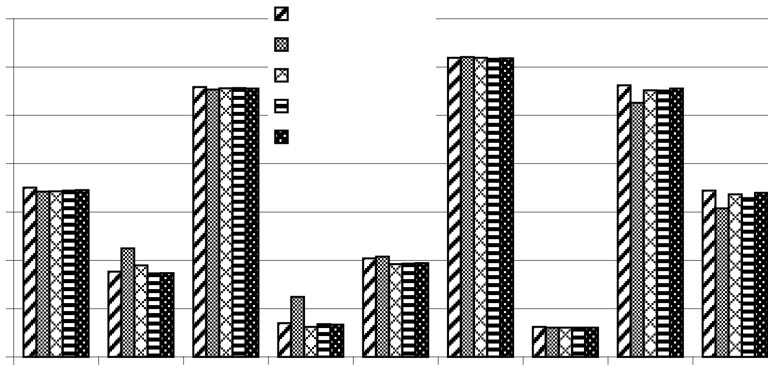
ratio for the critical cache or locality cache configurations. Furthermore, in contrast to my expectations, bzip2 shows an increase in the critical load miss ratio for both the 128k+128k locality/critical cache configurations, while gzip suffers an increase in the critical load miss ratio for the 128k+128k critical cache configuration. These increased miss ratios correlate with a commensurate decrease in performance relative to a traditional memory system.

I observe qualitatively similar results using larger caches, up to a 32KB L1 cache and up to a 1MB L2 cache. The above results indicate that retaining critical blocks longer by means of a critical cache does not decrease the critical load miss ratio enough to improve performance. I arrive at similar conclusions from experiments with a criticality-based cache replacement scheme that uses an aging policy to let critical blocks remain in the cache longer, and bypasses non-critical data when necessary.

The premise of criticality-based cache organizations or replacement policies is that critical loads are a small fraction of all references, and hence their working set will be much smaller than the overall program working set. The smaller working set should reduce competition in the criticality-based caches and produce lower critical load



(a) Overall Miss Ratio



(b) Critical Load Miss Ratio

Figure 4.5: L2 Content Management: Miss Ratio

miss ratios than traditional caches based on locality. However, my simulations do not show significant reductions in critical load miss ratios for criticality-based caches.

4.2.3 Critical Load Working Set

One possible explanation for the above results is that the programs access too much critical data. If this is true, it will be difficult for any practical cache organization or replacement policy to exploit criticality information. To further explore this hypoth-

esis and to gain insight into the above results, the remainder of this Section examines the working set sizes of the benchmarks.

I measure working set size by determining the smallest cache size required to obtain a specific miss ratio. I simulate cache sizes from 4KB to 16MB, with an associativity of 8, to minimize conflict misses, and a fixed cache line size of 16B. Since I am only interested in cache miss ratios and since the number of simulations required is large, I use trace driven simulation instead of detailed performance simulation. Using Simplescalar, I collect a reference trace for each of the benchmarks that includes criticality information for loads. I skip the first 3.9B instructions and gather traces over the execution of the next 200M instructions. In my cache simulations, the first 100M of the trace instructions are used to warm-up the caches. Thus, the period of execution I evaluate using traces matches the simulations in the rest of the thesis.

To estimate the overall working set, I simulate traditional caches managed based on locality and obtain the overall miss ratios (*locality.all*). To determine the critical load working set, I simulate criticality-based caches that allocate cache blocks only on critical load misses and obtain the critical load miss ratios (*criticality.crl*). I also note the critical load miss ratios for the locality based runs (*locality.crl*) which indicates how well the locality based caches are able to capture the critical load working set. For completeness, I also present the overall miss ratios for the criticality based runs (*criticality.all*). Figure 4.6 plots the overall and critical load miss ratios for both the locality and criticality based cache configurations.

Assume I define the working set as the smallest cache size required to capture 95% of the references (a 5% miss ratio, the solid line in Figure 4.6). To obtain the *overall working set*, I find the smallest cache size that captures 95% of *all references* (*overall miss ratio* less than 5%), whereas to get the *critical load working set*, I determine the smallest cache size that captures 95% of the *critical load references*.

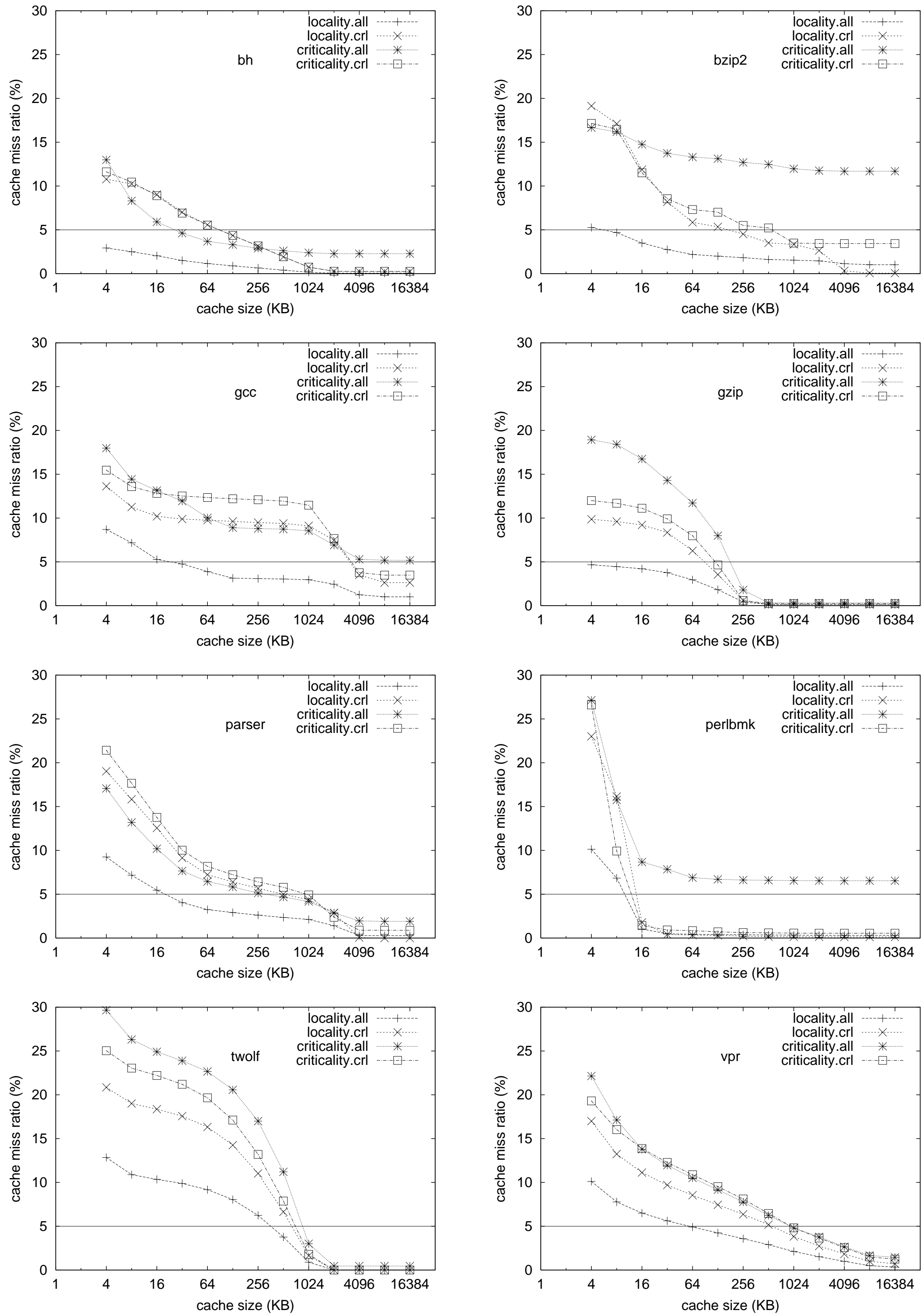


Figure 4.6: Working Set

Benchmark	Overall Working Set	Critical Load Working Set	
	locality	locality	criticality
bh	4	128	128
bzip2	8	256	1024
gcc	32	4096	4096
gzip	4	128	128
parser	32	512	1024
perlbmk	16	16	16
twolf	512	1024	1024
vpr	64	1024	1024

Table 4.1: Working Set: Cache size in KB required to achieve 95% cache hit ratio

Table 4.1 shows the overall and critical load working sets for the benchmarks. From Table 4.1, we can see that most benchmarks have quite large critical load working sets, 16KB to 4MB. The benchmark perlbmk, has the smallest critical load working set size of 16KB and it is also the only benchmark whose critical load working set is comparable in size to the overall working set. For these reasons, perlbmk is the only benchmark for which the critical cache shows performance gains at the L1 cache level, as seen earlier. Note that perlbmk's working set size is much smaller than the L2 cache size and hence a critical cache next to the L2 cache is not of much use.

The large critical load working sets are due to the fact that critical references are not concentrated on a few effective addresses, but rather are spread across a large portion of the virtual address space. This can be clearly seen from Figure 4.7(a) that shows the percentage of references that are critical, for each effective address (32B block) touched by twolf. This is true for the other benchmarks also (see Appendix 7.2).

This distribution of critical references across most of the effective addresses can in turn be attributed to different load PCs that touch different effective addresses

(a) Effective Addresses

(b) Program Counters

Figure 4.7: Critical Load Distribution

becoming critical over the lifetime of a program. Figure 4.7(b) shows a typical criticality distribution per load PC (Appendix 7.2 shows the criticality distribution per load PC for the rest of the benchmarks). From Figure 4.7(b), we can see that the criticality of load PCs vary, and that most load PCs are classified as critical at some point during program execution.

The instability in the criticality of load PCs can be understood by examining the individual critical load classification criteria. The Branch criteria classifies a load as critical based on whether a dependent branch is mis-predicted or not. Similarly, a load is classified as critical by the Cache criteria if a dependent load misses in the L1 cache. The number of independent instructions for a load (needed for the Issue criteria) is often determined by whether branches in the vicinity of the load are taken or not. The overall criticality distribution of load PCs reflects a combination of the distributions of branch mis-predictions, L1 cache misses, and branch directions.

Thus the very nature of programs and their execution, leads to large working sets for critical loads, and prevents criticality-based content management schemes from reducing competition for cache space by reducing the working set. This is the primary reason why criticality-based cache organizations do not decrease the critical load miss ratios significantly compared to a traditional locality-based memory system.

Load Criticality: Stability vs. Predictability

In spite of the instability in criticality of load PCs, I am able to achieve reasonably good criticality prediction ratios (average 83% accuracy). This is because a load's

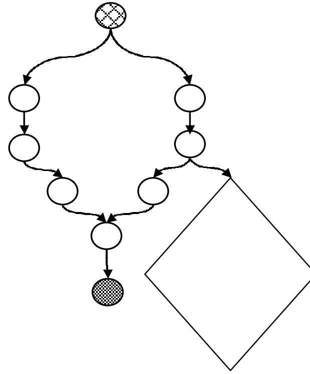


Figure 4.8: Load Criticality: Unstable, yet predictable

criticality could be predictable and still be changing over the execution of a program. Consider an example load *ld* shown in Figure 4.8. If the branch *br* is taken, we can see from Figure 4.8 that there are a lot of instructions independent of *ld* and *ld* will be non-critical. If *br* is not taken, then there are very few independent instructions while *ld* issues. In this scenario, the processor will be unable to tolerate *ld*'s latency and *ld* will be critical. Thus, the criticality of *ld* depends on the direction of *br* and could be changing throughout the execution of the program. But since my criticality prediction mechanism takes into account the path leading to a load (by correlating load criticality with global branch history), the criticality of *ld* could still be predictable.

Compulsory Critical Load Misses

Another observation from the critical load working set study (Figure 4.6) is that in most cases, locality-based caches achieve lower critical load miss ratios than criticality-based caches, particularly for bigger cache sizes. This is because in locality-based caches, non-critical misses can prefetch some critical data due to spatial locality between critical and non-critical sub-blocks within a block. Any criticality-based scheme that prevents non-critical misses from allocating a cache block discounts the

possibility of future critical accesses to the entire cache block and hence fails to exploit this property. This leads to a higher number of compulsory critical load misses for criticality-based caches than locality-based caches.

This effect can be clearly observed in the case of bzip2, and gcc in Figure 4.6. For bzip2, the critical load miss ratio settles down at 3.5% for criticality-based caches beyond a size of 1MB, while it continues to drop to almost 0% for locality-based caches. These misses seen for very large cache sizes are compulsory misses. As the cache size increases, compulsory critical load misses tend to dominate the overall number of critical load misses. Since locality-based caches have lower compulsory critical load misses compared to criticality-based caches, the former perform better for large cache sizes. This places the criticality-based cache organization schemes at a further disadvantage compared to locality-based schemes.

4.2.4 Summary

The results in this Section suggest that locality is a better property than criticality for managing cache content. The goal of the criticality-based schemes discussed so far has been to retain critical data in the cache longer and thereby convert subsequent critical load cache misses into hits. However, since the critical load working set is very large, simply modifying cache replacement or allocation policies alone is not sufficient to retain critical data for long enough periods of time to achieve significant reductions in the critical load miss ratio.

To move towards the 40% performance improvements that could be obtained by exploiting information on critical loads, I must investigate alternate solutions. Prefetching is a technique that can be used to bring in critical data just ahead of use. The advantage of prefetching over caching is that it does not require cache space to hold critical data for extended periods of time. I study prefetching for critical loads

in the next Section.

4.3 Criticality-based Prefetching

Prefetching exploits the spatial nature of accesses and the regularity of access patterns by predicting future accesses and initiating a memory access even before a load is actually issued. Accurate address prediction, proper timing, minimal cache pollution and minimal interference with regular load requests are essential ingredients of a good prefetching technique.

4.3.1 Prefetching Schemes

Early research concentrated on prefetching for all loads [5, 18, 21, 25, 34]. My goal is to reduce cache pollution and resource requirements by prefetching for critical loads alone. These selective prefetching techniques are aimed at achieving improvements in critical load cache hit ratios, thereby leading to higher performance.

I evaluate two proposed prefetching schemes: the Reference Prediction Table (RPT) [5] and the Spatial Footprint Predictor (SFP) [25]. RPT keeps track of the stride (current effective address - previous effective address) of each load PC, and issues a prefetch whenever a steady stride is observed. SFP groups multiple cache blocks into macro-blocks, and records the blocks within each macro-block that are touched – called the spatial footprint – during the macro-block's lifetime in the cache. When a miss to one of the blocks occurs, the entire spatial footprint of the macro-block is prefetched.

Comparing the two prefetching schemes RPT and SFP, my simulations show that SFP produces superior performance improvements for most of the benchmarks. Hence, I choose SFP over RPT for further evaluation. I make SFP critical load centric (*sfp-crl*) by modifying it to prefetch only the spatial footprint of critical loads, and

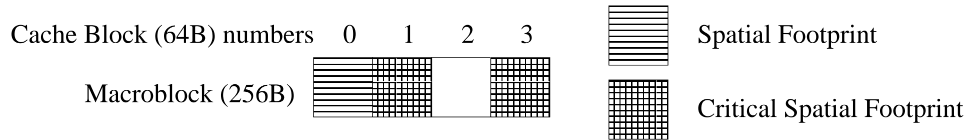


Figure 4.9: Prefetching Example

refer to the original SFP scheme as *sfp-all*.

I also evaluate another critical load centric prefetching scheme, called *mbp-crl* (Multiple Block Prefetch on critical load miss). *mbp-crl* prefetches an entire macro-block on a critical load cache miss and reverts to regular cache line fetch if a cache miss is from a non-critical load.

The motivation for *mbp-crl* is based on the following observation from my criticality analysis experiments: Cache lines that are nominated by a critical load (i.e. brought into the cache due to a critical load miss) are more likely to be referenced by subsequent critical loads than cache lines that are nominated by a non-critical load. I find that the average number of additional words touched by critical loads within a cache line is 4-5 times higher for cache lines nominated by a critical load than for cache lines nominated by a non-critical load. *mbp-crl* exploits this property at the macro-block granularity by prefetching an entire macro-block, whenever a cache line is nominated by a critical load, and thereby hopes to achieve more critical load hits.

The locality based scheme corresponding to *mbp-crl* is *mbp-all*, which prefetches an entire macro-block on all load misses. Store/write misses still fetch only one cache line which allows for finer granularity replacements, and thus *mbp-all* is different from having bigger cache lines.

To summarize the different prefetching schemes, consider the sample macro-block shown in Figure 4.9 that consists of 4 cache blocks. Assume that blocks 0, 1, and 3 constitute the spatial footprint of the macro-block, while the spatial footprint of critical loads consists of blocks 1 and 3. If there is a load miss to any of the blocks in

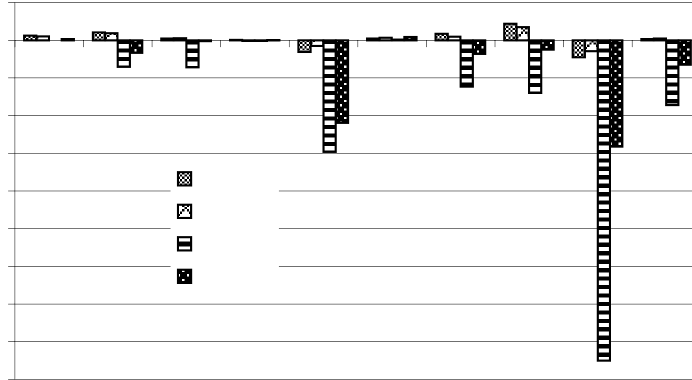


Figure 4.10: L1 Locality vs. Criticality Prefetching: IPC

the macro-block, *sfp-all* will prefetch blocks 0, 1, and 3, *sfp-crl* will prefetch blocks 1 and 3, and *mbp-all* will prefetch the entire macro-block. *mbp-crl* will prefetch the entire macro-block only if there is a critical load miss to blocks 1 or 3. In the remainder of this Section, I compare two criticality based prefetching schemes (*sfp-crl* and *mbp-crl*) with their corresponding locality based schemes (*sfp-all* and *mbp-all*) at both the L1 and L2 cache levels.

4.3.2 Prefetching Into the L1 Cache

When prefetching into the L1 cache, I simulate an ideal L2 cache to isolate the effects of prefetching into the L1 and L2 caches. I use an L1 cache line size of 16B and the macro-block size of 64B. I initiate prefetches on an L1 cache miss and check the L1 cache tags to issue prefetches only for those blocks that are not already in the cache. I find through experiments that queuing regular load requests the earliest in the MSHRs, followed by prefetch requests, followed by store misses, performs best, and hence, I use this ordering for all my runs.

Figure 4.10 shows the percentage improvement in IPC of the four prefetching

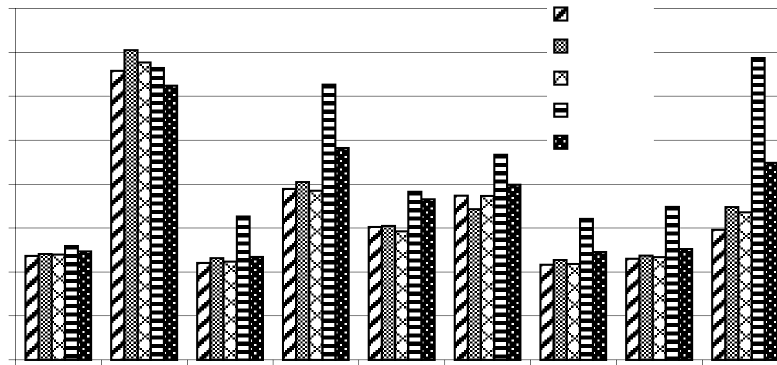


Figure 4.11: L1 Locality vs. Criticality Prefetching: Bandwidth

schemes over a traditional memory system. From Figure 4.10 we can see that neither the locality based schemes nor the criticality based schemes improve performance considerably. In fact, many decrease performance. The sfp schemes do not decrease performance for any of the benchmarks except parser and vpr, for which they decrease performance by less than 2%. However, *mbp-crl* decreases performance by 11% and 14% respectively for parser and vpr while *mbp-all* decreases performance by more than 4% for 6 out of the 9 benchmarks. In general, *mbp-all* decreases performance more than *mbp-crl* across all the benchmarks.

The main reason for the performance degradation due to prefetching at the L1 cache level is the resource constraints introduced by the prefetch requests. Prefetch requests can lead to performance degradation in two ways. First, they occupy MSHR entries and deny MSHR entries for regular load requests, thereby delaying them. Second, once a prefetch request has acquired the L1–L2 bus, a regular load request has to wait for the data transfer to be complete before acquiring the bus. This adds additional delays for regular loads.

Figure 4.11 shows the average completion delay of loads that hit in the L2 cache.

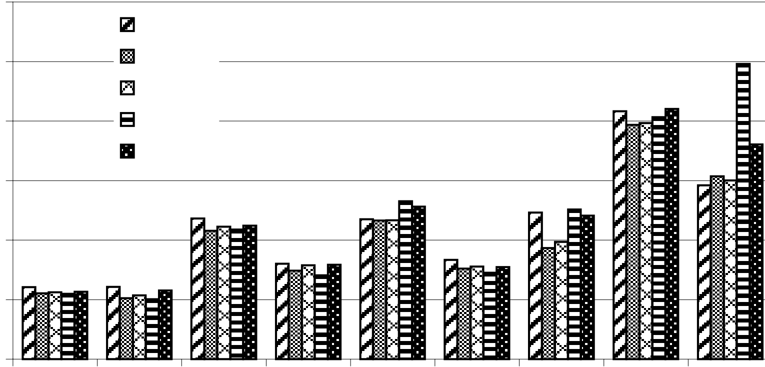
The completion delay of loads is the time interval between when a load's operands are ready and when the load completes. Ideally, the completion delay of loads that hit in the L2 cache should be 11 cycles (one cycle to issue the load + 10 cycle L2 cache access latency). Resource constraints such as L1 MSHR being full and the L2 bus being busy are the main reasons for the increase in completion delay of loads.

From Figure 4.11 we can see that for most of the benchmarks, both the MBP schemes increase the average completion delay of loads that hit in the L2 cache. In most cases (except for both the MBP schemes for gzip, and *mbp-crl* for perimeter), if a prefetch scheme increases the average completion delay of loads that hit in the L2 cache by more than 10% compared to *tmem*, it leads to performance degradation.

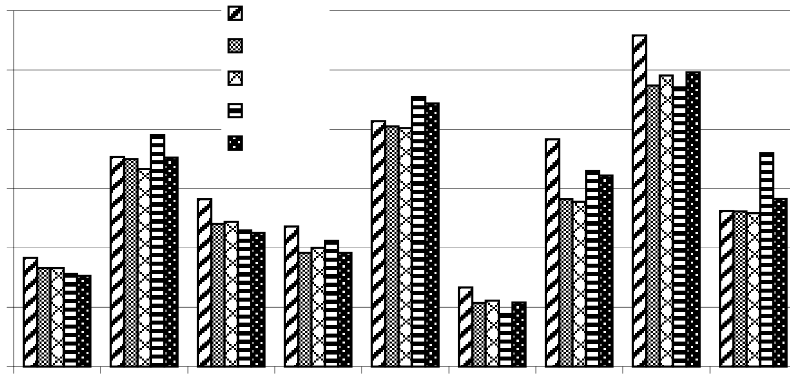
Figure 4.12(a) compares the L1 cache overall miss-ratio and Figure 4.12(b) compares the L1 cache critical load miss-ratio of the prefetching schemes with that of *tmem*. Prefetching reduces the overall miss ratio as well as the critical load miss ratio of the L1 cache in most cases. However, the overall miss-ratio increases for all four prefetching schemes for vpr and for both the MBP schemes for parser. The critical load miss ratio increases for both the mbp schemes for bzip2, parser and vpr. This suggests that pollution also becomes a problem for these cases.

Even for the benchmarks that show a decrease in L1 cache miss-ratios, the increase in the resource constraints due to prefetching more than offsets the advantages due to lower miss ratios. This results in either meager performance improvements of up to 2% or decreases in performance of up to 43% and 14% for the locality and criticality based schemes respectively, compared to a traditional memory system. Hence, I do not look any further at locality/criticality based prefetching into the L1 cache, and instead, turn to the L2 cache.

The results at the L1 cache level might not hold at the L2 cache level for two reasons. First, L2 caches are usually much bigger than L1 caches and so the chances



(a) Overall Miss Ratio



(b) Critical Load Miss Ratio

Figure 4.12: L1 Locality vs. Criticality Prefetching: Miss Ratio

of pollution in the L2 cache are lower than in the L1 cache. Secondly, the strain on the MSHR queue and the bus to the next level of memory could be very different due to the difference in the density/frequency of accesses to the two levels.

4.3.3 Prefetching Into the L2 Cache

When prefetching into the L2 cache, I use an L2 cache line size of 64B and a macro-block size of 256B. I make a prefetching decision (whether to prefetch or not, and

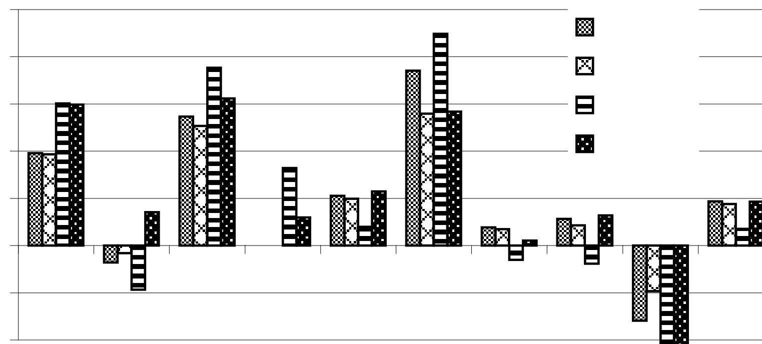
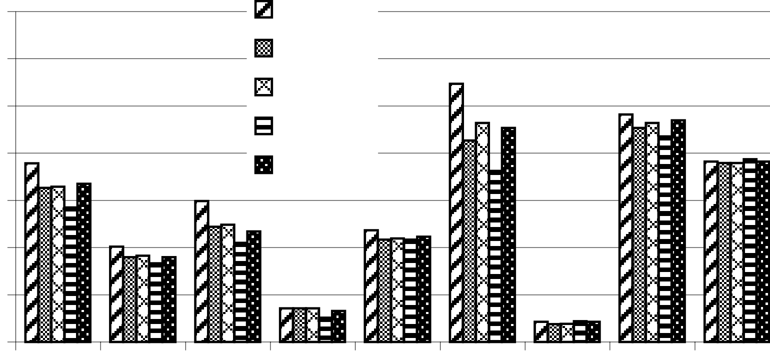


Figure 4.13: L2 Prefetching: IPC

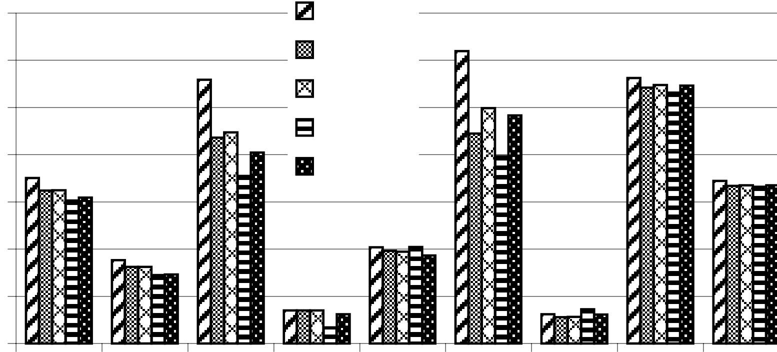
what to prefetch) whenever there is an L2 cache miss. Figure 4.13 shows the percentage improvement in IPC of the four prefetching schemes over a traditional memory system. Unlike at the L1 cache level, prefetching into the L2 cache shows performance gains in most cases. vpr is the only benchmark that does not show any performance benefits.

In general, whenever SFP increases performance, the locality based *sfp-all* produces higher performance numbers than the criticality based *sfp-crl*. This is not always true for MBP. For gcc, gzip, and perimeter, *mbp-all* does produce considerably higher performance improvements than *mbp-crl*. However, for three other benchmarks, bzip2, perlbmk, and twolf, *mbp-all* decreases performance, while the criticality based *mbp-crl* produces performance improvements. Across all four prefetching schemes, *mbp-crl* is the only scheme that increases performance for bzip2 and it also shows the most performance gains for parser and twolf.

To better understand these results, I look at both cache pollution and resource (L2 MSHR entries and memory bus) constraints. If pollution due to prefetching is a problem, it will lead to an increase in the number of regular load misses compared to



(a) Overall Miss Ratio



(b) Critical Load Miss Ratio

Figure 4.14: L2 Prefetching: Miss Ratio

---

tmem. However, my simulations reveal that none of the prefetching schemes increase the overall or critical load L2 local miss ratio considerably (see Figure 4.14). Using the reference traces collected before, I find this to be true for cache sizes ranging from 128KB to 16MB. Hence, I infer that pollution is not a problem at the L2 cache level.

Figure 4.15 shows the average completion delay of loads that access main memory. Ideally, the completion delay of loads that access main memory should be 310 cycles. From Figure 4.15, we can see that prefetching for vpr increases the resource

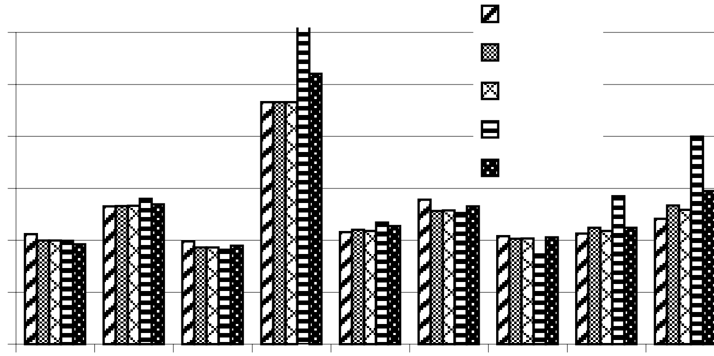


Figure 4.15: L2 Prefetching: Bandwidth

constraints by increasing the average completion delay of memory accesses compared to *tmem*. Furthermore, prefetching does not produce a significant decrease in the overall or critical load L2 local miss ratio for vpr. Hence, prefetching is not beneficial for vpr.

For the benchmarks gcc and perimeter, the locality based scheme, *mbp-all* produces considerably higher performance improvements than the other prefetching schemes. For these benchmarks, we can see from Figure 4.15 that prefetching does not introduce any resource constraints. For gcc and perimeter, prefetching decreases the average completion delay of memory accesses and hence does not delay regular loads. Since neither pollution nor resource constraints are a problem for these benchmarks, the most aggressive prefetching scheme – *mbp-all* – produces the most performance gains. For gzip, both MBP schemes increase the average completion delay of memory accesses, but the number of L2 cache misses for gzip is very small to begin with. The MBP schemes further reduce the number of loads going to main memory and hence produce performance gains in spite of the increase in average completion delay of memory accesses.

For the three benchmarks, bzip2, parser, and twolf, for which the criticality based scheme, *mbp-crl*, produces the most performance benefits, resource constraints do pose a problem. As can be seen from Figure 4.15, these are the only benchmarks (other than gzip and vpr) for which there is an increase in the average completion delay of memory accesses due to prefetching. Hence, carefully choosing what to prefetch becomes important for these benchmarks. For parser, and twolf, restricting the number of prefetches using either the locality based *sfp-all*, or the criticality based *mbp-crl* produces similar performance. However, for bzip2, *mbp-crl* has a lower critical load miss ratio than *sfp-all*, and hence produces performance benefits significantly higher than *sfp-all*. Even with a smaller number of MSHR entries, 24 instead of 128, the results are qualitatively similar.

4.3.4 Summary

In summary, I find that in general, prefetching based on either locality or criticality is not beneficial at the L1 cache level due to cache pollution and resource constraints. At the L2 cache level, I find that cache pollution is not a significant problem. If resource constraints are also not a problem, then the most aggressive locality based prefetching scheme, *mbp-all*, does best. For the benchmarks for which resource constraints are a problem, the selective prefetching scheme based on criticality, *mbp-crl*, gives the most performance benefits. However, the performance improvements of criticality based prefetching compared to locality based prefetching are small, and may not be worth the added complexity of detecting critical loads.

4.4 Related Work

Abraham et al. [1] have looked at which level in the memory hierarchy a particular load gets satisfied. They pass this information back to the compiler and selectively

schedule loads with the cache miss latency. Their goal is to keep the memory system fixed and increase the latency tolerance of accesses that are likely to miss in the L1 cache by scheduling them early. This thesis examines the inverse problem - given a schedule by the compiler, which level in the memory hierarchy best suits a load, based on its latency tolerance. My goal is to modify the memory system to conform to the requirements of loads, by managing caches such that loads with low latency tolerance stay closest to the processor.

Rivers and Davidson [41], and Gonzalez et al. [15] have partitioned memory accesses into those that have spatial locality and those that have temporal locality. Rivers and Davidson prevent non-temporal accesses from polluting the contents of the L1 cache by allowing them to bypass the L1 cache. Gonzalez et al explore the usefulness of separate spatial and temporal caches. Tyson et al [61] mark loads that account for most of the data cache misses as non-cacheable and let accesses by these loads bypass the L1 cache. Johnson and Hwu [20] increase the incumbency period of blocks with good temporal locality in the L1 cache by preventing them from being replaced by low temporal locality blocks. Blocks that are denied entry into the L1 cache end up bypassing the L1 cache.

All the above techniques deviate from the conventional "eager" policy of caches, whereby data that miss in the cache is always brought into the cache. These approaches tend to focus on the quantity and not the quality/nature of cache misses. In this thesis, I evaluate partitioning loads as critical and non-critical based on their latency requirements. My criticality based cache organization schemes are aimed at retaining critical loads in the cache longer than non-critical loads.

Fisk and Bahar use a Non-Critical Buffer (NCB) to hold loads classified as non-critical and attempt to free up primary cache space for critical data. My critical cache scheme is similar in principle, in that it tries to keep critical data close to the processor

for long periods of time. Fisk and Bahar report maximum performance improvements of 4% over a traditional memory system, with the SPEC95 benchmarks. However, they do not compare it with an equivalent locality based schemes (such as a victim cache), and fail to provide any insights on why further performance gains are not possible.

The idea of multi-block prefetching comes from previous work by Przybylski [39] that investigates optimal statically-determined fetch sizes. Johnson et al. [21] look at dynamically varying the fetch size by fetching bigger cache lines for blocks with good spatial locality. The *mbp-crl* scheme proposed in this thesis prefetches bigger cache blocks whenever a cache miss is generated by a critical load.

4.5 Conclusions

Current caches are designed to exploit locality of accesses and are unaware of the criticality of loads. In this Chapter, I compare the two properties, locality and criticality, in the context of several caching and prefetching schemes to answer the question: In practice, can criticality beat locality?

I find that my criticality based cache organization scheme that uses the critical cache is unable to significantly reduce the critical load miss ratio compared to a traditional memory system at both the L1 and L2 cache levels. This is because, the working set of critical loads is large and even retaining critical data alone, fails to reduce competition for cache space.

My criticality-based prefetching scheme, *mbp-crl*, selectively prefetches bigger cache lines on critical load cache misses. At the L2 cache level where pollution is not a problem, *mbp-crl* reduces resource constraints introduced by prefetching and achieves lower critical load miss ratios compared to a traditional memory system for the benchmarks (bzip2, parser and twolf) that have resource constraint problems.

Hence, for these benchmarks, *mbp-crl* achieves higher performance than locality based prefetching schemes. However, the performance gains are small compared to equivalent locality based schemes and may not be worth the added complexity of detecting critical loads.

Thus I find that even though, in theory criticality outperforms locality, in practice it is not worth violating locality to exploit criticality. Criticality based techniques that co-exist with and supplement locality may produce higher performance but requires further research. This thesis examines two such techniques that are dealt with in subsequent Chapters: (i) Scheduling critical loads early by means of Priority Scheduling, and (ii) Predicting branches using Load values.

Chapter 5: Priority Scheduling

5.1 Introduction

The total latency incurred by a load from the time it is ready to be issued to the memory system to the time the data is returned to the processor can be partitioned into three distinct components:

1. Access latency in the different levels of the memory hierarchy,
2. Bus transfer times, and
3. Queue delays.

Chapter 4 examined the first of these three components by attempting to provide critical loads with a short access latency (by bringing critical data close to the processor). The second component of a load's latency, bus transfer times, is usually fixed for a given architecture/level in the memory hierarchy. In this Section, I focus on the final component of a load's latency, queue delays. I investigate modifying queue scheduling policies to give priority to critical loads and thereby improve performance.

5.2 Queue Scheduling

Figure 5.1 shows the different queues that a load could encounter between the time it is ready to be sent to the memory system and the time the requested data comes back to the processor: Ready Queue, L1 and L2 MSHRs, and the Data Port Queue. There is often contention for the resources controlled by these queues, and the time spent in these queues constitutes a significant fraction of a load's latency. Hence,

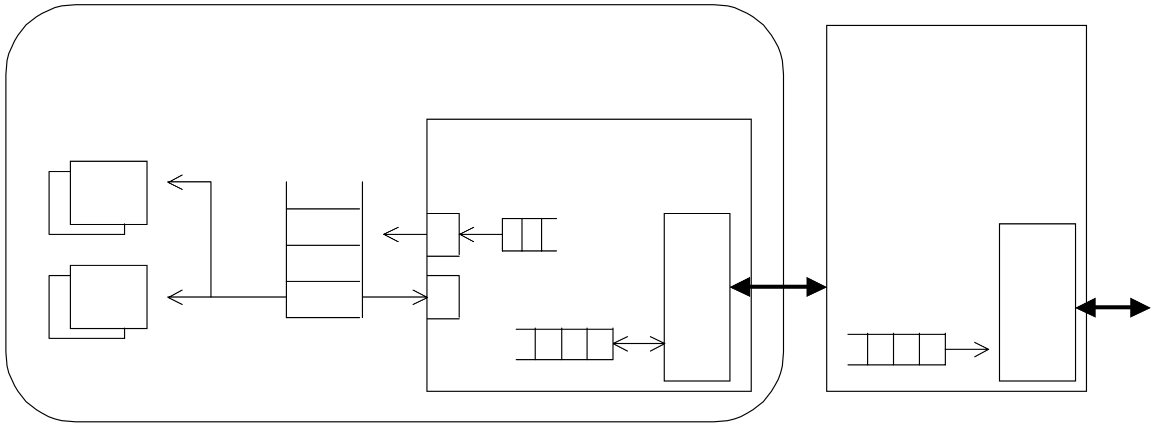


Figure 5.1: Queues on Load Path

having a good scheduling policy in the queues is crucial. The following subsection elaborates on each of these queues.

5.2.1 Queues on a Load Path

Ready Queue

When the address of a load is computed and there is no true data dependency on an earlier store, the load is ready to be issued to the memory system. At this point, the load is put into the Ready Queue, which holds instructions that are ready to be issued to functional units. The functional unit required by a load is a cache read port to access the cache. In the Ready Queue, a load competes with other instructions to be issued, and with other memory operations to obtain a cache read port.

Miss Status Handling Registers

When a load misses in a cache, the data has to be fetched from the next lower level in memory. The data band-width demands of today's superscalar processors require that caches be non-blocking/lock-up free. A non-blocking cache is capable of overlapping misses. Miss Status Handling Registers (MSHRs) [24, 50] are a standard means of implementing non-blocking caches. MSHRs are used to hold the status in-

formation of outstanding misses. The MSHRs free-up the cache to satisfy subsequent requests by taking care of:

1. Acquiring the bus and submitting the miss request to the next lower level in memory,

2. Merging secondary misses to cache blocks that already have misses outstanding, and

3. Updating the appropriate cache block and/or the CPU registers with the returned data.

With a no-fetch-on-write cache [23], only read misses occupy MSHR entries. However, with a fetch-on-write cache, write misses compete for MSHR entries too. The bus connecting two levels of memory is a major source of contention. Read requests from higher memory (closer to the processor), data returning from lower memory, as well as writes (or write-backs) from higher memory compete for the bus.

Data Port Queue

Once the data requested by a load is available, it has to be broadcast onto the result bus along with its tag. The wake-up logic present in registers and in the Register Update Unit (RUU) entries monitors the result bus and updates their contents when a matching tag is found. A limited number of entry points are available for returning loads to gain control of the result bus. For purposes of clarity, I call these entry points data ports. Just as the processor needs to acquire a cache port to submit a memory request to the L1 cache, a returning load needs to obtain a data port to provide data back to the processor. The number of data ports is typically equal to the number of cache ports (or cache read ports, in case of separate read/write ports). In a given cycle, L1 cache read hits and data returning from the L2 cache to satisfy earlier L1 cache misses contend for the data ports.

5.2.2 Default Scheduling

In the Ready Queue, I use a default policy that minimizes delays due to dependencies. This is done by separating instructions into two classes:

1. High impact operations – long latency operations (memory operations, and certain floating-point operations) and branches.

2. Low impact operations – the rest of the operations that have short latencies.

High impact operations are queued ahead of low impact operations in the Ready Queue. This scheduling policy is the default policy in the SimpleScalar toolset and is done for the following purposes. Long latency operations increase the wait times of dependent instructions and limit the look-ahead capabilities of the processor. Issuing them earlier frees up their dependent instructions sooner and helps increase processor utilization. The time it takes for a branch to be resolved controls the number of instructions the processor executes in the speculative state. Since speculatively executed instructions can potentially be useless (when the branch is mis-predicted), their execution should be minimized and issuing branches early enables this.

It must be noted that the classification takes place at an instruction-set level and is not on a per-instance granularity. For example, a particular instance of a load may have a short latency (due to an L1 cache hit), but will still be classified as a high impact operation.

In the L1 and L2 MSHRs, I use a default scheduling policy that queues load misses earlier than that of stores. Since only loads send data back to the processor, they alone use the Data Port Queue. I use a FIFO policy among loads in the Data Port Queue.

5.2.3 Priority Scheduling

The default scheduling policies in the queues treat all loads alike and hence, fail to match the latency requirements of critical loads. I change the scheduling policies in the queues to give priority to critical loads and thus, exploit the variation in load criticality.

In the Ready Queue, even though the default policy schedules high impact operations (that includes loads) early, it still treats all loads equal and on par with other high impact operations. I further refine loads based on their latency tolerance and give higher priority to critical loads in the Ready Queue.

In the MSHRs, the priority scheduling scheme queues critical loads first, followed by non-critical loads, followed by stores. This enables critical loads to acquire the L2 cache/memory bus early, which allows critical load misses to be handled faster.

With priority scheduling in the Data Port Queue, I let critical loads obtain a data port ahead of non-critical loads, irrespective of whether the data accessed by the critical load is a hit in the L1 cache or is returned from the L2 cache. This enables the critical data to be returned to the processor without being delayed by non-critical data.

By giving priority to critical loads in all the queues that it encounters, I attempt to minimize the queue delay component of their latencies. I use the hardware scheme alongwith the load criticality predictor described in Chapter 3 to classify loads as critical and non-critical. The following Section presents priority scheduling results.

5.3 Results

I find that priority scheduling is beneficial only for the Ready Queue and the L1 MSHR. This is because priority scheduling at the L2 MSHR and the Data Port Queue

potentially involves loads that have long latencies (L2 cache misses in the case of L2 MSHR and loads returning from lower memory in the case of the Data Port Queue). Delaying such loads in favor of loads classified as critical could prevent subsequent instructions from committing their results. This in turn would delay freeing up buffer space used by these instructions. This can aggravate the finite resources problem and will limit the look-ahead capability of the processor. Hence, a simple FIFO policy is best for the L2 MSHRs and the Data Port Queue. For all experiments in this Section, I employ priority scheduling only in the Ready Queue and the L1 MSHR.

I also find that with a memory latency of 300 cycles, the processor is waiting for memory most of the time and this masks the benefits seen due to priority scheduling. To highlight the benefits of priority scheduling, I use a more aggressive memory system with a latency of 60 cycles (44 cycle access latency + 16 cycle bus transfer time).

Other changes I make to the experimental setup used so far(Section 2.5) are: (i) I expand the scope of the Cache criteria to include loads feeding any load (not just L1 cache misses) as critical, and (ii) I add two more benchmarks, compress from the SPECint95 benchmark suite and treeadd from the Olden benchmark suite for which queue delays have a noticeable impact on performance.

Figure 5.2 shows the IPC numbers with (*prio*) and without (*base*) priority scheduling. We can see that priority scheduling is beneficial for all benchmarks except vpr. bzip2 shows the most gains of 5% while priority scheduling produces 3% gains for compress, gzip, and treeadd. The negative contribution from priority scheduling for vpr highlights the potential drawback of priority scheduling. A sustained presence of critical loads in the queues causes non-critical instructions to starve, thereby decreasing performance. This can be prevented by imposing a limit on the maximum number of cycles a non-critical instruction waits before it is given priority and qualifies to be

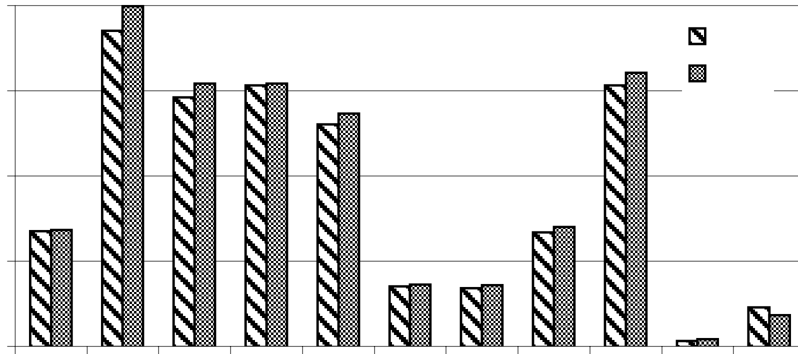


Figure 5.2: Priority Scheduling Benefits

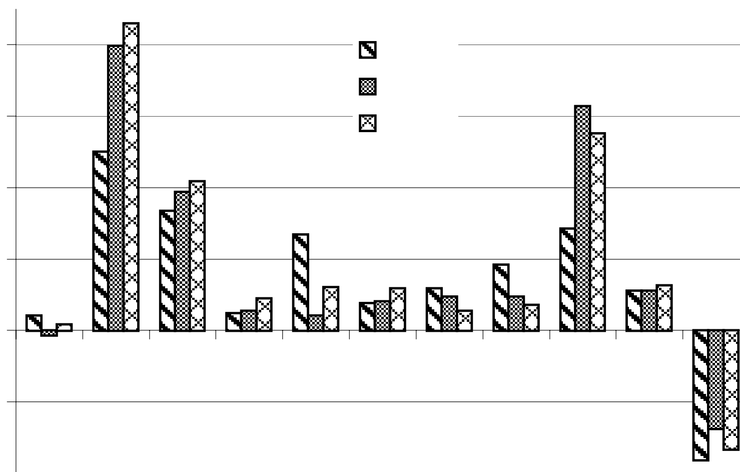


Figure 5.3: Priority Scheduling Benefits: Increasing L1 Cache Latencies

serviced first.

An increasing trend in modern microprocessors is to move towards multi-cycle L1 cache access latencies due to the growing disparity between processor and cache memory speeds. Figure 5.3 shows that for most of the benchmarks, the benefits due to priority scheduling increase with increasing L1 cache latencies. The performance gains for bzip2, compress and treeadd increase to 9%, 4% and 6% respectively, as the L1 cache latency increases from 1 to 3 cycles.

5.4 Conclusions

This Chapter explores giving priority to critical loads in the processor/memory system queues so as to schedule critical loads earlier than other non-critical instructions. I find that priority scheduling works well for the Ready Queue and the L1 MSHR alone. My results indicate that priority scheduling is capable of producing performance gains up to 9%.

In general, I find that scheduling critical loads alone earlier is not enough to improve performance for many benchmarks. Scheduling the entire chain of dependent instructions leading to critical loads may hold the key to higher performance. Currently, several researchers are looking at this problem.

Fields et al. [12] report that scheduling all instructions on the critical path earlier than other instructions produces significant benefits for all of their benchmarks. They use a clustered architecture [37] as their base configuration and apply critical path scheduling in the Ready Queue along with critical path based steering of instructions to appropriate clusters. They achieve performance improvements of up to 21% (10% on average).

Pre-execution [30, 53, 69] is a generic technique that is targeted at executing the entire backward slices [68] of critical instructions early. Thus pre-execution can be thought of as a superset of priority scheduling. However, pre-execution differs from priority scheduling in that the results of pre-computation are mostly used to predict future branches or prefetch for future loads and do not alter the architectural state of the processor. Pre-execution has been shown to be very effective, especially for processors with support for simultaneous multi-threading [60].

Chapter 6: Load-based Branch Prediction

This Chapter examines a subset of the critical loads: loads feeding mis-predicted branches, in detail. Branch mis-predictions are a major impediment to higher performance in today's processors. Previous studies [27, 40] highlight the detrimental effect of mis-predicted branches on the processor's ability to extract instruction-level parallelism. The hurdles posed by mis-predicted branches can be overcome in two ways:

1. Minimize the number of mis-speculated instructions executed by resolving a mis-predicted branch early, and/or
2. Reduce the number of branch mis-predictions by building better branch predictors.

So far in this thesis, I have followed the former approach by trying to complete loads feeding mis-predicted branches early. In this Chapter, I adopt the latter approach. I propose using the correlation between load values and dependent branch outcomes for better branch prediction.

6.1 Introduction

Prior branch prediction research has led to the use of sophisticated branch predictors that correlate the outcome of a branch with one or more of the following:

1. Path leading to the branch (global branch history) [35, 52, 67],
2. Previous instances of the branch (local branch history) [66], or

3. Operand values of the branch [16, 19].

Though these predictors perform well in general, there exist certain branches that are not captured by correlations exploited by existing branch predictors, even with huge predictor table sizes. This Chapter deals with capturing these hard-to-predict branches. I propose correlating the outcome of a branch with its input Data Set.

I define the Data Set of a branch as the set of load instructions on which it is dependent. I use load instructions because they bring new data into a program and often constitute the inputs for all computation leading to a branch. A change in the value fetched by a load (load value) gives an early indication of a potential change in a dependent branch outcome. The Load-based Branch Predictor (LBP) keeps track of histories of branch outcomes associated with load values to predict future instances of branches.

I propose using LBP in conjunction with other traditional branch predictors as base predictors because it may not always be possible to associate a branch with a set of input loads (due to lack of load–branch correlation), and since not all input load values may be available in time to predict a branch (a branch could immediately follow its input load). Therefore, whenever LBP has a prediction with confidence for a branch, I use LBP's prediction, otherwise, I revert to the prediction from the base branch predictor.

The contributions of this Chapter are two-fold:

1. To provide an analysis of the potential for exploiting the correlation between a branch and its input Data Set, and

2. To evaluate an implementation of the LBP scheme.

The rest of this Chapter is organized as follows. Section 6.2 discusses background and related work. Section 6.3 outlines my experimental setup and Section 6.4 explores

```
conv_str = Load (conversion string);  load1      Sample conv_str = "%8s %d"
index = 0;
while (index < Length(conv_str)) {
    next_char = conv_str[index];
    if (next_char == "%") {           branch1    Branch outcomes = 100010
        process (new_item);                      (1 = taken)
        new_item = NULL;
    } else {
        new_item = new_item + next_char;
    }
    index = index + 1;
}
```

Figure 6.1: Data Set Correlation Example

the potential benefits of using the correlation between branches and their Data Sets. Section 6.5 describes my implementation of the LBP scheme. Section 6.6 provides simulation results. Section 6.7 concludes this Chapter along with some future work.

6.2 Background

The idea behind LBP can be understood by examining the backward slices [65] of branches. The backward slice of a branch consists of all instructions that can affect the outcome of the branch. It typically has two parts:

1. A set of input values, called the Data Set of the branch. These input values are brought into the program often by means of load instructions, and

2. A sequence of computations on the input values, the end result of which determines the branch condition value(s).

Consider the program segment that parses a 'printf' statement, shown in Figure 6.1. A 'printf' statement has a conversion string that specifies the type of each

data item to be printed. A 'while' loop traverses the conversion string looking for a "%" character, which identifies the beginning of a new data item specification. Every time 'next_char' equals "%", branch1 is taken, and the new data item is printed.

For a given conversion string, the sequence of outcomes for branch1 is fixed. In the above example, whenever the conversion string is "%8s %d", the outcomes for the next 6 instances of branch1 will be 100010 (where 1 = taken). Hence, branch1 is said to strongly correlate with the conversion string. LBP builds the association between branch1 and the conversion string during the first execution of 'printf', and uses it to predict branch1 during subsequent executions for which the conversion string matches.

In this example, the main component of the input Data Set for branch1 is the conversion string, which is fetched by load1. LBP uses this correlation between a branch's input Data Set and its outcomes for better branch prediction. Note that if the length of the conversion string is bigger than the fetch-size of the load, multiple instances of the load may be required to fetch the entire conversion string. In this case, several instances of a branch will be dependent on several instances of the load.

6.2.1 Related Work

Most of the branch predictors widely used today take advantage of the correlation between branches (self-correlation between several instances of a branch or cross-correlation between different branches) [35, 38, 52, 67]. If two branches are correlated, knowing the outcome of the first branch could provide information about the direction of the second branch.

Figure 6.2 shows an example used by Evers et al. [10]. Here, if branch B1 is not taken, then B2 will not be taken. Global two-level branch predictors such as GAs [67] and gshare [32] are designed to capture such relationships. However, if a

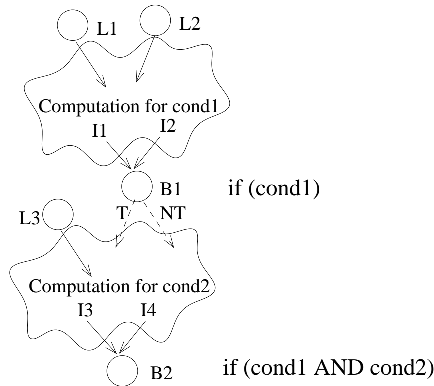


Figure 6.2: Path Based Correlation Example

predictor uses only correlation between branches, it will be unable to predict branches like B1, that do not correlate with a prior branch. Furthermore, these predictors rely on accurate knowledge of previous branches' behavior, including those still in flight. Processor delays preventing a branch from committing, and mis-predictions of previous branches could force such predictors to operate with incomplete information or stale information.

LBP relies on correlations between a branch and its Data Set. Branch B1 depends on condition (cond1) and its Data Set consists of loads L1 and L2. Since B2 depends on both conditions (cond1 and cond2), its Data Set will include an extra load L3 needed for the computation of cond2. Thus the Data Set for B2 will consist of all three loads and using the Data Set–branch correlation will be able to capture both branches B1 and B2. The Data Set–branch correlation is orthogonal to the correlations exploited by current branch predictors (such as between different branches). Hence, using LBP along with existing predictors is likely to increase the branch coverage (the branches for which a predictor is useful) of existing predictors.

Heil et al. [19] use Data values to correlate with branches. Their predictor uses the difference of the two branch source register operands, called the branch difference. In

Figure 6.2, to predict branch B1, they use the difference of the results of instructions I1 and I2. Thus, they use the results of the computation for B1, while LBP uses the input values that feed into the computation for B1, to predict B1.

Instructions I1 and I2 will always be very close to B1 and their results will not be available in time to predict B1. To solve this problem, they assume that the pattern of branch differences repeat. Hence they approximate the branch difference for the current instance of a branch using the last known branch difference from a previous instance of the branch and the dynamic count of branches fetched between the two instances. Changes in branch difference patterns will lead to branch mis-predictions in their scheme.

In contrast, LBP always relies on the input values that are responsible for the current instance of the branch. If a load value is not available in time, LBP looks for a prediction from a load further back in the dependence graph that correlates with the branch and whose value is available in time. For this purpose, LBP keeps track of more than one load that correlates with a branch.

Predicated execution [31, 62] is another technique that complements existing branch predictors and helps handle hard-to-predict branches. It involves conditional execution of instructions based on the result of a boolean condition (predicate). The main drawback of predicated execution is that operations whose predicates are not true, causing them not to be executed, must still be fetched and may take up space in the pipeline.

Farcy et al. [11], Zilles and Sohi [69], and Sundaramoorthy et al. [53] propose pre-executing relevant slices of a program. For example, the entire backward slice of hard-to-predict branches can be pre-executed and the pre-execution result can be fed into the primary execution's branch predictor. But pre-execution may not always be possible [69]. LBP could be helpful even in cases where pre-execution is not possible.

In such a case, regular execution of a load may not provide its value in time to predict the first few instances of a branch. However, if the branch history associated with the load value is sufficiently long, LBP can have useful predictions for later instances of the branch.

6.3 Experimental Setup

To evaluate branch prediction, I make several changes to the experimental setup of Section 2.5. They are:

1. I use a more aggressive default branch predictor and memory system,
2. To avoid cold start effects for the default branch predictor, I warm-up the branch predictor also (along with the caches) for the first 4 billion instructions while fast-forwarding, and
3. I choose 6 benchmarks from the SPECint2000 suite that have high branch mis-prediction ratios.

For experiments in this Chapter, my default branch predictor is a 16bit/64k-entry gshare. The branch mis-prediction penalty is 12 cycles. The predictor lookup happens at the Fetch stage and update is done in the Commit stage. I do not model speculative branch updates [44].

The default memory system consists of a 64KB, 4-way set associative L1 Data cache with 32B lines and a 1 cycle latency. The L2 cache is 1MB, 8-way set associative with 128B lines and a 10 cycle latency. There are 128 MSHR [24] entries at each cache level to handle outstanding misses. I assume ideal main memory with a latency of 70 cycles. Contention is modeled in all parts of the memory system. I do not simulate a TLB or an instruction cache.

Benchmark	Number of Static Branches	Number of Dynamic Branches	Base Predictor Accuracy (%)	IPC
gap	512	4738177	92.81	3.66
gcc	1344	17141763	96.81	2.96
gzip	153	8984764	91.35	2.59
parser	342	16026976	92.93	1.93
twolf	410	12899453	88.18	1.63
vpr	95	13414809	88.30	1.58

Table 6.1: LBP: Benchmark Characteristics

Table 6.1 lists the Benchmarks I simulate and their characteristics using the default configuration listed above. For the rest of this Chapter, I present performance improvements and percentage reductions in the number of mis-predictions over these base numbers.

6.4 Exploiting Data Set–Branch Correlation

I first study the characteristics of the correlation between a branch and its Data Set and the potential benefits of using this correlation for branch prediction. Then I present the details of a practical LBP implementation. Since the Data Set of a branch consists of load values which will be available only in the Write-back stage of the load, it may be too late to be of any use for certain branches. However, I want to find out the maximum possible benefits of exploiting the Data Set–branch correlation, and so I artificially eliminate this timing limitation. Also, I assume that there is space to hold an infinite number of load value, branch history correlations.

I modify the simulator to make load values available in the Fetch stage of the load itself, in time to be able to predict dependent branches. These early available load values are used only for branch prediction purposes i.e. dependent instructions will

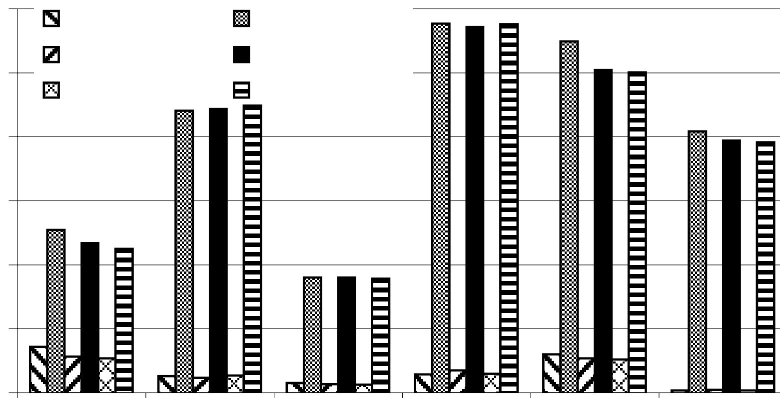
be able to make use of these load values only in the Write-back stage of the load, and so regular program execution proceeds as usual.

The Data Set of a branch consists of all loads that feed into the branch. For an arbitrarily long backward slice of a branch, the number of loads in its Data Set could be large. However, in practice, it is feasible to keep track of only a finite number of loads per branch. To limit the number of loads in the Data Set of a branch, each time the branch is executed I monitor the last 8 loads (in dynamic program order) that it is dependent on. Since the loads that constitute the last 8 can change from one instance of a branch to another instance of the same branch, I track up to 32 different loads for each branch over the entire run of the program. These 32 loads constitute the potential Data Set of the branch.

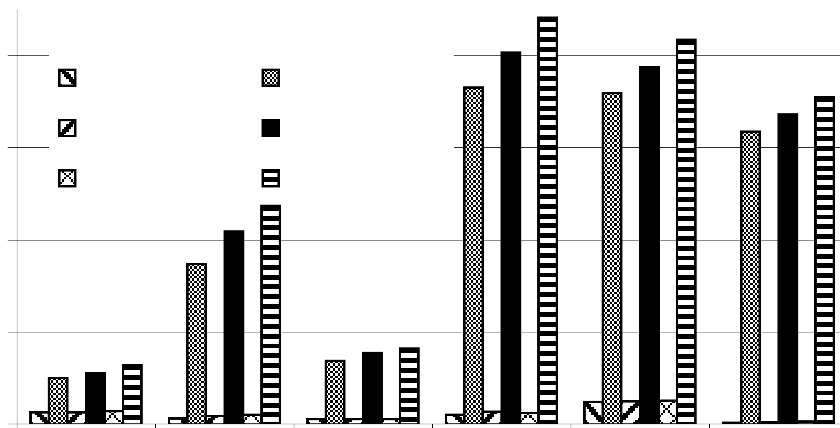
I gather a profile of the number of times a branch is dependent on each of the loads in its potential Data Set. Using this profile, I choose loads in decreasing order of the number of times a branch is dependent on a load to form the actual Data Set of the branch. My experiments suggest that using the 8 loads that feed the branch the most, to form its actual Data Set performs best for most benchmarks.

I first look for a useful prediction from the load on which the branch is dependent on the most. A useful prediction is one that satisfies a confidence mechanism, described below. If the first load does not provide a useful prediction, I turn to subsequent loads. If none of the loads have a useful prediction, I use the default branch predictor.

The confidence mechanism uses a counter that has four parameters (saturation threshold, use threshold, miss penalty, and hit bonus). The saturation threshold is the maximum value of the counter. If LBP correctly predicts a branch that is mis-predicted by the base predictor, the counter is incremented by the hit bonus. Conversely, if LBP mis-predicts a branch that is correctly predicted by the base



(a) Mis-predictions



(b) IPC

Figure 6.3: Data Set–Branch correlation: Effect of Pipeline Depth

predictor, the counter is decremented by the miss penalty. If LBP does not have a prediction, or if the predictions of both LBP and the base branch predictor are the same, the the confidence counter is not updated. A prediction from the LBP is used only if the counter value is above the use threshold. My experiments with different counter parameters suggest that the configuration (9,7,2,1) works well.

Figure 6.3 shows the potential benefits of using the Data Set–branch correlation with processors of different pipeline depths, when all load values are available in

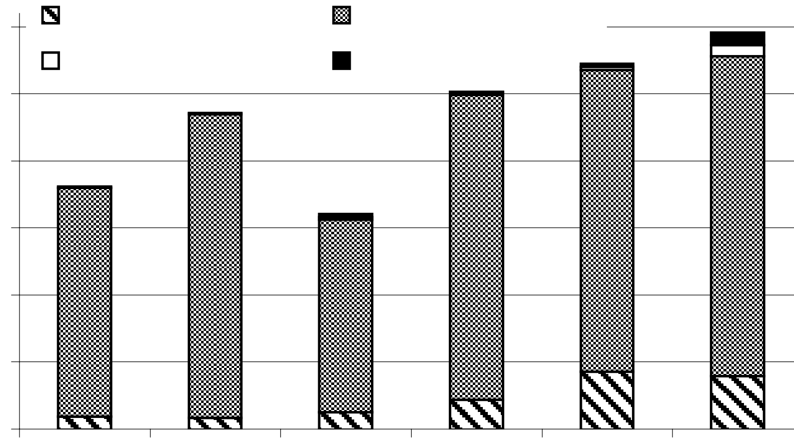
time for their dependent branches to be predicted. I model three pipeline depths of 5, 7, and 9. To increase the pipeline depth beyond the base pipeline depth of 5, extra stages are added between the Fetch and Dispatch stages of the pipeline. The branch mis-prediction penalties for the three pipeline depths are 12, 14, and 16 cycles respectively.

Figure 6.3(a) shows the percentage reduction in mis-predictions and Figure 6.3(b) shows the percentage improvements in IPC over the default 16bit/64K-entry predictor. For each pipeline depth, I compare a 17bit/128K-entry gshare (*17g*) and a 16bit/64K-entry gshare predictor that also exploits the Data Set–branch correlation (*16g+lbp*).

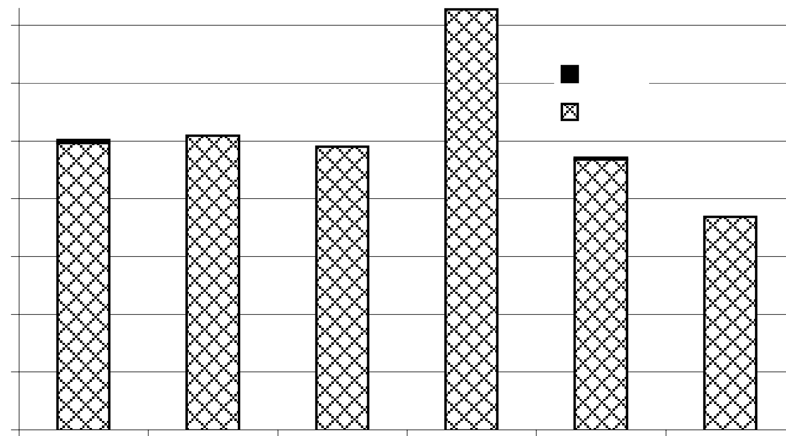
With the base 5-stage pipeline, I find that using the Data Set–branch correlation is capable of decreasing the number of mis-predictions by between 18% and 58% (average 40%), over the default predictor. This reduction in mis-predictions achieved by LBP translates into potential performance gains of up to 37% and 22% on average.

As the pipeline depth increases, we can see that the reductions in the number of mis-predictions are comparable. This is because for these experiments load values are available in the Fetch stage of the pipeline itself and increasing the pipeline depth does not alter the time-gap between the Fetch stages of a load and a dependent branch. However, as the branch mis-prediction penalty increases along with the increase in pipeline depth, we can see from Figure 6.3(b) that the performance impact of the mis-predictions saved also increases. The average improvements in IPC when using the LBP scheme increases from 22% for a pipeline depth of 5, to 27% for a 9-stage pipeline.

Note that for the above study, since it is assumed that load values are available in their Fetch stage, increasing the number of pipeline stages does not affect when a load's value is available for branch prediction. However, in a real implementation,



(a) Dynamic Branch Coverage



(b) Static Branch Coverage

Figure 6.4: Data Set–Branch Correlation: Branch Coverage

increasing the number of pipeline stages will increase the delay after which a load's value is available. This can decrease the number of branches for which load based branch prediction is available. This interaction is further examined in Section 6.6.

Figure 6.4(a) shows the percentage of dynamic branches for which LBP has a prediction that satisfies my confidence mechanism. The branches are further classified based on whether the base branch predictor and LBP predicted the branches correctly or not. I find that LBP has a useful prediction for between 32% to 60% (average

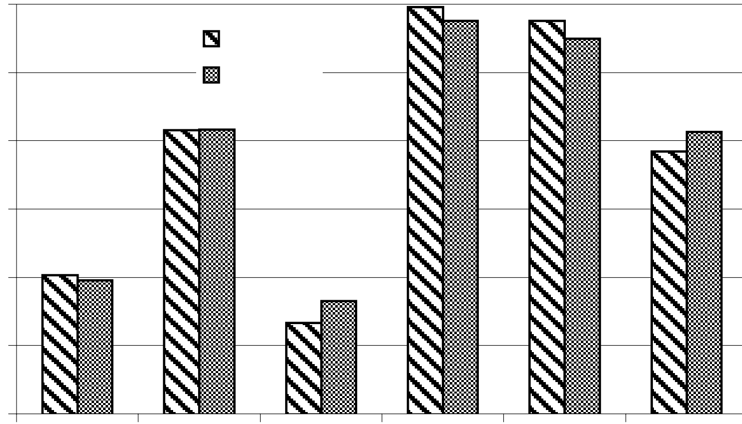
47%) of the dynamic branches. This shows that LBP cannot be used as a stand-alone predictor. LBP alone has a correct prediction for between 2% to 9% (average 5%) of the dynamic branches. This shows that LBP can supplement existing branch predictors well and serve to increase their branch coverage.

The average prediction ratio of the LBP scheme alone whenever it has a prediction with confidence is 97.8%. Since I look for a prediction from LBP first, it could provide an incorrect prediction for a branch that could have been correctly predicted by the base predictor (*lbp-bad, base-good*). Thus LBP could introduce new mis-predictions. However, for all of the benchmarks, I find that the number of mis-predictions averted is much more than those introduced by LBP, and there is a reduction in the overall number of mis-predictions.

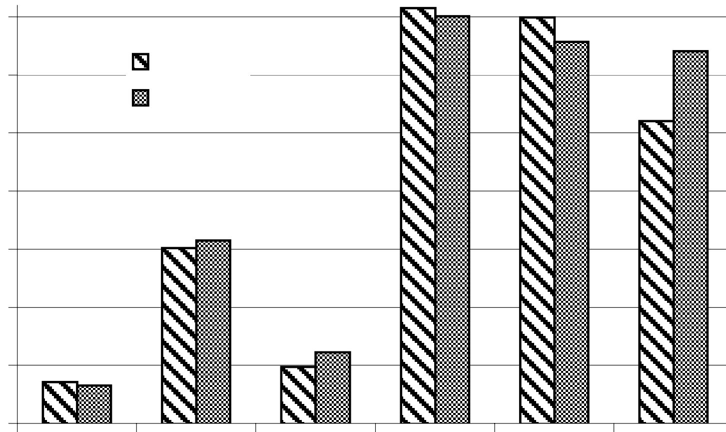
Figure 6.4(b) shows the percentage of static branches for which LBP is beneficial (more good predictions than bad predictions produced by LBP for a branch). From Figure 6.4(b) we can see that LBP is beneficial for between 37% and 73% (average 51%) of the static branches. There are very few static branches (3 for gap, 1 for twolf, and none for the rest of the benchmarks) for which LBP is detrimental.

Next, I evaluate the benefits of using the Data Set–branch correlation when used with bigger and better branch predictors. I use (i) a hybrid predictor [32], and (ii) a 20bit/1M-entry gshare predictor as base predictors for this purpose. The base hybrid predictor chooses between a 64K entry local history based predictor (no global branch history) and a 16bit/64K-entry gshare predictor (16 bits of global branch history). Figure 6.5(a) shows the percentage reduction in the number of mis-predictions and Figure 6.5(b) shows the percentage increase in IPC when using LBP along with the above base predictors. The improvements shown are over the respective base predictors.

We can see that LBP is capable of producing substantial benefits when used



(a) Mis-predictions



(b) IPC

Figure 6.5: Data Set–Branch correlation: Bigger and Better Base Predictors

with any of the above predictors. The average reduction in the number of branch mis-predictions are 39% when used with either base predictor. This translates into average performance gains of 20% and 21% for the hybrid predictor and the bigger gshare predictor respectively.

I find that just increasing the size of the base predictors to a 17bit hybrid predictor (*17h*: 128K entry local history based predictor + 17bit/128K gshare), or a 21bit/2M-entry gshare (*21g*) reduces the number of mis-predictions by less than 2% on average.

This suggests that just increasing the size of existing branch predictors is unlikely to yield benefits commensurate with the added costs. To lead us towards better branch prediction accuracies we need to exploit other types of correlations and this Section shows that the Data Set–branch correlation shows substantial promise. Hence, in the next Section I explore one possible implementation of LBP.

6.5 Implementation

A typical implementation of LBP consists of two parts. The first part deals with establishing the correlation between branches and their input Data Sets. This consists of (i) linking branches with the loads in its Data Set, and (ii) recording branch histories associated with load values. The second part utilizes the recorded load value, branch history correlations for branch prediction.

6.5.1 Establishing Data Set–Branch Correlations

Linking Branches with Loads in its Data Set

The first thing necessary to establish Data Set–branch correlations is to associate each branch with loads in their input Data Set. As discussed in Section 6.4, I statically determine the set of loads that constitute the Data Set of a branch. These are the loads that the branch is dependent on the largest number of times.

I make use of two tables, an Active Branch Table (ABT) and an Active Load Table (ALT) to store this information. Both tables are indexed using Instruction Program Counters (the branch PC for the ABT and the load PC for the ALT). Since the number of static branches and loads in programs is small, the sizes of these tables are also small. In practice, a 1024 entry table is sufficient for either table. During context switches, the Data Set information of a branch will also have to be saved as part of the context switch state.

Type	Name	Description
int	ld_num	Number of loads in branch's Data Set (maximum is 8)
unsigned long long	cur_val[8]	Current load value (deposited in the Write-back stage of loads)
unsigned int	fut_hist[8]	Branch history being accumulated for future use (updated during Commit stage of branch)
unsigned long long	prev_val[8]	Previous load value
unsigned int	past_hist[8]	Previously recorded branch history currently being used (looked up during Fetch stage of branch)
unsigned int	ldbr_order[8]	Dynamic sequence of loads and branches fetched (updated during Fetch stage of loads and branches)
int	cc[8]	Confidence counter

Table 6.2: Contents of ABT Entry

If a load is in the Data Set of a branch, the load's ALT entry will have the index of the branch's ABT entry, and the position of the load in the branch's Data Set (called the rank of the load). The position of a load in a branch's Data Set is determined based on the number of times the load feeds the branch and thus the load that feeds the branch the most will have a rank of 1. When a load completes, the ABT index and the rank fields are used to access the ABT entry of the dependent branch and deposit its value with the branch. The value is placed in the *cur_val* field of the branch's ABT entry. Table 6.2 lists the contents of an ABT entry and describes each of its fields.

Recording Branch outcomes associated with Load Values

When the branch reaches the Commit stage of the pipeline, for each load in its Data Set, it compares the *cur_val* field with a *prev_val* field. The *prev_val* field holds the previous value of the load. If the values are the same, the branch appends its outcome

(direction taken or not-taken) to a *fut_hist* field. If the values are different, it records the (*prev_val*, *fut_hist*) pair for future use in another table called the Load Value Branch History Table (LVBHT). It then begins accumulating branch history for the new value by copying *cur_val* to *prev_val* and initializing the *fut_hist* field with its current outcome.

The LVBHT is indexed using both the load value and the branch PC. Since a load can take many values during an entire run of the program, the size of the LVBHT could be large. I study the effect of LVBHT size on LBP in Section 6.6.

Continuing to accumulate branch history whenever *cur_val* and *prev_val* match presents a problem. When a previous load's influence on a branch ends and a different set of loads in the branch's Data Set start feeding the branch, *cur_val* will still match *prev_val* for the previous load. The time when a load's influence on a branch ceases, needs to be determined and accumulating branch history for the load has to be stopped at that point. For this purpose, I keep track of a limited dependence graph of programs. Specifically, I keep track of load dependencies of instructions (which loads an instruction is dependent on) using dependence vectors as described in Section 3.2. Similar structures are used to predict the critical path [4, 12] .

During the Commit Stage of a branch, I look up the dependence vectors to determine which loads it is dependent on. These are the loads that currently have an influence on the branch and so branch histories are accumulated only for these loads. The load value, branch history correlations thus accumulated and stored in the LVBHT are available for use by future branches.

6.5.2 Utilizing Data Set–Branch Correlations

Exploiting the established correlations begins when a load that is tracked by the ALT completes. Using the ABT entry indices stored in the load's ALT entry, the set

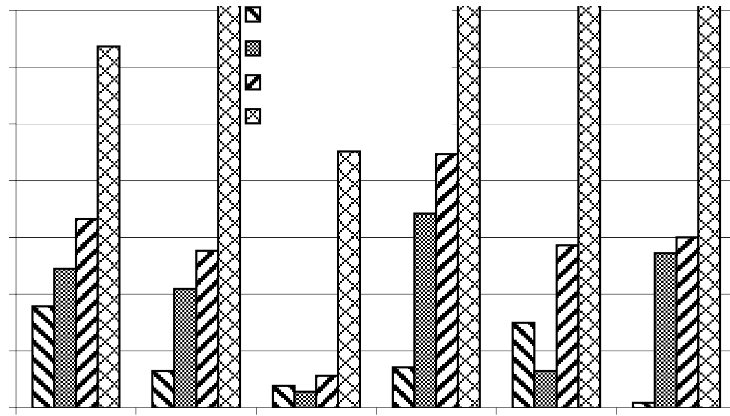
of branches dependent on the load is found. Using the load's value and dependent branch's PC, the LVBHT is accessed to look for any previously stored branch history. If one is found, it is placed in the *past_hist* field of the ABT entry of the dependent branch. When a subsequent instance of this branch is fetched, it will use the rightmost bit of *past_hist* as its prediction, and leave the rest for future instances of the branch.

A load's value becomes available in the Write-back stage of the pipeline. However, the branch histories that are made available by the load are used in the Fetch Stage of a branch. The time gap between these two events poses a problem. By the time a load's value is available, several instances of a branch could have been fetched. Hence, the entire branch history associated with the load's value should not be used. Instead, only a portion of the branch history, corresponding to the yet-to-be fetched instances of the branch can be used.

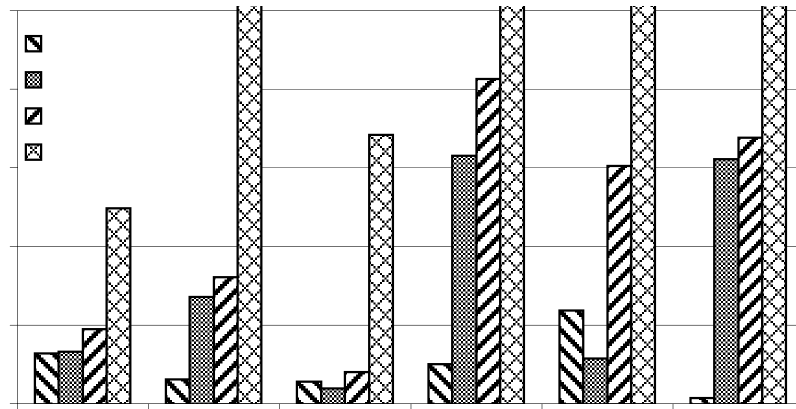
To overcome the time delay problem, the dynamic sequence of loads and branches fetched is kept track of using the *ldbr_order* field in a branch's ABT entry. When a load is fetched, a 1 is shifted in, when a branch is fetched, a 0 is shifted in. When a load completes, it counts the number of 0s following the leftmost 1 in the *ldbr_order* field to determine the number of branches (N) that have been fetched since its fetch. The last N bits are removed from the branch history that is obtained from the LVBHT and the adjusted branch history is placed in the ABT entry of the dependent branch.

6.6 Results

In this Section, I first examine how much of the potential benefits, shown in Section 6.4, can be realized using the LBP scheme. I then examine the impact of increasing pipeline depth on LBP performance. Finally, I present techniques for obtaining load values early and evaluate their performance.



(a) Mis-predictions



(b) IPC

Figure 6.6: LBP: "Ideal vs. Practical" Comparison

6.6.1 LBP: Ideal vs. Practical

Figure 6.6 compares the practical LBP scheme (load values available in the Write-back stage) with the ideal LBP scheme (*16g+fetch,inf-lbp*: 16bit/64K-entry base predictor + load values available in Fetch stage + infinite LVBHT) and a 17bit/128K-entry gshare predictor (*17g*). The practical LBP scheme is presented with either a finite (*16g+wb,fin-lbp*) or an infinite (*16g+wb,inf-lbp*) LVBHT.

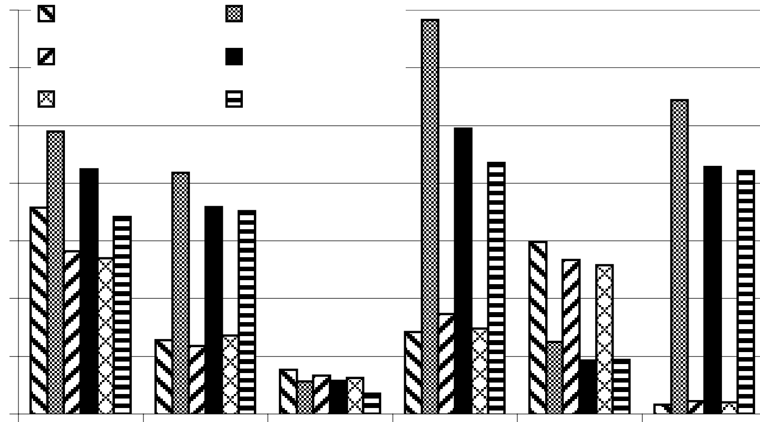
From Figure 6.6 we can see that the practical LBP schemes are able to achieve only a fraction of benefits of ideal LBP. The average reduction in the number of mis-predictions for practical LBP is 11% with an infinite LVBHT, and 8% with a finite LBP, down from 40% for ideal LBP. Comparing the *16g+fetch,inf-lbp* scheme with *16g+wb,inf-lbp*, we can see that most of the benefits are lost due to insufficient distance between loads and dependent branches. The three benchmarks, parser, twolf, and vpr that produce above 6% improvements in IPC with the *16g+wb,inf-lbp* configuration have longer branch histories associated with load values compared to the rest of the benchmarks. With longer histories, LBP's predictions can be used for later instances of a branch while leaving the initial instances of the branch for the base predictor to handle.

In addition to short load–branch distances, insufficient table space is a significant problem for twolf. For twolf, using a 16bit/64K-entry base predictor with a 64K-entry LVBHT (*16g+wb,fin-lbp*) performs worse than a 17bit/128K-entry gshare (*17g*), while using an infinite LVBHT (*16g+wb,inf-lbp*) does considerably better. This is because, for twolf, the loads that feed dependent branches take on a lot of different values increasing competition for LVBHT space.

For four of the benchmarks, gap, gcc, parser, and vpr, *16g+wb,fin-lbp* still performs better than *17g*. For parser and vpr, *16g+wb,fin-lbp* reduces the number of mis-predictions by 11% which translates into IPC gains of 5% and 6% respectively over *17g*.

6.6.2 LBP: Effect of Pipeline Depth

Figure 6.7 shows the effect of increasing the number of pipeline stages on LBP. As the number of pipeline stages increases, two opposing factors come into play. First, the delay in obtaining a load value will increase, reducing the number of branches for



(a) Mis-predictions

(b) IPC

Figure 6.7: LBP: Effect of Pipeline Depth

which LBP has a prediction. Second, the branch mis-prediction penalty will increase, which in turn will increase the IPC benefits of the mis-predictions saved.

Thus, in most cases we can see that the percentage reduction in mis-predictions goes down, while the percentage improvements in IPC either remain the same or do not decrease by as much, compared to their respective base predictors. For the benchmark parser, the detrimental effects of having less LBP predictions outweigh the advantages of increased benefits for the mis-predictions saved, leading to lower

IPC improvements for deeper pipelines.

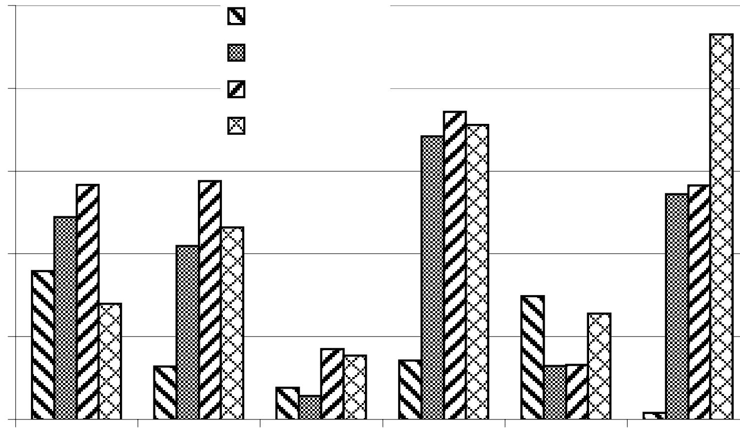
6.6.3 Obtaining Load Values Early

The LBP results so far indicate that the load–branch distance is small in many cases, and that the load values are not available in time for predicting dependent branches. To solve this problem, I investigate several means of obtaining load values early. The first approach is to identify store–load dependencies [6, 63] and utilize the store values themselves for branch prediction (str). For each load, I look for the store (if any) that it is dependent on the most. The second scheme is to predict the values of loads [29] and use the predicted values for branch prediction (lvp). Gonzalez et al. use a similar scheme [16], where they predict the values of the branch operands, while I use value prediction for loads that feed a branch.

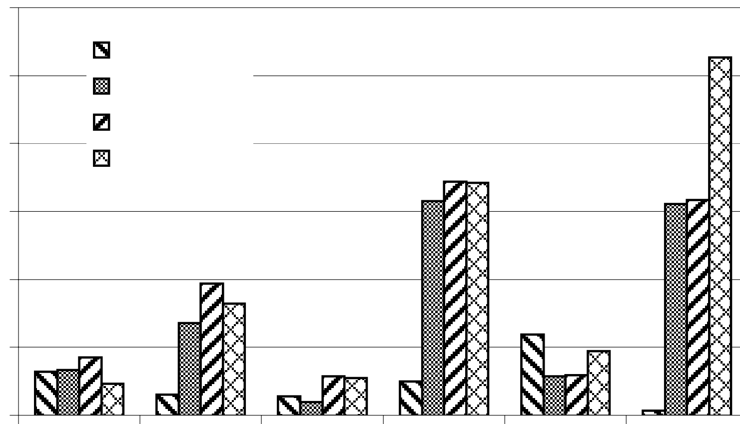
Figure 6.8 compares a 17bit/128K-entry gshare (*17g*) scheme with the baseline LBP scheme (*16g+lbp*: 16bit/64K-entry base predictor + 64K-entry LVBHT), an LBP scheme that uses store values when available (*16g+str-lbp*), and an LBP scheme the uses load value predictions when available (*16g+lvp-lbp*). Figure 6.8(a) shows the percentage reduction in the number of mis-predictions and Figure 6.8(b) shows the percentage improvements in IPC over just a 16bit/64K-entry predictor.

Using store values is beneficial for all benchmarks. The benefits are more for the benchmarks gap, gcc, and gzip compared to the rest of the benchmarks. LBP with store values enables gzip to perform better than the *17g* configuration. The average reduction in the number of mis-predictions for these three benchmarks goes up from 6% to 9% when store values are used for LBP.

Using load value prediction (LVP) is not uniformly beneficial for all benchmarks. It improves the benefits significantly for gzip, twolf, and vpr, while it decreases the benefits for gap. I find that for all benchmarks except vpr, LVP is more beneficial



(a) Mis-predictions



(b) IPC

Figure 6.8: LBP variations

when the base predictor accurately predicts a branch, than when the base predictor is wrong. This is consistent with my expectation that if a load's value is predictable, then the branch will also be predictable. For vpr, I find that for the branch for which LVP helps most, both the branch and the load feeding the branch are predictable (the branch's outcome is predictable if it has accurate global branch history). However, the branch suffers a lot of mis-predictions due to incomplete/inaccurate global branch history. Hence LVP is beneficial even when the branch is mis-predicted. When using

LVP, the percentage reduction in mis-predictions goes up to 19% for vpr, while the percentage improvement in IPC goes up to 11%.

6.7 Conclusions and Future Work

As the demand for better branch prediction increases for future processors, simply increasing the size of existing predictors is unlikely to yield benefits commensurate with added costs. In this thesis, I explore using the correlation between a branch and its input Data Set (the set of loads on which the branch is dependent) for better branch prediction. I propose using the Load-based Branch Predictor (LBP) along with existing predictors.

My experiments to establish the Data Set–branch correlation potential reveal that if all load values are available in time to predict a branch, reductions in mis-predictions of up to 58% and improvements in IPC of up to 36% can be achieved over the default predictor. Similar potential benefits are achieved even when LBP is used with bigger and better base predictors. This indicates that the Data Set–branch correlation is capable of predicting branches whose behavior is not captured by conventional correlating predictors.

Using a practical implementation of the LBP scheme, I am able to reduce the mis-predictions by up to 14% over the base predictor and up to 11% over a 17bit/128K-entry gshare. I find that insufficient load–branch distance and competition for space in the LVBHT are major impediments in realizing the full potential of LBP.

Examining the branches that have the highest discrepancy between ideal LBP and practical LBP, I find several instances where it is possible to increase the load–branch distance far beyond what a traditional compiler does. Hence, I plan to investigate using an executable editing tool [28] to move loads whose values can be used for LBP, as far apart from dependent branches as possible. Also, I plan to investigate

dynamically choosing a minimal set of loads that constitute a branch's Data Set so as to reduce competition for space in the LVBHT.

Chapter 7:
Conclusions

Load instructions form a crucial set of instructions because their latency has a high impact on processor performance. Limitations to instruction level parallelism such as data dependencies, finite resources and branch mis-predictions introduce a variation in the latency requirements of loads. The processor is able to tolerate the latencies of some loads while it needs other loads to complete early. Based on their latency requirements, this thesis classifies loads as critical and non-critical, and investigates using information about critical loads to improve processor performance.

Exploiting critical loads requires the following:

- Understanding the factors that make a load critical (critical load characterization),
- A practical scheme for computing the criticality of loads (critical load classification), and
- Practical optimization schemes that utilize information about critical loads to improve performance.

This thesis makes contributions in each of these areas.

7.1 Thesis Summary

To understand load criticality, I measure the latency tolerance of loads. The latency tolerance of a load is the maximum amount of time that load can be outstanding without adversely affecting performance. To measure load latency tolerance I use a processor simulator that does not complete loads based on a fixed memory hierarchy

but instead delays completion of loads as much as possible. This theoretical processor has the capability to look at future instructions and rollback its state so as to schedule load completions appropriately.

Studying the effect of different load completion policies on processor performance and load latency tolerance I arrive at the following set of conditions that make a load critical:

1. If a load feeds into a mis-predicted branch, or
2. If a load feeds into another load that misses in the L1 cache, or
3. If the processor is unable to find enough independent instructions to execute following a load.

Using the insights from this study I devise a practical scheme to dynamically compute the criticality of loads. This scheme uses information about the dependence sub-graph of loads that it obtains using a hardware structure called the Criticality Table. It also uses a load criticality predictor, similar to a two-level branch predictor to make criticality information available in time for use by criticality exploitation schemes.

This thesis examines two classes of criticality exploitation schemes, (i) those targeted at improving the memory hierarchy and which violate locality of references in favor of criticality, and (ii) those that co-exist with and hence do not violate locality.

First, this thesis compares practical criticality based cache organization as well as prefetching schemes with equivalent locality based schemes to answer the question, "In practice, is it worth violating locality to exploit criticality?" My experiments reveal that critical loads have large working sets and hence it is very difficult to retain critical data in the cache for long enough periods of time to translate into

significant performance gains. Criticality based selective prefetching is beneficial for benchmarks that are resource constrained but the benefits over equivalent locality based schemes is small. Thus I find that in practice, it is not worth violating locality to exploit criticality.

Next, I focus on criticality based schemes that do not violate locality, such as Priority Scheduling and Load-based Branch Prediction. Priority Scheduling queues critical loads earlier than other instructions which helps a subset of the benchmarks and produces performance gains of up to 9%. Load-based Branch Prediction uses the correlation between load values and branch outcomes to predict branches. It decreases the number of mis-predictions by up to 19% over an aggressive default branch predictor which produces gains in IPC of up to 11%.

7.2 Future Work

Power dissipation is an increasingly significant issue in modern processors. As clock rates and die sizes increase, power dissipation is predicted to soon become the key limiting factor on the performance of single-chip microprocessors [17, 56]. So far, my work examines criticality with the aim of maximizing processor performance. Extending my work to investigate criticality in the context of low power processors would be interesting and could lead to non-trivial insights very different from those presented in this thesis. I plan to pursue this line of work in the future.

Also, I plan to continue to work on the LBP scheme as mentioned in Section 6.7 to increase the load–dependent branch distance as much as possible, as well as dynamic means for reducing competition in the LVBHT.

Appendix A:
Critical Load Distributions

My experiments using a theoretical processor with look ahead/rollback capability indicate that completing loads based on their criticality outperforms satisfying loads based on locality of references. However, in practice I find that criticality based cache organization schemes are unable to beat existing locality based schemes.

The main reason for the poor performance of practical criticality based caching schemes is that the working sets of critical loads are large (critical loads touch a lot of data). This can be seen from the criticality distributions shown below. Each graph below has either an effective address(EA) referenced by a load or a load program counter(PC) on the X-axis and the percentage of times each EA/PC was critical on the Y-axis. The EA distributions show that critical loads are not concentrated on a few effective addresss but rather are spread across a vast portion of the virtual address space. Such large critical data working sets prevent criticality based cache organization schemes from reducing competition for cache space for critical loads and hence do not function better than equivalent locality based schemes.

The reason for the large critical data working sets is that critical loads are distributed across the different load load PCs as well, as shown by the PC distributions of the benchmarks. The criticality distribution of a load PC is in turn determined by the mis-prediction distribution of branches that it feeds into, the L1 cache miss distribution of other loads it feeds into, and the changes in direction of branches that influence the amount of instruction-level parallelism that is available during the time the load PC comes up for execution. Since these three distributions change over time, the criticality distribution of loads also change over time, which results in critical loads being distributed across most load PCs.

Bibliography

[1] S. G. Abraham, R. A. Sugumar, D. Windheiser, B. R. Rau, and R. Gupta. Predictability of Load/Store Instruction Latencies. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 139–152, Austin, TX, December 1993.

[2] T. M. Austin and G. S. Sohi. Dynamic Dependency Analysis of Ordinary Programs. In *Proceedings of the 19th Annual Symposium on Computer Architecture*, pages 342–351, May 1992.

[3] D. C. Burger, T. M. Austin, and S. Bennett. Evaluating Future Microprocessors - the SimpleScalar Tool Set. Technical Report 1308, Computer Sciences Department, University of Wisconsin–Madison, July 1996.

[4] B. Calder, G. Reinman, and D. Tullsen. Selective Value Prediction. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 64–75, June 1999.

[5] T. F. Chen and J. L. Baer. Effective Hardware-Based Data Prefetching for High-Performance Processors. *IEEE Transactions on Computers*, 44(5):609–623, May 1995.

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

[7] P. J. Denning. Virtual Memory. *ACM Computing Surveys*, 2(3):153–189, 1970.

[8] S. Dutta and M. Franklin. Block-Level Prediction for Wide-Issue Superscalar Processors. In *Proceedings of 1st International Conference on Algorithms and Architectures for Parallel Processing*, volume 1, pages 143–152, 1995.

[9] H. Dwyer and H. C. Torng. An Out-of-Order Superscalar Processor with Speculative Execution and Fast, Precise Interrupts. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 272–281, December 1992.

[10] M. Evers, S. J. Patel, R. S. Chappell, and Y. N. Patt. An Analysis of Correlation

and Predictability: What Makes Two-Level Branch Predictors Work. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 52–61, June 1998.

[11] A. Farcy, O. Temam, R. Espasa, and T. Juan. Dataflow Analysis of Branch Mispredictions and Its Application to Early Resolution of Branch Outcomes. In *Proceedings of the 31st Annual International Symposium on Micro architecture*, pages 59–68, Dec 1998.

[12] B. Fields, S. Rubin, and R. Bodik. Focusing Processor Policies using Critical-Path Prediction. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 74–85, 2001.

[13] B. R. Fisk and R. I. Bahar. The Non-Critical Buffer: Using Load Latency Tolerance to Improve Data Cache Efficiency. In *Proceedings of the IEEE International Conference on Computer Design*, October 1999.

[14] J. W. C. Fu, J. H. Patel, and B. L. Janssens. Stride directed prefetching in scalar processors. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 102 – 110, December 1992.

[15] A. Gonzalez, C. Aliagas, and M. Valero. A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality. In *Proceedings of ACM International Conference on Supercomputing*, pages 338–347, July 1995.

[16] J. González and A. González. Control-Flow Speculation Through Value Prediction for Superscalar Processors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques*, pages 57–65, California, Oct, 1999.

[17] M. K. Gowan, L. L. Biro, and D. B. Jackson. Power Considerations in the Design of the Alpha 21264 Microprocessor. In *Proceedings of the 35th Annual Design Automation Conference*, pages 726–731, San Francisco, California, Jun 15–19, 1998.

[18] D. Grunwald and D. Joseph. Prefetching Using Markov Predictors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 252–263, June 1997.

[19] T. Heil, Z. Smith, and J. E. Smith. Improving Branch Predictors by Correlating on Data Values. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 28–37, Nov, 1999.

[20] T. L. Johnson and W. W. Hwu. Run-time Adaptive Cache Hierarchy Management via Reference Analysis. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 315–326, June 1997.

[21] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time Spatial Locality Detection and Optimization. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 57–64, December 1997.

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

[23] N. P. Jouppi. Cache Write Policies and Performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201, May 1993.

[24] D. Kroft. Lockup-free Instruction Fetch/Prefetch Cache Organization. In *Proceedings of the 8th Annual International Symposium on Computer Architecture*, pages 81–87, May 1981.

[25] S. Kumar and C. Wilkerson. Exploiting Spatial Locality in Data Caches using Spatial Footprints. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 357–368, June 1998.

[26] L. Kurian, P. T. Hulina, and L. D. Coraor. Memory Latency Effects in Decoupled Architectures with a Single Data Memory Module. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 236–245, 1992.

[27] M. S. Lam and R. P. Wilson. Limits of Control Flow on Parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 46–57, Gold Coast, Australia, May, 1992.

[28] J. R. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In

Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation, pages 291–300, Jun 1995.

[29] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value locality and load value prediction. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 138 – 147, December 1996.

[30] C.-K. Luk. Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 40–51, Jun 30–Jul 4, 2001.

[31] S. A. Mahlke, R. E. Hank, R. A. Bringmann, J. C. Gyllenhaal, D. M. Gallagher, and W. mei W. Hwu. Characterizing the Impact of Predicated Execution on Branch Prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 217–227, San Jose, California, Nov 30–Dec 2, 1994.

[32] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[33] K. N. Menezes, S. W. Sathaye, and T. M. Conte. Path prediction for high issue-rate processors. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 178–188, November 1997.

[34] T. C. Mowry, M. S. Lam, and A. Gupta. Design and Evaluation of a Compiler Algorithm for Prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS V)*, pages 62–73, Boston, Massachusetts, October 1992.

[35] R. Nair. Dynamic Path-Based Branch Correlation. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 15–23, Ann Arbor, Michigan, Nov 29–Dec 1, 1995.

[36] A. Nicolau and J. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Transactions on Computers*, C-33 (11):968–976, November 1984.

[37] S. Palacharla, N. P. Jouppi, and J. E. Smith. Complexity-Effective Superscalar

Processors. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 206–218, June 1997.

[38] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Massachusetts, Oct, 1992.

[39] S. Przybylski. The Performance Impact of Block Sizes and Fetch Strategies. In *Proceedings of the 17th Annual Symposium on Computer Architecture*, pages 160–169, May 1990.

[40] E. M. Riseman and C. C. Foster. The Inhibition of Potential Parallelism by conditional Jumps. *IEEE Transactions on Computers*, C-21:1405–1411, 1972.

[41] J. A. Rivers and E. S. Davidson. Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume 1, pages 154–163, August 1996.

[42] E. Rotenberg, S. Bennett, and J. E. Smith. Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 24–34, Dec 1996.

[43] S. Sechrest, C.-C. Lee, and T. Mudge. Correlation and Aliasing in Dynamic Branch Predictors. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 22–32, May 1996.

[44] K. Skadron, M. Martonosi, and D. W. Clark. Speculative Updates of Local and Global Branch History: A Quantitative Analysis. *The Journal of Instruction-Level Parallelism*, 2, Jan 2000. http://www.jilp.org/vol2.

[45] A. J. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, 1982.

[46] J. Smith. A Study of Branch Prediction Strategies. In *Proceeding of 8th Annual Symposium on Computer Architecture*, pages 135–148, May 1981.

[47] M. D. Smith, M. Johnson, and M. A. Horowitz. Limits on Multiple Instruction Issue. In *Proceedings of the Third International Conference on Architectural Sup-*

port for Programming Languages and Operating Systems (ASPLOS III), pages 290–302, May 1989.

[48] A. Sodani and G. S. Sohi. Dynamic Instruction Reuse. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pages 194–205, June 1997.

[49] G. Sohi. Instruction Issue Logic for High Performance, Interruptable, Multiple Functional Unit, Pipelined Computers. *IEEE Transactions on Computers*, 39(3):349–359, March 1990.

[50] G. S. Sohi and M. Franklin. High-bandwidth Data Memory Systems for Superscalar Processors. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 53 – 62, April 1991.

[51] S. T. Srinivasan and A. R. Lebeck. Load Latency Tolerance in Dynamically Scheduled Processors. In *Proceedings of the 31st Annual International Symposium on Microarchitecture*, pages 148–159, December 1998.

[52] J. Stark, M. Evers, and Y. N. Patt. Variable Length Path Branch Prediction. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 170–179, San Jose, California, Oct, 1998.

[53] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Nov, 2000.

[54] J. E. Thornton. Parallel Operation of the Control Data 6600. In *Proceedings of the Fall Joint Computers Conference*, volume 26, pages 33–40, 1964.

[55] M. S. Thottethodi, S. Chatterjee, and A. R. Lebeck. Tuning Strassen's Matrix Multiplication for Memory Efficiency. In *Proceedings of the 1998 ACM/IEEE SuperComputing Conference*, November 1998.

[56] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing Power in High-Performance Microprocessors. In *Proceedings of the 35th Annual*

Design Automation Conference, pages 732–737, San Francisco, California, Jun 15–19, 1998.

[57] G. S. Tjaden and M. J. Flynn. Detection and Parallel Execution of Parallel Instructions. *IEEE Transactions on Computers*, C-19 (10):889–895, October 1970.

[58] R. M. Tomasulo. An Efficient Algorithm for Exploiting Multiple Arithmetic Units. *IBM Journal*, pages 25–33, January 1967.

[59] T. Tran and C. L. Wu. Limitation of Superscalar Microprocessor Performance. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 33–36, December 1992.

[60] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 191–202, May 1996.

[61] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A Modified Approach to Data Cache Management. In *Proceedings of the 28th Annual International Symposium on Microarchitecture*, pages 93–103, December 1995.

[62] G. S. Tyson. The Effects of Predicated Execution on Branch Prediction. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 196–206, San Jose, California, Nov 30–Dec 2, 1994.

[63] G. S. Tyson and T. M. Austin. Improving the Accuracy and Performance of Memory Communication Through Renaming. In *Proceedings of the 30th Annual International Symposium on Microarchitecture*, pages 218–227, Research Triangle Park, North Carolina, December 1–3, 1997.

[64] D. W. Wall. Limits of Instructional-Level Parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*, pages 176–188, April 1991.

[65] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[66] T.-Y. Yeh and Y. N. Patt. Two-Level Adaptive Training Branch Prediction. In *Proceedings of the 24th Annual International Symposium on Microarchitecture*, pages 51–61, Albuquerque, New Mexico, Nov, 1991.

[67] T.-Y. Yeh and Y. N. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 124–134, Gold Coast, Australia, May, 1992.

[68] C. B. Zilles and G. S. Sohi. Understanding the Backward Slices of Performance Degrading Instructions. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 172–181, Jun 2000.

[69] C. B. Zilles and G. S. Sohi. Execution-based Prediction Using Speculative Slices. In *Proceedings of the 28th Annual International Symposium on Computer Architecture*, pages 2–13, Jun 30–Jul 4, 2001.

Biography

Srikanth T. Srinivasan was born in Chennai, India on February 26, 1974. He did his schooling at Madras Christian College Higher Secondary School until 1991 and received his Bachelor of Technology (Honors) degree in 1995 in the field of Computer Science from Birla Institute of Technology and Science, Pilani, India. From 1995 to 2001, he attended Duke University and received his Ph.D. degree in Computer Science. He along with his advisor Prof. Lebeck won the "Best Paper Award" for their paper "Load Latency Tolerance in Dynamically Scheduled Processors" at the 31st Annual International Symposium on Microarchitecture in 1998. He is the principal author of the paper, "Locality vs. Criticality" that was published in the 28th Annual International Symposium on Computer Architecture.