University of California
Los Angeles

Scalable and Energy Efficient Instruction Scheduling

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Yongxiang Liu

2006

The dissertation of Yongxiang Liu is approved.

Milos D. Ercegovac

Sudhakar Pamarti

Yuval Tamir

Glenn Reinman, Committee Chair

University of California, Los Angeles

2006

In Loving Memory of Dr. Yongzhu Liu, my elder brother
who gave me the guidance, love and support
ever since I was a teenager

Table of Contents

1 Introduction **1**

1.1 Introduction and Motivation 1

1.2 Overview 7

2 Related Work **9**

2.1 Predicting Cache Misses and Access Times 9

2.2 Scheduling Adaptation to Cache Misses 10

2.3 Pre-scheduling Techniques 11

2.4 Counterflow Processors 12

2.5 Speculative Instruction Scheduling 12

2.6 Energy in Schedulers 16

2.7 3D Technologies 18

2.8 3D Microarchitectural Exploration 19

2.9 Delay, Power and Area Modeling 20

3 Load Latency Prediction **22**

3.1 Motivation 22

3.2 Experiment Methodology 24

3.3 Load Latency Prediction 26

3.3.1 Latency History Table (LHT) 27

3.3.2 Cache Latency Propagation (CLP) 28

3.3.2.1 Address Prediction 29
3.3.2.2 Address Precomputation 29
3.3.2.3 Cache Miss Predictor 30
3.3.2.4 Status of In-flight Load (SILO) Structure 31
3.3.2.5 Overall Algorithm of CLP 33
3.3.3 Hybrid Approach 35
3.4 Categorizing Mispredictions 36
3.5 Summary 40

4 Scaling Issue Queue Using Instruction Sorting 41
4.1 Introduction 41
4.2 Instruction Sorting 43
4.2.1 Instruction Waiting Times 44
4.2.2 Instruction Sorting Engine 45
4.3 Experiments and Results 48
4.3.1 Sorting Queue Selection 50
4.3.2 Issue Window Scaling Performance 53
4.3.3 IPC Speedup over Baseline 54
4.4 Summary 57

5 Investigation of Tornado Effects 58
5.1 Motivation 59
5.2 Methodology 62

5.3 The Tornado Effect 65
5.4 Dealing with Tornadoes 67
5.4.1 Reducing Replay Frequency 69
5.4.2 Limiting Threads to Prevent Tornadoes 70
5.4.3 Exploiting TLP 72
5.4.4 Cyclone+: Cyclone Extended with Load Latency Prediction 74
5.4.4.1 Tornadoes in Cyclone+ 76
5.4.4.2 Improving Cyclone+ 77
5.5 Zephyr Design and Performance Evaluations 79
5.5.1 The Zephyr Scheduler 79
5.5.2 Zephyr with the Sliding Window 81
5.6 Load Latency Prediction Analysis 85
5.7 Area and Energy 86
5.7.1 The Area Cost of Zephyr 88
5.7.2 Analysis of Energy Performance 88
5.8 Summary 90

6 Reducing Scheduling Energy 94
6.1 Motivation 94
6.2 Scheduling Techniques 97
6.2.1 Conventional Wakeup-and-Select vs Speculative Scheduling 97
6.2.2 Replays in Speculative Scheduling 97
6.2.3 Latency Prediction and Sorting 99

6.3 Methodology 101
6.3.1 Structures from Proposed Techniques 102
6.4 Experiments and Results 104
6.4.1 Prediction, Sorting and Buffering Structures 104
6.4.2 Speculative Scheduling 104
6.4.3 Register File Energy 105
6.4.4 Issue Queue Energy 105
6.4.5 Overall Performance and Energy Reduction 108
6.5 Summary 111

7 Scaling Issue Queue Using 3D IC Technologies 112

7.1 Introduction to 3D Technologies 113
7.2 Scaling Issue Queue in 3D 116
7.2.1 Issue Queues 116
7.2.2 3D IQ Design: Block Folding 118
7.2.3 3D IQ Design: Port Partitioning 118
7.2.4 Modeling Methodology 119
7.2.5 3D Issue Queue Performance 121
7.2.6 Scaling to Multiple Silicon Layers 122
7.2.7 Issue Queue Sizes 123
7.3 Extended Studies 123
7.3.1 Caches 124
7.3.1.1 3D Cache Design: Block Folding 125

7.3.1.2 3D Cache Design: Port Partitioning 126
7.3.2 Other Cache-Like Architectural Blocks 126
7.3.3 Modeling Methodology 127
7.3.4 3D Block Performance 128
7.3.5 Scaling to Multiple Silicon Layers 129
7.3.6 Impact of 3D Bonding Technology 129
7.3.7 Structure Sizes 134
7.4 Placement of 3D Issue Queue and Other Blocks 134
7.4.1 MEVA-3D Flow 135
7.4.2 Enhancements to MEVA-3D 135
7.4.2.1 Architectural Alternative Selection 135
7.4.2.2 Cube Packing Engine 136
7.5 Microarchitectural Exploration 137
7.5.1 Scaling Architectural Sizes 143
7.5.2 Frequencies 144
7.5.3 Number of Layers 145
7.6 Summary 146

8 Conclusion and Future Directions 147

8.1 Conclusions 147
8.2 Future Directions 149
8.2.1 Improving Load Latency Predictions 149

8.2.2 Latency Prediction of Cross-core Communications in Multi-core Processor 149

8.2.2.1 3D Processors 150

References **151**

List of Figures

2.1 Face-to-Back and Face-to-Face integration technologies 19

3.1 The Overall Prediction Structure 28

3.2 Precomputability 29

3.3 The Average Number of Loads that Alias In-Flight Memory Accesses 31

3.4 The Percentage of Memory Accesses (miss both L1 and L2) That Are Correctly Predicted 34

3.5 The Percentage of L1 Hits That Are Correctly Predicted 34

3.6 The Prediction Accuracy Rate of All Loads 34

3.7 The breakup by Prediction Methods in the Hybrid Approach . . . 36

4.1 the Overall Scheduling Architecture 43

4.2 Computation of Deterministic Waiting Time 44

4.3 A Sample Configuration of the Sorting Engine 45

4.4 The Potential of Scaling the Issue Window 49

4.5 The Histogram of Predicted Waiting Time for Selected Benchmarks 51

4.6 The relative performance of several sorting queue configurations . 52

4.7 The Scaling Effect of Our Approach 53

4.8 The IPC Speedup with 16-entry Issue Queue over a Baseline Configuration 54

4.9 The IPC Speedup with 32-entry Issue Queue over a Baseline Configuration 56

5.1 Snapshot of Cyclone Baseline 68

5.2 The Performance of Different Techniques to Combat Tornadoes on the Original Cyclone Scheduler. 68

5.3 Baseline Cyclone (Cyclone) and Cyclone extended with load latency prediction (Cyclone+). 74

5.4 Snapshot of Cyclone Extended with Load Prediction (Cyclone+) . 74

5.5 the IPC Performance of Cyclone Extended with Load Prediction (Normalized with Baseline Cyclone) 77

5.6 Zephyr Scheduler Architecture 78

5.7 Speedup of Zephyr and Zephyr with Sliding Window 78

5.8 Snapshot of Zephyr with Sliding Window 83

5.9 Percent of Loads that are Predicted by LHT, SILO and Default . 87

5.10 Prediction Accuracy of L2 Misses 87

5.11 the Histograms of Predicted Waiting Times for ammp.gcc and gzip2.gap. The smaller figures in the upper right corner present the full y-axis scale to better illustrate shorter latency instructions. 87

5.12 the Percentage of Total Energy Consumed by Zephyr Structures, including prediction structures and buffers 91

5.13 the Percentage of Reduced Register File Energy (per committed inst.) in Zephyr Sliding Window Compared with Cyclone 91

5.14 Comparing the Energy Per Committed Instruction: Cyclone vs Zephyr Sliding Window, Normalized with Cyclone 91

6.1 The Architecture to Perform Prediction, Sorting and Buffering . . 101

6.2 Performance in IPC 105

6.3 Energy Consumption in the Latency Prediction, Sorting and Buffering Structures (per instruction) 106

6.4 Reduction in Issue Queue Occupancy 106

6.5 Issue Queue Energy Consumption Per Committed Instructions . . 109

6.6 Reduction in Number of Scheduling Replays 110

6.7 Register File Energy Consumption Per Committed Instructions . 110

6.8 Total Energy Consumption Per Committed Instructions 111

7.1 (a): A Single IQ Cell with Four Tag Lines and Four Access Ports. Over 99% of the area is occupied by tags and access ports. . . . 119

7.2 Issue Queue Partitioning Alternatives: (a) An issue queue with 4 tag lines. (b) Block Folding: dividing the issue queue entries into two sets and stacking them. The tags are duplicated in every layer. Only the X-direction length is reduced. (c) Port Partitioning: the four tags are divided into two tags on each layer. Both X and Y direction lengths are reduced. 120

7.3 Performance of Scaled Issue Queue in 3D 124

7.4 Cache Block Alternatives (a) A 2-Ported Cache: the two lines denote the input/output wires of two ports. (b) Wordline Folding: Only Y-direction length is reduced. Input/output of the ports are duplicated. (c) Port Partitioning: Ports are placed in two layers. Both X and Y direction length are reduced. 125

7.5 The improvement in area, power and timing for dual layer vertical integration. 128

7.6 The improvement in area for multilayer F2B vertical integration. . 130

7.7 The improvement in timing for multilayer F2B vertical integration. 130

7.8 The improvement in power for multilayer F2B vertical integration. 131

7.9 Impact of Via Size on Timing using F2B, Port Partitioning 131

7.10 Impact of Via Size in Power using F2B, Port Partitioning 132

7.11 Latency Impact of Vertical Integration when Scaling the Size of Two Critical Blocks 133

7.12 Performance speedup for dual silicon layer architectures relative to a single layer architecture. 138

7.13 Floorplans for a single Layer Architecture (the best architectural configuration as determined by our modified version of MEVA-3D) 139

7.14 Floorplans for Dual Layer Architecture with 2D-only Blocks (the best architectural configuration as determined by our modified version of MEVA-3D) 140

7.15 Floorplans for Dual Layer Architecture with 2D and 3D Blocks (the best architectural configuration as determined by our modified version of MEVA-3D) 141

7.16 Average core temperature for the single layer architecture (shown at left) and the dual layer architecture with 3D blocks (the hottest layer is shown at right). 142

7.17 Performance when doubling critical resources. 143

7.18 BIPS performance for different clock frequencies. 144

7.19 Performance when scaling the number of silicon layers. 146

List of Tables

3.1 The Statistics of Baseline Benchmarks 25

3.2 Prediction based on Output of Miss Detection Engine. 33

3.3 Breakup of mispredictions by categories in percentages. 37

4.1 Effectiveness of the Sorting Engine. 47

5.1 Applications grouped by strong tornado effects, weak tornado effects, and the mixes. 63

5.2 Processor Parameters. 64

5.3 Comparison of the number of replays and structural hazards (i.e. switchback conflicts) in the scheduling queues. Symbols: "C" — Cyclone, "C+" — Cyclone+, "Z" — Zephyr, "zW" — zephyr sliding Window. 82

5.4 The Area of Zephyr Components 89

6.1 The benchmarks used in this study. 102

6.2 Processor Configuration. 103

7.1 Percentages of Reduction in Delay, Area and Power Consumption from 3D Design. Symbols: "nL" – n number of Layers, "F" – Folding, "PP" –Port/Tag Partitioning 121

7.2 Reduction in delay and energy obtained from HSpice and modified 3DCACTI and HSpice as compared to 2D blocks 127

7.3 Architectural parameters for the design driver used in this study. . 137

Acknowledgments

I would like to express my gratitude and appreciations to the many people have lent me their support, guidance, and help to make this dissertation possible.

First and foremost, I would like to thank my academic advisor, Professor Glenn Reinman for his support, encouragement, and guidance. It is my privilege to work with him these past four years. His deep knowledge in this field, his enthusiasm for research, and his trust and patience have been the guiding forces behind my research.

I am grateful for the opportunities to collaborate with professors Gokhan Memik and Jason Cong and for their invaluable guidance. I would also like to acknowledge and thank professors Yuval Tamir, Bill Mangione-Smith, Milos D. Ercegovac and Sudhakar Pamarti who served on my committee and provided many insightful comments.

It has been a pleasure working with so many talented and dedicated people at UCLA. I wish to express my special thanks to Anahita Shayesteh, Yuchun Ma and Eren Kursun with whom I collaborated on this research. I would also like to extend my appreciation to Si Chen, Thomas Yeh, Adam Kaplan, and Kanit Therdsteerasukdi for their help and for creating a stimulating research environment.

I am thankful to Dejun Wang, for his friendship and generous help on getting through the paperwork, and Hui Wang, for sharing so many rides and for making the long commute to UCLA possible.

My special love and gratitude go out to my wife, Jinping Wan for her unconditional support, understanding, and sacrifice without which I could not have accomplished this work. I dedicate this thesis to her.

Vita

1998	B.S.(Electrical Engineering), Nanyang Technological University, Singapore
1998-2002	Teaching Assistant, National University of Singapore
2002	M.S.(Computer Science), National University of Singapore
2003-2005	Research Assistant, University of California, Los Angeles
2004	M.S.(Computer Science), University of California, Los Angeles
2006	Doctor of Philosophy, University of California, Los Angeles

Publications

"Reducing the Energy of Speculative Instruction Schedulers." Yongxiang Liu, Gokhan Memik, and Glenn Reinman. In Proc. of the International Conference of Computer Design (ICCD), Oct 2005.

"Tornado Warning: the Perils of Selective Replay in Multithreaded Processors." Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman In Proc. of the ACM International Conference on Supercomputing (ICS'05), Boston MA, Jun 2005.

"The Calm Before the Storm: Reducing Replays in the Cyclone Scheduler." Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. In Proc. of the IBM T.J. Watson Conference on Interaction between Architecture, Circuits, and Compilers, Oct 2004.

"Scaling the Issue Window with Look-Ahead Latency Prediction." Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. In Proc. of the International Conference on Supercomputing, June 2004.

"TCP-CM: A Transport Protocol for TCP-friendly Transmission of Continuous Media." Yongxiang Liu, S. N. Srjith, L. Jacob, and A. L. Ananda. In Proc. of the 21st IEEE International Performance, Computing, and Communications Conference (IPCCC 2002), Phoenix, Arizona, April 2002.

Abstract of the Dissertation

Scalable and Energy Efficient Instruction Scheduling

by

Yongxiang Liu

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2006

Professor Glenn Reinman, Chair

In contemporary microprocessors, the performance of out-of-order instruction scheduling is critical. The IPC mainly depends on the amount of ILP (instruction level parallelism) that is exposed by instruction schedulers. The scheduling logic is also responsible for a large portion of the total energy consumption. In this dissertation, we explore several techniques to improve the IPC and energy performance of instruction scheduling.

In a conventional out-of-order superscalar, instructions enter the scheduling window in the original program order and may be executed in any order. Instructions that depend on a long latency load consume scarce issue queue slots even though they will spend a long time in the window. Essentially, if load latencies can be determined far enough in advance, we can avoid wasted space in a conventional scheduling window. However, predicting load latency is not trivial. Memory accesses may take anywhere from a few cycles if they hit in the level 1 cache to a few hundred cycles if they have to access the main memory due to a cache miss. In this dissertation, we present an accurate, lookahead prediction of

memory latency, which correctly predicts the latencies of 83% of the cache misses, and 98% of the cache hits. Assisted by memory latency predictions, the expected issue time of instructions can be predicted. We apply it with a simple structure that sorts instructions in their expected issue order based on the predicted waiting time – enabling a conventional scheduler using a 12-entry issue queue to achieve comparable performance to scheduler with a 24-entry issue queue.

In recently proposed speculative selective-replay schedulers (e.g. P4), the dependents of loads are scheduled based the assumption that loads always hit. The dependents are replayed repeatedly until a load's completion. This is the primary cause of the Tornado effect, which features a positive feedback loop where replays can spawn more and more replays and eventually consume most of the available issue bandwidth. We propose to schedule based on predicted load latencies instead of always assuming L1 cache hits, and to sort all instructions based on their waiting times. Our scheme can significantly reduce scheduling queue occupancies and replays, and therefore reduce the impact of the tornado effect. We further reduce this effect by proposing preventive measures to detect a tornado in its early stages of development, and dynamically limit the number of instructions in the scheduling queues. We are able to see a substantial drop in tornadoes when these techniques are combined together.

The energy due to scheduling logic and register file accesses make up a large portion of the total microprocessor energy consumption. Benefiting from reduced scheduling queue occupancies, our scheme significantly reduces the energy consumption in conventional schedulers. With the elimination of unnecessary accesses that stem from scheduling replays, our scheme even further reduces energy consumption in speculative schedulers.

Finally, we propose to scale scheduling queue sizes using physical level design

that leverages 3D via integration techniques. The instruction scheduling queue is a heavily-ported structure. By dividing the ports and placing them into different layers, significant amount of power and access time are reduced. Using this technique, the scheduling queue size can be doubled with an even faster access time. Our experiments show that a 3D architecture using this technique for issue queue and other critical structures achieves a 36% performance speedup.

CHAPTER 1

Introduction

1.1 Introduction and Motivation

Contemporary microprocessor designs often employ out-or-order scheduling to extract instructing level parallelism(ILP) from serial instruction streams. The IPC performance of such microprocessors mainly depends on the amount of ILP that is exposed by instruction schedulers (issue queue).

A larger issue queue size can certainly expose more independent instructions that can execute out-of-order. However, designing a large instruction window does not come for free. The hardware structure required by an issue queue is rather complex.

The issue queue (IQ) needs to examine all the instructions and chooses ready instructions for execution each cycle. It operates in two phases: wakeup and selection. During wakeup, the destination tags of the already scheduled instructions are broadcasted across the issue queue and are compared with each IQ entry associatively. An instruction wakes up when all sources operands become ready. In the selection phase, the instructions that wake up are selected for execution. The long wires used to broadcast the tags and the large number of comparators needed to implement an associative search cause significant access delay.

Further more, the issue queue is not suitable for a pipelined implementation.

The wakeup-and-selection loop has to complete in one cycle to ensure all dependent instructions are issued back-to-back in consecutive cycles [PJS97, SBP00, BSP01]. As shown by Palacharla et. al, the delay of the issue logic quadratically increase as IQ size scales [PJS97]. Hence, simply scaling issue queue size in the current designs is not a viable solution to improve scheduling performance.

In recent years, energy performance has become a critical design constraint. The issue queue is often a major contributor of the overall energy consumption of the chip. Significant amount of energy is dissipated due to the charging and discharging of the wire capacitance of the tag line and the gate capacitance of the comparators. In addition, the power is also dissipated when the instructions are written into the IQ (instruction dispatching) and when they are read out of the IQ (instruction issuing). In [FG01], it is estimated that instruction issue queue logic is responsible for around 25% of the total power consumption on average. Wilcox et al. [WM] showed that the issue logic could account for 46% of the total power dissipation in future out-of-order processors that support speculation.

In this dissertation, we present several effective solutions to improve scheduling performance without further complicating issue queue designs. We also present solutions to effectively reduce the IQ energy consumption.

Memory Access Latencies

The variability of memory accesses significantly complicates instruction scheduling and degrades microprocessor performance. Memory accesses may take anywhere from a few cycles if they hit in the level 1 cache to a few hundred cycles if they have to access the main memory due to a cache miss.

In a conventional out-of-order superscalar, the scheduling window (issue queue)

size is limited due to its circuit complexities. Instructions, in particular, the dependents of missed loads enter the scheduling window in the original program order, then consume the scarce issue queue slots even though they will spend a long time in the window.

In recently proposed speculative selective-replay schedulers (e.g. P4), the dependents of loads are scheduled based the assumption that loads always hit. The dependents are replayed repeatedly till loads completion. This is the primary cause of the Tornado effect, which features a positive feedback loop where replays can spawn more and more replays and eventually consume most of the available issue bandwidth.

Essentially, if load latencies can be determined far enough in advance, we can avoid wasted space in a conventional scheduling window and reduce replays in a speculative scheduler.

In this dissertation, we present an accurate, lookahead prediction of memory latency that can help predicting the expected issue time of instructions [LSM04b]. Our scheme correctly predicts the latencies of 83% of the cache misses, and 98% of the cache hits. Our scheme is able to predict memory latencies early in the renaming stage. Predicting latencies early in the pipeline stages allows intelligent proactive optimization on the scheduling.

Optimizations of Instruction Scheduling

As the variable memory latencies are known in advance, many optimization can be performed on instruction scheduling.

We first improve a conventional issue queue by using a simple structure that sorts instructions in their expected issue order based on the predicted waiting

time [LSM04b]. Instructions are buffered in the sorting architecture for their approximated waiting time. Hence, instructions that enter the issue queue are expected to be issued soon. Our experiment show such a scheme helps a conventional scheduler using a 12-entry issue queue to achieve comparable performance to scheduler with a 24-entry issue queue.

As future technologies push toward higher clock rates, traditional scheduling techniques that are based on wake-up and selection from an instruction queue fail to scale due to both timing and power constraints [EHA03]. Furthermore, the use of increasing larger physical register pools to deliver input operands puts the register file access on the load resolution loop. Speculative instruction schedulers that use selective replay to adjust to misscheduled instructions have been proposed as one solution to address these concerns [HSU01, EHA03].

However, such scheduling structures suffer from a phenomenon called the Tornado effect [Car04], a positive feedback loop where misschedulings can spawn more and more misschedulings and eventually consume all available issue bandwidth. One misscheduled instruction can cause further misschedulings due to data dependencies and structural hazards in the scheduling window. Replays can be frequent in speculative schedulers due to the underestimation of instruction waiting times. Speculative schedulers optimistically assume that memory accesses will always hit in the data cache in an effort to avoid increasing the back-to-back communication latency between loads and their dependents. However, variability in load latency and competition for functional units and other structural hazards creates uncertainty that cannot always be considered by a speculative scheduler.

Our studies [LSM04a, LSM] show that sorting instructions based on their predicted latencies prior to entering the scheduling window can substantially reduce the tornado effect. This benefit comes from the more accurate waiting

time prediction as load misses are taken into account, as well as reductions in scheduling hazards.

However, the difficulty in predicting uncertainty from structural hazards and load latency can still result in tornadoes without some method of dynamic adaptation. We propose preventive measures to detect a tornado in its early stages of development, and dynamically limit the number of instructions in the scheduling queues. Such adaptation is too restrictive without also doing latency prediction. As a result, we see little improvement when the adaptation is applied alone. When the adaptation is combined with latency prediction and buffering, we are able to see a substantial drop in tornadoes.

Energy Reduction Techniques

The energy due to scheduling logic and register file accesses make up a large portion of the total microprocessor energy consumption.

One source of wasted IQ energy is due to the long latency instructions. As mentioned, contemporary designs allow instructions with long waiting times (i.e. dependent on long latency operations) remain in the issue queue while waiting for their operands. An issue queue consumes proportional amount of energy with the number of active entries [BSB01, BKA03]. If the long latency instructions are known a priori, we can buffer their dependents before they enter the issue queue. In this way, the issue queue energy is reduced as a result of reduced number of active entries in IQ.

In speculative instruction schedulers, the dependents of cache misses are speculatively scheduled by assuming cache hits. These instructions will speculatively access the physical register files repetitively until their operands are ready. Sig-

nificant amount of energy is wasted due to speculative register file accesses.

In this dissertation, we reduce the energy consumption in the issue queue and register file by applying instruction sorting and load latency prediction. Benefiting from reduced scheduling queue occupancies, our scheme significantly reduces the energy consumption in IQ. With the elimination of unnecessary accesses that stem from scheduling replays, our scheme even further reduces energy consumption in speculative schedulers.

Scaled Issue Queue

Finally, we present solutions to directly scale issue queue by leveraging 3D integrated circuit technology. 3D integrated circuit technology are shown to be able to dramatically reduce the length of interconnection wires [CJM06]. The delay and energy are also reduced because the shorten wires have less capacitance and resistance.

In the issue queue, a significant contributor to delay and energy is the wire latency of the tag bits and match lines. Issue queue is a heavily-ported structure. As an example, a 4-way issue queue typically needs 12 ports – 4 ports each for comparison, read, and write. Our studies show that most of the silicon area is consumed by ports placement. In the wakeup logic, the ports occupy overall 99% of the total area.

One way to reduce tag line wire delay is to fold the issue queue entries and place them on different layers [TXV05]. This approach effectively shortens the tag lines.

A more efficient strategy to attack the port requirements is port partitioning, which places tag lines and ports on multiple layers, thus reducing both the height

and width of the issue queue. The reduction in tag and match-line wire length can help reduce both power and delay. The selection logic also benefits from this, as the distance from the furthest issue queue entry to the arbiter is reduced. This will speed up the comparison and also reduce power consumption. Our experiments show that a 3D issue queue using port partitioning can double its size with even decreased delay and comparable energy consumption.

1.2 Overview

The rest of this dissertation is organized as follows. In the next chapter, we survey related work in the fields of instruction scheduling and 3D integrated circuits.

In Chapter 3, we present our memory latency prediction techniques and analyze their prediction accuracy. We further categorize those load latencies that are mispredicted and highlight directions to further improve the prediction accuracy.

As the load latencies predicted in advance, we are able to obtain the expected waiting times for all instructions. In Chapter 4, we describe a sorting structure that sorts instructions in the expected waiting times. Further we evaluate the the IPC performance in a conventional wakeup and selection scheduler using the sorting structure.

In Chapter 5, we present a quantitative investigation of Tornado Effect. In this chapter, we also describe the Zephyr architecture, which effectively prevents the formations of tornadoes. The cost of Zephyr architecture, in terms of both area and energy are analyzed in details.

In Chapter 6, we propose to reduce the overall energy savings by reducing issue queue occupancies and by eliminating unnecessary accesses to the register

files. We compare the energy consumptions of a conventional wakeup and select issue queue and a conventional speculative issue queue with that of our proposal. Our experiments show that our scheme can save overall 20% of the total energy.

In Chapter 7, we present a scalable issue queue solution that leverages 3D integrated circuits technologies. We describe a novel 3D issue queue implementation that can significantly scale issue queue size while not increase the latency and energy consumption. We further explore the overall architecture performance in a 3D microprocessor that is equipped with 3D issue queue.

Finally, in Chapter 8, we identify a number of remaining challenges with our approach and present future directions for interested reader.

CHAPTER 2

Related Work

Much work on improving out-of-order scheduling performance and scaling issue queue have appeared in the literature. We now examine some of the more relevant prior work in details.

2.1 Predicting Cache Misses and Access Times

The Alpha 21264 uses the most significant bit of a global 4-bit history counter to determine whether a load hits in the cache [KMW98]. The saturation counter decrements by two on cycles when there is a load miss, otherwise it increments by one when there is a hit. However, it is difficult to capture cache misses based on global history. Yoaz et al. [YER99] propose to adapt branch predictors such as local history predictors, gshare [McF93] and gskew [MSU96] for load hit/miss prediction. However, branch predictors cannot predict cache hit/miss accurately.

The concept of load miss detection was explored by Memik et. al. [MRM03a] [MRM03b]. They use the cache miss detection engine (CMDE), and a previously accessed table (PAT) to determine the latency of memory accesses. They detect cache misses using the memory address operand at the execution stage. Peir et al. [PLL02] propose load miss detection based on a bloom filter, which is indexed by some bits of the effective address of memory instructions. Their prediction is

also performed in the execution stage. We propose to predict the load latencies far in advance of execution, as this is crucial in better scaling the scheduling window.

2.2 Scheduling Adaptation to Cache Misses

Several dynamic adaptation schemes have been proposed to handle non deterministic memory latencies.

Lebeck et al. propose moving instructions dependent on missed loads into a waiting instruction buffer (WIB) to prevent them from clogging the issue queue [LKL02]. Although not explicitly mentioned, the scheduler needs to speculate the presence of a load miss, and then migrate chains of dependent instructions into the WIB. It must also speculate when a load completes, and then migrate these instructions back into the issue queue to avoid exposing the pipeline stages from scheduling to execution. In addition, when instructions migrate into the WIB, they consume processor issue width. When instructions migrate back into the issue queue, they consume dispatch width. Instructions that re-enter the issue queue will experience the impact of the schedule to execute window (the pipestages of scheduling and register file access).

Another proposes Cyclone [EHA03], an effective broadcast free dynamic scheduler. Cyclone makes use of a switchback queue to delay the issue of instructions based on their deterministic latency, where loads are always predicted to hit in the level one cache. Both of these approaches handle load misses by consuming issue bandwidth to either send dependents of load misses to auxiliary storage (WIB) or to the head of the switchback queue (Cyclone). These techniques dynamically adapt to non-deterministic instruction latency, potentially consuming available

issue bandwidth to shuffle dependent instructions. The scheduler still does not have any knowledge of when the load will actually complete. Both approaches could leverage load latency prediction.

Our approach uses a similar concept of timing instructions to dynamically sort instructions prior to issue. Similar to Cyclone we predict the waiting time of instructions far in advance at rename. But, they optimistically assume that loads will hit in the first level cache and then dynamically adapt when the latency is not as expected. Our approach incorporates miss and in-flight load propagation structures, which can accurately predict load misses and propagate the times to their dependents.

2.3 Pre-scheduling Techniques

Several pre-scheduling techniques have appeared in the literature. Palacharla et al. [PJS97] propose to detect chains of dependent instructions and queue them into a set of FIFOs. In this way, only instructions at the FIFO heads are monitored for execution. This allows a faster clock and simplified wakeup and arbitration logic.

Michaud et al. [MS01] have proposed a technique called data-flow prescheduling to optimize instruction ordering based on data dependencies before they enter the issue stage. Instructions are arranged in data-flow order so that they pass through the issue queue quickly. With this technique, a large effective issue window can be build from a small and fast issue queue. However, they do not consider prediction of cache misses and prescheduling their dependents accordingly. Canal et al. [CG01a] [CG01b] have independently proposed a similar idea based on instruction latencies, but with a different implementation. Their scheme

works directy on the issue logic and requires a shorter pipeline than data-flow prescheduling [MS01].

The ILDP processor further refines dependence-based scheduling by including instruction set support for describing dependent instruction sets [KS02]. Raasch et al. [RBR02] propose to break the instruction queue into segments forming a pipeline. The flow of instructions is governed by a combination of data dependencies and predicted operation. Ideally instructions reach the final segment where they can be issued when their operands are available.

2.4 Counterflow Processors

Counterflow processors are proposed as an alternative to the superscalar approach by using highly localized communication to resolve scheduling issues [SSM94]. In the counterflow pipeline, instructions and data flow in opposite directions on circular queues. When instructions pass their input operands, they capture the data. Once an instruction has captured all its operands, it locates a functional unit, and leaves the queue to begin execution. Ernst et al's scheduler Cyclone [EHA03] also uses decentralized dependence analyses to design a broadcast free scheduler. Cyclone predicts the waiting times of instructions, which are then placed into a countdown queue where they will be delivered to the execution engine.

2.5 Speculative Instruction Scheduling

Stark et al. [SBP00] propose to pipeline scheduling with speculative wakeup to scale processor cycle time. The technique pipelines the scheduling logic into two cycles while still allowing back-to-back execution of dependent instructions.

It achieves this by speculating instruction with available operands are always selected for execution. Such scheduling approach is based on a monolithic issue window, which cannot scale proportionally as future clock speed doubles or even triples. One the other hand, the replay-based scheduling effectively release the scheduling loop from being the performance bottleneck.

Ernst et al. [EHA03] proposed Cyclone, a broadcast-free dynamic instruction scheduler with selective replay. The Cyclone scheduler relies on a simple one-pass scheduling algorithm to predict the time when instructions should execute. Once decided, this schedule is implemented with a timed queue structure. In the event of an incorrect scheduling, Cyclone is also able to support efficient selective instruction replay [HSU01]. Execution time prediction is accomplished with a timing table and MAX calculation. The timing table is indexed by logical register, and returns the expected delay until the logical register is ready. Instructions use the timing table to estimate when their input dependencies will be available, and are buffered in a countdown queue for this expected waiting time.

Instructions are injected into the tail of the Cyclone scheduler queue with a prediction of how far the instruction should progress down the countdown queue before turning around and heading back towards execution in the main queue. As mentioned, switching back from the countdown to the main queue can be a source of conflict and must be resolved. Once an instruction reaches the head of the main queue, a table of physical register ready bits is used to determine whether or not all input operands to the instruction are ready. If any operand is not ready, the instruction is routed back to the countdown queue and replays. Cyclone has an optimization option that allows a replayed instruction to consult the timing table to reevaluate its waiting time. We use this optimized version for fair comparison. Cyclone assumes that all loads hit in first level cache. Cache

misses will likely result in misscheduled instructions, creating replays.

Hu et al. [HVI04] propose WF-Replay (wakeup free replay), a 32-entry issue queue structure where instructions can be selected for issue from any queue slot. Each queue slot tracks the predicted waiting time for an instruction in a counter, decrementing this waiting time each cycle. The instructions "wake up" and can be selected for issue only when this counter reaches zero. This is designed to avoid the structural hazards in the switchback queues of Cyclone. Instructions must be replayed if their input dependencies are not ready or if they encounter a structural hazard (like insufficient functional units). As in Cyclone, a misscheduling in WF-Replay can potentially cause a chain of further misschedulings. Structural hazards can still spawn even more misschedulings – if one instruction is delayed due to functional unit contention, its dependents can wake up before their inputs are ready. The selection logic is also a potential bottleneck as every queue slot essentially participates in selection. This may limit the size of the scheduling window and the amount of load latency that may be tolerated. The Precheck enhancement [HVI04] checks the ready-bit register to avoid replays. However, this introduces new complexity in that ready-bit register update and instructions selection must complete in one cycle to ensure dependent instructions are executed back-to-back. The WF-Segment enhancement reduces this complexity by selecting instructions only from the segment of 0 waiting time [HVI04]. However, instructions in the other segments need to migrate to the lower segments at the appropriate time, provided that there are no hazards. As in the switchback queues of Cyclone, this instruction migration can result in an increase in structural hazards, which in turn can cause more misscheduled instructions.

Hu et al. [HVI04] further propose a Precheck enhancement, which introduces a ready-bit register that indicates whether or not the operands are ready for an

instruction in a queue slot. Instructions participate in selection only if their ready-bit register is set. However, this means that the ready-bit register update and instruction selection need to complete in one cycle to ensure that the dependents of single-cycle latency instructions can be issued without any additional latency. This is similar to the level of circuit complexity in a conventional wakeup-and-select issue queue. They further propose WF-Segment to reduce the pressure of wakeup-and-selection by dividing the issue queue into 4 segments. Instructions with waiting times of 0, 1-2, 3-4, or greater than 4 cycles are placed into segment 1, 2, 3, and 4 respectively. Instructions in the upper segments can migrate to the lower segments at the appropriate time, provided that there are no hazards. As in the switchback queues of Cyclone, this instruction migration can result in an increase in structural hazards, which in turn can cause more misscheduled instructions. Moreover, by collapsing instructions with waiting times longer than 4 cycles into one segment, the ability of WF-Segment to issue in the face of long latency instruction chains can be limited.

Both WIB and WF-Replay structures are examples of speculative scheduling mechanisms that can be impacted by misscheduled instructions and structural hazards beyond just contention for functional units (speculatively switching from IQ to WIB and back in the case of the WIB and switching between segments in WF-Segment). In this dissertation, we focus on Cyclone, which also exhibits misscheduled instructions and structural hazards (in the switchback queues), but our approach could easily be generalized to the WIB, WF-Replay, or any other approach that relies on speculative scheduling to avoid conventional wakeup and selection logic.

Ehrhart and Patel propose a speculative scheduling scheme by predicting the instruction waiting times using a PC-indexed history table [EP04]. However, the

waiting time of a static instruction varies dynamically. As a corrective measure, an allowance is added to the predicted waiting time. The allowance is dynamically and progressively adjusted by balancing between the amount of replays and the amount of wasted issue opportunities. Decreasing the allowance will lead to underestimation of waiting times, which causes scheduling replays, while increasing the allowance will cause instructions to wait unnecessarily long, thus wasting issue opportunities. Instructions replays or wasted issue opportunities are unavoidable to guide the process of adjustment. More critically, the time that dependents of loads need to wait varies largely from one to hundreds of cycles depending on cache behavior. A history based prediction scheme performs poorly to capture long latencies caused by cache misses. Therefore, replays due to cache misses will be frequent in such speculative schedulers.

2.6 Energy in Schedulers

Buyuktosunoglu et al. present an adaptive issue queue design by dynamically shutting down and re-enabling blocks of the issue queue [BSB01, BKA03]. By shutting down unused blocks of the issue queue, they are able to proportionately reduce the energy dissipated. Our work follows this trend to scale energy dissipation with the number of active entries in the issue queue.

In conventional issue queue design, the dependents of missed loads consume a substantial amount of power while waiting for loads completion. Gschwind et al. propose a recirculation buffer for misscheduled instructions in addition to the main issue queue [GKA01]. Similarly, Moreshet and Bahar [MB03] propose to use a Main Issue Queue(MIQ) and a Replay Issue Queue(RIQ). Load dependents in main queue are speculatively scheduled assuming cache hit. They will enter

replay/recirculation queue if the load misses. Power is saved by reducing the main queue size relative to a baseline issue queue. As we can see, energy is still consumed when the dependents of missed loads are misspeculated, and when they wait in the replay/recirculation queue. On the other hand, our approach anticipates load misses and then before their dependents can enter the issue queue, they are buffered in low-power FIFO structures.

Ponomarev et al. present a circuit-level low-power issue queue design [PKE03]. In their approach, energy is saved by using comparators that dissipate energy mainly on a tag match, using 0-B encoding of operands to imply the presence of bytes with all zeros and bitline segmentation. This is orthogonal to our work, and our approach can help to reduce the issue queue energy even further.

Karkhanis et al. propose to limit the number of in-flight instruction to save energy [BKA03, KSB02]. In their approach, the fetch engine dynamically throttles so that instructions are not fetched sooner than necessary. This reduces the processing of speculative instructions. However, ILP is sacrificed in this approach as the number in-flight instructions is limited. In our approach, we decouple instruction fetch and instruction issuing by introducing FIFOs buffers – available ILP is still exploited in such a design.

Folegnani and Gonzalez propose to save issue queue power by disabling the wakeup of empty issue queue entries or entries that have already been woken up previously [FG01]. In addition, they propose to dynamically reduce the effective issue queue size by monitoring the utilization of the issue queue. Our baseline model is a more ideal version of their approach. Our approach can help this technique work more effectively by reducing the occupancy of the issue queue, thus providing more opportunity to shut down parts of the queue.

Wilcox et al. [WM] demonstrate that the issue queue logic on the 8-way issue Alpha 21464 was expected to be 23% of the overall power of the core. They also argue that the issue logic could account for 46% of the total power dissipation in future out-of-order processors supporting speculation.

Lebeck et al. [LKL02] explore an alternative means of reducing issue queue occupancy, maintaining a secondary buffer of load dependents that have been misscheduled. However, this design does not use any form of load latency prediction, and therefore will not impact register file energy. They do not explore the energy implications of this design.

Kim and Lipasti explain in detail the problem of misschedulings due to load misses, and several misscheduling recovery mechanisms [KL]. We proposed latency prediction techniques to scale a conventional wakeup-and-select issue queue [LSM04b]. The Alpha 21264 uses a global 4-bit history counter to determine whether a load hits in the cache [Kes99]. However, it is difficult to accurately predict based on global history. Memik et al [MRM03b] propose to predict load/hit miss information during load execution time to reduce scheduling replays. In this dissertation, we propose to predict load access time far ahead at execution stage. In this way, we prevent the dependents of long latency loads from entering the issue queue too early, saving energy both from reduced replays and from more efficient use of the issue queue.

2.7 3D Technologies

While a number of 3D IC fabrication technologies have been proposed [MPP86, PSD99, MS99], we consider the use of wafer bonding [BSK01, BNW04, DCR03] in this study. In this technology, fully processed wafers are bonded together, and

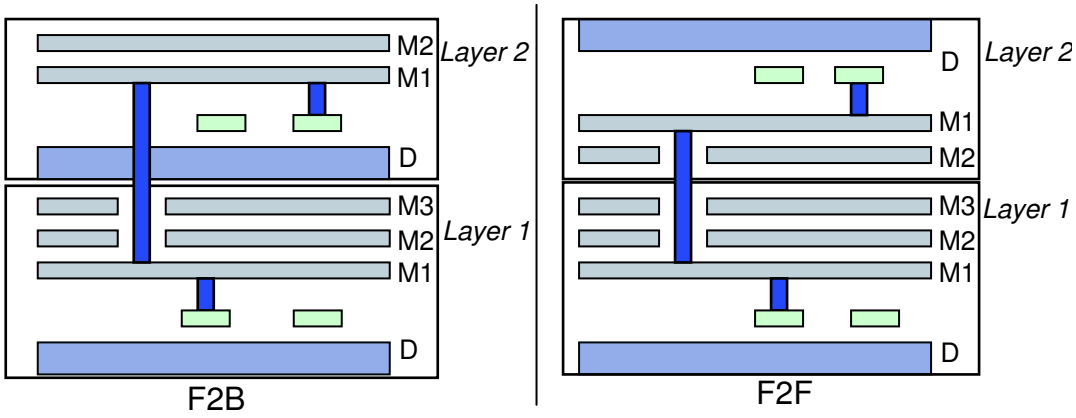

Figure 2.1: Face-to-Back and Face-to-Face integration technologies

devices are fabricated on these wafers. Interlayer vias that connect different layers are etched after metalization and prior to wafer bonding. Two main kinds of wafer bonding strategies have been evaluated in prior work [BNW04, DCR03]: Face-to-Back (F2B) placement and Face-to-Face (F2F) placement (Figure 2.1). Vias in F2B cut through device layers in addition to metal layers. In F2F placement, the top device layer is flipped to face the lower device layer. Metal layers are placed between the facing device layers. Hence, vias cut through metal layers only. However, F2F cannot scale beyond two layers without also employing F2B layers.

2.8 3D Microarchitectural Exploration

MEVA-3D [CJM06] is an automated exploration framework that can explore a 3D design space for an optimal placement of 2D architectural blocks into multiple device layers. MEVA-3D optimizes a cost function that is configured to weigh

latencies of critical microarchitectural loops, temperature, and die area. The critical loop latency is the sum of individual block latencies along the loop and inter-block wire latencies. Critical loop latencies relate to performance (IPC) as in [SC02]. The algorithm returns a floorplan with the best overall performance, temperature and die area for a given target frequency. MEVA-3D leverages SimpleScalar [BA97] to validate its performance estimate. MEVA-3D can also perform automated thermal via insertion to help mitigate areas of high power density. However, MEVA-3D does not currently support the exploration of 3D designs using 3D blocks.

Ozturk et. al. proposed a 3D topology optimization algorithm [OWK06]. The algorithm considers the optimal placement of a few processor cores that are associated with a large number of storage blocks. The algorithm is able to improve performance by placing these cores and blocks in 3D so that the cores are closer to their most frequently accessed storage blocks. However, this algorithm does not consider the placement of actual microarchitectural blocks such as the ALU, issue queue, branch predictor, etc, and does not consider the latency reduction of critical microarchitectural loops. The algorithm is also not able to explore the placement of 3D-designed blocks.

2.9 Delay, Power and Area Modeling

Prior work has provided block models for various architectural structures including caches [WJ96], register files [FJC95, PJS97], and wakeup and select logic [PJS97]. CACTI [WJ96, RJ, SJ01] is an analytical model that provides timing, area, and power results for different cache configurations. CACTI models different levels of associativity, multiporting, sub-banking, and ideally scales to

different feature sizes using 0.80μm cache data. Tsai et al [TXV05] extended CACTI to explore 3D cache designs. However, they only consider folding blocks by wordlines or bitlines, and not by port partitioning. In addition, they do not explore the impact of this 3D design on the overall microarchitecture (i.e. performance, temperature, layout), or the impact of 3D stacking on area in general. Puttaswamyy et al. [KG05] showed the delay benefit and the reduction of power consumption in a stacked cache design by bank-stacking or array-splitting. There has been no prior work that explores partitioning cache ports. Palacharla et al [PJS97] built detailed transistor-level models for critical structures in dynamically scheduled processors, analyzing critical timing paths and the scalability of these structures. However, this study is limited to single layer structures.

CHAPTER 3

Load Latency Prediction

3.1 Motivation

With processor speeds scaling at a rate far greater than that of memory, the latency to memory has grown to hundreds of cycles, and is projected by some to exceed 1000 cycles in the next few years [BS04]. Contemporary microprocessors typically use multiple levels of cache structures to better tolerate these long memory latencies, leveraging the locality of reference exhibited by most applications to reduce the average load latency seen. However, this adds an element of nondeterminism to load latency.

This nondeterminism can cause significant problems for the scheduling window of an out-of-order superscalar processor. Instructions enter this window in-order, and remain until issued. The dependents from long latency loads can clog this expensive window, preventing the exploitation of distant instruction level parallelism (ILP). Palacharla et. al. point out that wakeup and selection in a scheduling window must complete in one cycle to ensure that dependent instructions can be executed back-to-back [PJS97]. And as cycle times continue to scale, the size of the scheduling window will likely be constrained.

One prior study looked at dynamically adapting the scheduling window to long latency loads by shuttling instructions to an auxiliary scheduling window [LKL02].

Another looked at flushing a thread (in a multithreaded environment) after a certain amount of time has passed during a load's execution to prevent clogging the scheduling window [TB01]. Such dynamic adaptation can be difficult because the latency of a load operation is not even known at time of issue - it is only known when the load actually is resolved, which may be too late to allow for dynamic adaptation. These techniques, and others, could all benefit from early knowledge of load latency.

However, predicting load access times far in advance of the execution stream is not trivial. While most instructions have deterministic execution latency, the latencies of memory instructions can significantly vary due to the gradient of latencies in the cache hierarchy. Even worse, cache blocks can be in-flight (i.e. in the process of arriving from some level of the memory hierarchy) and therefore load latencies can range anywhere from the L1 cache hit latency to a full memory access. Loads frequently alias with outstanding load misses due to the larger block sizes used in lower level caches to exploit spatial locality. In addition, loads may hit or miss in the TLB during address translation, adding more nondeterminism in architectures with low-overhead page faults. Latency may also be affected by possible structural hazards or contention in the pipeline or memory buses. Finally, stores may directly forward values to loads if the memory operations alias one another – potentially bypassing the entire memory hierarchy.

In this chapter, we propose a mechanism to predict load instruction latency early in the pipeline. We will present its application in the next two chapters. In Chapter 4, we will demonstrate its application on the scheduling window by sorting instructions into their expected issue order before they enter this window. The end result of this process is that instructions that are not ready to issue spend less time in the scheduling window, potentially allowing instructions that

are ready to issue to enter the window even sooner. In Chapter 5, we demonstrate its application to reduce Tornado effect.

We make the following contributions:

- We present techniques to efficiently predict load latency, including hybrid approaches that arbitrate among multiple latency predictors. Our approach can easily be integrated into the processor pipeline without impacting the critical path.
- We investigate the causes of mispredictions and categorize loads whose latencies are difficult to predict.

The remainder of this chapter is organized as follows. In section 6.3 we describe the experimental methodology for evaluating our design. We then present the design components of the load latency predictor in section 3.3. In section 3.4, we investigate and categorize the causes of load latency mispredictions. Finally, we summarize this chapter in section 3.5.

3.2 Experiment Methodology

We make use of the SimpleScalar 3.0 tool set [ALE02] [BA97] to evaluate our design. We simulate 10 floating-point and 10 integer benchmarks that were randomly selected from the SPEC 2000 benchmarking suite. The applications were compiled with full optimisation on a DEC C V5.9-008 and Compaq C++ V6.2-024 on Digital Unix V4.0. We simulate 100 Million instructions after fast-forwarding an application-specific number of instructions according to Sherwood et al. [SPC01].

Table 5.3 shows the applications we used, including the percent of load in-

Data set	**%load**	**%L1 MR**	**%L2 MR**	**Branch Acc.**	**IPC(IQ32)**
ammp	27.5	8.2	24.5	0.938	0.935
applu	30.2	18.9	16.4	0.917	1.123
apsi	24.4	2.6	1.9	0.907	1.431
art	26.9	40.6	66.3	0.927	0.714
bzip2	28.4	0.8	9.2	0.992	2.683
crafty	30.2	5.5	0.3	0.919	1.02
eon	33.3	2.6	0	0.918	1.252
equake	40.7	18.4	31.3	0.948	0.42
gap	26	1.1	6.1	0.992	1.63
gcc	24.3	12.6	6	0.951	0.741
gzip	20	9.7	0.5	0.921	1.645
galgel	39.6	15.3	9.7	0.952	1.997
lucas	12.5	20.3	33.3	0.993	0.717
mesa	27.1	1.9	7.3	0.957	2.055
mgrid	31.7	15.4	17.7	0.957	1.753
perl	28.6	0.4	5.7	0.99	2.427
twolf	26.8	7.5	14.6	0.9	0.708
vortex	29.8	0.6	2.9	0.99	1.396
vpr	42.6	5	41	0.879	0.71
wupwise	22.6	3.2	31.6	0.96	1.861

Table 3.1: The Statistics of Baseline Benchmarks

structions, the L1 and L2 data cache miss rates, and branch prediction accuracy. The L1 miss rate ("L1 MR") does not include in-flight data.

The IPC shown in Table 5.3 is from the simulation of baseline architecture which models a future generation microprocessor. We simulated our baseline using a 256-entry ROB and a 128-entry LSQ. The issue window is varied from 16 to 128 (16, 24, 32, 64 and 128 entries). We model an 8-way out-of-order superscalar processor. The cache parameters are based on the P4 with an 8 KB, 32-Byte block size, 4-way set associative L1 data cache with 2 cycle latency. The L1 instruction cache is a 16KB, 2-way set associative cache with a 32-byte block size. The unified L2 cache is a 512KB, 64-Byte block size, 4-way set associative cache with a 12-cycle hit latency. The total round trip time to memory is 164 cycles. The processor has 8 integer ALU units, 2-load/store units, 2-FP adders, 2-integer MULT/DIV, and 1-FP MULT/DIV. The latencies are: ALU 1 cycle, Integer MULT 5 cycles, Integer DIV 25 cycles, FP ADD/CMP/CVT 2 cycles, FP MULT 10 cycles, FP DIV 30 cycles, and SQRT 60 cycles. All functional units, except DIV and SQRT units, are fully pipelined allowing a new instruction to initiate execution each cycle. We use a 4K-entry combining predictor to select from a 4K-entry bimodal predictor and an 8K-entry gshare predictor. The minimum branch misprediction penalty is 20 cycles.

3.3 Load Latency Prediction

Our load latency prediction architecture is made up of two components: a Latency History Table (LHT), which predicts latency using a last value predictor, and Cache Latency Propagation (CLP), which makes use of existing cache miss detection techniques [MRM03a]. The LHT can be accessed as early as instruction

fetch as it only requires the instruction PC to make a prediction. The CLP can also be accessed early on in the pipeline, but requires address prediction (indexed via PC).

3.3.1 Latency History Table (LHT)

Value prediction [LWS96] has been used in the past to predict load values. We use the same design to predict load latencies. The LHT predicts whether or not a given load will experience the same latency as the last access of that load. As shown in Figure 3.1, the LHT has 2K entries, is indexed by PC, and has 3-bit confidence counters to avoid less confident predictions. The confidence threshold to predict is four.

If the LHT cannot make a confident prediction, we guess the L1 data cache latency for the load latency. Note that this is a conservative assumption, but our intention is not to degrade performance. If the waiting time of an instruction is predicted to be longer than the actual delay, the instruction will be unnecessarily delayed at the sorting stage (Refer to Section 4.2), which is likely to degrade performance if this instruction is on the critical path of the application. On the other hand, if we underestimate the waiting time, we may miss the opportunity to reduce issue queue clogging, but we will not degrade performance below that of our baseline.

LHT can accurately predict loads that frequently hit in the L1 cache. However as shown later, it is only able to predict 24% of the main memory accesses (i.e. misses in both L1 and L2 data caches, and are not aliased with any in-flight data blocks). Our arbitration policy ensures that the LHT outperforms a scheme that always predicts a cache hit, by capturing L1 misses with predictable latency. The

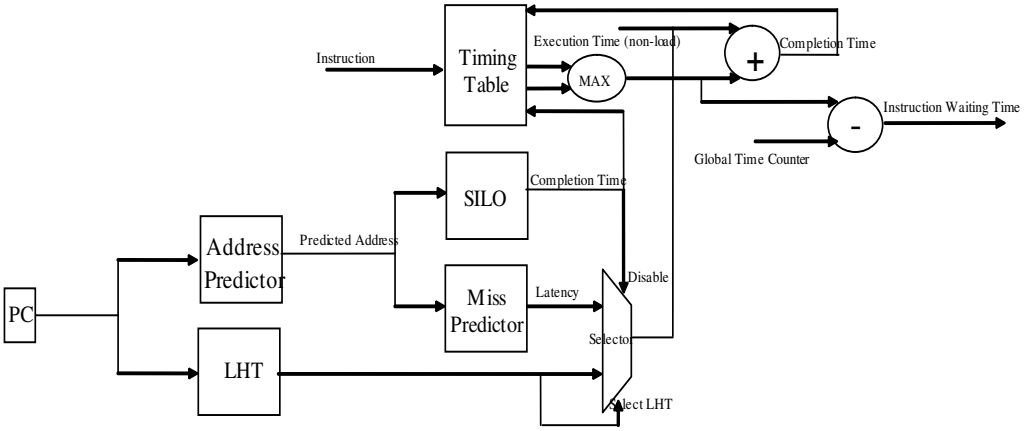

Figure 3.1: The Overall Prediction Structure

LHT is not good at handling aliases to in-flight loads.

3.3.2 Cache Latency Propagation (CLP)

A load hit can be handled like a deterministic instruction when calculating the instruction waiting time. The execution latency of a load that will hit in the cache can simply be set to the L1 cache hit latency. However, a load miss needs special handling because it impacts not only its own dependent instructions, but also any subsequent aliasing loads and their dependent instructions. For instance, if load A misses, and subsequent load B aliases with A, A may hide some of the latency seen by B. This will, in turn, impact B's dependent instructions.

Our Cache Latency Propagation (CLP) scheme identifies cache misses, and then propagates the completion time of cache misses to any aliasing loads via a structure that stores the Status of In-flight Loads (SILO). The CLP approach needs an address predictor, which produces load addresses that are used by a cache miss predictor.

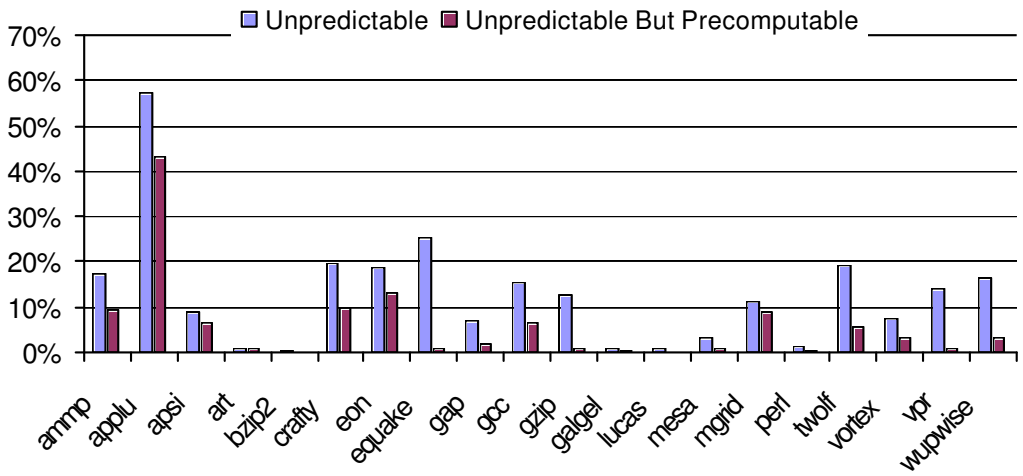

Figure 3.2: Precomputability

3.3.2.1 Address Prediction

Our address predictor makes use of a hybrid address predictor similar to that proposed by Wang and Franklin [WF97]. It consists of a 2K entry stride predictor and a 4K entry Markov predictor. Prediction is guided by 3-bit confidence counters to reduce mispredicted addresses. The confidence threshold is four.

3.3.2.2 Address Precomputation

For many loads, the effective address can be precomputed because the input operand for the address calculation is ready far in advance of the execution of load operations. We can perform precomputation for an unpredictable load address if its address operand is ready at the renaming stage. As shown in Figure 3.2, an average of 13% of the load addresses are not predictable. Out of these, about half of them can be precomputed.

A precomputation approach needs to monitor the status of load address operands, and requires additional register file ports or a shadow register file to access load address operand values. Therefore, we propose precomputation as an *optional* feature, and the precomputation is only performed if the address is not predictable. We will consider results for architectures with and without address precomputation.

3.3.2.3 Cache Miss Predictor

In CLP, we look for a more rigid cache miss predictor as we have additional knowledge, i.e. load addresses that are produced by the address predictor.

Memik et al. [MRM03a] used cache miss prediction to reduce the cache access times and power consumption in a processor with multiple cache levels. However, in their approach, the detection engine was accessed along with the data cache access, which is too late for our approach. We use a similar technique for cache miss detection using a small hardware structure to quickly determine whether or not an access will miss in the cache. It stores information about the block address placed and removed from the caches. We use the hybrid cache miss detector as described in [MRM03a]. We assume a two level cache structure in our architecture, and therefore make use of three detection engines to indicate whether a load access is a miss in the L1 cache, a miss in L2 cache, or a miss in the TLB. The miss detection engine never indicates a miss when the block is in the cache, but it can wrongly indicate a hit when the block is actually not in the cache (i.e. there are almost no false predictions on the detected misses) [MRM03a]. This matches our desire to be conservative in latency estimation, as we will not guess a longer latency if the load is currently in the cache.

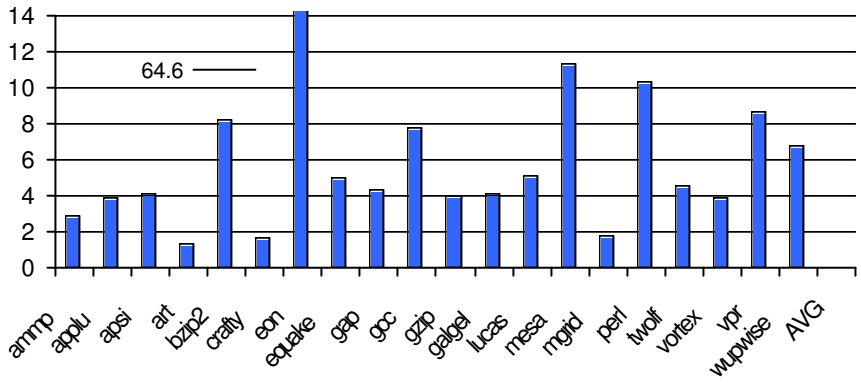

Figure 3.3: The Average Number of Loads that Alias In-Flight Memory Accesses

3.3.2.4 Status of In-flight Load (SILO) Structure

The miss detection engine cannot detect in-flight data accesses. For instance, when the engine claims a block is a miss in the cache, two cases are possible:

1. There is an earlier load operation on the same block that has been renamed but has not reached the cache access stage of the pipeline.

2. The access has already started but the data block has not yet arrived in the cache.

Figure 3.3 indicates the frequency and importance of capturing in-flight loads. We design the Status of In-flight LOads (SILO) structure to capture the initial miss and propagate its completion time to the aliasing loads. Without the SILO, a load instruction that aliases with another in-flight load instruction may be mistaken as a miss.

SILO is a small, cache-like structure addressed by the predicted address of load instructions. Our SILO is similar to the PAT structure presented in [MRM03b].

PAT stores the time remaining for in-flight data blocks to reach the cache. In SILO, on a load miss, we store its completion time and propagate the time to later aliasing loads, if any exists.

Speculative loads that have been renamed but are squashed before execution may pollute the SILO. If such loads record their completion times into the SILO, a subsequent aliasing load will wrongly conclude that an earlier load has accessed the same cache block. We solve this problem by maintaining two versions of the SILO: One is updated at issue time like a PAT [MRM03b], and the other is updated speculatively at prediction time. On a branch misprediction, we use the nonspeculative SILO to recover the speculative SILO. We use the issue time SILO to invalidate erroneous entries.

Often, the L2 cache block size is larger than the L1 cache block size to take better advantage of locality and reduce main memory accesses. For instance, in P4 [HSU01], the L2 cache block size is 64B, twice the L1 block size. On a load miss, not only the L1 block but its adjacent block is brought into the L2 cache. We maintain two separate SILO structures (and two separate recovery structures) to addresses this. One SILO tracks the L1 blocks, and the other tracks their adjacent L1 block. In the latter SILO, the value stored is the time for the block to arrive in the L2 cache. If both SILOs have a valid hit, we defer to the L1 cache's SILO.

Our simulation results indicate that there is little need to track the in-flight status of TLB misses.

We use 8 entries for each SILO structure. The SILO tag length is 25 bits. Each record is 10 bits. Hence, each SILO structure we use is not more than 35 Bytes.

Cases	Dl2 Miss?	Dl1 Miss?	Tlb Miss?	Prediction
Case A	Yes	Don't Care	Don't Care	Mem. Lat.
Case B	No	Don't Care	Yes	TLB M Lat.
Case C	No	Yes	No	L2 Lat
Case D	No	No	No	L1 Lat

Table 3.2: Prediction based on Output of Miss Detection Engine.

Assumption: Mem. Lat. >TLB Miss Lat >L2 Lat. >L1 Lat.

3.3.2.5 Overall Algorithm of CLP

The SILO is probed using the predicted address, If a load hits in the SILO, it skips Step One, and directly performs Step Two by recording the completion time obtained from the SILO into the timing table. In this way, the completion time is propagated to the dependents of the aliasing loads.

The cache miss predictor is accessed in parallel with SILO. SILO is given priority over the cache miss predictor, so that loads aliased with in-flight loads can always receive the propagations from SILO. The cache miss predictor produces a load access time by arbitrating among three levels of miss detection structures as shown in Table 3.2. Note that predictions from the different cache detector may not agree. For instance, the L1 structure claims that there may be a hit, while the L2 structure claims that there is a miss. Because the miss detection structure claims a miss ("yes") only when it is sure, but maybe ("no") when it is unsure, we use the prediction from the L2.

The CLP approach is efficient at capturing loads misses As shown in Figure 3.4, the CLP with an ideal address predictor is able to capture 84% of the main memory accesses. The CLP also accurately captures the L1 cache hits.

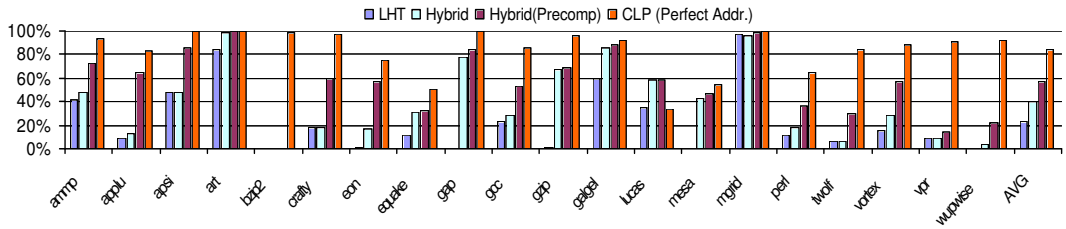

Figure 3.4: The Percentage of Memory Accesses (miss both L1 and L2) That Are Correctly Predicted

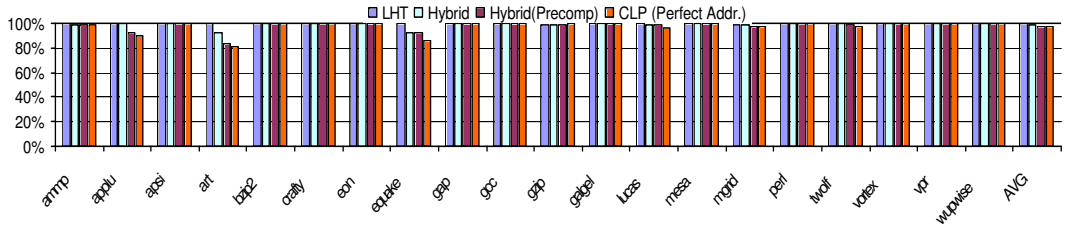

Figure 3.5: The Percentage of L1 Hits That Are Correctly Predicted

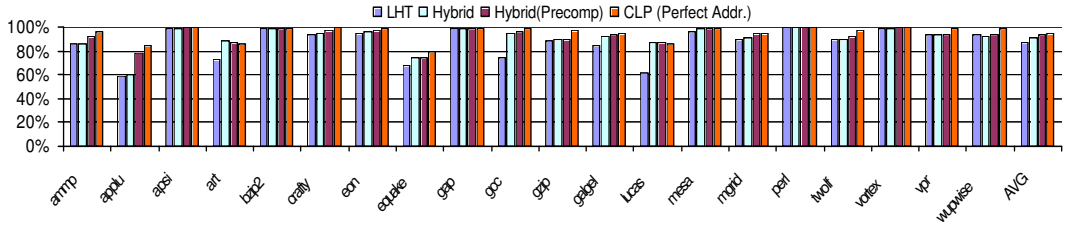

Figure 3.6: The Prediction Accuracy Rate of All Loads

Figure 3.5 shows that CLP captures 97% of the L1 cache hits, which is slightly outperformed by the LHT. Overall, CLP can predict up to 95% of L1 cache hits.

However, CLP is limited by address prediction. Our simulations show that a realistic address predictor accurately covers 68% of addresses on average. We use CLP in a hybrid approach for those loads that are not predictable by LHT, but are address predictable.

3.3.3 Hybrid Approach

In our hybrid design, LHT is given priority when LHT and CLP both predict confidently, as accessing the CLP structure is more expensive in terms of hardware cost. If both techniques fail, the default value of the L1 hit latency is used.

Figure 3.1 shows the overall architecture of the hybrid load latency predictor, which is combined with the structures for computing instruction waiting time. Predictor look-ups start in parallel with the rest of the pipeline as early as the fetch stage. This can hide the latency of the prediction structures, as the prediction starts in fetch and is required during rename. The predictors allow four accesses per cycle. Prior work has demonstrated multi-access techniques to predictors [LY00]. Address predictors are updated speculatively with predicted values when a prediction is made. In the commit stage, confidence counters are updated non-speculatively for the address predictor and LHT. We also update the last latency seen for the LHT and the stride of the address predictors in commit. Cache changes are updated to the cache miss predictors in the write-back stage.

Figure 3.7 shows the distribution of predictions by the three prediction methods, namely LHT, CLP and default (always guess the L1 latency). The loads

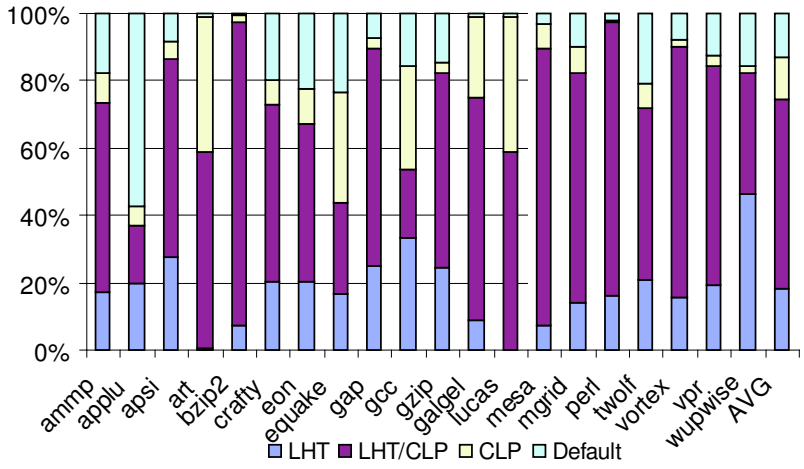

Figure 3.7: The breakup by Prediction Methods in the Hybrid Approach

that are both predictable by LHT and CLP are categorized as "LHT/CLP". The benchmarks that have less LHT coverage but are address-predictable can benefit from the hybrid approach. As shown in Figure 3.4, the Hybrid approach is able to capture 40% of the main memory accesses. The hybrid approach predicts 87% of the latencies correctly if all level 1 cache misses are considered. The hybrid approach predicts 99% of the cache hits correctly.

3.4 Categorizing Mispredictions

Despite the accuracy of load latency prediction, there are still some load latencies that cannot be determined a priori. Table 3.3 shows the breakdown of mispredictions by their cause for the hybrid predictor. They are categorized as follows:

1. *Overestimated by SILO*: Our SILO predicts a load latency that is longer than the actual latency. SILO predictions occur when a load aliases another

Bench marks	Over-est. by SILO			Under est. by SILO	Evic tion	TLB	BUS Conten tion	Mispre diction in LAT	Unpre dict Addr	Others
	Haz	Dep	Mis Addr							
ammp	5.53	0.14	1.05	0.55	6.22	0.02	0.00	18.66	64.90	2.95
applu	2.51	0.00	9.32	4.92	27.39	0.00	0.00	7.56	41.65	6.65
apsi	0.44	0.02	4.46	0.39	7.63	0.00	0.00	35.98	40.04	11.06
art	0.07	0.04	4.62	0.31	82.95	0.01	0.07	1.97	0.20	9.77
bzip2	0.00	0.00	0.00	0.00	4.19	0.00	0.00	37.33	57.67	0.81
crafty	0.09	0.00	0.52	0.02	1.17	0.00	0.00	18.15	78.11	1.94
eon	0.26	0.00	1.59	0.02	0.47	0.00	0.00	19.51	74.59	3.58
equake	5.20	0.07	1.91	3.19	8.90	0.00	0.00	2.54	76.91	1.27
gap	0.94	0.00	1.15	0.04	7.63	0.01	0.00	26.97	53.36	9.92
gcc	0.10	0.00	0.67	0.02	1.65	0.00	0.00	16.24	79.46	1.87
gzip	0.01	0.00	0.05	0.00	0.42	0.00	0.00	24.76	74.63	0.13
galgel	0.59	0.02	4.51	1.41	37.25	0.04	0.04	27.69	20.99	7.49
lucas	0.10	0.00	0.15	5.06	69.50	1.85	0.00	2.21	7.72	13.42
mesa	9.54	0.00	0.92	0.15	1.93	0.00	0.00	27.70	55.50	4.26
mgrid	1.30	0.00	0.11	0.20	17.59	0.02	0.00	39.60	9.08	32.13
perl	0.52	0.00	0.29	0.04	3.14	0.02	0.00	42.17	49.75	4.10
twolf	0.06	0.00	0.42	0.01	0.84	0.00	0.00	12.88	84.94	0.85
vortex	0.64	0.00	1.90	0.01	1.59	0.11	0.00	23.16	50.12	22.56
vpr	0.01	0.00	0.04	0.01	0.59	0.22	0.00	23.47	75.01	0.66
wupwise	0.05	0.00	0.29	0.01	2.15	0.03	0.00	26.54	69.44	1.53
Avg	1.40	0.01	1.70	0.82	14.16	0.12	0.01	21.75	53.20	6.85

Table 3.3: Breakup of mispredictions by categories in percentages.

in-flight memory operation. If load B aliases load A for example, load B can be overestimated by our SILO for the following reasons:

(a) *Structural Hazards*: Structural hazards can impact prediction accuracy. For example, if load B cannot enter the issue window because the window is full, the load will see a shorter latency than expected because the aliased block is being brought in during the structural hazard. An instruction dependent on B might expect a longer latency than is actually seen at B's issue. `Ammp`, `equake` and `mesa` have a noticable (5% to 10%) mispredictions that fall into this category. These applications have a large number of in-flight instructions and the issue window is completely filled most of the time. The issue queue filling is the predominant cause of mispredictions in this category. Contention for functional units or memory ports also contributes in a few cases.

(b) *Data Dependencies*: The load's address operand takes more time than expected to be produced. We observe that most benchmarks do not suffer much from this.

(c) *Mispredicted Address*: If the load address is not correctly predicted, the load may mistakenly be thought to alias with an unrelated load.

2. *Underestimatd by SILO*: This happens when the original load, which produces the in-flight data block, experiences extra issue latency. Similarly, the extra issue latency can be due to the above mentioned causes. However, further categorization is not performed as underestimations are less frequent.

3. *Eviction*: Prediction is performed during rename, but the actual execution of loads may be many cycles after renaming. Hence, a cache block present in

the cache at the time of prediction may have been evicted by intermediary loads by the time of the load's execution. `Art` is a typical example in this category. As shown in Table 5.3, `art` has frequent cache misses. In `art`, the average latency from renaming to load exeuction is also long. Hence, a cache block present at prediction time may be evicted by other cache misses before the load's execution. `Applu`, `galgel` and `lucas` also have a large percentage of mispredictions due to eviction. Note that evictions result in underestimation of latency – which means that we are simply unable to exploit as much distant ILP as we might have been able to exploit. It does not lengthen the perceived latency of a load as in the case of an overestimation.

4. *Bus Contention*: Unexpected extra delay in accessing the memory bus may cause underestimations in load latencies. Only `art` and `galgel` observe some mispredictions in this catogery. This is due to a relatively high pressure of bus contention as a result of frequent cache misses in these two benchmarks.

5. *TLB*: This is due to the mispredictions from the TLB MNM. As shown in the table, mispredictions in this category are rare.

6. *Mispredicted by LAT*: The overall prediction scheme attempts LAT first. Hence, mispredictions in LAT can be a major source of overall mispredictions. This is especially the case in applications like `bzip2`, `perl` and `apsi`. These applications have very few cache misses as shown in Table 5.3. In these applications, the predictions are mostly predicted by LAT as shown in Figure 3.7. Correspondingly, these applications see more than one third of their overall misspredictions from LAT mispredictions.

7. *Unpredictable Address*: If LAT cannot predict confidently, and the address is not predictable either, a default L1 latency is assumed. Mispredictions due to this make up a large percentage of the overall mispredictions for many applications.

8. *Others*: other causes such as misprediction from MNMs, cold starts in the predictors etc.

Overall, as shown in Table 3.3, mispredictions due to *Unpredictable Address* makes up the largest portion – an average of 53%. This suggests that our prediction scheme could be significantly improved with better address prediction. Other main sources of mispredictions come from *Mispredicted by LAT* and *Eviction*, at an average of 22% and 14% respectively.

3.5 Summary

In this chapter, we have develped novel schemes to predict load execution time accurately, based on data reference history. Combined with waiting time computation, we are able to estimate the waiting time of instructions accurately. The accurate prediction on instruction waiting time enables many potential applications.

CHAPTER 4

Scaling Issue Queue Using Instruction Sorting

4.1 Introduction

In this chapter, we propose a dynamic instruction sorting mechanism that provides more ILP without increasing the size of the issue window. The proposed sorting mechanism relies on load latency prediction technique presented in the earlier chapter. With load latency prediction, our approach can accurately predict the waiting time that an instruction will endure before its operands are ready for execution. Instructions then enter the sorting structure, which consists of a number of differently sized FIFO queues. Instructions with longer waiting times (i.e. "slow" instructions) enter a FIFO queue of longer length, which will delay the instruction from entering the issue queue. Instructions with shorter waiting times (i.e. instructions with distant ILP, or "fast" instructions) enter a FIFO queue with a shorter length, and are delivered to the issue queue with less, or no delay. All instructions are placed into a final FIFO queue, the Pren-issue Buffer, which then feeds the issue window in-order. Instructions with different latencies can enter the Pre-issue Buffer out-of-order with respect to one another. This effectively prevents "slow" instructions from clogging the issue window when there is available distant ILP in the application, without consuming available issue bandwidth.

The load latency predictor plays an important role in guiding the sorting process. A large fraction of the executed instructions directly or indirectly depend on load operations, particularly in applications with frequent register spills and/or large data structures.

We make the following contributions:

- We develop a simple sorting structure with a number of FIFO queues to enable instructions to enter the issue queue out-of-order. The queues delay instructions with longer waiting times and allow instructions with shorter waiting times to pass through quickly. The pre-issue buffer of this sorting structure provides a scalable approach to decoupling resource allocation and register renaming from instruction issue.

- We demonstrate the improvement achievable through instruction sorting that is assisted by latency prediction compared to increasing the issue queue size. By keeping the issue queue size smaller, and by performing latency prediction off the critical path, our approach provides a more scalable improvement in ILP.

- We demonstrate an application of load latency prediction. We show the load latency prediction technique can assist effective instruction sorting.

The remainder of this chapter is organized as follows. In section 4.2, we describe the mechanism of the instructions sorting engine. The performance results from our simulations are presented in section 4.3, which is then followed by the summary.

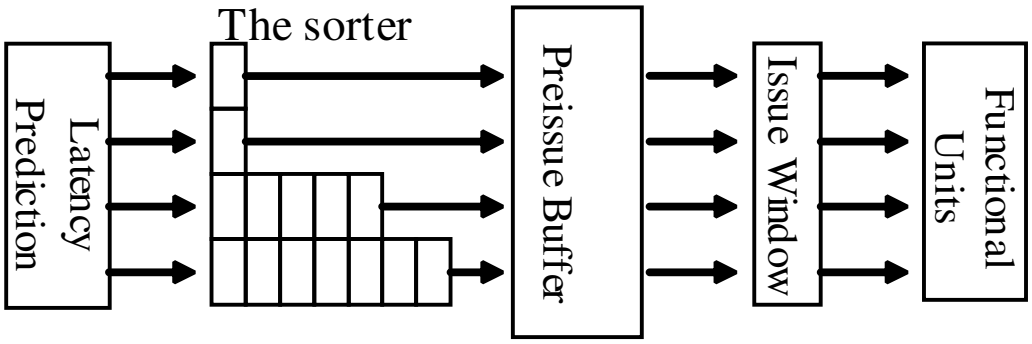

Figure 4.1: the Overall Scheduling Architecture

4.2 Instruction Sorting

Figure 4.1 illustrates the overall architecture to optimize instruction scheduling by utilizing load latency prediction. It consists of three major components: a Latency Prediction component, which estimates the waiting time of instructions, a sorting structure, which consists of a few FIFO sorting queues, and a Pre-issue Buffer (PB), where instructions are buffered before entering the issue queue. The PB provides a temporary storage space for the sorted instructions, but does not make use of wakeup and select logic, as in the case of the less scalable issue queue. Instructions are sequentially fed from the PB into the issue queue every cycle when space is available.

Throughout this paper we use the following terms: instruction waiting time and execution latency. The instruction waiting time refers to the number of cycles that must elapse before an instruction's operands are ready for execution. The execution latency of an instruction refers to the time it takes for the functional unit to execute the instruction. It is referred to as load access time in the case of load operations.

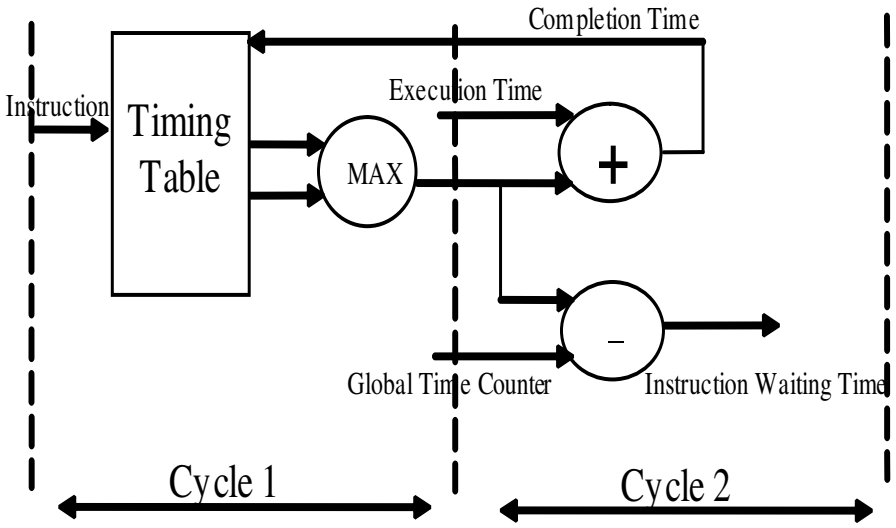

Figure 4.2: Computation of Deterministic Waiting Time

4.2.1 Instruction Waiting Times

Figure 4.2 illustrates the architecture we use to compute the waiting time of deterministic instructions. Similar to the instruction pre-scheduler in Cyclone [EHA03], our design incorporates a timing table indexed by logical register names, which stores the time when logical register value is going to be produced. It allows four accesses in a row. The computation involves two steps:

Step 1, MAX Computation: Each instruction obtains its ready time by accessing a timing table indexed with its input operands, and take the maximum completion times.

Step 2, Waiting Time and Completion Time Computation: result from Step.1 is added to the instruction execution to get the ready time of the instruction's destination register. This result is then written back to the timing table. Completion time is calculated by comparing the ready time with a global time counter.

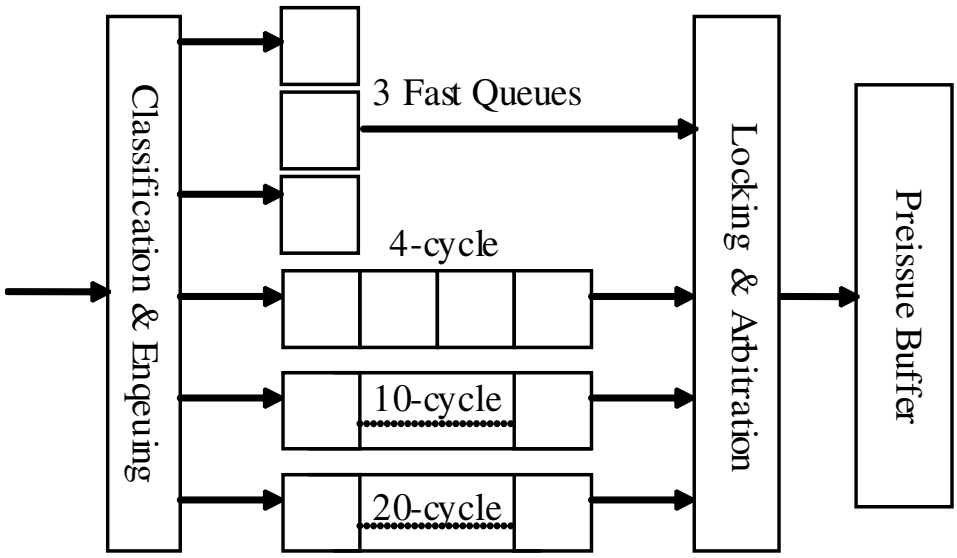

Figure 4.3: A Sample Configuration of the Sorting Engine

It is possible to have dependencies among instructions that are renamed in the same cycle. As in [EHA03] we limit the dependency chain that can be handled by the timing table to two cascaded MAX computations. If a dependency-chain is longer than two, we force the computation to the next cycle, effectively stalling the front-end. Our simulations (in agreement with [EHA03] revealed that such stalls cause very little performance degradation, because longer dependency chains in the same cycle are not common in the SPEC2000 suite.

4.2.2 Instruction Sorting Engine

Once its waiting time is predicted, an instruction is classified and placed into one of the FIFO queues in the sorting engine. The primary function of the FIFO queues is to hold the instructions until their waiting time has elapsed. Instructions with very short waiting times are placed into fast queues and visa

versa. Figure 4.3 shows a simple configuration where there are 3 fast queues, which let instructions go immediately in the next cycle, and 3 slower queues, with hold instructions for 4, 10 and 20 cycles respectively. Instructions progress one slot per cycle, then are ultimately released into the Pre-issue Buffer (PB).

During classification, we round down waiting time to the closest latency class. For example, an instruction with waiting time of 12 is rounded down to the 10-slot queue. This ensures that instructions are not held for more than their waiting time. Hence, our sorting engine will not affect instructions on the critical path. Instructions are placed in a round-robin fashion in the same class. The front end of the processor is stalled if an instruction cannot find an available queue.

It is possible that an instruction reaches the queue head while its parent instruction is still inside FIFOs due to a waiting time misprediction. If we let the child instruction leave for the PB first, deadlock can occur. If the issue window is filled with dependents of instructions in the scheduling queue, forward progress will become impossible.

We address this problem using a small structure called Locking Table, which is indexed by physical registers. An instruction sets the "lock" flag for its destination physical register when it is enqueued, and resets the flag when it is dequeued. A subsequent instruction that consumes the register will check the locking table for a "lock" flag when it reaches the head of the FIFO. An instruction is released into PB if no "lock" is present. We limit the number of sorting queues to 8, to keep the accesses to the locking table less than 16 at each cycle (two accesses per instruction). As described above, our locking table is a very simple structure that returns a single bit for each register index, and can be easily replicated to reduce port costs.

Old Seq	Instruction	RT	EL	WT	New Seq
1	Ldt f25,-320(r9)	0	12	1	1
2	Addq r3,r24,r3	0	1	10	7
3	Ldt f18,0(r3)	0	12	12	8
4	Mult f25,f18,f18	1	10	23	9
5	Addt f0,f18,f0	1	2	35	10
6	Stt f0, 0(r25)	2	0	36	11
7	Ldq r3,-32(r27)	3	6	1	2
8	Ldt f30,-256(r9)	3	0	1	3
9	Addq r3,r24,r3	3	1	7	5
10	Ldt f16,0(r3)	3	164	9	6
11	Mult f30,f16,f16	4	10	172	12
12	Addt f0,f16,f0	4	2	186	13
13	Stt f0,0(r25)	5	0	187	14
14	Ldq r3,-24(r27)	6	12	1	4

Table 4.1: Effectiveness of the Sorting Engine.

Symbols Used: RT(Renaming Time) – relative time when instruction is renamed. EL (Execution Latency) – instruction execution latency. WT(Waiting Time) – instruction waiting time. **Dependency Chains:** 2–>3–>4–>5–>6, 1–>4–>5–>6, 7–>9–>10–>11–>12–>13, 8–>11–>12–>13, 14 is independent.

Table 4.1 shows the sequence of instructions before and after sorting from one of our experiments. The sorting queue configuration has 3 0-slot queues, 2 5-slot queues, 1 10-slot queue, 1 20-slot queue, and 1 150-slot queue. The 14 instructions above are renamed in 6 consecutive cycles as shown in the third column. There are four chains of instruction dependencies: a). 2,3,4,5 and 6. b). 1,4,5 and 6. c). 7,9,10,11,12 and 13. d) 8,11,12 and 13, e) 14. From the predicted values, we notice instructions 1, 7, 8 and 14 have their operands ready sooner than any other instructions. Our sorting engine successfully places these instructions ahead of others. Instruction 9 and 10 observe waiting times of 7 and 9 cycles respectively. As mentioned, their waiting times will both be rounded down to the nearest sorting class, in this case 5 cycles, and they are both placed into a 5-slot queue. Following the same analysis, instruction 2,3,4,5,6 are placed into a 10-slot queue. Hence, after sorting, instructions 9 and 10 are in front of 2,3,4,5 and 6. Instructions 11,12 and 13 observe large waiting times as a result of memory access. They are placed into the 150-slot queue, and hence, they come out last. Overall, the sorting engine effectively places the "slow" instructions after the "fast" instructions.

4.3 Experiments and Results

One application of load latency prediction is to enable instruction sorting based on predicted waiting times. We demonstrate the benefits of load latency prediction in this section, exploring how a latency-based sorting mechanism can effectively scale the instruction scheduling window.

Figure 4.4 demonstrates the performances of issue window sizes 16, 24, 32, 64, 128, and 256 in a machine with the baseline configuration. The IPC increases

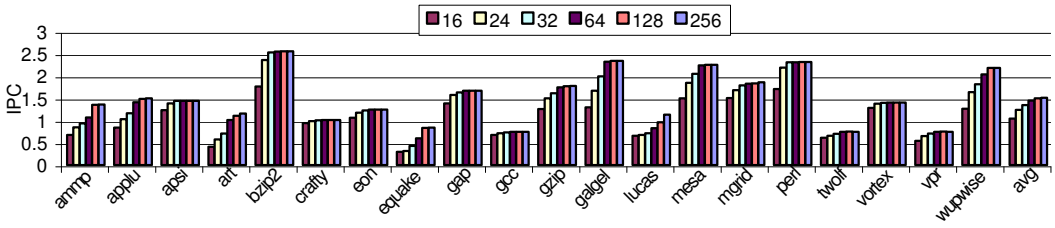

Figure 4.4: The Potential of Scaling the Issue Window

substantially as the issue window size increases. From a window size of 16 to a window size of 256, the average improvement in IPC is 43%. A larger window size allows the architecture to exploit distant ILP, even in the face of instructions that wait for long periods of time in the scheduling window. A smaller issue window can easily clog when too many instructions with long waiting times occupy the window – limiting the amount of ILP that can be exploited. If the instruction waiting times are known in advance, then we apply the FIFO-based sorting mechanism to prevent instructions with long waiting time from entering the issue window prematurely, and allow the distant independent instructions to enter the issue window earlier. In this way, we can use a small issue window size to attain similar or improved performance compared to a much larger window size.

The ability to use a smaller issue window with performance comparable to a larger issue window addresses several critical issues in contemporary microprocessor design. The issue window employs a fully associative structure that can potentially wakeup and select new instructions for execution every cycle from any slot. As a result, in future deeply pipelined microprocessors, the issue window could limit the achievable clock cycle time [PJS97]. Due to its fully associative nature, the issue window is also a major contributor to the overall power

consumption of the chip, and can often be a hot spot [BAS02, BKA03] on the core. In [FG01], it is estimated that instruction issue queue logic is responsible for around 25% of the total power consumption on average. Wilcox et al. [WM] showed that the issue logic could account for 46% of the total power dissipation in future out-of-order processors that support speculation. The proposed techniques effectively address the timing and power issues of the issue window by improving IPC with a relatively smaller issue window size.

4.3.1 Sorting Queue Selection

We profiled all 26 SPEC2000 benchmarks to obtain the histograms of instruction waiting time. Four of these are presented in Figure 4.5. Each graph is zoomed in its right-upper corner to reflect the distributions in lower range latencies. There are wide variations among the histograms of all benchmarks. We choose a simple sorting configuration to balance the distribution of latencies. The histograms indicate that a large fraction of instructions have waiting times of more than 150 cycles, therefore we simulated a slow queue of 150 slots to buffer these instructions. Similarly, we need a few fast queues to expedite the instructions with very short waiting time, and a few intermediate queues with 5, 10, and 20 slots. We need fewer sub-queues for the slow FIFOs, and more for the fast FIFOs, as there is more buffering capacity in a slow FIFO. In this paper, our results are based on a configuration of three 0-slot queues, two 5-slot queues, one 10-slot queue, one 20-slot queue, and one 150-slot queue.

Figure 4.6 shows the relative performance of other queue configurations. The first three configurations have 4 classes of queues, namely 0-slot, 8-slot, 20-slot and 150-slot. The last four configurations have 5 classes, namely 0-slot, 5-slot, 10-slot, 20-slot and 150-slot. Each configuration is labeled with the number of

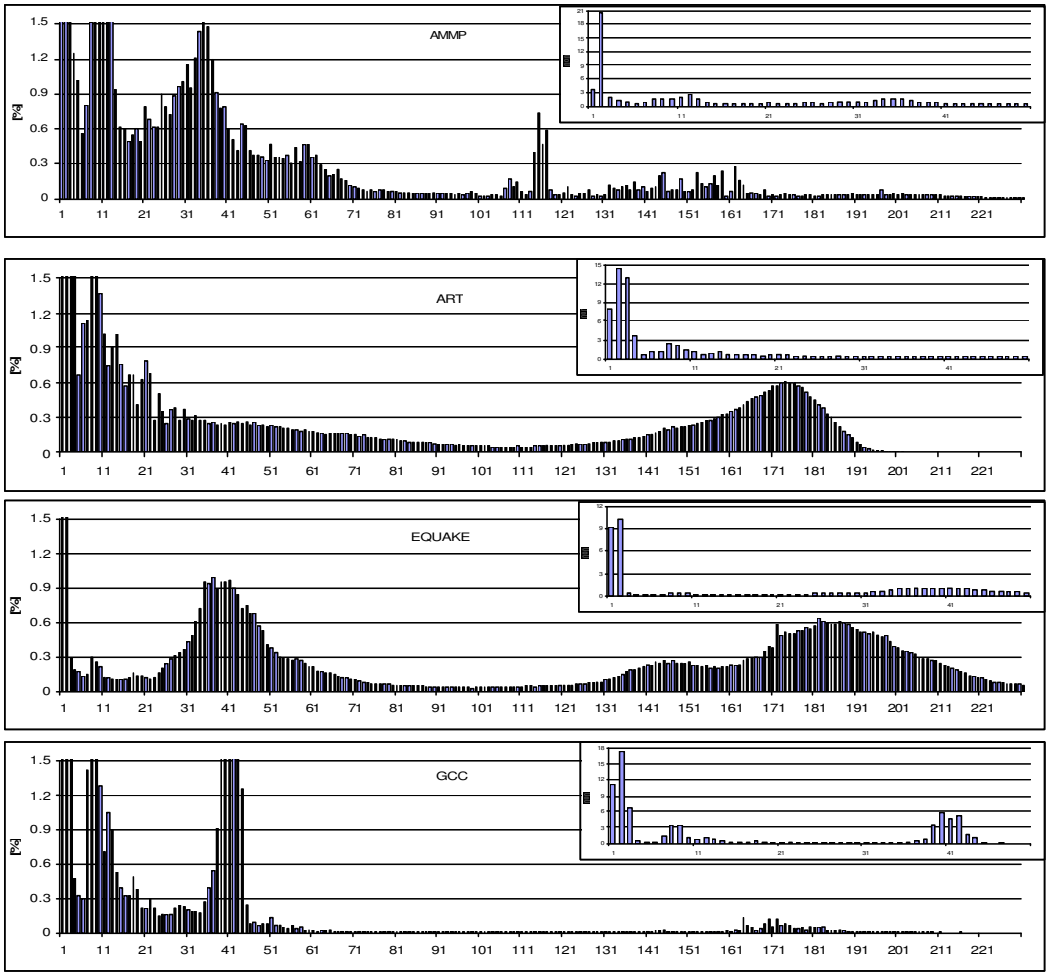

Figure 4.5: The Histogram of Predicted Waiting Time for Selected Benchmarks

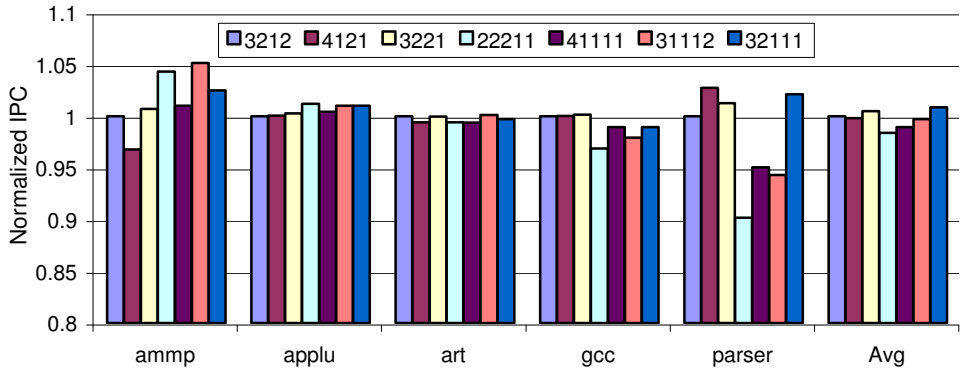

Figure 4.6: The relative performance of several sorting queue configurations

queues of each class that are present. For example, in the case of “3212”, there are 3 0-slot queues, 2 8-slot queues, 1 20-slot queue and 2 150-slot queues. From left to right, the 7 bars represent “3212”, ”4121”, ”3221”, ”22211”, ”41111”, ”31112”, ”32111” respectively. In general, the number of queues in each class should match to the histogram distribution. As the capacity of short queues is small and because instructions with short latencies are more frequent, more queues need to be dedicated to the short queues. On the other hand, long queues have large capacity, and the long latency instructions are less sensitive to the extra delay caused by enqueuing conflicts, thus fewer queues are needed for the long queues. The 7 configurations mostly follow these guidelines. Hence, the relative performance is quite similar from one to another. However, we still observe subtle differences, e.g. “31112” vs “32111”. As the 150-slot queue has a large capacity, and an instruction classified into this queue can tolerate a few cycles even if there is a conflict, one queue is enough. Therefore, we observe slight improvements from “31112” to “32111”, where the two 150-slot queues are reduced to one, and the 5-slot queues are increased to two.

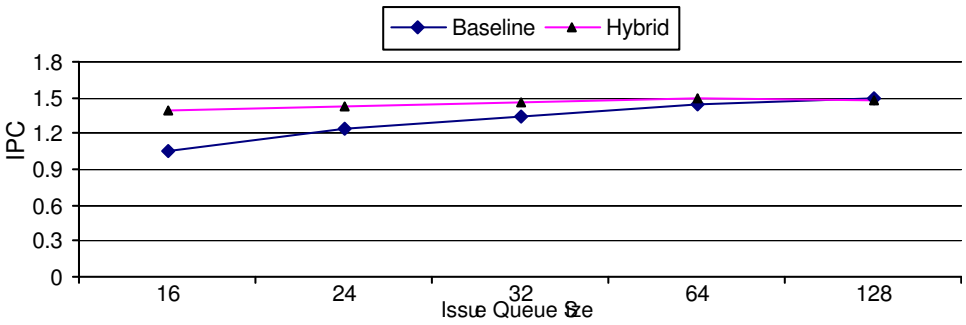

Figure 4.7: The Scaling Effect of Our Approach

4.3.2 Issue Window Scaling Performance

Figure 4.7 shows the scaling effect by comparing the average IPC performance using instruction sorting with baseline configurations with issue windows of 16, 24, 32, 64 and 128 entries. The instruction sorting engine is guided by latency prediction using the Hybrid approach. We use the parameters described in prior sections for the hardware components. All prediction structures are accessed off the critical path, during instruction fetch. We lengthen the pipeline by two additional cycles to account for enqueuing and dequeuing delays.

We observe that a 16-entry issue window using our approach is able perform slightly better than a 32-entry issue window with a baseline configuration, and 24-entry issue window using our approach performs close to that of 64-entry baseline. This is because the sorting engine is able to prevent long waiting time instructions from entering the issue window prematurely, and can accelerate the entry of more distant instructions with shorter waiting times into the issue window. Effectively this allows a small issue window to exploit the same amount of ILP as a large issue window with the baseline configurations.

If our approach is compared with a baseline architecture of the same size issue

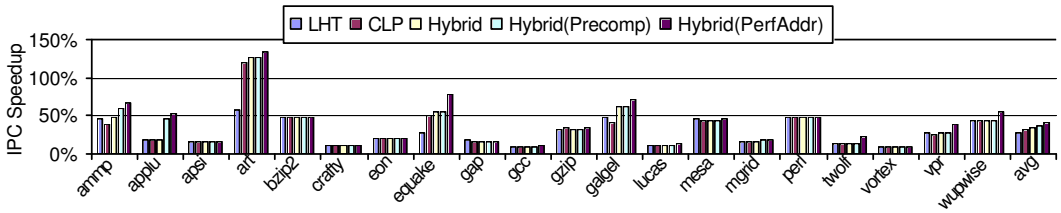

Figure 4.8: The IPC Speedup with 16-entry Issue Queue over a Baseline Configuration

window, then at 16, 24, 32, and 64 entries, we observe a speedup of 31%, 14%, 7% and 1% respectively. The amount of speedup decreases with increased issue window size because the baseline machine is able to extract most of the distant ILP using a large issue window. Our sorting engine can only help in cases where the ILP is further ahead in the fetch stream than can be included in the issue window. As the instruction window increases, such long distance ILP becomes rare. We even observe a slight 1% degradation at an issue window size of 128. This is because the performance we gain from distant ILP is effectively canceled out by the penalties due to additional pipe-stages in latency prediction, sorting, and buffering in our approach.

4.3.3 IPC Speedup over Baseline

Figure 4.8 shows the speedup for a 16-entry issue queue over a baseline architecture with the same issue queue size, but without the extra branch misprediction penalty. The sorting engine guided by LHT shows an average speedup of 27%. The sorting engine guided with CLP shows an average speedup of 32%. The Hybrid approach has an average of 33% speedup. The improvement from CLP to Hybrid is small because most of the load latencies that can be predicted by

LHT can also be predicted by CLP. However, our hybrid scheme still incorporates LHT due to its simplicity and low prediction cost. A load that can be accurately predicted by LHT does not need to access CLP. As mentioned, the CLP suffers from address misprediction. The hybrid approach can perform even better if the CLP has better address prediction. As we can see from the figure, the speedup is increased to 35% if pre-computation is added, and 41% if an ideal address predictor is used.

The best speedup is seen by `art`, which achieves 56%, 120%, 127%, 128% and 135% speedup respectively for a LHT, Hybrid, Hybrid with Precomputation and Hybrid with a perfect address predictor. As shown in Table 1, `art` has a large amount of cache misses. This is also confirmed in Figure 4.5. The dependent instructions of these misses need to wait a substantial amount of time. In our approach, the cache misses are detected, and their dependents placed into the "slow queue", giving way to the "fast" instructions to the issue window. As the Hybrid predictor can more accurately detect misses, additional IPC speedup is observed. Further speedup is observed for our ideal address predictor as most of the cache misses are predicted accurately.

Some benchmarks achieve less speedup. For example, `gcc` and `crafty` observe only 5-9% speedup. Table 1 shows that these two benchmarks have less cache misses than other benchmarks. Furthermore, Figure 4.5 confirms that `gcc` has only a few instructions that have a very long waiting time. While `gcc` has a large number of instructions that have shorter waiting times, our sorting engine is not able to differentiate these instructions due to our selection of queue sizes. As a consequence, the speedup from these benchmarks is small.

Benchmarks that have higher misprediction rate tend to have less speedup. As shown in Figure 3.6 and Figure 3.4, lucas has a poor overall prediction rate,

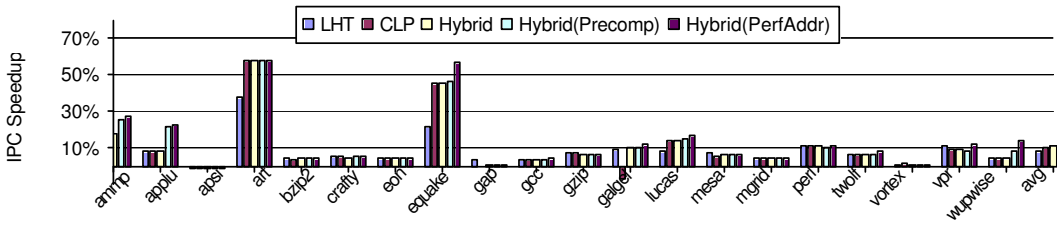

Figure 4.9: The IPC Speedup with 32-entry Issue Queue over a Baseline Configuration

and a poor rate at identifying cache misses. Thus, lucas observes a speedup of only 10%.

Figure 4.9 shows the speedup of our approach over a baseline architecture with a 32-entry issue queue. The average speedup is 11% with a Hybrid approach, and 15% with a Hybrid approach using a perfect address predictor. As we can see, the memory intensive applications, such as `ammp, art, equake` still observe significant speedup while others observe less speedup. This is because there are more instructions with long waiting times in the memory intensive applications. Our approach effectively prevents these instructions from entering issue queue too earlier than necessary, thus allowing more distant ILP to be extracted. The benchmarks `apsi` and `gap` almost observe no speedup – in these applications, the distant ILP that not extractable by a 32-entry issue queue is rare. `Galgel` observes over 7% speedup with other prediction methods except CLP – when CLP is used, a slight degradation is observed. We found that this is because some cache hits that can be easily predicted by LHT are mispredicted as cache misses by CLP due to a wrongly predicted address.

4.4 Summary

In this chapter, we look into scaling issue queue using instruction level techniques. We propose an instruction sorting engine guided by proactive waiting time prediction. We have combined our schemes to predict load access time with waiting time computation to estimate the waiting time of instructions accurately. Guided by the estimated waiting time, our sorting engine can efficiently sort "fast" instructions ahead of "slow instructions". Simulation results show that our approach is able to exploit significantly more application ILP than a comparably sized issue window.

CHAPTER 5

Investigation of Tornado Effects

As future technologies push towards higher clock rates, traditional scheduling techniques that are based on wake-up and select from an instruction window fail to scale due to their circuit complexities. Speculative instruction schedulers can significantly reduce logic on the critical scheduling path, but can suffer from instruction misscheduling that can result in wasted issue opportunities.

Misscheduled instructions can spawn other misscheduled instructions, only to be replayed over again and again until correctly scheduled. These "tornadoes" in the speculative scheduler are characterized by extremely low useful scheduling throughput and a high volume of wasted issue opportunities. The impact of tornadoes becomes even more severe when using Simultaneous Multithreading. Misschedulings from one thread can occupy a significant portion of the processor issue bandwidth, effectively starving other threads.

In this chapter, we propose Zephyr, an architecture that inhibits the formation of tornadoes. Zephyr makes use of existing load latency prediction techniques as well as coarse-grain FIFO queues to buffer instructions before entering scheduling queues. On average, we observe a 23% improvement in IPC performance, 60% reduction in hazards, 41% reduction in occupancy, and 48% reduction in the number of replays compared with a baseline scheduler. Together, these allow Zephyr to reduce energy by 13% on average, at an area cost of only 4%.

5.1 Motivation

The performance of an out-of-order superscalar processor relies on the discovery and exploitation of instruction-level parallelism (ILP) and/or thread-level parallelism (TLP). However, the amount of ILP and TLP that a processor can extract is constrained by the design of the instruction scheduler and the size of the issue window. The instruction scheduler and issue window may prove difficult to scale to future technology goals due to the impact of wire latency. Circuit-level studies of dynamic scheduler logic have shown that broadcast logic dominates performance and power [PJS97, EA02], which complicates the scaling of the issue queue.

A number of prior studies [EHA03, HVI04, KL04, HSU01] have examined instruction schedulers that eschew the need for complex wakeup and selection logic and are able to hide the latency of the schedule-to-execute window through speculative scheduling. One example of this is Cyclone [EHA03], which relies on a simple mechanism to predict the expected issue time of each instruction, and then delays the issue of the instruction based on this prediction via scheduling queues. However, misschedulings can occur for the dependents of loads that miss in the first level data cache. In addition, structural hazards in the switchback paths, also known as switchback conflicts/hazards, happen when an instruction wishes to cross from the replay queue to the main queue but cannot do so if that slot is already occupied. These hazards subsequently cause misschedulings of its descendants.

Prior work demonstrates that a large fraction of instructions directly or indirectly depend on load operations [LSM04b]. If a load misses, its dependents may be replayed many times before the load completes. Replayed instructions are

likely to prevent independent instructions from moving through the switchback queues, further contributing to structural hazards. Hazards are likely to increase in processors with Simultaneous Multithreading (SMT) [TEL95, TEE96], where overall switchback queue utilization increases. SMT also results in a greater number of loads and more contention in shared cache resources, which can lead to more replays. The positive feedback loop between replays and structural hazards can degrade performance dramatically for an SMT processor, as we will demonstrate in Sections 5.3 and 5.4. The feedback loop eventually results in many instructions requiring replay, circulating around the Cyclone queues many times before correctly scheduling. This has been called the *Tornado Effect* [Car04].

One solution might be to apply simple techniques such as increasing the replay interval, limiting the number of instructions in the scheduler, or flushing threads on a cache miss [TB01] to prevent tornadoes. However, these techniques inevitably decrease the amount of ILP, and as our experiments show, degrade performance.

In Chapter 3, we developed techniques to predict the execution time of load instructions in the early stages of the pipeline in an effort to scale the size of the instruction window for a conventional instruction scheduler. We use simple FIFOs with different buffer lengths that buffer instructions based on their predicted execution time to prevent instructions from entering the issue window before their operands are ready – effectively providing out-of-order entry into the issue window. However, these technique does not include any dynamic adaptation to mispredicted load latency, and dependents of a misscheduled load can still clog the issue queue and degrade performance.

A natural solution to these challenges might be the addition of load latency prediction techniques as presented in Chapter 3 to the Cyclone scheduler. Such

a solution should decrease the number of hazards and replays, thereby improving scheduler performance. However, our experimental results demonstrate that this naive combination creates even more structural hazards, and degrades processor performance. The additional structural hazards come from the increased utilization of the scheduling queues. We will explore this impact on the Tornado Effect in more depth in later sections.

As an alternative, we propose Zephyr, an architecture that effectively prevents the formation of tornadoes. Zephyr buffers instructions using coarse-grain FIFO queues. Instructions are released into the scheduling queues only when they are close to their scheduled execution time. This way, we keep the scheduling queue occupancy low to maintain its switchback efficiency. The switchback queues still provide dynamic adaptation to mispredicted instruction latency and selective replay. Our results show that Zephyr is able to eliminate a substantial amount of structural hazards and replays, improving IPC significantly.

However, Zephyr does not eliminate replays completely, and some instructions still enter the scheduler prematurely due to the underestimation of instruction waiting times. This can be due to imperfect load latency prediction or structural hazards such as conflicts for functional units. We further propose to detect the onset of a tornado early on and limit the number of instructions in the scheduling queues for a thread on the verge of forming a tornado. Our results demonstrate that Zephyr with this kind of preventive scheme further eliminates structural hazards and replays, thereby improving overall IPC.

Our contributions over prior work include:

- An investigation of the impact of structural hazards and replays on Cyclone in an SMT environment.

- A quantitative study of the Tornado Effect, characterized by low execution core throughput due to a high volume of misschedulings and structural hazards. This phenomenon may occur in any generalized speculative scheduler using selective replay, but we limit our analysis to the Cyclone scheduler.

- An analysis of the limitations of a simple integration of load latency prediction and Cyclone.

- The Zephyr architecture, which effectively prevents the formation of tornadoes. Zephyr is an integration of a load latency predictor, a sorting engine implemented with different length FIFOs, and a Cyclone-style scheduler. Zephyr is able to improve IPC significantly over both a baseline Cyclone scheduler and a simple integration of Cyclone with a load latency predictor.

The rest of this chapter is organized as follows. We describe the experimental methodology in Section 5.2. Section 5.3 describes the tornado phenomenon observed in the Cyclone scheduler. Several simple remedies are introduced in Section 5.4. Section 5.5 presents the Zephyr architecture. Summary and remarks follow in Section 5.8.

5.2 Methodology

The simulator used in this chapter was derived from the SimpleScalar/Alpha 3.0 tool set [BA97], a suite of functional and timing simulation tools for the Alpha AXP ISA. We have made significant modifications to SimpleScalar to model Simultaneous Multithreading (SMT) as in [TEE96]. The applications were compiled with full optimization on a DEC C V5.9-008 and Compaq C++ V6.2-024 on Digital Unix V4.0. We simulate at least 100 million instructions for

each thread after fast-forwarding an application-specific number of instructions according to Sherwood et al. [SPC01]. The processor configuration used for most simulations is shown in Table 6.3.

Strong	**ammp**.**gcc**
	art.**parser**
	ammp.**art**.**gzip**.**equake**
Weak	*bzip2.gap*
	crafty.mgrid
	bzip2.crafty.mesa.vortex
Mix	**ammp**.*bzip2*
	gcc.*gap*
	art.*crafty*
	parser.*mgrid*
	ammp.**gzip**.*bzip2.mesa*
	art.**equake**.*crafty.vortex*

Table 5.1: Applications grouped by strong tornado effects, weak tornado effects, and the mixes.

The benchmarks in this study are taken from the SPEC 2000 suite. As shown in Table 5.1, we rank the benchmarks by the number of misschedulings per issued instruction and the number of hazards caused by misscheduled instructions. The overall ranking is obtained by sorting the average of the two rankings – benchmarks with the same overall ranking are grouped in parentheses. Benchmarks with more misscheduled instructions and hazards suffer from stronger tornadoes. We select six benchmarks with strong tornadoes to form the "strong" group. Although `mcf` has dramatic tornadoes due to very frequent cache misses, we exclude

it to avoid skewing our results and to make them more representative as it receives an extremely large speedup (over 200%) from our approach. We select six benchmarks with weak tornadoes to form the "weak" group. These two groups, shown in Table 5.1, form the multithreaded workloads presented in this chapter. Three multithreaded runs are formed exclusively from the strong group, three are formed from the weak group, and six are formed from a mix of the strong and weak groups.

Parameters	Value
Issue Width	8
ROBs	256 entries
LSQs	128 entries
Queue Length	Cyclone: 100, PIB: 64, FIFOs: 1,5,10,20, or 150
Cache Block Size	L1: 32B, L2: 64B
Shared L1 Cache	16KB, 4-way, 2-cycle lat.
Shared L2 Cache	512KB, 2-way, 12-cycle lat
Memory Latency	164 cycles
Integer FUs	8 ALU, 2 Ld/St, 2 Mult/Div
FP FUs	2 FP Add, 1 FP Mult/Div
Integer FU Latency	1/5/25 add/mult/div (pipelined)
FP FU Latency	2/10/30 add/mult/div (all but div pipelined)
Branch Pred.	Private 4k BBTB, 8k gshare
Branch Penalty	20, additional 2 for latency prediction

Table 5.2: Processor Parameters.

5.3 The Tornado Effect

In this section, we examine the impact of the Tornado Effect on a speculatively scheduled SMT processor. We will consider the Cyclone scheduler as our representative speculative scheduler.

The Cyclone timing table [EHA03] is indexed by logical register and thread number. It returns the expected ready time of a particular logical register. Our Cyclone scheduler uses a switchback queue length of 100 – all threads share a common switchback queue. Our experiments demonstrate that there is no benefit from further lengthening or shortening the queues, even with latency prediction. We use ICOUNT [TEE96] for thread selection, where priority to enter the Cyclone queue is given to the thread with the least number of instructions in the Cyclone queues. Hu et. al. [HVI04] simulated Cyclone with an instruction placement strategy that places instructions into paths with different forwarding lengths based on predicted latency. This avoids instructions congregating in rows 0 and 1 of the queues. In our implementation, a round-robin placement strategy is used and it effectively avoids this problem.

Cyclone [EHA03] as described in Section **??** assumes that all loads hit in the first level cache, and schedules their dependent instructions based on this assumption. However, loads that miss in the cache and their dependents account for a large fraction of all instructions [LSM04b]. Ignoring long latency memory accesses can result in a large number of replays and structural hazards. The situation becomes worse in a simultaneously multithreading processor where switchback queue utilization is higher and misschedulings on one thread can waste issue bandwidth for other threads. Moreover, an increase in replays increases the probability of switching conflicts. The benchmarks `gap` and `gcc` can help to il-

lustrate this. When run alone, `gap` and `gcc` see an average of 2 replays per issued instruction. Table 5.3 presents statistics on hazards, replays, and occupancy for Cyclone. As we can see, when they run together on a 2-threaded SMT with Cyclone, around 5 replays per issued instruction are seen. Similarly, `gap` sees 2 switchback hazards per cycle on average – `gcc` sees 3 on average. But when run together, they see around 9 hazards on average each cycle.

One misscheduled instruction can directly cause instructions dependent on that instruction to be misscheduled. Furthermore, replays can create collisions – hazards in the switchback queues of Cyclone (stalling the progress of other instructions), thereby causing even more misschedulings – even on independent instructions. This positive feedback loop can result in the formation of a *tornado* [Car04]. A tornado is characterized by a period of low useful throughput and a high volume of replays, when instructions that are in the process of selectively replaying circulate through the Cyclone queues over and over again until they are correctly scheduled or squashed from a branch misprediction.

Figure 5.1 shows a snapshot from the execution of applications `gcc` and `gap`, running together on a 2-threaded SMT processor with Cyclone (processor configuration is as described in section 5.2). Statistics are collected and averaged over 10-cycle intervals (shown along the x-axis). On the left side, we show the composition of the issue width for each 10-cycle interval. For the 8-issue processor we consider, this figure shows the amount of time the issue bandwidth is used for correctly scheduled instructions from `gcc`, correctly scheduled instructions from `gap`, incorrectly scheduled instructions from `gcc`, incorrectly scheduled instructions from `gap`, and when the issue slots are idle. Instructions that have their input operands ready at issue are allowed to continue to the execution engine and are classified as correctly scheduled instructions. If an instruction has been

incorrectly scheduled (i.e. it issues before its input operands are ready), it is replayed to the countdown queues. The right side shows the average Cyclone queue occupancy broken down into the component contributed by `gcc` and `gap`, the presence of cache misses, and the average number of hazards in the switch-back queues. The two occupancies statistics are cumulative (i.e. the height of the greater occupancy line is the occupancy of the Cyclone queue for both `gcc` and `gap`) and are shown on the primary y-axis. The number of hazards uses the secondary y-axis.

The cache miss experienced by `gcc` causes a chain of misscheduled instructions that occupy the instruction bandwidth of the processor from interval 5 to interval 15. This period is characterized by relatively high queue occupancy by the thread that spawned the tornado (`gcc`) and a large number of hazards. After interval 15, `gap` is able to get some of the issue bandwidth and issue correctly scheduled instructions. Before interval 15, `gap` had instructions that could have issued, but were not even able to get into the Cyclone queues due to the dramatic number of replays. The Cyclone occupancy of `gcc` eventually drains as the instructions that made up the tornado are scheduled correctly after the cache miss is satisfied. However, the lapse in incorrectly scheduled instructions is shortlived before another cache miss starts another tornado.

5.4 Dealing with Tornadoes

In this section, we explore a number of techniques to combat the Tornado Effect on speculative schedulers.

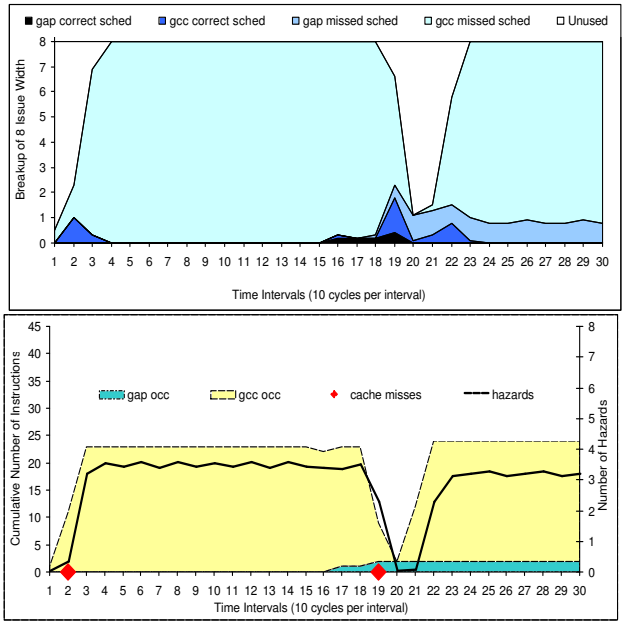

Figure 5.1: Snapshot of Cyclone Baseline

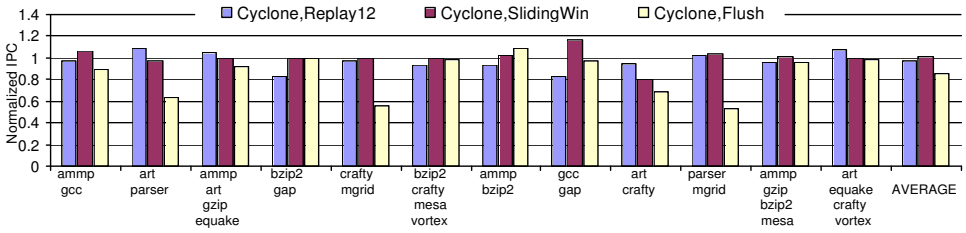

Figure 5.2: The Performance of Different Techniques to Combat Tornadoes on the Original Cyclone Scheduler.

5.4.1 Reducing Replay Frequency

One approach to reduce the formation of tornadoes is to replay instructions less frequently. A natural choice is a replay interval of 12 cycles – the L2 cache latency. This effectively coarsens the granularity of replays. Since misscheduled instructions replay every 12 cycles, it is possible for an instruction to wait for a longer time than required by true data dependence latency.

However, our experiments show an average 4% degradation in performance when using this approach. Figure 5.2 demonstrates the performance of this longer replay interval, Replay12. While a few application combinations like `art.parser`, `ammp.art.gzip.equake`, `art.equake.crafty.vortex` see improvement, most of the benchmarks perform worse with a longer interval. The benchmarks `art`, `ammp`, and `equake` have L2 cache miss rates of over 25%. This approach helps such applications to reduce hazards, switchback queue occupancy, and replays. In addition, these applications have longer average load latencies, and are therefore able to tolerate waiting a few extra cycles to speculatively issue a misscheduled instruction. Longer average load latencies mean significantly more replays and longer tornadoes - and therefore a coarser granularity can prove highly effective here. However, it degrades the performance of most of the other applications, as these do not have such long average load latencies. The impact of waiting an unnecessarily longer amount of time outweighs any benefit from reduced replays.

We also explored a replay interval of 6 cycles, and observed a similar degradation in performance.

5.4.2 Limiting Threads to Prevent Tornadoes

Once a tornado develops, it exhibits symptoms like excessive replays and a high volume of switchback hazards in the scheduling queues. One solution would be to detect tornadoes in their early stages and take preventive measures to avoid the full onset of the tornado.

We have developed effective algorithms to detect tornadoes in the context of the Cyclone scheduler. Although we have considered a number of different policies for dealing with tornadoes, the most effective approach we have found is the *Sliding Window* scheme. In this approach, we set a per-thread window size (*WIN*) which caps the total number of instructions from a given thread that are allowed in the Cyclone queues. This limit can be increased or decreased for each thread to control tornado formation.

The Sliding Window can be easily implemented as a part of the ICOUNT policy. To adequately guide this mechanism, we need to know when a thread is likely to spawn a tornado (*WIN* should be decreased) or when a thread is too severely restricted, potentially impairing ILP (*WIN* should be increased). The former condition will be referred to as *overflow* and the latter condition will be referred to as *underflow*. We will determine whether a thread is in underflow or overflow by considering how many replays are required in a given period of time. Too many replays means that the thread may be forming a tornado – too few replays means that the thread may be too constrained and is not being aggressive enough in speculative scheduling.

We define two thresholds: the overflow threshold *(OF_{th})*, which determines when a thread is likely to form a tornado, and the underflow threshold *(UF_{th})*, which determines when a thread is probably too severely restricted. To ensure

that *WIN* does not oscillate wildly, we require that a thread exceeds OF_{th} for more than $Decr_{th}$ consecutive cycles before decrementing *WIN*. Similarly, a thread must not replay more than UF_{th} for $Incr_{th}$ consecutive cycles for *WIN* to be incremented.

Our implementation of this algorithm uses three counters, an *Increment_Flag*, and a *Decrement_Flag* per thread. The first counter (*R_Counter*) counts the number of replays per cycle. It resets every cycle. The second counter (*OF_Counter*) counts the number of consecutive cycles a thread remains in *overflow*. The *OF_Counter* resets whenever the thread is not in *overflow*. The third counter (*UF_Counter*) counts the number of consecutive cycles in *underflow*. The *UF_Counter* resets whenever the thread is not in *underflow*.

If $OF\_Counter >= Decr_{th}$, the *Decrement_Flag* is turned on. Similarly, the *Increment_Flag* is turned on if $UF\_Counter >= Incr_{th}$. When either flag is triggered, the triggering counter (OF_Counter or UF_Counter) resets. Both flags are reset every cycle, after being tested to see if *WIN* will change in a given cycle. We performed extensive experiments to tune these parameters (results not shown). Our data demonstrates that in an 8-way cyclone scheduler, the following parameters can detect tornadoes effectively: $OF_{th} = 6$, $UF_{th} = 2$, $Decr_{th} = 10$, $Incr_{th} = 5$.

WIN is reduced upon the detection of tornado symptoms (i.e. if the *Decrement_Flag* is on), is increased when instructions are smoothly scheduled (i.e. if the *Increment_Flag* is on), and not changed if neither flag is set. Initially *WIN* is set to "unlimited", which does not cap the number of instructions at all. Upon detection of a potential tornado (*Decrement_Flag*), *WIN* is set to 24. When *Decrement_Flag* is set, this value is decremented by 4, and when *Increment_Flag* is set, this value is incremented by 4. Incrementing beyond 24 sets *WIN* to "un-

limited". *WIN* cannot be decreased below 4. To maintain fairness, *WIN* is reset to "unlimited" every 10,000 cycles.

Overall, we observe only a slight performance improvement of 1%. As shown in Figure 5.2, `gcc.gap` sees the most speedup (16%), while the remaining benchmarks see less than 6% improvement, some even performing worse than baseline cyclone. Unfortunately, this approach can significantly limit the amount of ILP that can be exploited from many applications. When Cyclone is operating smoothly, and there are no tornado effects, high occupancy in the cyclone queues can be extremely constructive, allowing the processor to see a larger window of issuable instructions. The benchmark mix of `gcc.gap` suffers from a dramatic number of tornadoes, and therefore is able to see benefit from this approach.

5.4.3 Exploiting TLP

Prior work [TB01] has demonstrated that overall throughput can be improved in an SMT architecture with conventional issue queue by stalling or flushing a thread when that thread suffers an L2 cache miss. The intuition here is that the thread is consuming resources that could be used for other threads while waiting for the long latency operation. The authors propose several mechanisms to detect an L2 miss (detection mechanism) and two ways of acting on a thread once it is predicted to have an L2 miss (action mechanism). The detection mechanism that presents the best results is to predict a miss every time a load spends more cycles in the cache hierarchy than needed to see a hit in the L2 cache, including possible resource conflicts (15 cycles in the simulated architecture of [TB01]). Two action mechanisms provide good results. The first is STALL, which consists of fetch-stalling the offending thread. The second, FLUSH, flushes the instructions after the load that missed in the L2 cache, and then stalls the offending thread until the

load is resolved. As a result, the offending thread temporarily does not compete for resources, and what is more important, the resources used by the offending thread are freed, giving the other threads full access to them.

We apply the same approach to Cyclone. In this chapter, an ideal version of FLUSH is considered. We detect an L2 cache miss ideally by probing the cache structures after selection by ICOUNT. This should give FLUSH the best performance potential by avoiding any wasted issue bandwidth when an L2 miss will occur. The thread is restarted once the load instruction that missed goes to writeback. However, this approach gives an average 15% slowdown as shown in Figure 5.2. Only application mix `ammp.bzip2` has a noticeable speedup of 9%. This is because the L2 miss rate of `ammp` is very high and few independent instructions closely follow an L2 cache miss in program order. The program `bzip2` has few L2 misses but has very rich ILP. When instructions from `ammp` are flushed due to a L2 miss, `bzip2` is able to improve by utilizing more of the available issue width.

However, a program may have a significant amount of ILP that can be exploited even after a level 2 miss is encountered, `art` being a notable example. In addition, other threads that are NOT flushed may have little ILP to exploit the scheduling resources emptied by the flushed thread. FLUSH does poorly on such applications. The typical examples are `art.parser`, `mgrid.crafty` and `parser.mgrid`. We observe that `art` and `mgrid` have relatively higher L2 miss rates. The programs `crafty` and `parser` have few cache misses, however, unlike `bzip2`, they benefit little from increased scheduling resources due to the lack of ILP. Consequently, these applications observe over a 30% slowdown.

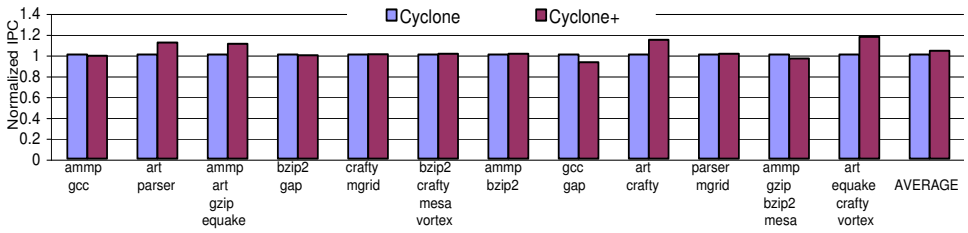

Figure 5.3: Baseline Cyclone (Cyclone) and Cyclone extended with load latency prediction (Cyclone+).

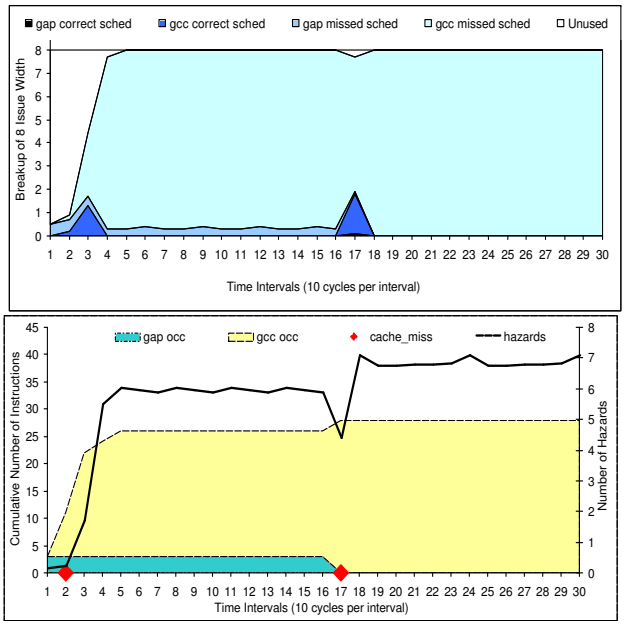

Figure 5.4: Snapshot of Cyclone Extended with Load Prediction (Cyclone+)

5.4.4 Cyclone+: Cyclone Extended with Load Latency Prediction

As shown in the previous sections, conservatively guessing an L1 cache hit for the latency of loads that do not alias stores is a major cause of tornadoes in the Cyclone scheduler. It seems much of the problem would be solved if the actual latency for each load was known.

To verify this, we extend Cyclone with the load latency prediction techniques recently proposed in [LSM04b]. The techniques in [LSM04b] capture 83% of the

load misses, and 99% of the cache hits. More accurate load latency prediction should allow Cyclone to more precisely schedule instructions and reduce the number of switchback structural hazards and replays. Address and latency predictors, as well as the miss detection engine and SILO, are shared by the threads. We limit the number of load latency predictions to two in each cycle to reduce the number of ports required on these structures. Our experiments show there is no benefit from increasing the number of ports any further.

Figure 5.3 shows the performance results for this extended Cyclone architecture. The first bar shows performance for the baseline Cyclone and the second shows the performance for Cyclone enhanced with latency prediction. Contrary to our expectations, predicting load latency only improves the performance of a handful of benchmarks (like `art.parser` and `art.crafty`) and actually degrades performance for a few application mixes (like `ammp.gcc` and `gcc.gap`). On average, Cyclone+ only shows a slight IPC speedup of 4%.

Our investigation shows that this is due to a dramatic increase in stalls for some applications. Table 5.3 (presented on page 82) presents the average number of structural hazards seen in the switchback queues per cycle, the average occupancy of the queues, and the average number of replays seen per cycle. Note that these behaviors are bursty and tend to occur in clusters – however, the average behavior is still useful for purposes of comparison. The first column shows the benchmark mixes we considered, and the first two columns of the hazards and structural hazard results show data for the baseline Cyclone and Cyclone enhanced with load latency prediction (Cyclone+) respectively. Cyclone+ sees significantly more structural hazards – except for a few application mixes (like `art-parser` and `art-crafty`) where there is actually a drop in hazards.

As shown in Table 5.3, we observe a substantial increase in queue occupancy.

On average, queue occupancy is 45% larger with Cyclone+ than baseline Cyclone - with some applications seeing double the occupancy with Cyclone+. When load latency prediction is applied, although the descendants of missed loads obtain their waiting times accurately, these waiting times are much longer than baseline Cyclone which assumes loads always hit the cache. These instructions can progress further towards the end of the Cyclone queues – the furthest point in the switchback queues from the execution engine. This increases the occupancy of the scheduling queues, thus creating more switchback hazards. When queue occupancy is high, a tornado can be formed.

As an example, an instruction is not switched on time due to a structural hazard (i.e. queue conflict due to high occupancy or replay). The dependents of that instruction may have to be replayed even if they do not encounter any hazards. Such replays occupy queue spaces, which can further introduce more conflicts and replays. When a tornado is active, the scheduler experiences extremely low useful throughput, and a high volume of replays and hazards. The benefit of reduction in replays through load latency prediction is effectively canceled for some applications by the dramatic increase in structural hazards and switchback queue occupancy – all of which feeds the Tornado Effect.

5.4.4.1 Tornadoes in Cyclone+

Figure 5.4 shows a snapshot from the execution of `gcc.gap` in Cyclone+. We observe relatively larger queue occupancies and relatively more structural hazards. At interval 2, a cache miss results in the formation of a tornado. By interval 17, the cache miss is satisfied and a few instructions from `gcc` are issued. Before the old instructions drain, however, a new cache miss brings more instructions into the scheduling queue. The tornado is sustained and further deteriorates.

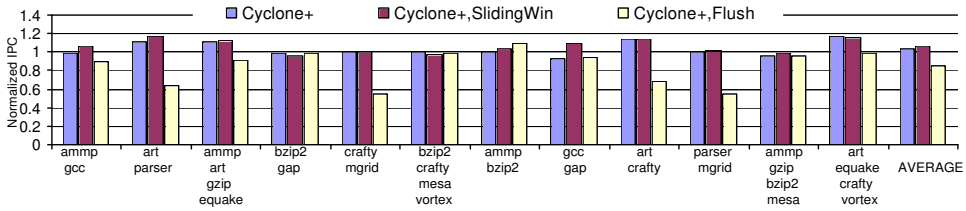

Figure 5.5: the IPC Performance of Cyclone Extended with Load Prediction (Normalized with Baseline Cyclone)

5.4.4.2 Improving Cyclone+

The tornadoes affect Cyclone+ more than the original Cyclone scheduler, as evidenced by the increased occupancies, switchback hazards, and replays. In this section, we attempt to mitigate this by applying techniques to prevent the formation of tornadoes.

Figure 5.5 shows the normalized IPC (with respect to the original baseline Cyclone) of Cyclone extended with load latency Prediction (Cyclone+), the further extension of Cyclone+ with our sliding window approach (Cyclone+, Sliding-Window) and Cylone+ with FLUSH (Cyclone+,Flush). Cyclone+ with Sliding-Window performs 2% better on average than Cyclone+ alone. As was the case when applied to baseline Cyclone, the sliding window approach only improves a handful of application mixes that suffer from extremely strong tornadoes, but does not help and even hurts other application mixes. Cyclone+ with FLUSH has a comparable performance to that of baseline Cyclone with FLUSH, often degrading performance as it limits the available ILP.

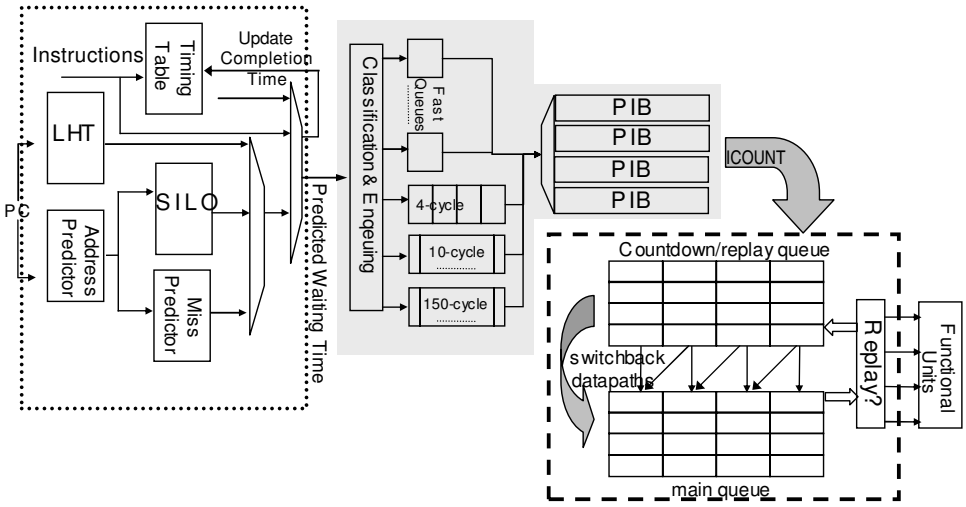

Figure 5.6: Zephyr Scheduler Architecture

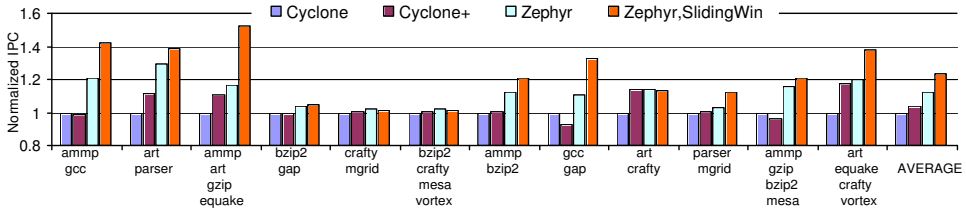

Figure 5.7: Speedup of Zephyr and Zephyr with Sliding Window

5.5 Zephyr Design and Performance Evaluations

Our prior attempts to combat tornadoes either sacrificed ILP to reduce tornadoes or increased queue occupancy beyond Cyclone's ability to effectively switchback instructions. In this section, we propose Zephyr, an architecture that reduces replays and structural hazards without sacrificing ILP and without straining Cyclone's queue structure. Zephyr has the potential to improve any speculative scheduler plagued by the tornado effect, but we consider the impact on Cyclone alone for brevity.

5.5.1 The Zephyr Scheduler

Figure 5.6 illustrates the high-level architecture of the Zephyr scheduler. Zephyr prevents the formation of tornadoes by sorting instructions in their predicted execution order and then using Cyclone to adapt to misschedulings. Zephyr effectively allows instructions to enter Cyclone out of order. Figure 5.6 is divided into three components: the latency prediction engine (delineated with a dotted box), the coarse-grain sorting engine (colored in grey), and the fine-grain sorting engine (surrounded with a dashed box). The latency prediction engine features a timing table (similar to [EHA03]) that is accessed on every instruction, and a latency prediction structure (similar to [LSM04b]) that is accessed on every load instruction. The result of this prediction stage is a predicted wait time for each instruction before all input operands are ready.

Instructions are then enqueued in the coarse-grain sorting engine – the FIFOs. Instructions with very short waiting times are placed into FIFOs with a buffering length of 1, and can progress to the next stage in one cycle. Instructions with longer waiting times are placed into different FIFOs with buffering lengths of more

than one. During the classification, we round down waiting times to the closest granularity queue available, ensuring instructions are not delayed beyond their estimated waiting time. We adopt the same queue configurations as in [LSM04b], but double the number queues to accommodate the additional bandwidth of SMT. We have six 0-slot queues, four 5-slot queues, two 10-slot queues, two 20-slot queues, and two 150-slot queues. Instructions enter the FIFOs in program order, but can leave the FIFOs out of order.

Instructions are buffered in the PreIssue Buffer (PIB) after sorting. Each thread has a PIB, and we use the ICOUNT [TEE96] thread selection policy to choose a PIB from which to pull instructions. ICOUNT measures the number of instructions from each thread that are currently in the fine-grain sorting engine. Note that ICOUNT cannot pull instructions from an empty PIB. Since the coarse-grain sorting has absorbed some of the expected wait time from instructions in the PIB, if there are no available instructions in the PIB, it indicates that there is currently no ILP to exploit in a given thread. Therefore, the sorting engine enables a more intelligent ICOUNT which has some notion of available parallelism in a given thread – and will selectively pull from threads that have such parallelism.

Instructions leave the PIB and enter the fine-grain sorting engine (Cyclone). Here, instructions may encounter structural hazards or may need to be replayed if they have been misscheduled. However, the coarse-grain sorting engines of Zephyr are able to absorb some of the instruction latency to keep the countdown/replay queue occupancy low. This reduces the structural hazards in Cyclone and inhibits the formation of tornadoes.

Figure 5.7 shows the relative performance of Zephyr, using Cyclone as the baseline. Zephyr has an average of 13% speedup over Cyclone. This benefit

comes from the more accurate waiting time prediction as load misses are taken into account, as well as the reduction in occupancy, and scheduling hazards. As seen from the graph, applications with strong tornadoes have an average of 22% speedup. The mixed applications have an average of 13% speedup. The benefits on applications with weak tornadoes are small. But still, an average of 3% speedup is observed.

As mentioned in [EHA03], the greatest contributors to IPC loss with Cyclone are structural hazards (switchback conflicts) and replays in the scheduling queues. Zephyr reduces the number of hazards by buffering the instructions in the coarse-grain sorting structure. Instructions entering the scheduling queues are expected to have their operands ready in a short time. We observe an average of 31% reduction in queue occupancy compared with baseline Cyclone. Table 5.3 illustrates this reduction with the data labeled "Z" – Zephyr. Zephyr observes a 42% reduction in structural hazards from the original Cyclone, and a 60% reduction from our Cyclone+. Zephyr also observes a 24% reduction in the number of replays relative to the original Cyclone design. The applications with strong tornadoes and the mixed applications observe large reductions in structural hazards and replays. Correspondingly, these applications see significant speedup over the Cyclone baseline in Figure 5.7. Applications from the group of weak tornadoes see less of a drop in queue occupancy, hazards, and misschedulings, and therefore see a smaller performance improvement.

5.5.2 Zephyr with the Sliding Window

Zephyr cannot eliminate all replays. There are only a finite number of sorting queues, and therefore the fine grained sorting in the Cyclone queues can still cause switchback hazards even for instructions with highly predictable latencies.

	hazards per cycle				occupancy				replays per inst.			
	C	C+	Z	zW	C	C+	Z	zW	C	C+	Z	zW
ammp.gcc	21	23	12	5	101	126	63	36	6.2	6.1	4.7	1.6
art.parser	12	10	6	1	56	101	40	28	6.0	4.4	3.1	1.4
ammp.art–gzip.equake	25	41	10	11	82	166	49	63	4.7	4.0	3.9	2.2
bzip2.gap	6	7	3	3	34	37	21	21	1.4	1.4	1.3	1.2
crafty.mgrid	5	5	4	3	29	30	21	20	1.5	1.5	1.4	1.2
bzip2.crafty–mesa.vortex	6	6	5	4	33	36	27	24	1.0	1.0	1.0	0.9
ammp.bzip2	22	22	12	9	87	92	55	42	2.2	2.2	1.9	1.4
gcc.gap	9	16	4	1	45	88	30	16	5.3	5.6	3.7	1.5
art.crafty	10	2	1	1	51	57	19	18	4.3	1.3	1.2	1.0
parser.mgrid	5	7	5	4	31	51	33	28	2.6	2.4	2.1	1.4
ammp.gzip–bzip2.mesa	17	20	6	8	61	71	33	40	1.7	1.7	1.3	1.2
art.equake–crafty.vortex	13	21	10	5	56	128	52	40	3.3	2.4	2.6	1.2

Table 5.3: Comparison of the number of replays and structural hazards (i.e. switchback conflicts) in the scheduling queues. Symbols: "C" — Cyclone, "C+" — Cyclone+, "Z" — Zephyr, "zW" — zephyr sliding Window.

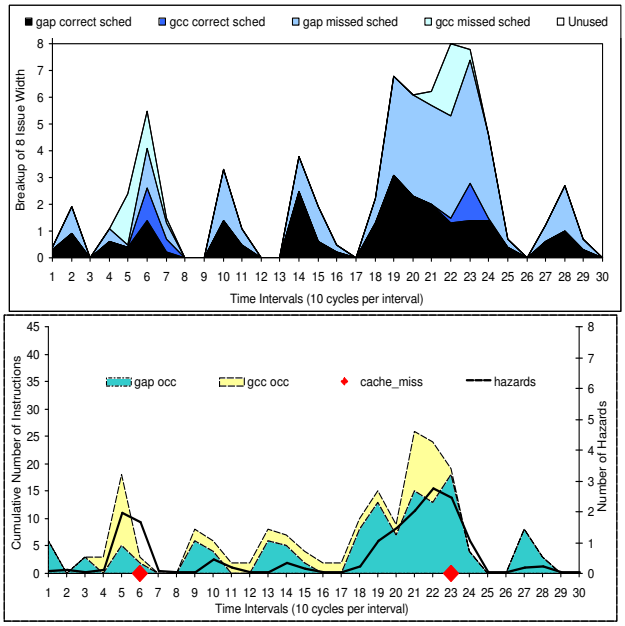

Figure 5.8: Snapshot of Zephyr with Sliding Window

Moreover, if the latency prediction engine is not able to confidently report a latency for a load instruction, it conservatively guesses the latency of a cache hit. On average, 17% of load misses are not captured. In addition to underestimating loads that are not address or latency predictable, an instruction's waiting time may be underestimated due to other types of structural hazards such as contention for functional units, memory ports, and the memory bus. This underestimation potentially issues instructions prematurely, spawning replays and potentially more structural hazards.

We consider using the preventive measures presented in Section 5.4 to reduce replays spawned by unpredictable loads and structural hazards. The best performing approach with Zephyr is the Sliding Window from Section 5.4.2. We use this approach to throttle instructions from entering the Cyclone queues via the PIBs on a per thread basis. We use the same set of parameters as described in section 5.4.2.

Since Zephyr is able to safeguard ILP by effectively buffering those instructions that are waiting on input dependencies, we expect our Sliding Window to be more effective in reducing tornado formation without impacting performance.

Figure 5.7 shows the performance of the Sliding Window on Zephyr. This approach shows significant speedup from the baseline Cyclone. On average, we observe a 23% improvement in performance, 60% less hazards, a 41% drop in occupancy, and 48% fewer replays. This represents a further performance improvement from Zephyr alone. Applications in the strong group observe further speedups ranging from 40% to 50%. The mixed applications observe an average of 23% speedup. A single thread degradation of 13% is seen by `gcc` in the 2-thread run `gcc.gap`, and a degradation of 15% by `parser` in the 2-thread run `parser.mgrid`. Other than this, we see no more than a 4% per thread degradation.

Load latency prediction is unable to capture 10% to 30% of load misses in `art`, `ammp`, and `equake`. In `gcc`, `parser` and `gzip`, frequent conflicts for FUs cause a great deal of underestimation. In these application mixes, the Sliding Window scheme successfully limits the underestimated instructions from entering the scheduling queues before the onset of a tornado. As a result, these applications experience substantial performance improvement.

The applications from the weak group see fewer tornadoes. Hence, these applications see no further improvement using our sliding window. The mix of `bzip2.crafty.mesa.vortex` even experiences a slight degradation of 1% relative to Zephyr alone. This is because the application has very few tornadoes, and the sliding window can still limit ILP.

Figure 5.8 shows a snapshot of 300 cycles from the execution of `gap.gcc`

using Zephyr with our Sliding Window. In general, we observe more accurate scheduling and throughput and fewer hazards and Cyclone queue occupancy. When cache misses are captured by the predictor, their dependents cannot enter the scheduling queues immediately, but have to go through the sorting queues and PIBs. They can only enter the scheduling queue when they are close to their predicted issuing time. This reduces unnecessary queue occupancy and switchback hazards. As can be seen in the figure, instructions are soon issued after they enter the scheduling queues.

As shown in the figure, there are still a substantial amount of premature instructions being replayed though at much reduced scale. This is because instructions can enter the scheduling queue several cycles in advance due to the finite granularity in the sorting stage. This can also be due to underestimated waiting time caused by load latency mispredictions, hardware hazards, and scheduling hazards. However, Zephyr with our Sliding Window can take preventive measures before premature instructions can form tornadoes. When early tornado symptoms are observed via the (*Decrement_Flag*), the underestimated instructions are forced to wait longer in the PIBs. This helps these instructions to enter the scheduling queues at the right time. Overall, Zephyr with our Sliding Window prevents tornadoes, and is able to schedule instructions correctly with sustained useful throughput.

5.6 Load Latency Prediction Analysis

One critical component in Zephyr is the load latency predictor. In this section, we evaluate the prediction performance.

Figure 5.9 shows the percent of loads that are predicted by the LHT, SILO and

default prediction. Each of these predictors is enhanced with two bit confidence counters to guide predictability – the counters are incremented on a correct prediction and decremented on an incorrect prediction. Our implementation gives priority to the LHT if the LHT can predict confidently. Other loads are predicted by the SILO (with MNM) if the address predictor can predict an address confidently. If neither the LHT nor the SILO can make a confident prediction on a given load, the load is assumed to hit in the level one cache. The LHT structure is much smaller than the alternative SILO scheme, which needs an address predictor and structures to detect cache misses and to track in-flight loads. As shown in Figure 5.9, about 75% of loads are predicted by the less expensive LHT.

Figure 5.10 shows the percent of L2 misses that have their latencies correctly predicted. In general, the prediction scheme is affected by the low coverage of address predictors. We show that with more accurate address prediction, we can ideally achieve over 85% prediction accuracy for L2 misses.

Figure 5.11 shows the histogram distribution of predicted waiting times for all instructions. Note that the scale of the figure has been adjusted to show longer latency instructions more clearly. The smaller figures in the upper right corner of the figures in Figure 5.11 present the full y-axis scale to better illustrate shorter latency instructions. From this distribution, we choose a FIFO queue configuration that has more queues for short-latency instructions, and fewer queues for long-latency instructions.

5.7 Area and Energy

In this section, we examine the area cost and energy consumption of the Zephyr architecture.

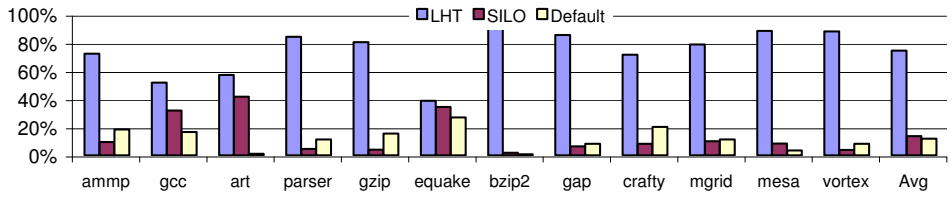

Figure 5.9: Percent of Loads that are Predicted by LHT, SILO and Default

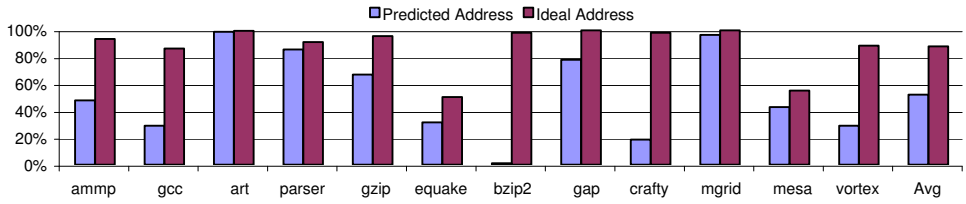

Figure 5.10: Prediction Accuracy of L2 Misses

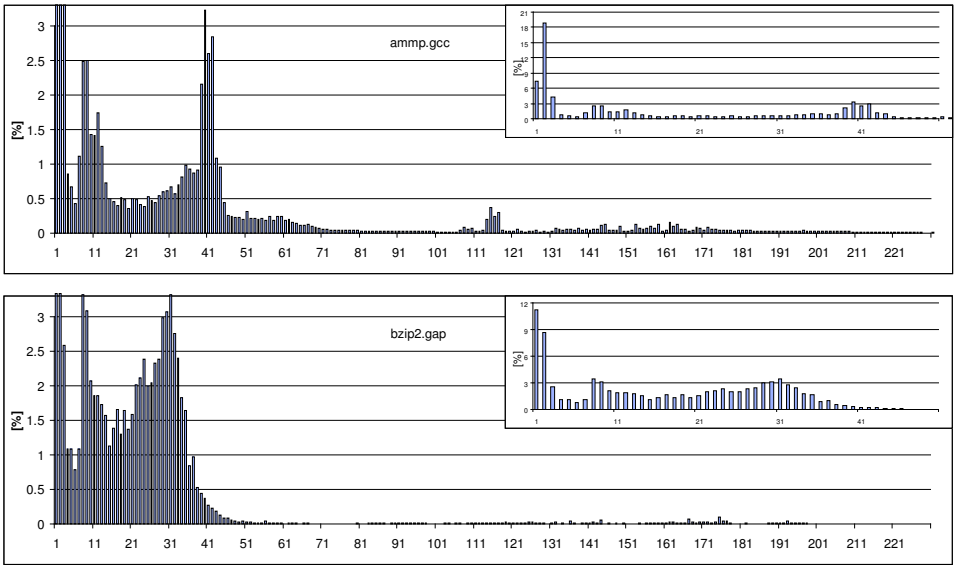

Figure 5.11: the Histograms of Predicted Waiting Times for ammp.gcc and gzip2.gap. The smaller figures in the upper right corner present the full y-axis scale to better illustrate shorter latency instructions.

5.7.1 The Area Cost of Zephyr

We first obtain the silicon area of the different microarchitectural components. These components are modeled based on 70nm process technology using parameters from [TXV05]. The area of instruction and data caches are obtained using the CACTI tool [SJ01]. We obtain the area of functional units from [PJS97] using scaled feature size. Other structures, such as branch predictor, renamer and load-store queue are modeled as the tagless data arrays in Cacti [WJ, RJ, SJ01]. The area of Zephyr structures, including the address predictor, LHT, SILO and buffers that include sorting queues and PIBs are also modeled as tagless data arrays.

We obtain the overall microprocessor area by summing up the area of individual components. More accurate results could be obtained using tools that perform detailed routing and floorplanning. However, we find that estimation at the components level is sufficiently accurate – the total area of the Alpha 21264 [Kes99] using our estimation method is only 5% smaller than the actual size.

The area results are summarized in 5.4. The Zephyr specific structures comprise only about 4% of the total area budget.

5.7.2 Analysis of Energy Performance

We integrate Wattch [BTM00] with the SimpleScalar 3.0 tool set [BA97] to evaluate the energy performance, using this to estimate the power for a 70nm process technology at 3 GHz. We model the FIFO buffers in the same way as Orion [WZP02] using SRAM arrays. The address predictors, SILO and LHT are modeled similarly to branch predictors as in Wattch [BTM00].

	Area(mm^2)	Percentage
Address Predictors	0.342	1.9%
Buffers	0.153	0.8%
LHT	0.051	0.3%
SILO	0.165	0.9%
Total (Zephyr-specific Blocks)	0.711	3.9%
Total Core Area	18.072	100.0%

Table 5.4: The Area of Zephyr Components

We compare the energy performance of original scheduler with the Zephyr, Sliding Window approach. We use the energy per committed instruction in order to compare the two scheme on a fair basis. We obtain the total energy dissipation and then divide it by the number of committed instructions to produce the average energy per instruction.

The percentage of total energy consumed by Zephyr-specific structures are shown in Figure 5.12. On average, we observe only 2.4% of total energy is consumed by Zephyr-specific structures. In benchmark `art.crafty`, more energy is consumed by the additional structures because there are more memory accesses and strong tornadoes. `Bzip2.gap` has the least energy consumed by the additional structures because of few memory accesses and weak tornadoes.

As shown in Table 5.3, Zephyr with a sliding window is able to reduce scheduling replays significantly. The reduction comes from two sources. First, the dependents of missed loads that are correctly latency predicted tend not to be misscheduled because they do not enter the scheduling queue until they have been buffered. Second, dependents of loads that are not correctly latency predicted

will have a minimal effect thanks to the sliding window.

The reduction in scheduling replays translates into energy savings in the register file. In Zephyr, in addition to reduced energy from reduced replays, there is also a net energy reduction purely from the performance gain (i.e. an intrinsic drop in energy per instruction when IPC is higher) since the energy overhead of Zephyr-specific structures is small. As shown in Figure 5.13, the new approach saves energy in the register file by an overall average of 60% – saving 72% in strong-tornado applications and 50% in weak-tornado applications. Strong-tornado applications observe a much larger energy savings due to a larger amount of cache misses.

The reduction in energy dissipation in the register file, combined with the improved utilization of the scheduling queue, results in an overall drop in energy dissipation. As shown in Figure 5.14, the strong-tornado applications observe an average 30% reduction in energy per instruction. In these applications, a large amount of misschedulings are effectively eliminated, and therefore a significant energy reduction is observed. The weak-tornado applications observe a 19% reduction. These applications, though have less energy savings from eliminated replays, have large savings in scheduling queue energy. On average, we observe a 13.2% energy reduction using Zephyr with a sliding window.

5.8 Summary

While Cyclone is able to provide scalable instruction scheduling for deeply pipelined processors, the structural hazards and replays that are possible with Cyclone can severely degrade performance in a multithreaded environment. Useful issue bandwidth can be wasted on misscheduled instructions, limiting the amount of thread

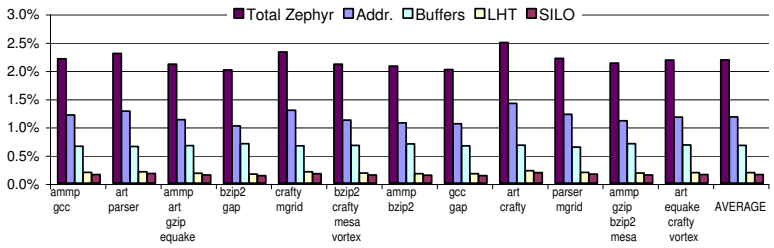

Figure 5.12: the Percentage of Total Energy Consumed by Zephyr Structures, including prediction structures and buffers

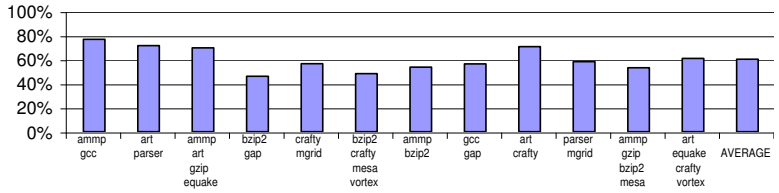

Figure 5.13: the Percentage of Reduced Register File Energy (per committed inst.) in Zephyr Sliding Window Compared with Cyclone

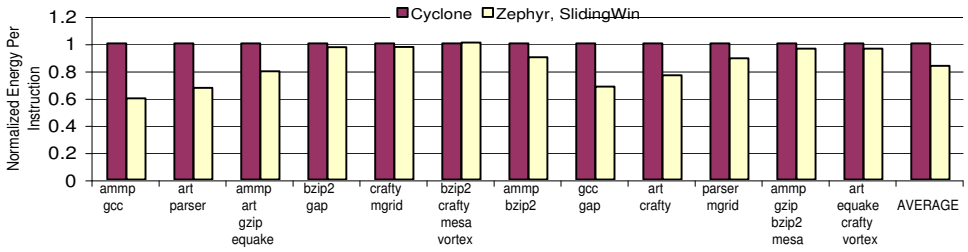

Figure 5.14: Comparing the Energy Per Committed Instruction: Cyclone vs Zephyr Sliding Window, Normalized with Cyclone

level parallelism that can be exploited. There exists a positive feedback loop between structural hazards and replays that can result in the misscheduling of a large portion of Cyclone-issued instructions. This is characterized by instructions from different threads continually shuffling around the scheduler, with low useful scheduling throughput. This *Tornado Effect* can also happen with other replay-based schedulers that employ speculative scheduling.

In this chapter, we present Zephyr, an architecture that intelligently schedules instructions to avoid tornadoes in multithreaded processors. Scheduling instructions dependent on loads is extremely challenging as load access time is highly nondeterministic, particularly when considering loads that alias with in-flight memory requests. Assisted by prior work on load latency prediction, Zephyr is able to predict instruction waiting times accurately, including instructions dependent on loads. Zephyr buffers instructions in a coarse-grain sorting engine, resulting in an approximate execution ordering for each thread. This ordering is buffered in a per-thread buffer that can then steer thread selection for execution. Instruction scheduling is still done via Cyclone, with fine-grain sorting handled by the Cyclone switchback queues. Cyclone is also able to dynamically adapt to unpredictable load latencies and misschedulings using selective replay.

We further propose a Sliding Window algorithm to enhance Zephyr, which dynamically caps the number of instructions allowed in the scheduling queues by observing the symptoms that lead to the formation of a tornado. With this option, the Zephyr architecture delivers consistently higher IPC than the baseline Cyclone. Our experiments show Zephyr has a 23% improvement in IPC performance, 60% less hazards, a 41% drop in occupancy, and 48% fewer replays compared with a baseline scheduler.

Our analysis shows that Zephyr needs only 4% additional area. By reducing

the scheduling queue occupancy, scheduling replays and accesses to register file, Zephyr is able to reduce energy by an average of 13%.

CHAPTER 6

Reducing Scheduling Energy

Energy dissipation from the issue queue and register file constitutes a large portion of the overall energy budget of an aggressive dynamically scheduled microprocessor. In this chapter, we propose techniques to save energy in these structures by reducing issue queue occupancy and by reducing unnecessary register file accesses that can result from speculative scheduling.

6.1 Motivation

In contemporary microprocessors, the out-of-order issue queue logic and register file access are responsible for a large portion of the total energy dissipated. The issue queue employs a fully associative structure that can potentially wakeup and select new instructions to issue every cycle from any slot in the queue. As a result, the issue queues are often a major contributor to the overall power consumption of the chip, and can be a hot spot [BAS02, BKA03] on the core. In [FG01], it is estimated that instruction issue queue logic is responsible for around 25% of the total power consumption on average. Wilcox et al. [WM] showed that the issue logic could account for 46% of the total power dissipation in future out-of-order processors that support speculation.

Similarly, register files also represent a substantial portion of the energy

budget in modern high-performance, wide-issue processors. It is reported that modern register files represent about 10% to 15% of processor energy [PPV02]. The development of speculative scheduling in recent microarchitectures [HSU01, Kes99] will further worsen the energy dissipation in register files.

Speculative scheduling emerged in response to the growing pipeline depth in recent microprocessor designs. The pipeline depth of dynamically scheduled processors between instruction scheduling and execution – the schedule to execute (STE) window has grown to multiple cycles. For example, the recent P4 design features a 7 stage STE window [HSU01]. Conventional schedulers broadcast instruction completion to instructions in the issue queue, and then select candidates for execution from the pool of ready instructions [Tom]. The throughput of such schedulers in a deep-pipelined processor is extremely low as every dependency is exposed to the depth of the STE pipeline. Speculative schedulers [HSU01, Kes99], on the hand, are designed to hide the latency of the STE pipeline by anticipating operand ready time and scheduling the instruction further ahead, even before their parent instructions have completed execution.

Speculative execution works well for instructions that have predictable latencies, but load latency is highly nondeterministic. As shown in Chapter 3, this nondeterminism comes from cache misses, loads that alias with in-flight data blocks, and memory bus contention. Load instructions can take anywhere from several cycles to several hundred cycles in current generation processors. Current generation designs [HSU01, Kes99] speculatively schedule load dependents by assuming cache hits. If a load misses in the cache, the processor must prevent the load's dependents from executing and then attempt to reschedule these instructions until they are correctly scheduled.

Such misscheduled instructions will speculatively access the physical register

files before their operands are ready, and will therefore consume even more energy in the register file. If the latency of load instructions can be determined prior to scheduling, then the misscheduling of their dependents can be avoided, saving register file energy.

Another source of wasted energy is in the issue queue. In contemporary designs, instructions with long waiting times (i.e. dependent on long latency operations) will remain in the issue queue while waiting for their operands. Prior research [BSB01, BKA03] demonstrates that an issue queue consumes power proportionately to the number of active entries in issue queue. If the long latency instructions are known a priori, we can buffer their dependents before they enter the issue queue – effectively allowing instructions to enter the scheduling window out of order. In this way, we can reduce the energy consumption in the issue queue by reducing the issue queue occupancy. In this chapter, we look into modern speculative schedulers and we propose several techniques to reduce register file accesses and issue queue occupancy in such schedulers.

Particularly, in this work, we make the following contributions:

- We investigate the energy impact of speculative scheduling on deeply pipelined processors,
- We reduce the energy consumption in the issue queue and register file by applying latency prediction and instruction sorting in a speculative scheduler,
- We examine the energy cost of the added latency prediction and instruction sorting hardware.

The rest of this chapter is organized as follows. In Section 6.2, We describe

energy reduction techniques. Section 6.3 describes our experimental methodology. Section 6.4 presents our simulation results. Summary follows in Section 6.5.

6.2 Scheduling Techniques

In this chapter, we apply latency prediction to reduce the energy from misschedulings in a speculative scheduler and the energy of the issue queue.

6.2.1 Conventional Wakeup-and-Select vs Speculative Scheduling

In recent microprocessor designs, the number of pipeline stages from the stage of Scheduling To Execution (STE) has grown to accommodate the latency needed for reading the register file and performing other book-keeping duties. Conventional instruction schedulers let issued instructions wake up their dependent instructions [Tom]. As the pipeline stages from scheduling to actual execution grow beyond a single stage, conventional wakeup-and-select can no longer schedule and execute the dependent instructions back-to-back. Each back-to-back scheduling exposes the depth of the schedule to execute window.

Modern microprocessors address the problem by speculatively waking up and selecting dependents instructions several cycles ahead [Kes99, HSU01]. In this way, instructions that have back-to-back dependencies can still be executed in consecutive cycles.

6.2.2 Replays in Speculative Scheduling

In current microprocessors, the instructions dependent on a load (and their eventual dependents) are scheduled with the assumption that the load will hit in the

cache. This assumption usually increases the performance as most of the loads actually hit in the cache. However, performance can be adversely affected in the case of cache misses.

Consider a load operation that is scheduled to execute. If the STE pipeline latency is n, and the processor is an m-way machine,
$(m \times (n + time\_to\_identify\_miss))$ instructions may be scheduled before the scheduler knows whether the load will hit in the cache, Once the load instruction misses in the cache, these dependent instructions will be replayed. Misscheduled instructions represent wasted issue bandwidth (as something useful could have been issued in place of the instruction) and wasted issue energy (wasted energy in the issue logic and register file).

Prior work has suggested two replay mechanisms: flush recovery (used in the integer pipeline of the Alpha 21264 [Kes99]) and selective recovery (used by Pentium 4 [HSU01]). In this chapter, we focus on selective replay, where the processor only re-executes the instructions that depend on the missed load.

A speculative scheduler needs to know when to reschedule instructions dependent on a load miss, but load latency is highly nondeterministic. A load may hit in different levels of the memory hierarchy. In addition, a load miss may alias an already in-flight data block. Bus contention and memory port arbitration may also impact the load latency. In current processors, the latency of load can range from a few cycles to a few hundred cycles. The current strategy [HSU01, Kes99] is to replay instructions every so often until the instructions are correctly scheduled. Waiting longer before replaying a misscheduled instruction can lengthen the perceived load latency. Replaying at too fine a granularity (i.e. replaying more frequently) can increase wasted issue energy and bandwidth. We have seen that replaying at the granularity of the L2 cache latency is a reasonable compromise.

Therefore, on a load miss, a misscheduled dependent instruction will continue to execute every `T` cycles, where `T` is the latency of the L2 cache hit.

6.2.3 Latency Prediction and Sorting

In this chapter, we explore the use of latency prediction techniques to reduce the number of replays in speculative schedulers. We use the lookahead latency prediction described in our previous work [LSM04b]. With correctly predicted load latency, speculative schedulers are able to schedule the dependent instructions to execute precisely after loads complete.

We propose to reduce the number of replays by allowing only instructions that are that can be executed soon to enter the issue queue, while buffering other instructions with long waiting times before they enter the issue queue. This way, instructions with long waiting times will not be mistakenly selected for execution. In addition, instead of attempting every T cycles, the scheduler makes decisions based on the accurately predicted latencies. This way, even if the dependents of a missed load enter the issue queue too early due to imperfect buffering , most of the scheduling replays can still be avoided. The speculative scheduler is also able to get better utilization out of the existing issue queue space, potentially improving performance.

Figure 6.1 shows the overall architecture of the proposed technique. In the early pipeline stages, we predict how long an instruction needs to wait before it can be issued, i.e, the waiting time for its operands to be produced. The "latency prediction" structure implements the techniques described in [LSM04b]. The prediction structure captures 83% of the load misses, and 99% of the cache hits. In the renaming stage, we let the PC access a load address predictor and

a latency history table (LHT). The latency history table performs a last latency prediction. If the LHT can confidently report a latency, then the predicted load latency is obtained. Otherwise, we use the predicted address to access a cache miss detection engine [MRM03a] – a small, energy-efficient structure to tell if a load address will miss in a given level of the memory hierarchy. We also access the SILO (Status of In-flight Loads) structure – a small structure to tell if a load aliases with an in-flight data block. The latency of loads are predicted based on whether a load needs to access the L1 or L2 cache, main memory, or if it aliases with an in-flight data block.

The latencies of instructions other than loads are deterministic. Each instruction saves its expected completion time in a timing table [EHA03], which can be implemented with architectural register files, so that its dependents can obtain the waiting times by checking their parents' completion times.

Once its waiting time is predicted, an instruction is placed into one of the FIFO queues in the sorting engine. The primary function of the FIFO queues is to hold the instructions until their waiting time has elapsed. Instructions with very long waiting times are placed into the long queues and those with short waiting times into the short queues. Instructions should only leave the sorting queues when they can be executed soon. This way, dependents of missed loads that are correctly latency predicted will not be misscheduled. At the same time, issue queue occupancy is reduced. The sorting queues feature a locking mechanism [LSM04b] that prevents instructions from entering the issue queue before their parents.

A Preissue Buffer (PIB) is inserted between the sorting structure and issue queue to provide an inexpensive buffering of sorting instructions in cases where the issue queue fills. Instructions may *not* directly issue from the PIB - they must

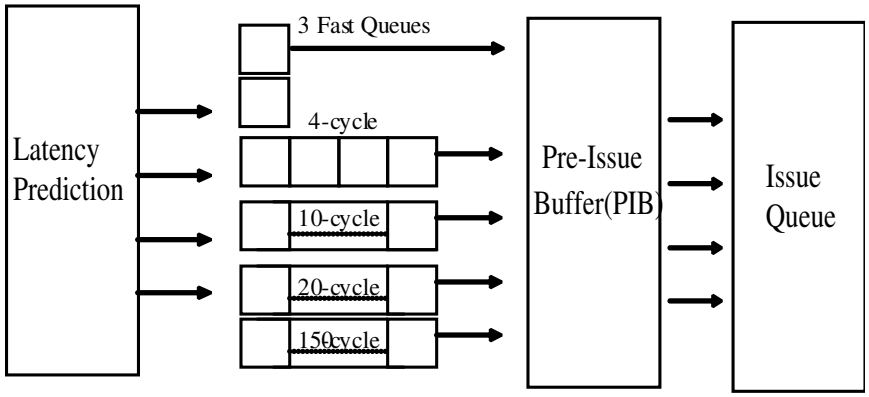

Figure 6.1: The Architecture to Perform Prediction, Sorting and Buffering

pass to the issue queue first.

6.3 Methodology

We integrate Wattch [BTM00] with the SimpleScalar 3.0 tool set [BA97] to evaluate the energy performance of our design. We simulate the power and performance for a 100nm process technology at 3 GHz. We obtain the total energy dissipation and then divide it by the number of committed instructions to produce the average energy per instruction for each application.

The benchmarks in this study are taken from the SPEC 2000 suite. We rank the benchmarks by the percentage of level 2 cache misses. As listed in Table 1, we take 6 benchmarks with high miss rates as memory-intensive applications, and another 6 benchmarks with low miss rates as ILP-intensive applications. The benchmark `mcf` has extremely large miss rate and sees benefit far beyond any other benchmark from our proposed techniques. We exclude it from the memory-intensive group to make our results more representative.

ILP-intensive applications	apsi,crafty,eon, gcc,gzip,vortex
Memory-intensive applications	ammp,applu,art equake,lucas,twolf

Table 6.1: The benchmarks used in this study.

The applications were compiled with full optimization on a DEC C V5.9-008 and Compaq C++ V6.2-024 on Digital Unix V4.0. We simulate 100 Million instructions after fast-forwarding an application-specific number of instructions according to Sherwood et al. [SPC01].

The processor configuration used for the base simulations is shown in Table 2.

6.3.1 Structures from Proposed Techniques

We model the address predictors and latency history table (LHT) as tagless arrays as in [WJ, BTM00]. The address predictor we use has 8K entries and the LHT has 2K entries. Each entry in these structures has 40 bits. Our SILOs (Status of In-flight Loads) are modeled as a 8-entry fully associative caches, with 40 bit block sizes. We model the timing table as part of the renaming table by extending each entry of the table by 10 bits.

We model the preissue buffers and sorting queues as FIFO queues. As in Orion [WZP02], we model FIFO queue energy with SRAM arrays [WJ, BTM00]. Our PIB has length of 64. The sorting queues have FIFO lengths of 1,5,10,20 and 150. The number of FIFOs are 3, 2, 1, 1, and 1 respectively.

Our 8-issue architecture would require 16 read ports and 8 write ports on the register file. To reduce the energy dissipation of the register file, we use a

Parameters	Value
Issue Width	8 instructions per cycle
ROBs	128 entries
LSQs	64 entries
Issue Queue	32 or 64 entries
Cache Block Size	L1: 32B, L2: 64B
L1 Cache	8KB, 4-way, 2-cycle latency
L2 Cache	512KB, 2-way, 12-cycle latency
Memory Latency	164 cycles
Integer FUs	8 ALU, 2 Ld/St, 2 Mult/Div
FP FUs	2 FP Add, 1 FP Mult/Div
Integer FU Latency	1/5/25 add/mult/div (pipelined)
FP FU Latency	2/10/30 add/mult/div (all but div pipelined)
Branch Predictor	4k BTB/Comb/Bimod/gshare
Branch Penalty	12, additional 2 for latency prediction

Table 6.2: Processor Configuration.

common technique: we maintain two copies of the register file, each with only 8 read ports and 8 write ports. Half of the functional units are connected to one register file and half are connected to the other. All writes go to both register files. Despite having to write each value twice (once per register file), we still save the net energy by reducing the ports on each individual register file.

6.4 Experiments and Results

In this section we examine the energy data for our latency prediction engine.

6.4.1 Prediction, Sorting and Buffering Structures

We use Wattch to record the total energy dissipation in these structures and then divide it by the number of committed instructions to produce the average energy per instruction for each application (shown in Figure 6.3). We use energy dissipation per committed instruction for a fair comparison among different schedulers. The total energy from FIFOs in the sorting engine in our simulation is similar to the result obtained from Orion's FIFO implementation [WZP02]. Overall, the energy dissipation from all additional structures for our latency prediction, sorting, and buffer engine is 0.3 nJ/instruction, which constitutes 4% of the overall energy consumption.

6.4.2 Speculative Scheduling

Speculative scheduling helps to hide the schedule to execute latency that is exposed with conventional wakeup and select logic. Figure 6.2 demonstrates that most of the benchmarks observe a large speedup with speculative scheduling, an

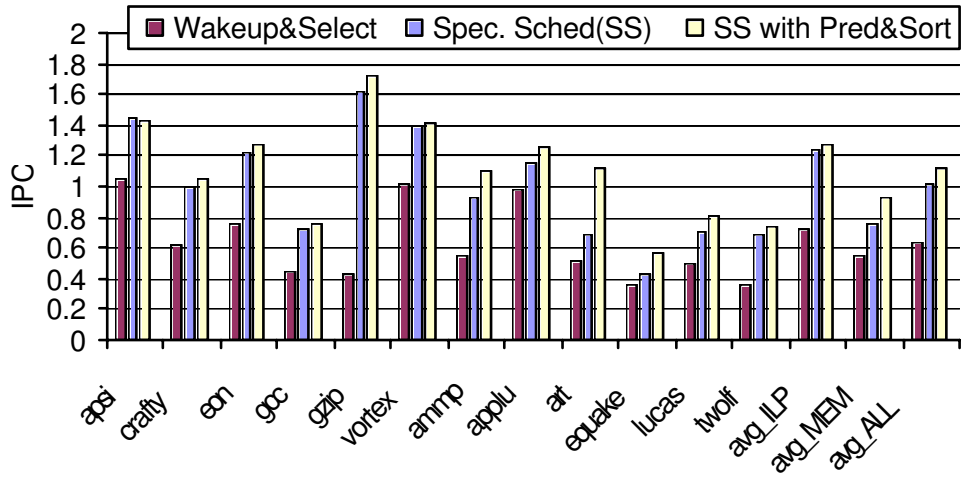

Figure 6.2: Performance in IPC

average 60% improvement. Without speculative scheduling, back to back instructions see the full schedule to execute window. In the remainder of this section, we will focus on reducing the energy consumption of speculatively scheduled processors.

6.4.3 Register File Energy

6.4.4 Issue Queue Energy

In the baseline speculative scheduler, instructions with long waiting times consume energy waiting for their operands in the issue queue. When we use latency prediction and instruction sorting to buffer these instructions before they enter the issue queue, we observe significant reductions in issue queue occupancy and energy. Note that this can provide benefit to any instruction that must wait for their operands in the issue queue, not just those instructions dependent on load misses.

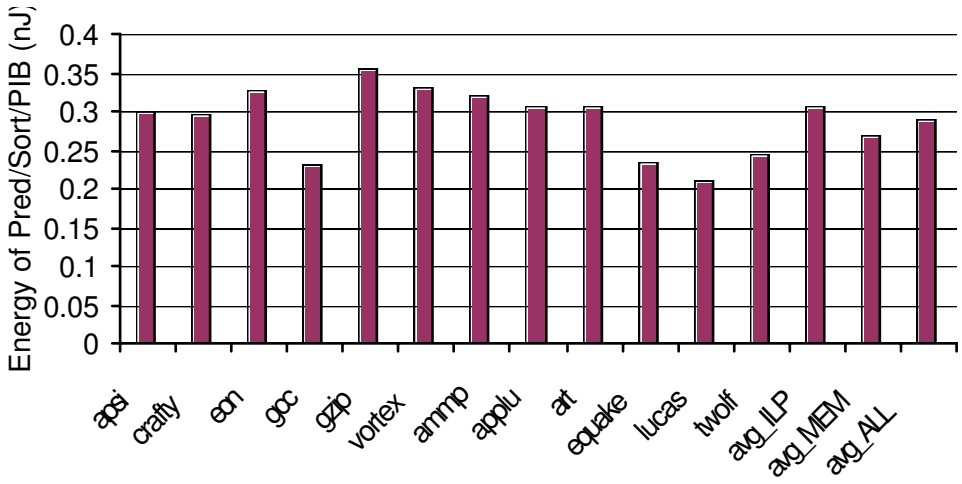

Figure 6.3: Energy Consumption in the Latency Prediction, Sorting and Buffering Structures (per instruction)

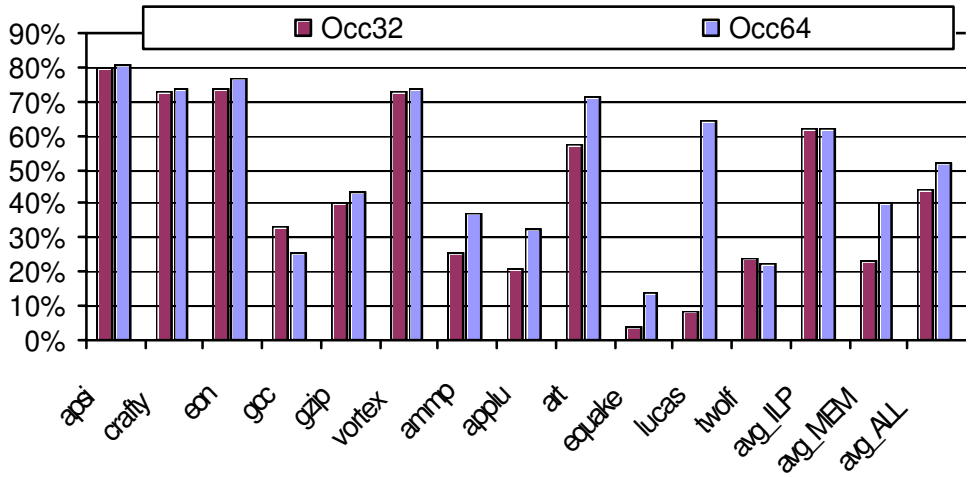

Figure 6.4: Reduction in Issue Queue Occupancy

As shown in Figure 6.4, we observe a 44% reduction in issue queue occupancy with a 32-entry issue queue configuration, and a even larger reduction of 53% with a 64-entry issue queue. The ILP-intensive applications observe larger reductions in issue queue occupancy because they have a larger number of in-flight instructions. The latency prediction and buffering mechanism effectively prevents these instructions from entering the issue queue earlier than necessary. In memory-intensive applications, dependents of missed loads can still enter issue queue early due to the coarser granularity of sorting queues for very long latencies.

Figure 6.5 shows the energy reduction in a 32-entry issue queue configuration. We observe an average of 52% reduction in issue queue energy. The ILP-intensive applications are able to reduce their issue queue energy by nearly two thirds. The memory-intensive applications also observe a large energy savings, around 45%.

The baseline speculative scheduler also suffers from frequent misschedulings in the face of load nondeterminism. Without knowledge of load latencies, the scheduler has to optimistically issue load dependents to assume the load will hit in the cache. When a load misses in the cache, its dependents will have been misspeculated and will need to be rescheduled. Moreover, to avoid exposing the latency of the schedule to execute window, a speculative scheduler will replay the load's dependents at certain intervals until they are able to correctly schedule. This can impact performance in two ways: 1) by coarsening the granularity of load latencies based on the interval at which instructions replay and 2) when misscheduled instructions consume issue bandwidth that could be used to execute useful instructions. This latter component can also dramatically impact energy consumption, particularly in the issue logic and register file.

When we use latency prediction and instruction sorting to implement out-of-

order entry into the scheduling window, we observe a large reduction of scheduling replays (Figure 6.6): the number of replays are reduced by 87% in a 32-entry issue queue configuration and by 90% in a 64-entry issue queue. The reduction comes from two sources: the dependents of missed loads that are correctly latency predicted tend not to be misscheduled because they do not enter the issue queue until they have been buffered in the sorting queues. Even if the dependents of missed loads enter issue queue prematurely, misschedulings are still rare because the scheduler speculates load latency based on the accurately predicted load latencies.

The reduction in scheduling replays translates into energy savings in the register file. As shown in Figure 6.7, the new approach saves energy in the register file by an overall average of 13% – saving 22% in memory-intensive applications and 2% in ILP-intensive applications. Memory-intensive applications observe a much larger energy savings due to a larger amount of cache misses.

6.4.5 Overall Performance and Energy Reduction

When we use latency prediction and instruction sorting to implement out-of-order entry into the scheduling window, we observe a substantial improvement in IPC performance (Figure 6.2). On average, the memory-intensive benchmarks observe a 22% speedup. The benchmark `art` observes the largest improvement (61%). This is because these applications have frequent cache misses. With our proposed techniques, dependents of loads that miss in the cache are prevented from entering the issue queue, while other instructions with shorter waiting times are allowed to enter the issue queue earlier (and out-of-order). Therefore, the scheduler is able to better exploit memory level parallelism. The ILP-intensive benchmarks see less of an improvement because load misses are less frequent. In the case of

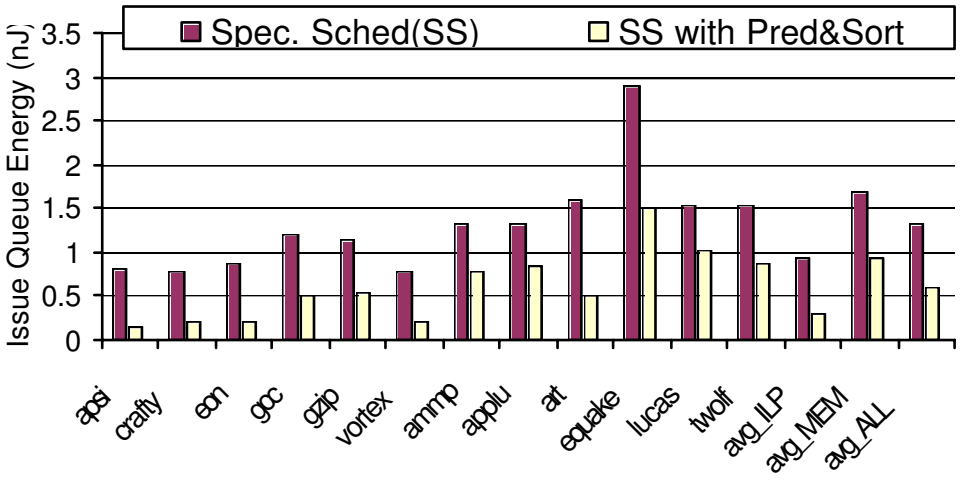

Figure 6.5: Issue Queue Energy Consumption Per Committed Instructions

`apsi`, we even observe a slight degradation of 1.5%. Our results show that `apsi` has relatively few cache misses. For this application, any benefit from latency prediction and sorting is canceled out by the impact of the additional pipe stages to perform latency prediction, sorting and preissue buffering.

The reduction in energy dissipation in the register file and issue logic, combined with the improved utilization of the issue queue, results in an overall drop in energy dissipation. As shown in Figure 6.8, the memory intensive applications observe an average 22% reduction in energy per instruction. In these applications, a large amount of misschedulings are effectively eliminated, and therefore a significant energy reduction is observed. The ILP intensive applications observe a 23% reduction. These applications, though have less energy savings from eliminated replays, have large savings in issue queue energy.

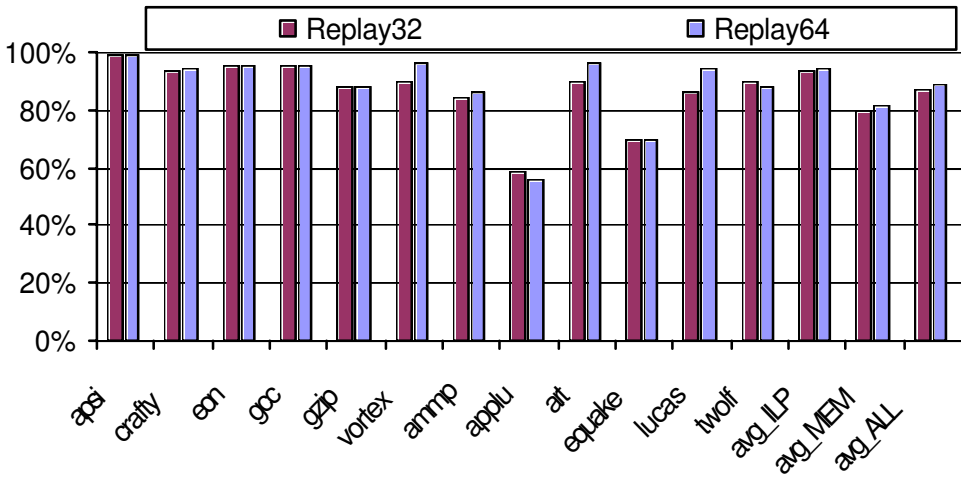

Figure 6.6: Reduction in Number of Scheduling Replays

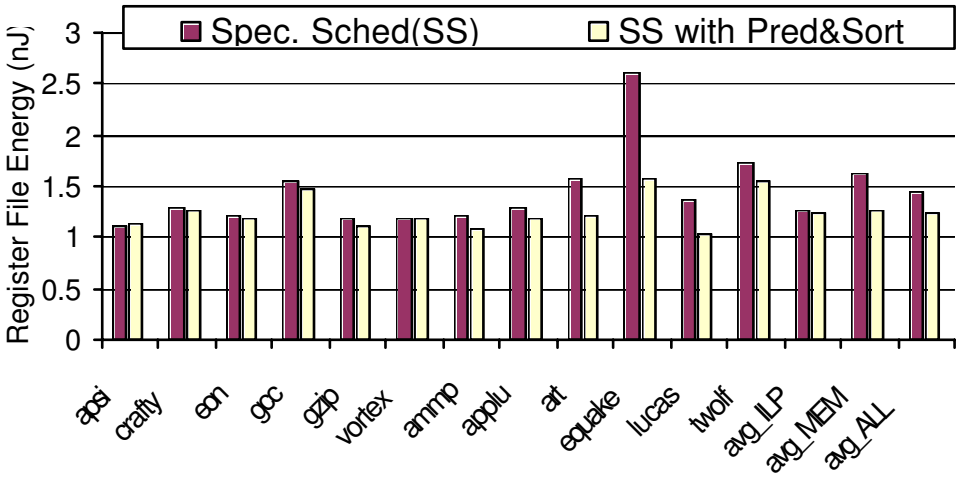

Figure 6.7: Register File Energy Consumption Per Committed Instructions

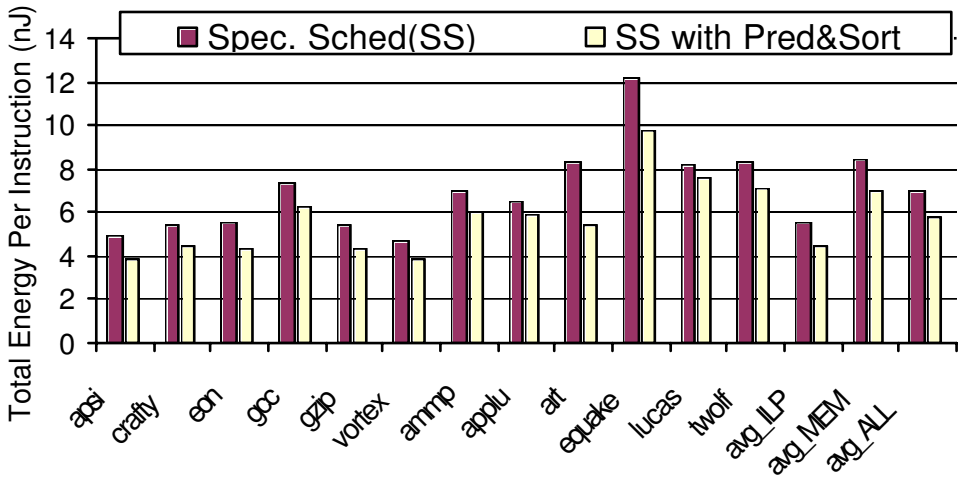

Figure 6.8: Total Energy Consumption Per Committed Instructions

6.5 Summary

In this chapter, we investigate the energy reductions in issue queue and register file accesses using look-ahead load latency prediction and proactive instruction buffering. With predicted latencies, the schedulers can avoid unnecessarily miss-scheduling the dependents of missed loads. With the buffering mechanisms, we prevent instructions from entering issue queue earlier than necessary. Our results show that these savings translate into 52% savings in issue queue power, 13% savings in register file power, and 22% overall energy savings.

CHAPTER 7

Scaling Issue Queue Using 3D IC Technologies

Vertical integration (3D ICs) has demonstrated the potential to reduce inter-block wire latency through flexible block placement and routing. However, there is untapped potential for 3D ICs to reduce intra-block wire latency through architectural designs that can leverage multiple silicon layers in innovative ways. The scheduling logic (issue queue) features multiple read/write ports and compare ports. Ports in such a structure is responsible for a large portion of access time and energy.

In this chapter, we introduce port partitioning to construct multi-layer issue queue. Port partitioning significantly reduces the access time and energy of same size issue queue in 2D. It also scales issue queue to doubled size yet with faster access and slightly increased energy consumption. Similar performance improvements are observed when we further extend this technique to other critical structures with multi-ports such as data cache and register file.

Study of stand-alone 3D issue queue has limited usage if the entire processor configuration is not considered. We plug the 3D issue queue into a recent microprocessor configuration. We extend MEVA-3D, a microarchitectural exploration tool to include the ability to floorplan multi-layer blocks. For this design driver, we see an average 36% improvement in performance (measured in BIPS) over a single layer architecture. The on-chip temperature is kept below 40°C.

The rest of the chapter is organized as follows: We introduce 3D technologies and motivate our study in 7.1. Next, we detail and evaluate our scalable 3D issue queue, then extend the study to other microarchitectural blocks in Section 7.2. Our 3D block placement enhancements are detailed in Section 7.4. We finally explore a design driver microarchitecture in Section 7.5 and then summarize in Section 7.6.

7.1 Introduction to 3D Technologies

Microprocessor architecture design faces greater challenges in the deep submicron era due to technology scaling and an increasing number of available transistors [RAK01]. As a result of the demand for higher performance and more sophisticated functionality, chip area increases continuously, outweighing the effects of feature scaling. Hence, more transistors are packed in higher density on larger wafers. Interconnect delay and complexity have already become major limitations. These problems are expected to worsen unless a paradigm shift occurs [ITR05].

Vertical integration [MDA79, SNT83, YN86, TKY89, KSP01] leverages multiple layers of silicon to allow physical designers more flexibility in component layout. One approach to using this technology is to place single-layer (i.e. 2D) blocks in one of the silicon layers and running both horizontal and vertical interconnect between blocks. The flexibility that this design affords has the potential to dramatically reduce inter-block interconnect latency in a design [DCR03, BSK01, BNW04, CJM06].

However, this approach does little to help intra-block wire latency. And despite the advantage of almost completely eliminating inter-block wire latency, we

find that the placement of 2D blocks in two layers improves performance by 6% on average for a particular architecture (described in section 7.5). Additional gains from the use of vertical integration must attack the intra-block wire latency.

Furthermore, the emergence of technology like vertical integration can have a dramatic impact on microarchitecture design – a field that is heavily reliant on physical planning and technological innovation. However, physical planning is not meaningful without consideration for microarchitectural loop sensitivities: some loose loops [BTM02] are better able to tolerate latency than others [SC02]. A floorplan with a 5% reduction in wirelength may actually be better than a floorplan with a 7% reduction in wirelength – if the former reduces the length of more critical microarchitectural loops than the latter. Similarly, architectural innovations are not meaningful without understanding their physical design implications.

Recently, the MEVA-3D [CJM06] framework was proposed to bridge the gap between physical planning and microarchitectural design. The framework uses microarchitectural loop sensitivities in the floorplanning process to guide block placement. With this framework, architects can obtain accurate loop latencies to feed to a cycle-accurate simulation framework. This can help evaluate the impact of new and emerging technologies on microprocessor design.

In this chapter, we explore the architectural impact and potential of finer granularity vertical integration, where individual blocks are placed across multiple layers. The challenge from the architectural side is the construction of blocks that can span multiple layers. The challenge for physical design is to automate the process of placing blocks in multiple layers.

To address these challenges, we make the following contributions:

- 3D Architectural Blocks: We propose *port partitioning*, an approach to place architectural blocks like register files, issue queues, and caches in multiple silicon layers. We compare port partitioning with wordline/bitline partitioning [TXV05] with respect to area, timing, power, and required vertical interconnect.

- 3D Microarchitectural and Physical Design Co-Optimization: We extend the MEVA framework [CJM06] to handle fine-grain 3D exploration. Our modified framework can automatically choose between 2D and 3D implementations of a given block. Given a frequency target, an architectural netlist, and a pool of alternative block implementations, this framework can find the best solution in terms of performance (in BIPS), temperature, or both.

- 3D Design Driver Exploration: Using our modified framework, we explore the design space of different partitioning schemes for a particular design driver architecture, using one to four layers of silicon. In addition to exploring the use of single layer and multilayer blocks, we consider growing the sizes of different architectural structures, using the timing slack from vertical integration. In some cases, the timing slack can enable the use of larger instruction or scheduling windows, or larger caches.

In addition to helping latency, this reduction in wire RC delay can reduce power dissipation. However, the stacking of components can adversely impact the temperature of the microprocessor. It is therefore essential for any study using vertical integration to make use of accurate temperature modeling to demonstrate the effectiveness of any architecture. All of our explorations are enhanced with a state-of-the-art, accurate, temperature simulator tool. We also consider

automated thermal via insertion to help mitigate the impact of temperature.

7.2 Scaling Issue Queue in 3D

To reduce intra-block interconnect latency, we evaluate two main strategies for designing blocks in multiple silicon layers: *block folding* and *port partitioning*. Block folding implies either a vertical or horizontal folding of the block - potentially shortening the wire-length in one direction. Port partitioning places the access ports of a structure in different layers - the intuition here is that the additional hardware needed for replicated access to a single block entry (i.e. a multi-ported cache) can be distributed in different layers, which can greatly reduce the length of interconnect within each layer. In this section, we describe the use of these strategies for scaling the issue queues and in the next section, we extend the study to various cache-like blocks in our design driver architecture.

7.2.1 Issue Queues

The issue queue is a critical component of out-of-order microprocessor performance and power consumption. Recent research [CJM06] has shown that every additional pipeline stage of latency seen in the scheduling loop causes an average 5% performance degradation. Moreover, Folegnani and Gonzalez [FG01] have found that the issue queue is responsible for an average 25% of a processor's total power consumption.

The issue queue stores renamed instructions and performs out-of-order instruction scheduling. The issue queue we studied in this dissertation is based on Palacharla's implementation [PJS97]. There are two main stages of issue queue functionality: the wakeup stage where tags from completing register values are

compared against input register tags stored in issue queue entries, and a selection stage where ready instructions (as determined by the wakeup stage) are selected for execution.

Each issue queue entry must track and compare the input register tags required by a given instruction in that entry. Figure 7.1 shows a single CAM cell used to store one bit of a register tag for an issue queue entry. Assuming that at most four register values can be written back each cycle, and at most four new instructions can enter the issue queue each cycle, an individual cell would have four different 1-bit tags to compare against and have four write ports. In a processor with a 128-entry physical register file, register tags are 7-bits. Therefore each row would need seven CAM cells for each operand, for a total of fourteen CAM cells. In general, an n-entry issue queue has n such rows.

In the wakeup stage, the match lines for each issue queue entry are precharged high and the tag lines are driven with the register tags of completed instructions. A match line only remains high if the register tag stored at the issue queue entry is the same as a certain one of the register tags driven on the tag lines. If any match line for a given input register remains high, the ready bit for that operand is set in the issue queue. Once both ready bits are set, the operand is eligible for issue (i.e. has woken up). In this stage, most of the delay comes from tag broadcasting and matching.

In the selection stage, the select logic picks instructions to execute [PJS97] among all instructions that are eligible for issue.

For example, a selection tree for a 32-entry issue queue consists of three levels of arbiters. Each arbiter takes four input requests (i.e. four eligible instructions) and grants one request (i.e. selects one eligible instruction). In general, an N-

entry issue queue needs a selection tree of level $L = log_4 N$.

In the issue queue, the delay due to wakeup logic contributes a large portion of the overall delay. Our simulations show that wakeup takes about 60% of the delay in a 32-entry issue queue with four incoming register tags to compare against, and four access ports. A significant contributor to delay is the wire latency of the tag bits and match lines. A 3D integrated issue queue can significantly reduce the length of these wires.

7.2.2 3D IQ Design: Block Folding

One way to reduce tag line wire delay is to fold the issue queue entries and place them on different layers. Figure 7.2 (a) shows a single layer issue queue with four incoming register tags that are compared against entries in the issue queue. In Figure 7.2(b), the issue queue is folded into two sets and they are stacked in two layers. This approach effectively shortens the tag lines.

7.2.3 3D IQ Design: Port Partitioning

In an issue queue with four tag comparison ports and four read/write ports, as shown in Figure 7.1(a), most of the silicon area is allocated to ports. The wire pitch is typically five times the feature size [RJ, SJ01, PJS97]. For each extra port, the wire length in both X and Y directions is increased by twice the wire pitch [RJ, SJ01]. On the other hand, the storage, which consists of 4 transistors, is twice the wire pitch in height, and has a width equal to the wire pitch. Hence, in a cell as shown in Figure 7.1(a), the storage area is less than 1% of the total area, while tags and access ports occupy over 99% of the total area.

One strategy to attack the tag and port requirements is port partitioning,

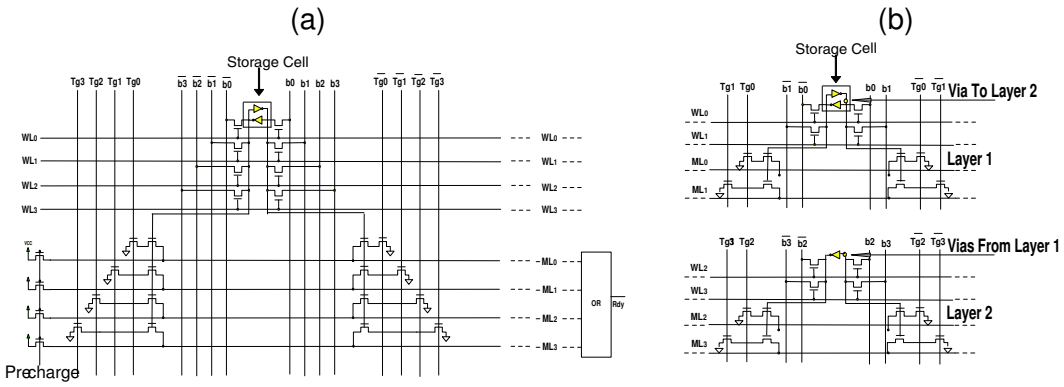

Figure 7.1: (a): A Single IQ Cell with Four Tag Lines and Four Access Ports. Over 99% of the area is occupied by tags and access ports.

(b): Port Partitioning. Tags and access ports are distributed into two layers. Width and height of each bit are reduced by half, and area by 75%.

which places tag lines and ports on multiple layers, thus reducing both the height and width of the issue queue. The reduction in tag and matchline wire length can help reduce both power and delay. The selection logic also benefits from this, as the distance from the furthest issue queue entry to the arbiter is reduced. This will speed up the comparison and also reduce power consumption.

7.2.4 Modeling Methodology

We use Hspice to model issue queue. We assume a supply voltage of $1.0V$ and a $70nm$ process technology. Transistor and wire scaling parameters are derived from [TXV05, MF95], and we assume copper interconnect in our simulation. Further transistor parameters are obtained from [CSS00]. The 3D via resistance is estimated to be $10^{-8}\Omega cm^2$ [TXV05]. The height of the 3D vias is assumed to be $10\mu m$ per device layer. Current dimensions of 3D via sizes vary from $1\mu m \times 1\mu m$

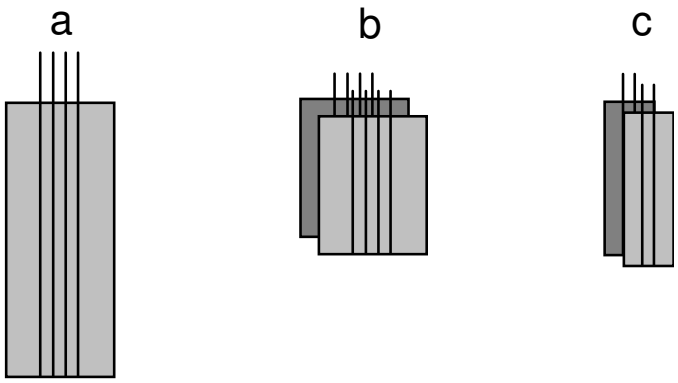

Figure 7.2: Issue Queue Partitioning Alternatives: (a) An issue queue with 4 tag lines. (b) Block Folding: dividing the issue queue entries into two sets and stacking them. The tags are duplicated in every layer. Only the X-direction length is reduced. (c) Port Partitioning: the four tags are divided into two tags on each layer. Both X and Y direction lengths are reduced.

	F2F, 2L(%)		F2B, 2L(%)		F2B, 3L(%)		F2B, 4L(%)	
	BF	PP	BF	PP	BF	PP	BF	PP
Delay	26.9	36.8	23.7	30.3	33.1	43.2	39.0	49.4
Area	20.2	73.1	17.2	70.0	34.5	80.7	42.2	90.0
Power	-2.5	28.8	-2.9	24.3	-8.1	33.3	-8.5	38.4

Table 7.1: Percentages of Reduction in Delay, Area and Power Consumption from 3D Design. Symbols: "nL" – n number of Layers, "F" – Folding, "PP" –Port/Tag Partitioning

to $10\mu m \times 10\mu m$ [TXV05, DFC04]. As 3D technology advances, the 3D via size will decrease even further. In this study, we assume the via pitch is $1.4\mu m$. An area of $0.7\mu m \times 0.7\mu m$ is reserved for each 3D via for the upper layers in F2B technology.

7.2.5 3D Issue Queue Performance

Table 7.1 shows the percent reduction in delay, area, and power consumption using 3D techniques.

In general, 3D stacking significantly reduces both delay and area. The energy reduction is also significant using port partitioning(PP). We observe a maximum delay reduction of 50%, a maximum area reduction of 90%, and a maximum reduction in power consumption of 40%.

For example, in the 2 layer F2F implementation of block folding (BF), delay is reduced by 27%. Port partitioning (PP) sees even more improvement (37% reduction in delay). PP reduces both tag wire lengths and match wire lengths, and wire lengths to the selection logic. On the other hand, BF only reduces tag

wire lengths. The match wire lengths are even increased due to 3D via insertions for every tag and bit line. As a result, we observe over 70% reduction in area for PP, with only a 20% reduction for BF. Note that the area shown is the maximal area in any one layer for that block, and while the footprint of the block may be reduced, the sum of the area occupied in all layers may actually increase relative to the 2D baseline.

The power consumed in CMOS circuits is represented as $P = 0.5 * a * f * C * Vdd^2$, where f is the clock frequency, a is the activity factor, Vdd is the supply voltage and C is the switching capacitance. The power consumption rate is proportional to the switching capacitance. In BF, although tag wire lengths for each layer are reduced, the tag wires are duplicated on different layers. The aggregate wire length is still the same. In addition, there is an increase in match line lengths mentioned above. Thus, the total switching capacitance is slightly increased due to the increased total wire length. As a result of this, the power consumption of BF is slightly increased (Table 7.1). On the other hand, PP is able to reduce power consumption by 29%.

7.2.6 Scaling to Multiple Silicon Layers

For a dual layer implementation, F2F is able to outperform F2B since the 3D vias in F2B impact the silicon footprint in the top silicon layers. For example in the PP results, the F2B area is about 5% larger than that of F2F due to the increased silicon footprint. The delay and power consumption are larger than those of F2F as well. However, F2B allows more layers to be stacked. It may be possible to stack two F2Fs in back to back fashion; however, we do not consider this alternative in this dissertation.

Table 7.1 also shows timing, power, and area results with F2B blocks for two, three, and four layers of silicon. All measurements are normalized to the performance of a single layer block. In general, we observe that the reduction of area, power and delay is further increased as the number of layers is increased. Furthermore, PP consistently outperforms BF.

For the issue queue (IQ) with PP, area reduction increases to 80% with 3 layers, and to 90% with 4 layers. Reduction in issue queue delay increases to 43% with 3 layers, and to 50% with 4 layers. Reduction in power consumption grows as high as 38% with 4 layers.

For the issue queue with block folding, there is less reduction in area and delay with additional layers. However, the impact on match line wire length from stacking more layers increases the power consumption for folding to 9% with 4 layers.

7.2.7 Issue Queue Sizes

Figure 7.3 shows the timing performance when the issue queue is scaled from the default size to 16 times larger. As shown in the figure, using 3D integration technology, the access latency of double-sized structures is still less than in 2D. Hence, the 3D IC technology effectively enables issue queue scaling.

7.3 Extended Studies

Regular structures that are consists of storage arrays can also benefit from the same scaling techniques described for issue queue. We therefore extend our studies to structures such as Caches, Register Files etc.

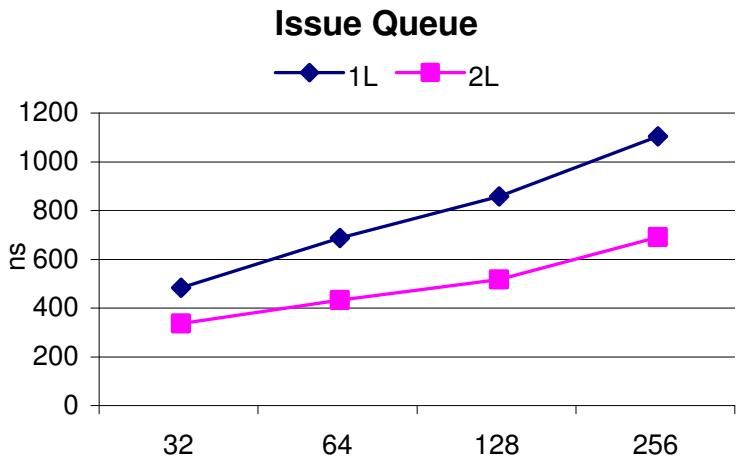

Figure 7.3: Performance of Scaled Issue Queue in 3D

In Section 7.3.6, we also extend our studies on the performance trends when 3D bonding technology advances.

7.3.1 Caches

Caches are commonly found architectural blocks with regular structures - they are composed of a number of tag and data arrays. Figure 7.4(a) demonstrates a high level view of a number of cache tag and data arrays connected via address and data buses. Each vertical and horizontal line represents a 32-bit bus – we assume two ports on this cache, and therefore the lines are paired. Each box of the figure is a tag or data array, which is composed of a mesh of horizontal wordlines and vertical bitlines. Every port must have a wordline for each cache set and a pair bitlines for each bit in a cache set. The regularity of caches means that their components can easily be subdivided – the tag and data arrays for example can easily be broken down into subarrays. We make use of CACTI [SJ01] to explore the design space of different subdivisions and find an optimal point for

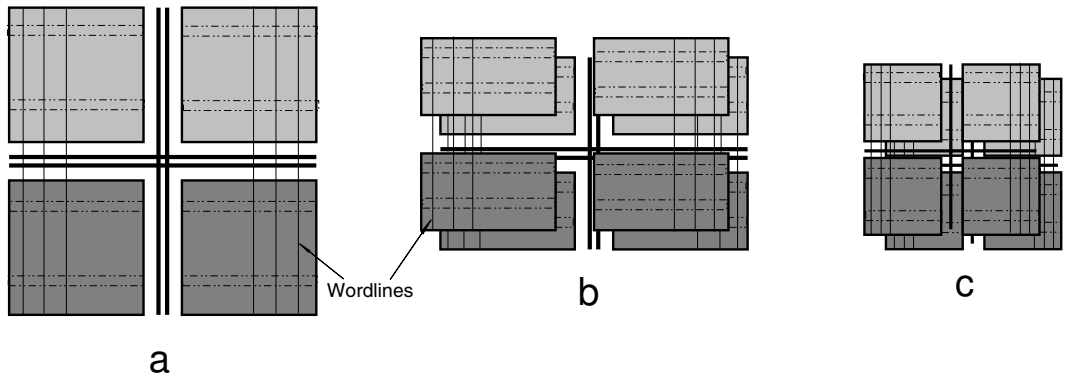

Figure 7.4: Cache Block Alternatives (a) A 2-Ported Cache: the two lines denote the input/output wires of two ports. (b) Wordline Folding: Only Y-direction length is reduced. Input/output of the ports are duplicated. (c) Port Partitioning: Ports are placed in two layers. Both X and Y direction length are reduced.

performance, power, and area.

7.3.1.1 3D Cache Design: Block Folding

Prior research [TXV05] looks into two folding options: wordline and bitline folding. In the former, the wordlines in a cache sub-array are divided and placed onto different silicon layers. The wordline driver is also duplicated. The gain from wordline folding comes from the shortened routing distance from predecoder to decoder and from output drivers to the edge of the cache.

Similarly, bitline folding places bitlines into different layers. This approach needs to duplicate the pass transistor. The sense amplifier can be duplicated to improve timing performance at a cost of increased power consumption. The cost is significant because sense amplifiers can make up a significant portion of total cache energy consumption. The other approach is to share sense amplifiers across

layers, but this dramatically reduces the improvement in timing.

Our investigation shows that wordline folding has a better access time and lower power dissipation in most cases compared with a realistic implementation using bitline folding. In this dissertation, we only present results using wordline folding.

7.3.1.2 3D Cache Design: Port Partitioning

The port partitioning strategy that we proposed for the issue queue can also be leveraged for caches. For example, a 3-ported structure would have a port area to cell area ratio of approximately 18:1. Hence, there is a significant advantage to partitioning the ports and placing them onto different layers. In a two layer design, we can place two ports on one layer, one port and the SRAM cells on the other layers. The width and height are both approximately reduced by a factor of two, and the area by a factor of four.

7.3.2 Other Cache-Like Architectural Blocks

Register files are similar to caches, sharing the regularity of a cache. We therefore adapt our CACTI to model this structure as well. However, they are not associative and typically have more ports than caches do. Register files dissipate relatively large amounts of power due to their porting requirements, and the size of the physical register file can constrain the size of the instruction window in a dynamically scheduled superscalar processor. We will consider the same folding schemes for the register files as we used for caches.

The register mapping units, load-store queue, and branch predictors can be approximated using only the data array portion of the cache.

	DCache		Reg. File	
	Hspice	Cacti	Hspice	Cacti
Timing	74.7%	77.0%	72.3%	72.9%
Power	93.6%	89.5%	83.7%	83.3%

Table 7.2: Reduction in delay and energy obtained from HSpice and modified 3DCACTI and HSpice as compared to 2D blocks

We implemented our issue queue models using HSpice to obtain accurate timing and power data. The area of the issue queue is approximated by 3D-CACTI using a similarly sized cache. Our 2D issue queue is derived from Palacharla et al's model [PJS97].

7.3.3 Modeling Methodology

We have modified 3D-CACTI [TXV05] to model caches and cache-like structures. First, we add port partitioning to 3D-CACTI in addition to wordline/bitline folding. Second, we add area estimation, including the area impact of 3D vias on the transistor layer. Both 3D bonding technologies are available: F2B and F2F. We validated our modifications to 3D-CACTI with HSpice. In table 7.2, we show the comparisons of HSpice simulation results with 3D-CACTI. The savings in delay and power using 3D F2F technology are compared. We observe that the delay and power savings predicted by our modified 3D-CACTI are similar to the actual design.

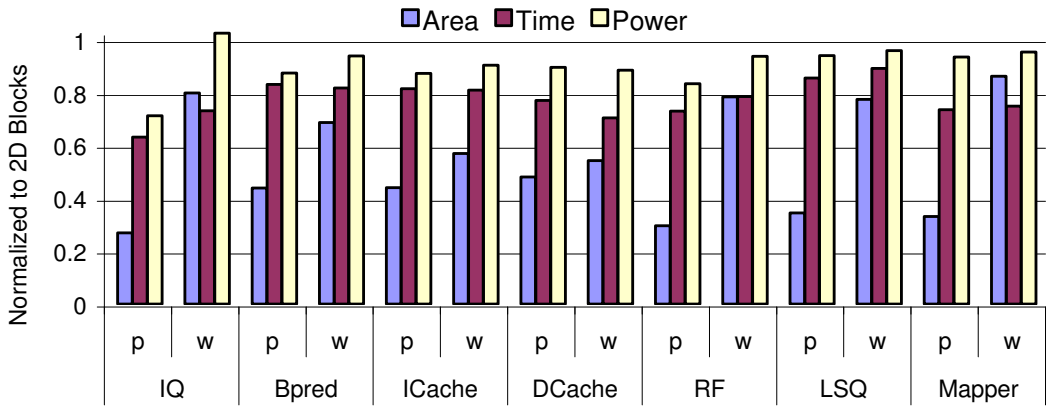

Figure 7.5: The improvement in area, power and timing for dual layer vertical integration.

7.3.4 3D Block Performance

Figure 7.5 demonstrates the effectiveness of 3D block design on area, power, and timing for dual layer F2F blocks. We also include the performance data from issue queue for a comparison with cache and cache-like blocks. The y-axis is normalized to the area of a single layer baseline block. The x-axis represents different folding techniques for each architectural block investigated. The letters in the label of a bar represents the type of folding: either port partitioning (PP) or block folding (BF). All results are shown normalized to the 2D implementation of the block. In F2F technology, the via starts from the surface of one layer and ends on the surface of the other layer. Therefore, vias do not impact the layout of transistors.

For the caches and cache-like structures, PP is extremely effective in heavily ported structures. For example, the register file with PP sees a 27% reduction in delay, a 17% reduction in power, and an impressive 70% reduction in area.

However, for structures with fewer ports, BF can be more effective. The data cache sees a 30% reduction in delay with BF, and a 23% reduction in delay with PP. While PP does reduce both wordline and bitline length, this reduction is proportional to the number of ports that can be partitioned to other silicon layers. For structures with very few ports, BF is able to reduce wordline length more than PP. Hence in structures that have significant wordline delay, the overall reduction in delay with BF can be greater than PP.

The diversity in benefit from these two approaches demonstrates the need for a tool to flexibly choose the appropriate implementation based on the constraints of an individual floorplan.

7.3.5 Scaling to Multiple Silicon Layers

Figures 7.7, 7.8, and 7.6 show timing, power, and area results (respectively) with F2B blocks for two, three, and four layers of silicon. All measurements are normalized to the performance of a single layer block. In general, we observe the cache and cache-like blocks have more reductions of area, power and delay as the number of layers is increased.

7.3.6 Impact of 3D Bonding Technology

3D via size has rapidly scaled down as 3D bonding technology has advanced. 3D via size has reduced from $10\mu m$ to $1.75\mu m$ in MIT Lincoln Laboratory's 3D process technology [Lab06] at $180nm$. We expect the 3D via size to continue to scale at smaller feature sizes. In this dissertation, we have assumed a $0.7\mu m$ via size for a $70nm$ feature size.

To demonstrate the impact of scaling via size, we plot the performance of

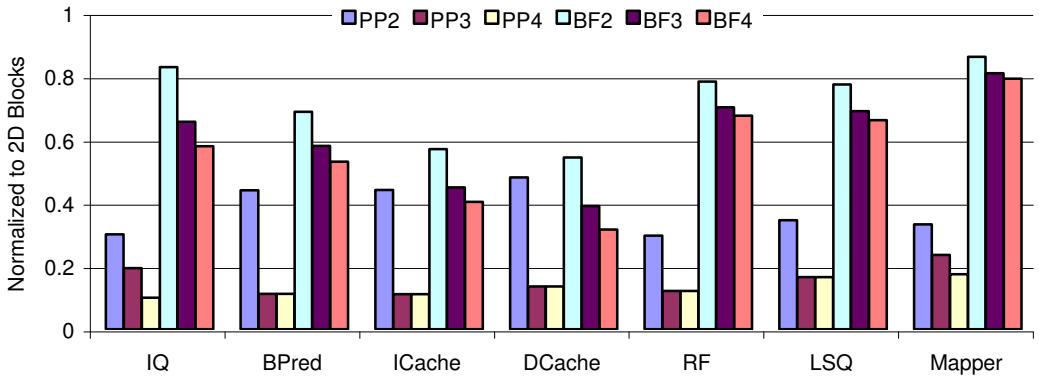

Figure 7.6: The improvement in area for multilayer F2B vertical integration.

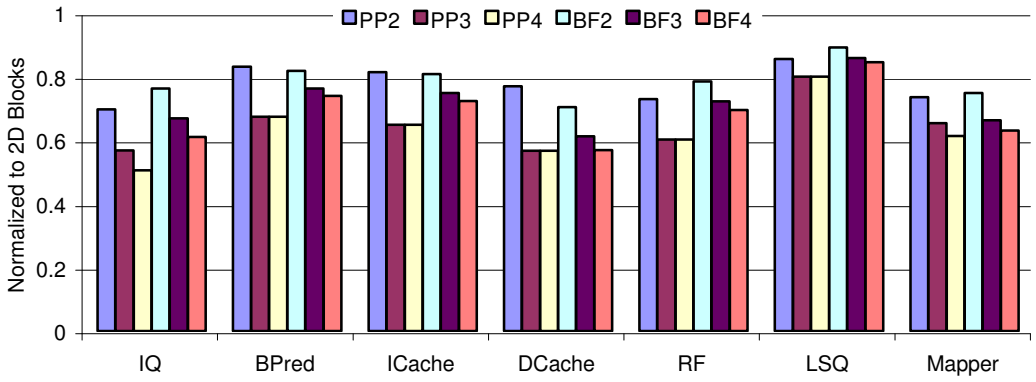

Figure 7.7: The improvement in timing for multilayer F2B vertical integration.

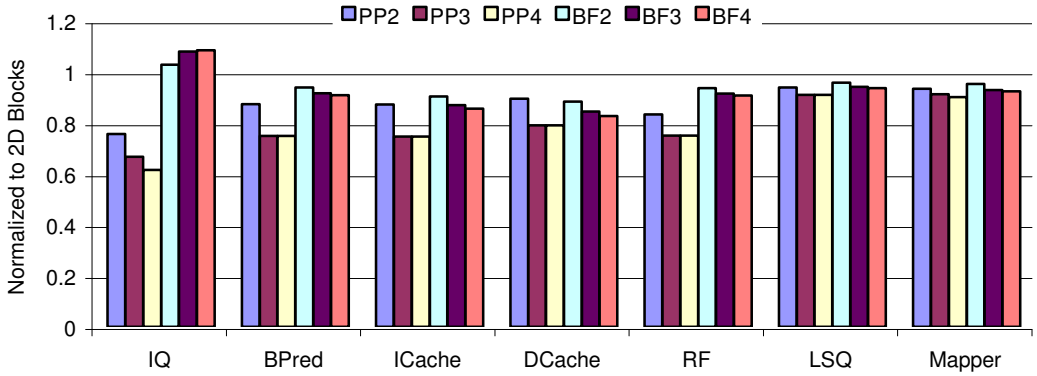

Figure 7.8: The improvement in power for multilayer F2B vertical integration.

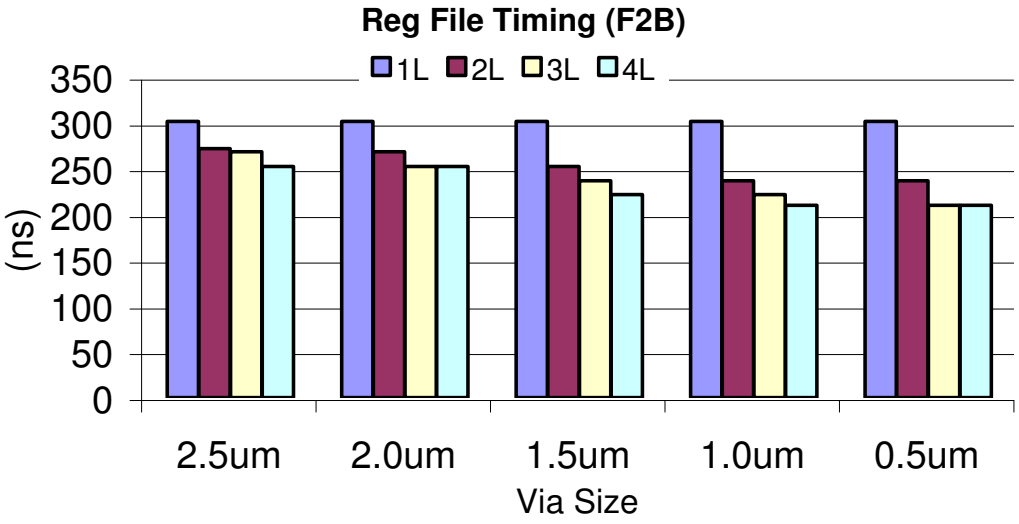

Figure 7.9: Impact of Via Size on Timing using F2B, Port Partitioning

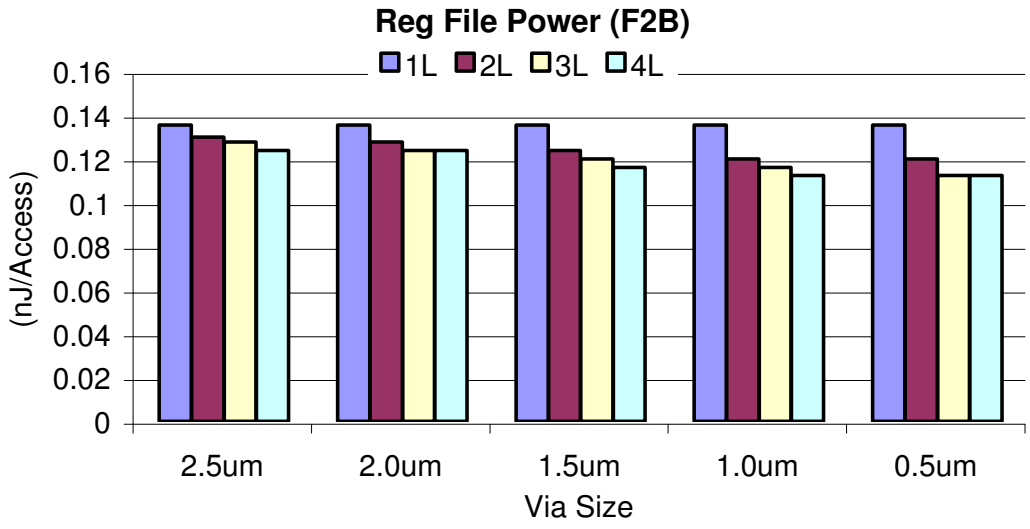

Figure 7.10: Impact of Via Size in Power using F2B, Port Partitioning

the register file for via sizes ranging from $2.5\mu m$ to $0.5\mu m$ in $70nm$ technology. The via pitch is twice the via size. The register file has four read ports and four write ports. A single cell size is approximately $5.6\mu m$ x $5.6\mu m$. In F2B bonding technology, 3D vias occupy silicon area in all layers except for the bottom layer. Taking 2-Layer partitioning as an example, when via size is $2.5\mu m$, the best solution is to place seven ports in the bottom layer, and one port in the top layer, which only slightly reduces the wire-length. When the via size is scaled to $0.5\mu m$, the best solution places four ports in each layer. The wire-length is almost cut in half in both X and Y directions. As shown in Figures 7.9 and 7.10, the larger reduction in wire-length reduces both delay and power as the via size is scaled from $2.5\mu m$ to $0.5\mu m$.

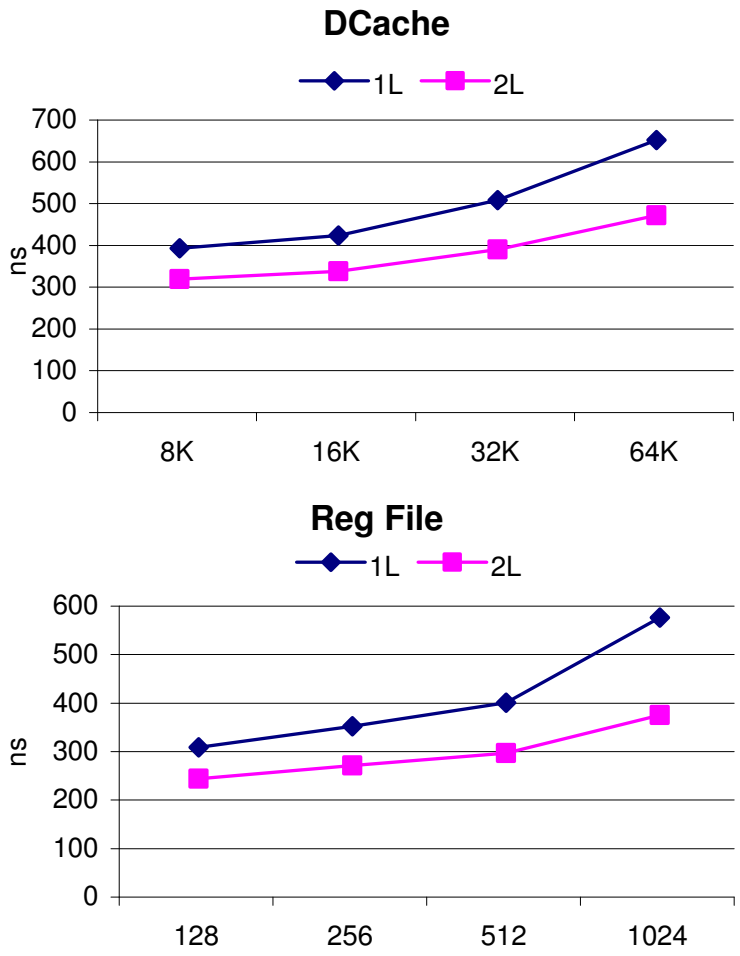

Figure 7.11: Latency Impact of Vertical Integration when Scaling the Size of Two Critical Blocks

7.3.7 Structure Sizes

Figure 7.11 shows the timing performance when register file and L1 data cache scaled from the default size to 16 times larger. As shown in the figure, using 3D integration technology, the access latency of double-sized structures is still less than in 2D. These two structures can even quadruple their sizes while still outperforming the default blocks in 2D. In this dissertation, we limit our study to doubled sizes.

7.4 Placement of 3D Issue Queue and Other Blocks

Microprocessor throughput, as measured in IPC, is influenced by the latency of critical architectural loops such as the scheduling loop, branch resolution loop, inter-cluster communication loop, etc [SC02]. Vertical integration can help to reduce the latency of these critical loops. Critical loops differ in the magnitude of their impact on throughput, and therefore the exploration of the use of vertical integration on microprocessor design requires consideration for both physical design and architecture. Existing work on this type of co-design exploration [CJM06] has only explored the use of vertical integration to reduce inter-block latency in these critical loops. However, as demonstrated in section 7.2, there is tremendous potential for vertical integration to reduce the latency of blocks along critical loops. In this section, we detail our modifications to the co-design framework of [CJM06].

7.4.1 MEVA-3D Flow

MEVA-3D [CJM06] is an automated physical design and architecture performance estimation flow for 3D architectural evaluation which includes 3D floorplanning, routing, interconnect pipelining, automated thermal via insertion, and associated die size, performance, and thermal modeling capabilities.

First, MEVA-3D takes a microarchitectural configuration, a target frequency, architectural critical path sensitivities, and power density estimates and uses 2D/3D floorplanning to optimize for performance and temperature. Then routing and thermal via planning are performed to provide physical design information to our microprocessor simulation. Critical loop latencies are passed from the floorplanner to the simulator for accurate determination of performance. MEVA-3D makes use of the SimpleScalar [BA97] simulator to obtain performance in IPC and utilization counts of individual blocks.

7.4.2 Enhancements to MEVA-3D

MEVA-3D currently only considers 2D architectural blocks. We make the following modifications to extend it to 3D blocks. In the following section, we will make use of this modified framework to explore an architectural design driver.

7.4.2.1 Architectural Alternative Selection

3D component design gives us different configurations for each component: the number of layers that the component will occupy. When we choose another configuration for a component, the dimensions, timing characteristics, and power values change as well, which usually results in a significant change to the floorplan. In order to explore the combinations of the different configurations, we

introduce a new type of move in the optimization approach, called architecture alternative selection. When a new configuration is selected, the 2D dimensions, layer information, and delay information are updated. Accordingly, the distribution of the power, including leakage, is updated for all blocks in the design. The new packing result may be accepted or rejected depending on the cost function evaluation.

7.4.2.2 Cube Packing Engine

Because of the limitation of the packing engine used in MEVA-3D, each component can only occupy one layer, and therefore 3D components are not allowed in the original MEVA-3D flow. To enable the packing of 3D components which may occupy more than one layer, we constructed a new packing engine which is a true 3D packing engine – 3D components in our design can be treated as cubic blocks to be packed in 3D space. The dimension of the block in the Z direction represents the layer information. The 3D packing algorithm is extended from the CBL floorplanner [MHS05].

Our 3D CBL(3-Dimensional Corner Block List) packing system represents the topological relationship between cubic blocks with a triple (S,L,T), where each element is a list. List S records the packing sequence by block names. List L and T represent the topological relationship between cubic blocks in terms of covering other packed blocks.

We use simulated annealing to optimize the cubic packing. The number of layers is given as a constraint on the maximal height in the Z direction of the packing. We extended the floorplanner to optimize chip area, performance (using microarchitectural loop sensitivities), and temperature at the same time.

Processor Width	6-way out-of-order superscalar, two integer execution clusters
Register Files	128 entry integer (two replicated files), 128 entry FP
Data Cache	8KB 4-way set associative, 64B blocksize
Instruction Cache	128KB 2-way set associative, 32B blocksize
L2 Cache	4 banks, each 128KB 8-way set associative, 128B blocksize
Branch Predictor	8K entry gshare and a 1K entry, 4-way BTB
Functional Units	2 IntALU + 1 Int MULT/DIV in each of two clusters 1 FPALU and 1 MULT/DIV

Table 7.3: Architectural parameters for the design driver used in this study.

7.5 Microarchitectural Exploration

In this section, we use the modified MEVA framework to investigate the ability of vertical integration to reduce both intra-block and inter-block architectural latencies.

We constructed a design driver based loosely on the Alpha 21264 [KMW98], and along with the architectural blocks from Section 7.2 (functional unit blocks are based on [CJM06]), we feed this driver into our modified version of MEVA-3D. The architectural parameters are shown in Table 7.3. We measure architectural performance on all 26 programs of the SPEC CPU2000 suite.

Figure 7.12 presents performance results relative to a single layer design driver. The first bar represents the benefit from using two layers of silicon with 2D blocks (as in [CJM06]) and the second bar represents the benefit from using two layers of silicon with 3D blocks. All three configurations (single layer, dual layer 2D blocks, dual layer 3D blocks) are running at 4GHz. On average, the use of 2D blocks in a two layer design improves performance by 6%. Since the blocks themselves do not take advantage of vertical integration, any performance gain can only

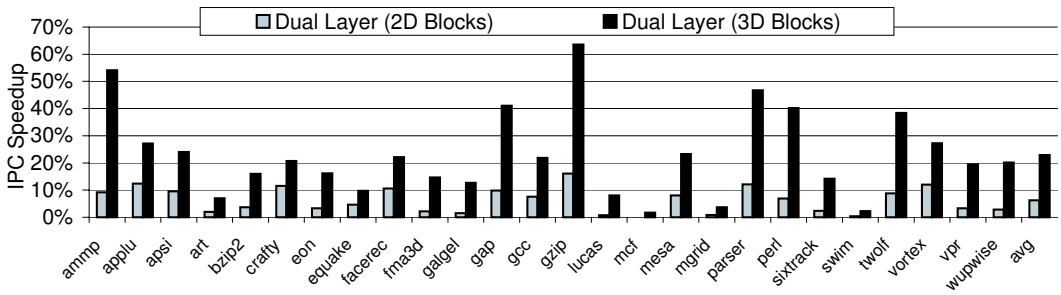

Figure 7.12: Performance speedup for dual silicon layer architectures relative to a single layer architecture.

come from a reduction in the inter-block wire latency. For example, the branch misprediction loop has a total latency of 815ps at 4GHz for a single layer design – 238ps of this total latency is from inter-block wire delay. When using 2D blocks in two layers, this inter-block wire delay is reduced to only 63ps. However, the overall reduction in path delay is not enough to reduce the loop by a cycle of our 4GHz clock. Thus, while timing slack is certainly increased, the benefit of this has not been exploited in Figure 7.12. When we allow MEVA-3D to select 3D block alternatives, we see a performance improvement of 23% on average over the single layer architecture. This can be attributed to the ability of 3D blocks to reduce the intra-block latency of critical processor loops.

We show floorplans for all three architectures in Figure 7.13, 7.14 and 7.15. The single layer design occupies $3.4 \times 3.4 mm^2$ in one silicon layer. The dual layer design with 2D blocks occupies $2.8 \times 2.8 mm^2$ in each silicon layer. The dual layer design with 3D blocks occupies $2.3 \times 2.3\ mm^2$ in each silicon layer.

Temperature issues are considered to be a major concern for vertical integration. Therefore, an accurate and fast thermal simulation framework was very crucial for our experimental analysis. We used the finite element method (FEM)

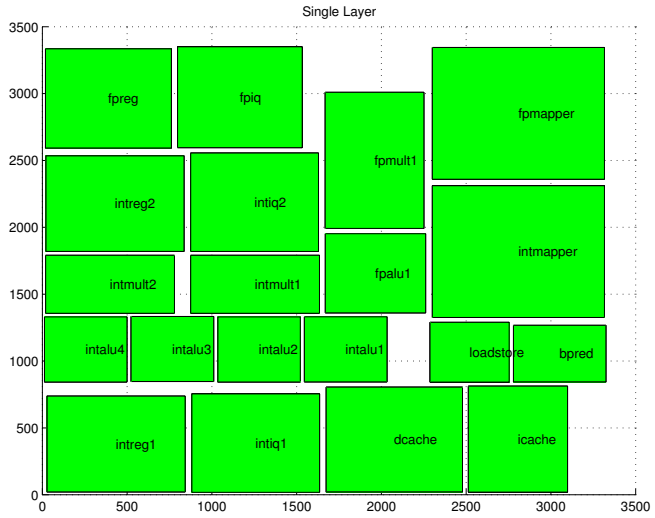

Figure 7.13: Floorplans for a single Layer Architecture (the best architectural configuration as determined by our modified version of MEVA-3D)

based CFD-ACE+ temperature simulator [PAM04]. Further details on heat sink and thermal parameters we used can be found in [PAM04]. Figure 7.16 presents the average core temperature for the single layer architecture (shown at left) and the dual layer architecture with 3D blocks (the hottest layer is shown at right). The average and maximum temperature for the single layer architecture was 30.6°C and 32.7°C. The average and maximum temperature for the dual layer architecture with 2D blocks was 30.6°C and 32.6°C. The average and maximum temperature for the dual layer architecture with 3D blocks was 30.3°C and 34.1°C.

Thermal vias can help to relieve thermal problems in 3D microarchitectures. We used the algorithm proposed in [JY05] for thermal via insertion. In our multi-layer designs, we designate 5% of the area as dead space on each layer, which provides sufficient space for thermal vias.

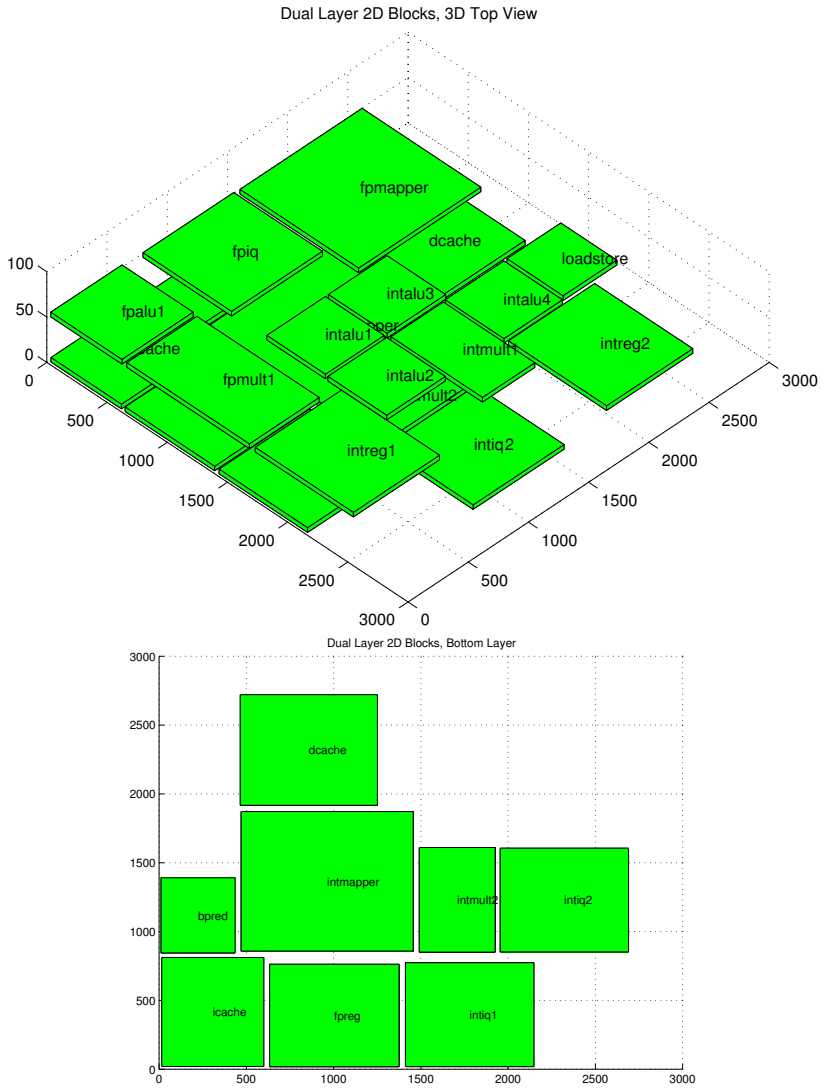

Figure 7.14: Floorplans for Dual Layer Architecture with 2D-only Blocks (the best architectural configuration as determined by our modified version of MEVA-3D)

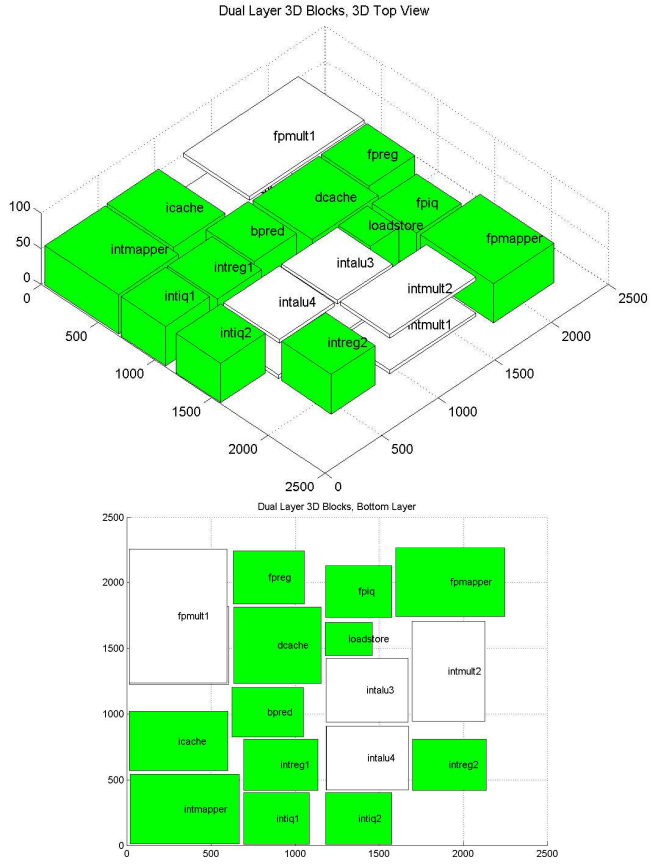

Figure 7.15: Floorplans for Dual Layer Architecture with 2D and 3D Blocks (the best architectural configuration as determined by our modified version of MEVA-3D)

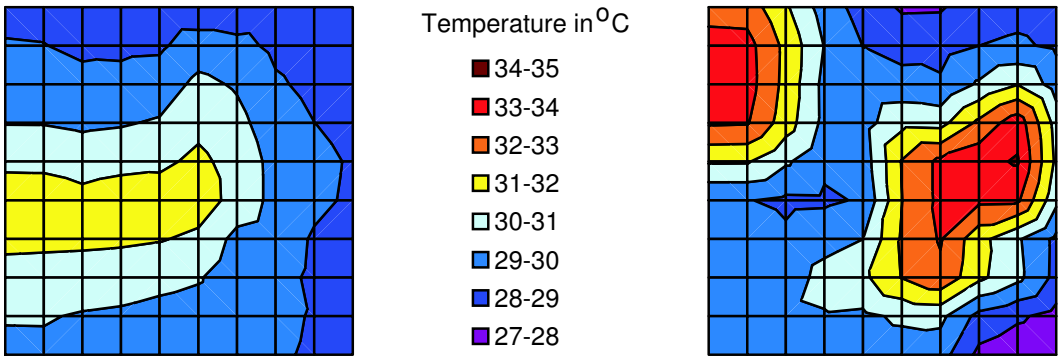

Figure 7.16: Average core temperature for the single layer architecture (shown at left) and the dual layer architecture with 3D blocks (the hottest layer is shown at right).

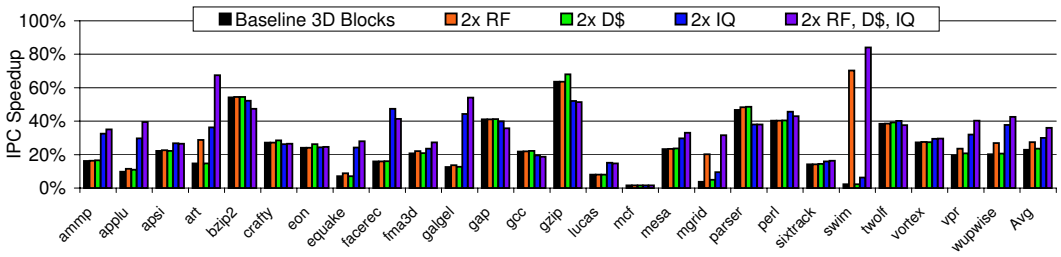

Figure 7.17: Performance when doubling critical resources.

7.5.1 Scaling Architectural Sizes

Even in the 3D block architecture, there are still cases where we are able to increase the timing slack within a given cycle of a critical loop, without actually reducing the number of cycles in that critical loop. Figure 7.17 presents one approach to leveraging this extra slack: we double the size of the data cache, issue queue, and register file.

As mentioned earlier, using 3D integration technology, the access latency of double-sized structures for these three critical blocks is still less than in 2D. The register file and data cache can even quadruple their sizes while still outperforming the default blocks in 2D. In this dissertation, we limit our study to doubled sizes.

As shown in Figure 7.17, the performance is increased by an additional 5% with a doubled cache, an additional 1% with a doubled register file, and an additional 7% with a doubled IQ. The best performance is observed when doubling the size of all three structures. Overall, there is a 36% gain over the 2D architecture and a 13% gain over the 3D architecture with our default block sizes.

These larger structures will dissipate more power than regular-sized 3D blocks.

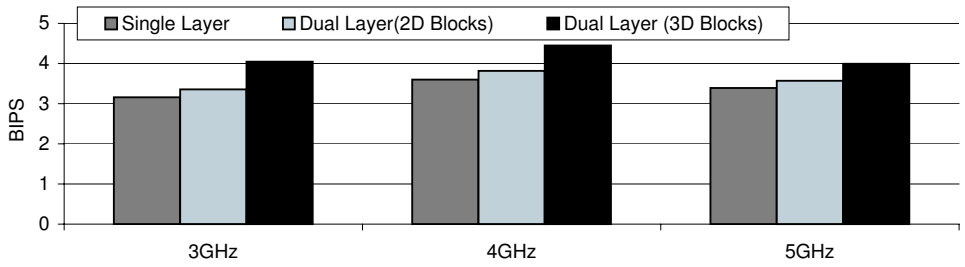

Figure 7.18: BIPS performance for different clock frequencies.

But despite the increase in power, the increased area of these larger designs saw an average slight decrease in temperature of 0.8°C for the case where all three resources were doubled. The maximal temperature in this case was 34.1°C.

7.5.2 Frequencies

The results presented so far feature a 4GHz clock frequency. Figure 7.18 demonstrates the performance in BIPS when using different clock frequencies: 3GHz, 4GHz, and 5GHz. The first bar is the single layer architecture, the second bar is the dual layer architecture with 2D blocks, and the final bar is the dual layer architecture with 3D blocks.

At 3GHz, the larger latency of wire using 2D architecture or 2D blocks is better tolerated by the more forgiving clock rate – but the use of 3D blocks can still provide a 10% gain over a single layer architecture. The overall performance at 5GHz decreases slightly from 4GHz due to the lengthened critical loops at a higher frequency.

7.5.3 Number of Layers

In this subsection, we demonstrate the performance of vertical integration when scaling beyond two silicon layers. Figure 7.19 illustrates this gain for 4GHz architectures. Due to the challenge in scaling to more layers with F2F blocks, we only use F2B in this study.

The performance of two layers in this figure is slightly worse than in Figure 7.12. The F2B bonded blocks used in this section perform slightly worse than F2F blocks because of the impact of 3D vias on the silicon layer.

The first bar shows the performance when using only 2D blocks, and as shown in the figure, there is little gain from scaling the number of silicon layers for this design driver. Without tackling intra-block latency, even the near elimination of inter-block latency can only improve performance by so much. However, the gain from 3D blocks over the single layer architecture grows from 22% at two layers to 28% at four layers. Although inter-block wire latencies have been nearly eliminated in two layers, the latency reduction in four-layer microarchitectural blocks further reduced cycles in critical loops.

There is little gain from two layers to three layers using 3D blocks. As shown in Figure 7.7, many architectural blocks have little or no reduction in latency from two layers to three layers. Furthermore, while there was an increase in slack for many critical loops, there was no overall reduction in cycles for these critical loops. However, it would certainly be possible to leverage this slack in other ways.

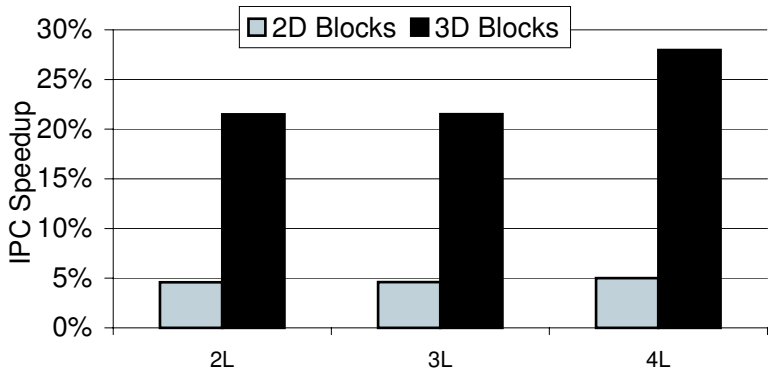

Figure 7.19: Performance when scaling the number of silicon layers.

7.6 Summary

Vertical integration has tremendous potential to reduce both inter-block and intra-block wire latency. We have proposed and evaluated tag partitioning for the issue queue, for caches, and for cache-like blocks. And we have enhanced the MEVA-3D exploration framework to evaluate the use of 3D blocks in multiple layers of silicon. When using two layers of silicon with 3D blocks, we see an average 36% improvement in performance over a single layer architecture and 29% improvement in performance over two layers with single layer blocks, for the architectural design driver we explored. Temperature is kept below 40°C using a two heat sink F2F design.

CHAPTER 8

Conclusion and Future Directions

In this chapter, we conclude the dissertation and discuss several further directions to explore several remaining challenges.

8.1 Conclusions

In this dissertation, we have proposed several techniques to improve microprocessor scheduling performance and to reduce power consumption in the scheduling logic.

We present a lookahead memory latency prediction scheme. Our prediction scheme features over 80% rate for cache misses, and over 98% for cache hits. Assisted by memory latency predictions, we are able to obtain expected waiting times of instructions in the issue queue. We apply it with a simple structure that sorts instructions in their expected issue order based on the predicted waiting time. In a conventional out-of-order superscalar, the scheduling window (issue queue) size is limited due to its circuit complexities. Our sorting mechanism effectively prevent instructions with long waiting times, in particular, the dependents of missed loads from entering the scheduling window too early. Otherwise, these instructions enter the issue queue in the original program order and then consume the scarce issue queue slots for a long time. Our experiment shows the

proposed technique enables a conventional scheduler using a 12-entry issue queue to achieve comparable performance to a scheduler with a 24-entry issue queue.

We also investigate the Tornado effect and propose effective scheme to reduce its scale. Tornado effects happen in recently proposed speculative selective-replay schedulers (e.g. P4). Our investigation shows that the primary cause of the tornado effect is because the dependents of loads are scheduled based the assumption that loads always hit. The dependents are replayed repeatedly till loads completion. In addition, the structural hazards in schedulers also cause instruction replays. There is a positive feedback loop where replays can spawn more and more replays and eventually consume most of the available issue bandwidth. We propose to buffer instructions based on their predicted latencies, and the scheme substantially reduce the tornado effect. We further reduce this effect by proposing preventive measures to detect a tornado in its early stages of development, and dynamically limit the number of instructions in the scheduling queues. We are able to see a substantial drop in tornadoes when these techniques are combined together.

The above techniques feature instruction level mechanisms that scale issue queue performance. A larger physical issue queue size allows more ILP be exploited. However, building larger issue queue using conventional physical designs will significantly increase the issue queue access time [PJS97]. We present port partitioning as an alternative solution to scale issue queue size. Scaling issue queue size can significant improve microprocessor scheduling performance. In traditional designs, however, the delay increases quadratically as issue queue size scales. Leveraging existing 3D via insertion technology, port partitioning effectively scales issue queue with less complexities. The instruction scheduling queue is a heavily-ported structure. By dividing the ports and placing them into differ-

ent layers, significant amount of power and access time are reduced. Using this technique, the scheduling queue size can be doubled with an even faster access time.

Our proposed techniques effectively reduce the overall energy consumption. The energy due to scheduling logic and register file accesses make up a large portion of the total microprocessor energy consumption. Benefiting from reduced scheduling queue occupancies, our scheme significantly reduces the energy consumption in conventional schedulers. With the elimination of unnecessary accesses that stem from scheduling replays, our scheme even reduces more energy consumption in speculative schedulers. As the wire capacitance is reduced as a result of wire length reduction, our proposed 3D issue queue consumes substantially less energy.

8.2 Future Directions

8.2.1 Improving Load Latency Predictions

As highlighted in Chapter 3, a significant amount of load latency mispredictions are caused by load address mispredictions. One future direction is to improve load address prediction to better utilize the our scheme's capability of accurately predicting caches misses with given load addresses.

8.2.2 Latency Prediction of Cross-core Communications in Multi-core Processor

Chip-multiprocessor (CMP) architectures[HNO97] are a promising design alternative to exploit an ever-increasing transistor density. However, many applica-

tions can not easily be parallelized across the multiple cores. CMPs may have to execute data-dependent threads of an application. Hence, in the CMP environment, there are uncertainties on how long an execution core needs to wait for its depending data from another execution core. If the expected waiting time can be predicted in advance, an execution core can switch to other threads if the waiting time is too long.

An interesting challenge is to extend our latency prediction techniques to the prediction of cross-core communications. Cross-core dependences can be handled dynamically via synchronization and communication either at the register level or at the memory level[KT99]. The added complexities for synchronization and communication may create additional challenges on predicting waiting times.

8.2.2.1 3D Processors

We have observed the performance speedup of microprocessors that are equipped with 3D architectural blocks. In future CMP microprocessors, the number of processor cores and L2/L3 sizes are expected grow substantially. The interconnect is increasingly critical to the performance and power of future design [HMH01]. It is interesting to study the use of 3D interconnects and 3D architectural blocks to alleviate the problem.

References

[ALE02] T. Austin, E. Larson, and D. Ernst. "SimpleScalar: an Infrastructure for Computer System Modeling." *IEEE Computer*, **35**(2), February 2002.

[BA97] D. C. Burger and T. M. Austin. "The SimpleScalar Tool Set, Version 2.0." Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[BAS02] Alper Buyuktosunoglu, David H. Albonesi, Stanley Schuster, David Brooks, Pradip Bose, and Peter Cook. "Power-efficient issue queue design." In *Power Aware Computing*, pp. 35–58. Kluwer Academic Publishers, 2002.

[BKA03] Alper Buyuktosunoglu, Tejas Karkhanis, David H. Albonesi, and Pradip Bose. "Energy efficient co-adaptive instruction fetch and issue." In *Proceedings of the 30th annual international symposium on Computer architecture (ISCA'03)*, pp. 147–156. ACM Press, 2003.

[BNW04] B. Black, D. W. Nelson, C. Webb, and N. Samra. "3D Processing Technology and its Impact on IA32 Microprocessors." In *Proc. Of ICCD, pp.316-318*, 2004.

[BS04] David F. Bacon and Xiaowei Shen. "Braids and Fibers: Language Constructs with Architectural Support for Adaptive Response to Memory Latencies." In *Proc. of the First Watson Conference on Interaction between Architecture, Circuits, and Compilers (P=ac2)*, October 2004.

[BSB01] Alper Buyuktosunoglu, Stanley Schuster, David Brooks, Pradip Bose, Peter W. Cook, and David H. Albonesi. "An Adaptive Issue Queue for Reduced Power at High Performance." In *Proceedings of the First International Workshop on Power-Aware Computer Systems-Revised Papers*, pp. 25–39. Springer-Verlag, 2001.

[BSK01] K. Banerjee, S. Souri, P. Kapur, and K. Saraswat. "3-D ICs: A Novel Chip Design for Improving Deep-Submicrometer Interconnect Performance and Systems-on-Chip Integration." In *In Proc. of the IEEE, 89(5):602-633*, May 2001.

[BSP01] M. D. Brown, J. Stark, and Y. N. Patt. "Superscalar architectures: Select-free instruction scheduling logic." In *34th International Symposium on Microarchitecture*, December 2001.

[BTM00] David Brooks, Vivek Tiwari, and Margaret Martonosi. "Wattch: a framework for architectural-level power analysis and optimizations." In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*, pp. 83–94. ACM Press, 2000.

[BTM02] E. Borch, E. Tune, S. Manne, and J. Emer. "Loose Loops Sink Chips." In *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, 2002.

[Car04] D. Carmean. "Distinguished Lecturer Series Presentation at UCLA." 2004.

[CG01a] R. Canal and A. Gonzalez. "A Low-Complexity Issue Logic." In *Proceedings of the 15th International Conference on Supercomputing (ICS01)*, May 2001.

[CG01b] Ramon Canal and Antonio González. "Reducing the complexity of the issue logic." In *ICS '01: Proceedings of the 15th international conference on Supercomputing*, pp. 312–320, New York, NY, USA, 2001. ACM Press.

[CJM06] J. Cong, A. Jagannathan, Y. Ma, G. Reinman, J. Wei, and Y. Zhang. "An Automated Design Flow for 3D Microarchitecture Evaluation." In *Proc. Asia and South Pacific Design Automation Conf*, January 2006.

[CSS00] Y. Cao, T. Sato, D. Sylvester, M. Orshansky, and C. Hu. "New paradigm of predictive MOSFET and interconnect modeling for early circuit design." In *Proc. of Custom Integrated Circuit Conference*, 2000.

[DCR03] S. Das, A. Chandrakasan, and R. Reif. "Design Tools for 3-D Integrated Circuits." In *Proc. Asia and South Pacific Design Automation Conf., pp. 53-56*, January 2003.

[DFC04] S. Das, A. Fan, K. Chen, and C. Tan. "Technology, Performance, and Computer-Aided Design of Three-Dimentional Integrated Circuits." In *Proc. International Symposium on Physical Design*, April 2004.

[EA02] D. Ernst and T. Austin. "Efficient Dynamic Scheduling through Tag Elimination." In *29th Annual International Symposium on Computer Architecture*, May 2002.

[EHA03] D. Ernst, A. Hamel, and T. Austin. "Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay." In *Proceedings of the 30th Annual International Symposium on Computer Architecture (ISCA'03)*, June 2003.

[EP04] E. Ehrhart and Sanjay J. Patel. "Reducing the Scheduling Critical Cycle Using Wakeup Prediction." In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture (HPCA10 2004)*, pp. 222–231, 2004.

[FG01] Daniele Folegnani and Antonio Gonzalez. "Energy-effective issue logic." In *Proceedings of the 28th Annual International Symposium on Computer Architecture (ISCA'01)*, pp. 230–239. ACM Press, 2001.

[FJC95] K. Farkas, N. Jouppi, and P. Chow. "Register File Design Considerations in Dynamically Scheduled Processors." In *Proceedings of the Second International Symposium on High Performance Computer Architecture*, 1995.

[GKA01] M. Gschwind, S. Kosonocky, and E. Altman. "High Frequency Pipeline Architecture Using the Recirculation Buffer." In *IBM Research Report(RC23113)*, 2001.

[HMH01] Ron Ho, Kenneth W. Mai, and Mark A. Horowitz. "The future of wires." *PROCEEDINGS OF THE IEEE*, **89**(4):490–504, April 2001.

[HNO97] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. "A Single-Chip Multiprocessor." *IEEE Computer*, **30**(9):79–85, 1997.

[HSU01] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. "The Microarchitecture of the Pentium 4 Processor." *Intel Technology Journal Q1*, 2001.

[HVI04] J. S. Hu, N. Vijaykrishnan, and M. J. Irwin. "Exploring Wakeup-Free Instruction Scheduling." In *Proceedings of The Tenth International Symposium on High-Performance Computer Architecture (HPCA10 2004)*, February 2004.

[ITR05] "The International Technology Roadmap for Semiconductors (ITRS)." 2005.

[JY05] J.Cong and Y.Zang. "Thermal-Driven Multilevel Routing for 3-D ICs." In *Asia Pacific Design Automation Conference*, pp. 121–126, 2005.

[Kes99] R. E. Kessler. "The Alpha 21264 Microprocessor." *IEEE Micro*, **19**(2):24–36, 1999.

[KG05] K.Puttaswamy and G.H.Loh. "Implementing Caches in a 3D Technology for High Performance Processors." In *International Conference on Computer Design*, 2005.

[KL] Ilhyun Kim and Mikko H. Lipasti. "Understanding Scheduling Replay Schemes." In *10th International Conference on High-Performance Computer Architecture (HPCA'04), 14-18 February 2004, Madrid, Spain*, pp. 198–209.

[KL04] Ilhyun Kim and Mikko H. Lipasti. "Understanding Scheduling Replay Schemes." In *HPCA '04: Proceedings of the 10th International Symposium on High Performance Computer Architecture*, p. 198, Washington, DC, USA, 2004. IEEE Computer Society.

[KMW98] R.E. Kessler, E.J. McLellan, and D.A. Webb. "The Alpha 21264 Microprosessor Architecture." In *International Conference on Computer Design*, December 1998.

[KS02] H-S. Kim and J. E. Smith. "An instruction set and microarchitecture for instruction level distributed processing." In *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 71–81, June 2002.

[KSB02] T. Karkhanis, J. E. Smith, and P. Bose. "Saving energy with just in time instruction delivery." In *Proceedings of the 2002 International Symposium on Low Power Electronics and Design (ISLPED'02)*, pp. 178–183. ACM Press, 2002.

[KSP01] K.Banerjee, S.Souri, P.Kapur, and K.C. Saraswat. "3-D ICs: A Novel Chip Design for Improving Deep-Submicrometer Interconnect Performance and Systems-on-Chip Integration." In *IEEE Special Issue Interconnections - Addressing The Next Challenge of IC Technology, Vol. 89, No. 5*, pp. 602–633, 2001.

[KT99] Venkata Krishnan and Josep Torrellas. "The Need for Fast Communication in Hardware-Based Speculative Chip Multiprocessors." p. 24, 1999.

[Lab06] MIT Lincoln Laboratory. "MITLL Low-Power FDSOI CMOS Process: Design Guide." March 2006.

[LKL02] A.R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan, and E. Rotenberg. "A Large, Fast Instruction Window for Tolerating Cache Misses." In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA'02)*, May 2002.

[LSM] Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. "Proceedings of the 19th Annual International Conference on Supercomputing, ICS 2005, Cambridge, Massachusetts, USA, June 20-22, 2005." pp. 51–60. ACM Press.

[LSM04a] Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. "The Calm Before the Storm: Reducing Replays in the Cyclone Scheduler." In *Proc. of the First Watson Conference on Interaction between Architecture, Circuits, and Compilers (P=ac2)*, October 2004.

[LSM04b] Yongxiang Liu, Anahita Shayesteh, Gokhan Memik, and Glenn Reinman. "Scaling the issue window with look-ahead latency prediction." In *Proceedings of the 18th Annual International Conference on Supercomputing (ICS'04)*, pp. 217–226. ACM Press, 2004.

[LWS96] M.H. Lipasti, C.B. Wilkerson, and J.P. Shen. "Value Locality and Load Value Prediction." In *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 138–147, October 1996.

[LY00] S. Lee and P. Yew. "On Some Implementation Issues for Value Prediction on Wide-Issue ILP Processors." In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2000.

[MB03] Tali Moreshet and R. Iris Bahar. "Power-aware issue queue design for speculative instructions." In *Proceedings of the 40th Conference on Design Automation (DAC'03)*, pp. 634–637. ACM Press, 2003.

[McF93] S. McFarling. "Combining Branch Predictors." Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[MDA79] M.W.Geis, D.C.Flanders, D.A. Antoniadis, and H.I.Smith. "Crystalline Silicon on Insulators by Graphoepitaxy." In *IEDM Technical Digest*, pp. 210–212, 1979.

[MF95] G. McFarland and M. Flynn. "Limits of Scaling MOSFETS." CSL TR-95-62, Stanford University, November 1995.

[MHS05] Y. Ma, X. Hong, and C.K. Cheng S. Dong. "3D CBL: An Efficient Algorithm for General 3-Dimensional Packing Problems." In *In IEEE International Midwest Symposium on Circuits and Systems*, August 2005.

[MPP86] M.A.Crowder, P.G.Carey, P.M.Smith, R.S.Sposili, H.S.Cho, and J.S.Im. "Low Temperature Single Crystal Si TFTs fabricated on Si–filmw processed via Sequential Lateral Solidification." In *IEEE Electron Device Letters, Vol. 19, No. 8*, pp. 306–308, 1986.

[MRM03a] G. Memik, G. Reinman, and W. H. Mangione-Smith. "Just Say No: Benefits of Early Cache Miss Determination." In *Proceedings of the 9th International Symposium on High-Performance Computer Architecture (HPCA'03)*, February 2003.

[MRM03b] G. Memik, G. Reinman, and W. H. Mangione-Smith. "Precise Scheduling with Early Cache Miss Detection." CARES Technical Report No. 2003_1, 2003.

[MS99] M.Rodder and S.Aur. "Utilization of Plasma Hydrogenization in stacked SRAMs with pli-Si PMOSFETs and bulk Si NMOS FETs." In *IEEE Electron Device Letters, Vol. 12*, pp. 233–235, 1999.

[MS01] P. Michaud and A. Seznec. "Data-flow prescheduling for large instruction windows in out-of-order processors." In *Proceedings of the 7th International Symposium on High Performance Computer Architecture (HPCA-6)*, January 2001.

[MSU96] Pierre Michaud, Andre Seznec, and Richard Uhlig. "Skewed Branch Predictors." Technical Report RR-2978, IRISA, June 1996.

[OWK06] Ozcan Ozturk, Feng Wang, Mahmut T. Kandemir, and Yuan Xie. "Optimal topology exploration for application-specific 3D architectures." In *ASP-DAC*, pp. 390–395, 2006.

[PAM04] P.Wilkerson, A.Raman, and M.Turowski. "Fast, Automated Thermal Simulation for Three-Dimensional Integrated Circuits." In *Intersociety Conference on Thermal and Thermomechanical Phenomena in Electronic Circuits, Itherm*, 2004.

[PJS97] S. Palacharla, N. P. Jouppi, and J. E. Smith. "Complexity-Effective Superscalar Processors." In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, pp. 206–218, June 1997.

[PKE03] Dmitry V. Ponomarev, Gurhan Kucuk, Oguz Ergin, Kanad Ghose, and Peter M. Kogge. "Energy-efficient issue queue design." *IEEE Trans. Very Large Scale Integr. Syst.*, **11**(5):789–800, 2003.

[PLL02] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. "Bloom filtering cache misses for accurate data speculation and prefetching." In *ICS '02: Proceedings of the 16th international conference on Supercomputing*, pp. 189–198, New York, NY, USA, 2002. ACM Press.

[PPV02] Il Park, Michael D. Powell, and T. N. Vijaykumar. "Reducing register ports for higher speed and lower energy." In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture (MICRO'02)*, pp. 171–182. IEEE Computer Society Press, 2002.

[PSD99] S. Pae, T.-C. Su, J.P. Denton, and G/W. Neudeck. "Multiple Layers of Silicon-on-Insulator Islands Fabrication by Selective Epitaxial Growth." In *IEEE Electron Device Letters, Vol. 20, No. 5*, 1999.

[RAK01] R.Ronnen, A.Mendelson, K.Lai, S-L Liu, F.Pollack, and J.P.Shen. "Coming Challenges in Microarchitecture and Architecture." In *Proceedings of the IEEE, Vol. 89, No.3*, pp. 325–340, 2001.

[RBR02] S. Raasch, N. Binkert, and S. Reinhardt. "A Scalable Instruction Queue Design Using Dependence Chain." In *Proceedings of the 29th Annual International Symposium on Computer Architecture (ISCA-29)*, May 2002.

[RJ] G. Reinman and N. P. Jouppi. "CACTI 2.0 Beta." In *In http ://www.research.digital.com/wrl//people/jouppi/CACTI.html, 1999.*

[SBP00] J. Stark, M. D. Brown, and Y. N. Patt. "On pipelining dynamic instruction scheduling logic." In *33rd International Symposium on Microarchitecture*, December 2000.

[SC02] E. Sprangle and D. Carmean. "Increasing Processor Performance by Implementing Deeper Pipelines." In *29th Annual International Symposium on Computer Architecture*, 2002.

[SJ01] P. Shivakumar and Norman P. Jouppi. "CACTI 3.0: An integrated cache timing, power, and area model." In *Technical Report*, 2001.

[SNT83] S.Kawamura, N.Sasaki, T.Iwai, M.Nakano, and M.Takagi. "Three-Dimensional CMPS ICs Fabricated using Beam Recrystallization." In *IEEE Electron Devices Vol. Ed. 4*, pp. 366–369, 1983.

[SPC01] T. Sherwood, E. Perelman, and B. Calder. "Basic Block Distribution Analysis to Find Periodic Behavior and Simulation Points in Applications." In *International Conference on Parallel Architectures and Compilation Techniques (PACT'01)*, September 2001.

[SSM94] R. F. Sproull, I. E. Sutherland, and C.E. Molnar. "The Counterflow Pipeline Processor Architecture." *IEEE Design and Test of Computers*, **11**(3):48–59, 1994.

[TB01] D. Tullsen and J. Brown. "Handling long-latency Loads in a Simultaneous Multithreading Processor." In *34th International Symposium on Microarchitecture*, 2001.

[TEE96] 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 *23rd Annual International Symposium on Computer Architecture*, May 1996.

[TEL95] Dean Tullsen, Susan Eggers, and Henry Levy. "Simultaneous Multithreading: Maximizing On-Chip Parallelism." In *Proceedings of the 22rd Annual International Symposium on Computer Architecture (ISCA)*, June 1995.

[TKY89] T.Kunio, K.Oyama, Y.Hayashi, and M.Morimoto. "Three Dimensional ICs, Having Four Stacked Active Decide Layers." In *IEDM Technical Digest*, pp. 837–840, 1989.

[Tom] R. Tomasulo. "An efficient algorithm for exploring multiple arithmetic units." In *IBM Journal of Research and Development, vol. 11, no. 1, pp. 25-33, Jan. 1967.*

[TXV05] Y. Tsai, Y. Xie, N. Vijaykrishnan, and M. Irwin. "Three-Dimensional Cache Design Exploration Using 3DCacti." In *International Conference on Computer Design*, October 2005.

[WF97] K. Wang and M. Franklin. "Highly Accurate Data Value Prediction using Hybrid Predictors." In *30th Annual International Symposium on Microarchitecture*, pp. 281–290, December 1997.

[WJ] S. Wilton and N. P. Jouppi. "CACTI: An Enhanced Cache Access and Cycle Time Model." In *IEEE Journal of Solid-State Circuits, pages 677-687, 1996.*

[WJ96] S. Wilton and N. Jouppi. "CACTI: An Enhanced Cache Access and Cycle Time Model." In *IEEE Journal of Solid-State Circuits*, May 1996.

[WM] K. Wilcox and S. Manne. "Alpha processors: A history of power issues and a look to the future." In *Cool-Chips Tutorial, November 1999. Held in conjunction with MICRO-32.*

[WZP02] Hang-Sheng Wang, Xinping Zhu, Li-Shiuan Peh, and Sharad Malik. "Orion: a power-performance simulator for interconnection networks." In *Proceedings of the 35th annual ACM/IEEE International Symposium on Microarchitecture (MICRO'02)*, pp. 294–305. IEEE Computer Society Press, 2002.

[YER99] Adi Yoaz, Mattan Erez, Ronny Ronen, and Stephan Jourdan. "Speculation techniques for improving load related instruction scheduling." In *ISCA '99: Proceedings of the 26th annual international symposium on Computer architecture*, pp. 42–53, Washington, DC, USA, 1999. IEEE Computer Society.

[YN86] Y.Akasaka and T. Nishimura. "Concept and Basic Technologies for 3D IC Structure." In *IEDM Technical Digest*, pp. 488–491, 1986.