Research Commons

http://researchcommons.waikato.ac.nz/

Research Commons at the University of Waikato

Copyright Statement:

The digital copy of this thesis is protected by the Copyright Act 1994 (New Zealand).

The thesis may be consulted by you, provided you comply with the provisions of the Act and the following conditions of use:

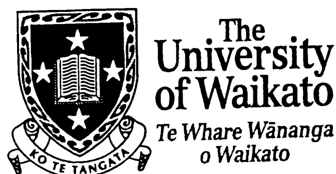
- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement will be made to the author where appropriate.
- You will obtain the author's permission before publishing any material from the thesis.

Controlling Speculative Execution Through a Virtually Ordered Memory System

A thesis
submitted in partial fulfilment
of the requirements for the degree
of
Doctor of Philosophy
at the
University of Waikato
by

J A David McWha

Department of Computer Science



Hamilton, New Zealand

January 12, 2003

Abstract

Processors which extract parallelism through speculative execution must be able to identify when mis-speculation has occurred. The three places where mis-speculation can occur are register accesses, control flow prediction and memory accesses. Controlling register and control flow speculation has been well studied, but no scalable techniques for identifying memory dependence violations have been identified. Since speculative execution occurs out of order this requires tracking the causal order, as well as the addresses of memory accesses.

This thesis uses simulations to investigate tracking the causal order of memory accesses using explicit tags known as virtual timestamps, a distributed and scalable method. Realizable virtual timestamps are necessarily restricted in length and it is demonstrated that naive allocation schemes seriously constrain execution by inefficiently allocating virtual timestamps. Efficiently allocating virtual timestamps requires analysis of the number required by each section of code. Basic statically and dynamically evaluated analysis methods are established to avoid virtual timestamp allocation becoming a resource bottleneck.

The same analysis is also used to efficiently allocate state saving resources in a fixed hardware order. The hardware order provides an alternative way of maintaining the causal order using a simple hardware organization. The ability to predict the resources required by regions of code is used as a way of selecting instructions to execute speculatively. This enables resources to be allocated efficiently and is shown to allow large amounts of parallelism to be extracted. It also promotes the effectiveness of speculative execution by issuing less instructions that will ultimately be rolled back.

Using a hierarchy of hardware ordering modules, themselves ordered by explicit virtual timestamps, a scalable ordering system is proposed. This hierarchy forms the basis of a

twisted memory system, a multiple version memory system capable of identifying speculative memory dependence violations. The preliminary investigations presented here show that twisted memory has the potential to support aggressive speculative parallel execution. Particular attention is paid to memory bandwidth requirements.

Acknowledgments

There are many people without whom preparing this thesis would not have been possible, or as fruitful and enjoyable as it has been.

I would like to thank my supervisor John Cleary for his support and guidance throughout the process. A special thanks must go to Richard Littin for his help and forbearance as a colleague, flatmate and friend. I would also like to thank the other members of the WarpEngine project over the years for contributing to such a great environment.

I also want to acknowledge the support of all the friends I have made during my years at Waikato, in the tearoom, on the soccer field, on the stage, in the tutors' room. Thanks for keeping me sane.

Thank you to Chris, Yvonne, Gerard, Eddy, Richard and, of course, John, for providing proof reading assistance in the final stages under time pressure.

I owe a debt of gratitude to the School of Computing and Mathematical Sciences and the Department of Computer Science for the vital financial support and resources they have provided me with.

Finally, and most importantly, thanks to my family for their support. Without the support and encouragement of my parents throughout my life I would never have contemplated a PhD.

Thanks guys.

Contents

Abstract ii

Acknowledgments iv

List of Figures xv

List of Tables xvi

1 Introduction 1

1.1 The Challenge 2

1.2 Thesis Claims 3

1.3 Thesis Overview 4

2 Controlling Instruction Level Parallelism 6

2.1 Instruction Dependencies 7

2.1.1 Data Dependence 8

2.1.2 Control Dependence 8

2.1.3 Limit Studies 9

2.2 Conservative Parallel Execution 10

2.2.1 Vector 11

2.2.2 Dataflow 12

2.2.3 Very Long Instruction Word 13

2.2.4 Out-of-order Execution 14

2.3 Speculative Parallel Execution 16

2.3.1 Control Speculation 16

2.3.2 Data Speculation 18

2.3.3 Architectures 19

2.4 Recovering from Mis-speculation 22
2.4.1 Pipeline Flushing 22
2.4.2 Checkpoint Repair and History Buffer 23
2.4.3 Reorder Buffer 23
2.4.4 Split Instruction Window Squashing 25
2.4.5 Selective Undo 25
2.4.6 Memory Consistency 27
2.5 Summary 31

3 The WarpEngine **32**
3.1 Time Warp 33
3.1.1 Global Virtual Time and Fossil Collection 34
3.1.2 Cancelback 35
3.1.3 Time Warp in Computer Architecture 35
3.2 Supporting Speculative Execution 36
3.2.1 Block Structured Execution 37
3.2.2 Tree Structured Execution 38
3.2.3 Virtual Ordering 40
3.3 Architecture 42
3.3.1 Instruction Set 43
3.3.2 Frames 44
3.3.3 Code Store 44
3.3.4 Function Units 45
3.3.5 Synchronization Mechanism 46
3.3.6 Instruction Execution 46
3.3.7 Memory System 47
3.4 Related Architectures 51
3.4.1 Superscalar 52
3.4.2 Dataflow Processors 52
3.4.3 Multiscalar 53
3.5 Summary 54

4 Simulation **55**
4.1 Virtual Order Simulation 55

4.1.1 Assumptions 56
4.1.2 Limiting Resources 56
4.1.3 State History 57
4.2 Simulation Test Bench 58
4.2.1 Simulation Parameters 60
4.2.2 Virtual Order Simulation Effects on Performance 62
4.3 Test Programs 63
4.3.1 Matrix Manipulation 65
4.3.2 Sorting 66
4.3.3 Dynamic structures 67
4.3.4 Recursion 68
4.4 Compilation 69
4.5 Summary 70

5 Explicit Virtual Timestamps **72**
5.1 Concept 72
5.1.1 Requirements 75
5.2 Symbolic Representation 77
5.3 Fixed Length VTS Schemes 78
5.3.1 Length Representation 79
5.3.2 Exponential Representation 80
5.3.3 Ideal Representation 82
5.3.4 Minimum Size VTSs 83
5.4 Rescaling VTSs 85
5.4.1 Root Rescaling 86
5.4.2 Moving Head Rescaling 89
5.4.3 Speedup 91
5.5 Summary 94

6 Variable Virtual Timestamp Ranges **97**
6.1 Concept 97
6.2 Execution Tree Size Calculation 98
6.2.1 Encoding 100
6.2.2 Static Analysis 100

6.2.3 Dynamic Analysis 103
6.3 Minimum Size VTSs 104
6.4 Rescaling Variable Range VTSs 108
6.4.1 Preserved Tree Rescaling 110
6.4.2 Disconnected Subtree Rescaling 111
6.4.3 Moving Head Rescaling 112
6.5 Speedup 114
6.5.1 VTS Constrained 114
6.5.2 Frame Limit Sensitivity 116
6.6 Unresolved Issues 120
6.7 Summary 121

7 Selective Speculation **123**
7.1 Related Work 124
7.2 Selective Speculation in the WarpEngine 127
7.3 Resource Allocation 128
7.3.1 Assigning Frames to Resource Blocks 131
7.4 Speedup 133
7.5 Resource Blocks and Frame Limiting Compared 139
7.6 Speculation Effectiveness 141
7.7 Summary 145

8 Twisted Memory **147**
8.1 Explicit VTSs 149
8.2 Twisted Memory 150
8.2.1 Write 153
8.2.2 Read 155
8.2.3 Anti-write 157
8.2.4 Anti-read 159
8.2.5 Fossil Collection 159
8.3 Evaluation 164
8.4 Memory Bandwidth 165
8.4.1 Average Memory Bandwidth 165
8.4.2 Bandwidth Profile 173

8.5 Optimizations 175
8.6 Summary 176

9 Summary and Conclusions **179**
9.1 Virtual Timestamps 179
9.2 Resource Blocks 180
9.3 Virtually Ordered Memory System 181
9.4 Conclusions 181
9.5 Future Work 182

Appendices

A WarpEngine Instruction Set **184**

B Test Code **188**

C Additional Graphs **196**
C.1 Minimum Size Fixed Length VTSs 196
C.2 Speedup for Length and Exponential VTSs 198
C.3 Minimum Size Variable Range VTSs 200
C.4 Variable Range VTS Speedup 201
C.5 Variable Range VTS Frame Limit Sensitivity 203
C.6 Speculation Effectiveness 206

D Twisted Memory Bandwidth Profile Graphs **208**
D.1 AVL(200) 208
D.2 Binary Tree (300) 212
D.3 Fibonacci (15) 215
D.4 Gauss-Jordan (10) 218
D.5 Matrix Multiply (20) 221
D.6 Quicksort 1 (200) 224
D.7 Enlarged Regions of Level Zero Bandwidth Profile Graphs 227

E Glossary of Terms and Abbreviations **230**

Bibliography **233**

List of Figures

2.1 Code with control independence 9

3.1 Producing tree structured execution from sequential code 39
3.2 Execution tree for loop iterations 39
3.3 An equivalent binary execution tree 40
3.4 Equivalent CFGs for single flow of control and tree structured execution . . 41
3.5 Components of the WarpEngine and their connections 43
3.6 Layout of a frame 45
3.7 Instruction flow within a WarpEngine frame. 46
3.8 Components of the virtually ordered memory system 48
3.9 Resatisfying a read due to an out-of-order write. 49

4.1 Processing input history lists to form an output history list. 58
4.2 Components of the WarpEngine virtual order simulator 59

5.1 Returning the correct value from a read request and rollback due to a write . 73
5.2 Returning the correct value from a read request and rollback due to a write using VTSs 74
5.3 Code for a while loop and later independent code 76
5.4 Execution tree for a while loop and later independent code 76
5.5 Conceptual VTSs for a binary tree 78
5.6 VTSs in length representation 80
5.7 Percentage of children generated with child number zero 80
5.8 VTS in exponent representation 81
5.9 Minimum VTS length necessary to execute AVL tree insertion without re-scaling 83
5.10 Minimum VTS length necessary to execute Fibonacci without rescaling . . 84

5.11 Minimum VTS length necessary to execute quicksort 1 without rescaling . 85
5.12 Root rescaling of an execution tree 87
5.13 Cancelback and root rescaling in an exhausted execution tree 87
5.14 Moving head VTS representation 90
5.15 Rescaling the moving head representation 90
5.16 Comparison of speedup for AVL tree insertion with exponent and length VTSs 92
5.17 Comparison of speedup for Gauss-Jordan with exponent and length VTSs . 92
5.18 Comparison of speedup for matrix multiply with exponent and length VTSs 93

6.1 VTS trees with implicit and explicit upper bounds 98
6.2 Execution tree for a dynamically bounded loop 99
6.3 Encoding for subtree size calculations for a fixed number of VTSs (top) and proportion of available range (bottom) 100
6.4 Calculating the number of frames required to execute an acyclic code region 101
6.5 Calculating the number of frames required to execute a `for` loop 102
6.6 Minimum VTS length necessary to execute AVL tree insertion without rescaling 104
6.7 Minimum VTS length necessary to execute Fibonacci without rescaling . . 105
6.8 Minimum VTS length necessary to execute Gauss-Jordan without rescaling 106
6.9 Minimum VTS length necessary to execute quicksort without rescaling . . 106
6.10 Percentage overhead of dynamic VTS analysis methods for quicksort1 . . . 108
6.11 Rescaling a variable range VTS tree 112
6.12 Speedup for AVL tree with variable range VTSs with rescaling 115
6.13 Speedup for Fibonacci with variable range VTSs with rescaling 115
6.14 Speedup for quicksort1 with variable range VTSs with rescaling 116
6.15 Speedup for AVL(500) with varying frame limitations and variable range VTSs 118
6.16 Speedup for quicksort1 (500) with varying frame limitations and variable range VTSs 119
6.17 Speedup for Fibonacci (15) with varying frame limitations and variable range VTSs 119

7.1 Speculation effectiveness for different speculative techniques 125
7.2 Allocating frames in a resource block 130

7.3 Program forcing cancelback for a circular queue of resource blocks 131
7.4 Speedup for AVL tree using resource blocks in a circular queue 134
7.5 Speedup for binary tree using resource blocks in a circular queue 135
7.6 Speedup for Fibonacci using resource blocks in a circular queue 136
7.7 Speedup for quicksort1 using resource blocks in a circular queue 136
7.8 Speedup for AVL tree using reorderable resource blocks 137
7.9 Speedup for binary tree using reorderable resource blocks 137
7.10 Speedup for Fibonacci using reorderable resource blocks 138
7.11 Speedup for quicksort1 using reorderable resource blocks 138
7.12 Speedup for AVL tree with frame limited and resource block limited execution 139
7.13 Speedup for binary tree with frame limited and resource block limited execution 140
7.14 Speedup for Fibonacci with frame limited and resource block limited execution 140
7.15 Speedup for quicksort1 with frame limited and resource block limited execution 141
7.16 Speculation effectiveness of resource block and frame limiting for AVL tree 142
7.17 Speculation effectiveness of resource block and frame limiting for quicksort1 143
7.18 Speculation effectiveness of resource block and frame limiting for Gauss-Jordan 144

8.1 Layout of a twisted memory system with four resource blocks 151
8.2 Propagation of a write through twisted memory 152
8.3 Algorithm for a twisted memory write operation 154
8.4 Propagation of a read through twisted memory 156
8.5 Algorithm for a twisted memory read operation 156
8.6 Algorithm for a twisted memory anti-write operation 158
8.7 The effects of an anti-write in twisted memory 160
8.8 Algorithm for a twisted memory anti-read operation 161
8.9 The effects of an anti-read in twisted memory 162
8.10 Algorithm for a twisted memory fossil collection 163
8.11 Ordering time-space cache entries using an epoch counter 163
8.12 AVL average memory bandwidth per cycle for twisted memory 167
8.13 Binary tree average memory bandwidth per cycle for twisted memory . . . 168

8.14 Fibonacci average memory bandwidth per cycle for twisted memory 169
8.15 Gauss-Jordan average memory bandwidth per cycle for twisted memory . . 170
8.16 Matrix multiply average memory bandwidth per cycle for twisted memory . 171
8.17 Quicksort average memory bandwidth per cycle for twisted memory 172

A.1 Conditional C code. 187
A.2 WarpEngine assembly for the C code in Figure A.1. 187

C.1 Minimum VTS length necessary to execute binary tree insertion without rescaling 196
C.2 Minimum VTS length necessary to execute Gauss-Jordan without rescaling 197
C.3 Minimum VTS length necessary to execute matrix multiply without rescaling 197
C.4 Comparison of speedup for binary tree insertion with exponent and length VTSs 198
C.5 Comparison of speedup for Fibonacci with exponent and length VTSs . . . 199
C.6 Comparison of speedup for quicksort 1 with exponent and length VTSs . . 199
C.7 Minimum VTS length necessary to execute binary tree insertion without rescaling 200
C.8 Minimum VTS length necessary to execute matrix multiply without rescaling 200
C.9 Speedup for binary tree with variable range VTSs with rescaling 201
C.10 Speedup for matrix multiply with variable range VTSs with rescaling . . . 202
C.11 Speedup for Gauss-Jordan with variable range VTSs with rescaling 202
C.12 Speedup for AVL(1000) with varying frame limitations and variable range VTSs 203
C.13 Speedup for binary tree (1000) with varying frame limitations and variable range VTSs 204
C.14 Speedup for quicksort 1 (200) with varying frame limitations and variable range VTSs 204
C.15 Speedup for Gauss-Jordan (20) with varying frame limitations and variable range VTSs 205
C.16 Speedup for matrix multiply (30) with varying frame limitations and variable range VTSs 205
C.17 Speculation effectiveness of resource block and frame limiting for binary tree 206
C.18 Speculation effectiveness of resource block and frame limiting for Fibonacci 207

C.19 Speculation effectiveness of resource block and frame limiting for matrix multiply 207

D.1 Twisted memory bandwidth profile to level zero for AVL(200) 209
D.2 Twisted memory bandwidth profile to level one for AVL(200) 209
D.3 Twisted memory bandwidth profile to level two for AVL(200) 210
D.4 Twisted memory bandwidth profile to spatial memory for AVL(200) 210
D.5 Twisted memory bandwidth profile to level two for AVL(200) 211
D.6 Twisted memory aggregate bandwidth profile to spatial memory for AVL(200) 211
D.7 Twisted memory bandwidth profile to level zero for binary tree (300) 212
D.8 Twisted memory bandwidth profile to level one for binary tree (300) 212
D.9 Twisted memory bandwidth profile to level two for binary tree (300) 213
D.10 Twisted memory bandwidth profile to spatial memory for binary tree (300) . 213
D.11 Twisted memory bandwidth profile to level two for binary tree (300) 214
D.12 Twisted memory aggregate bandwidth profile to spatial memory for binary tree (300) 214
D.13 Twisted memory bandwidth profile to level zero for Fibonacci (15) 215
D.14 Twisted memory bandwidth profile to level one for Fibonacci (15) 215
D.15 Twisted memory bandwidth profile to level two for Fibonacci (15) 216
D.16 Twisted memory bandwidth profile to spatial memory for Fibonacci (15) . . 216
D.17 Twisted memory bandwidth profile to level two for Fibonacci (15) 217
D.18 Twisted memory aggregate bandwidth profile to spatial memory for Fibonacci (15) 217
D.19 Twisted memory bandwidth profile to level zero for Gauss-Jordan (10) . . . 218
D.20 Twisted memory bandwidth profile to level one for Gauss-Jordan (10) . . . 218
D.21 Twisted memory bandwidth profile to level two for Gauss-Jordan (10) . . . 219
D.22 Twisted memory bandwidth profile to spatial memory for Gauss-Jordan (10) 219
D.23 Twisted memory bandwidth profile to level two for Gauss-Jordan (10) . . . 220
D.24 Twisted memory aggregate bandwidth profile to spatial memory for Gauss-Jordan (10) 220
D.25 Twisted memory bandwidth profile to level zero for matrix multiply (20) . . 221
D.26 Twisted memory bandwidth profile to level one for matrix multiply (20) . . 221
D.27 Twisted memory bandwidth profile to level two for matrix multiply (20) . . 222

D.28 Twisted memory bandwidth profile to spatial memory for matrix multiply (20) 222
D.29 Twisted memory bandwidth profile to level two for matrix multiply (20) . . 223
D.30 Twisted memory aggregate bandwidth profile to spatial memory for matrix multiply (20) 223
D.31 Twisted memory bandwidth profile to level zero for quicksort 1 (200) . . . 224
D.32 Twisted memory bandwidth profile to level one for quicksort 1 (200) 224
D.33 Twisted memory bandwidth profile to level two for quicksort 1 (200) 225
D.34 Twisted memory bandwidth profile to spatial memory for quicksort 1 (200) 225
D.35 Twisted memory bandwidth profile to level two for quicksort 1 (200) 226
D.36 Twisted memory aggregate bandwidth profile to spatial memory for quicksort 1 (200) 226
D.37 Enlarged region of level zero bandwidth profile graphs for AVL(200) 227
D.38 Enlarged region of level zero bandwidth profile graphs for binary tree (300) 227
D.39 Enlarged region of level zero bandwidth profile graphs for Fibonacci (15) . 228
D.40 Enlarged region of level zero bandwidth profile graphs for Gauss-Jordan (10) 228
D.41 Enlarged region of level zero bandwidth profile graphs for matrix multiply (20) 229
D.42 Enlarged region of level zero bandwidth profile graphs for quicksort 1 (200) 229

List of Tables

4.1 Typical instruction latencies 61

4.2 Complexity of test algorithms 64

5.1 Number of levels which can be represented by different sizes of length and exponential representation VTS 82

8.1 Procedure descriptions for twisted memory pseudocode 154

Chapter 1
Introduction

In recent years integrated circuit process technology has advanced rapidly, providing computer architects with an enormous number of transistors on each CPU. This trend looks set to continue for some time. The computer architect's aim is, as ever, to produce CPUs capable of faster execution of a wide range of programs. Using those resources effectively in a processor provides a substantial architectural challenge.

There are several approaches to faster computation. Clock frequencies are being aggressively pushed beyond the 1GHz mark, largely governed by the physical process technology, although CPU architecture has some influence. Some architects are proposing ever deeper pipelines [Intel, 2000] in an effort to allow higher clock frequencies.

In the past architectures have been designed with the aim of reducing the instruction count of programs [Thornton, 1964]. However this has largely been abandoned in favor of RISC instruction sets [Patterson, 1985], which contain simpler instructions, allowing higher clock frequencies and execution in fewer cycles than a typical CISC instruction.

Parallel processing has long been used in high performance scientific computing to speed up program execution. Since the 1990's many of the same techniques have been adopted in general purpose processors. By executing many instructions in parallel the enormous resources available on a modern integrated circuit (IC) can be used to speed up computation.

Early parallel processing relied on programmer intervention to control the parallelism, indicating instructions that could be executed concurrently. This requires skilled programmers to extract maximum performance and is only cost effective for performance critical code.

There is a substantial cost associated with speeding up legacy code by parallelizing it in this manner.

A more practical alternative, particularly for general purpose processing, is to *automatically parallelize* programs through a combination of processor and compiler assistance. Extracting high levels of parallelism relies on *identifying* and *exploiting* the available parallelism in the program.

1.1 The Challenge

Identifying parallelism is a matter of identifying independent instructions—the only instructions that can be safely executed in parallel. Memory operations can only be established as independent by comparing the addresses of the memory accesses, which are frequently calculated at runtime.

A large proportion of existing programs for general purpose processors are written in pointer based languages such as C and Pascal. It is often impossible to determine the address of a pointer statically, and even at runtime it may be unknown until just before the instruction executes. Typically most memory accesses will be to different addresses, but this parallelism can not be identified until the addresses are known.

As the differential between processor and memory performance continues to increase, parallelizing memory operations is becoming increasingly important as a way to hide memory latency.

Speculative execution, the approach utilized in this thesis, assumes all memory operations are independent and relies on rectifying any incorrect parallel execution after the fact. This requires that the order of operations be tracked, so that dependency violations can be identified. Previously proposed methods of tracking dependencies, described in Chapter 2, do not scale well and place a fundamental limit on the degree of parallelism that can be extracted by speculation.

Tracking the instruction order can be done using fixed hardware structures or explicit ordering tags. Hardware structures have been extensively explored in the literature and are reviewed in Chapter 2. This thesis investigates the feasibility of attaching an explicit tag,

known as a virtual timestamp (VTS), to each memory operation as a scalable, distributed method of tracking causal ordering.

Realizable VTSs are necessarily a finite, linear resource, since their order is vital. Allocating VTSs efficiently at the instruction level requires analysis of the number of instructions in each region of code well in advance. This thesis develops a framework for such analysis and proposes some basic methods. Allocation of other execution resources can also benefit from similar analysis.

A more efficient use of resources can be achieved by restricting speculation to code for which execution resource usage can be determined. More detailed resource use analysis also facilitates allocation, enabling a simpler organization of resources.

Out of this work a hierarchical ordering system is developed using fixed hardware ordering and explicit tags at different levels. The hierarchy forms part of a multiple version memory system used to recover from mis-speculation once it has been identified.

All of these ideas are investigated within the framework of simulations of a theoretical architecture known as the WarpEngine. The WarpEngine is an aggressively speculative architecture capable of exercising the instruction ordering mechanisms and memory system to the limits of the parallelism available in the test programs. This makes it a useful test-bench for exploring structures designed to operate with very high levels of parallelism.

1.2 Thesis Claims

The WarpEngine simulator can be used to show the effects on parallelism extracted from programs caused by different speculation tracking and selection mechanisms. The restricting effects of the different mechanisms can be isolated because the WarpEngine fundamentally allows a large amount of the available parallelism in a program to be extracted. In this context the simulation results presented in this thesis support the following major claims:

- Analysis of program resource requirements is necessary to allow aggressive speculation on sequential programs.
- Speculating preferentially on code which is amenable to analysis simplifies the hard-

ware, but does not substantially constrain performance.

- Allocating execution resources hierarchically allows a processor to take advantage of easily analyzed code, while limiting the difficulties of opaque code.
- Splitting programs into analyzable and unanalyzable tasks allows speculation resources to be used more effectively.
- A hybrid hardware ordered and tagged speculative memory system using a network of distributed caches can operate scalably to maintain the causal consistency of speculative memory accesses without exceeding reasonable bandwidth requirements.

1.3 Thesis Overview

Chapter 2 considers some popular *instruction level parallelism (ILP)* techniques. In all ILP mechanisms the dependencies between instructions (real and perceived) limit the amount of parallelism which can be extracted. The ways in which these dependencies arise are discussed and some methods that have been used to detect them and expose parallelism available in programs are outlined.

Although the WarpEngine draws on several ILP techniques, the primary method is speculative execution. It is important to understand the distinction between the different classes of speculation. A number of existing commercial and research architectures use speculative execution, albeit on a less aggressive basis than proposed in the WarpEngine. A discussion of the main techniques used for controlling speculative execution, particularly recovery after a mis-speculation, concludes the chapter.

Chapter 3 provides a general overview of the WarpEngine architecture, first laid out by Littin [2000]. The WarpEngine paradigm is based on the Time Warp algorithm and is developed from this background. It is compared and contrasted with other prominent ILP paradigms. An implementation of the essential WarpEngine components is described as a baseline for performance analysis to which the advanced systems developed later in the thesis can be added.

Chapter 4 describes the simulation methodology used throughout the thesis. The micro-benchmarks used in simulations are characterized, and the particular issues involved in

compiling and running them on the WarpEngine simulator are considered. An overview of important earlier results obtained from simulations of the WarpEngine is also included.

The following four chapters propose and investigate several novel mechanisms for tracking the causal order of memory operations using several approaches. Chapters 5 and 6 consider different schemes using explicit VTSs. Chapter 5 exposes the shortcomings of a naive VTS allocation method. Chapter 6 describes allocating VTSs more efficiently by analyzing the requirements of each region of the program. Constraints imposed by VTSs are compared to the effects of constraining other execution resources.

Chapter 7 expands VTS requirement analysis to control resource allocation and speculation more generally. A new method of selective speculation, based on the ability to determine resources required, is proposed. The fundamental limit this places on parallelism, for the benefit of easier implementation is explored through simulations. Building on these resource allocation techniques Chapter 8 proposes a hierarchical speculative memory system which uses explicit VTSs and hardware ordering to maintain memory dependencies. Simulation results for the WarpEngine using the *twisted memory system* are presented, with particular focus on the memory bandwidth consumed at different levels of the memory system.

Chapter 9 draws out the key conclusions from the work presented, along with promising aspects for further investigation.

Chapter 2

Controlling Instruction Level Parallelism

This chapter examines issues involved in extracting and controlling *instruction level parallelism (ILP)* in general purpose computation. The particular focus is on techniques applicable to the large volume of existing general purpose code written in imperative languages. The cost of rewriting this code in special purpose parallel languages is too great to be feasible for all but the most performance critical applications. Indeed, it is difficult to persuade the majority of programmers to learn a new language for performance reasons alone. For these reasons automatically extracting parallelism, using either hardware or compiler methods, will continue to be important.

It is the dependencies within a program that place fundamental restrictions on the parallelism that can be extracted. The following sections classify the dependencies that may exist and outline some of the ILP studies that have been done.

The architectural techniques discussed below for exploiting ILP are not necessarily mutually exclusive, nor are there clear boundaries between the categories. Many prominent commercial and research architectures use techniques from several classes. For example, the EPIC architecture devised by Intel and Hewlett-Packard [Schlansker and Rau, 2000] is a very long instruction word (VLIW) architecture, but also includes support for speculation. Superscalar architectures, such as the Intel x86, Compaq Alpha and IBM PowerPC, have dominated the market in recent years. In the quest for more parallelism and increased performance they have begun to exploit other ILP techniques, and much academic research has

focused on this area.

Although the WarpEngine includes dataflow and superscalar techniques (see Chapter 3), it is primarily used to investigate speculative execution techniques. The remainder of this chapter contains a taxonomy and description of speculative execution and an overview of the major research architectures exploiting this type of parallelism. Of principle interest in the context of this thesis is how mis-speculation is identified and recovered from, particularly in the memory system.

2.1 Instruction Dependencies

The execution time of a program depends on three factors:

- the number of instructions executed per clock cycle (IPC)
- the clock period
- the total number of instructions executed

The emphasis in this thesis is on improving the first factor, executing more useful instructions per clock cycle through ILP, without significantly impacting the other two factors. If ILP is increased to the detriment of clock frequency or the dynamic instruction count overall performance may actually decrease.

Extracting maximum ILP from a program means executing as many instructions as possible in parallel. Even assuming that sufficient computational resources are available there is a limit to the parallelism imposed by the structure of the program. During the course of executing a program some instructions will require input data to be calculated by another part of the program (*data dependence*). At other times a decision to branch to an optional section of code or to continue a loop will need to be made. Typically this depends on data calculated previously and is known as *control dependence*.

2.1.1 Data Dependence

Data dependencies place a fundamental limit on parallel execution, since the value of intermediate results must be computed before they can be used in further computation. These dependencies are also known as *true data dependencies* or *read after write hazards*.

Some apparent dependencies result from the way the program has been implemented, rather than the underlying data usage, these are *false data dependencies*. They typically result from register or memory location reuse, for example two instructions writing to the same register (a *write after write hazard* or *output dependence*) must be done in the defined order, but the operations do not rely on each other. Similarly, where a write to a register follows a read from the same register (a *write after read hazard* or *anti-dependence*) they must be done in the proper order, although there is no logical dependence in the underlying computation.

However, it is not always possible to determine in advance whether a data dependence exists between two instructions [Tjaden and Flynn, 1970]. This is often the case with memory operations which calculate the address at runtime. The memory access is potentially dependent on all other memory operations until the address is known.

2.1.2 Control Dependence

The results of computation can determine not just the input data for further instructions, but also which instructions execute. The outcome of a conditional branch instruction, based on some previously calculated value, determines which instruction executes next. A *control dependence* exists between the instruction following the branch and the instruction calculating the value used in the branch. Control dependencies determine the dynamic execution path of the program. This property is necessary in order to implement dynamically bounded loops (for example `while` loops) and conditionally executed statements (for example `if...else` constructs).

Certain code structures can cause the possible execution paths of a program to re-converge after a branch. In Figure 2.1 either region B or region C executes depending on the outcome of A, but region D always executes, regardless of A's result. D is *control independent* [Rotenberg et al., 1999b] of A, and D can be executed in parallel with A (and B or C), as

long as data dependencies are respected.

Control independence is useful in determining code that can be executed speculatively because it is independent of any intervening branch predictions which may be made incorrectly. In Section 3.2.2 we discuss a novel way for identifying and exploiting this property.

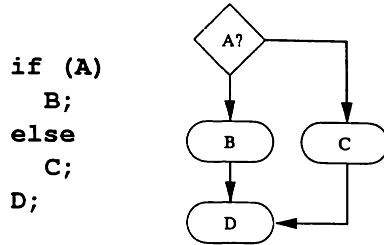


Figure 2.1: Code with control independence

2.1.3 Limit Studies

Modern CPU process technology has advanced to the stage where it is a challenge to keep all the available resources on a chip busy with computation. Fundamentally this means executing more instructions in parallel. However, dependencies place a critical theoretical upper bound on how much parallelism can be extracted. A number of studies have been done to identify the limits under different conditions. They differ in the mechanisms allowed and the assumed accuracy of these mechanisms.

Early studies of available ILP are quite pessimistic. Jouppi and Wall [1989] and Smith et al. [1989] suggest that a maximum of around 2 instructions per cycle (IPC) are available from a variety of integer applications (floating point applications tend to have more inherent parallelism) for realistic designs. Investigations by Butler et al. [1991] are more encouraging, finding that removing constraints not required by the semantics of the program allows up to 17 IPC. With hardware constraints reasonable for the time this was limited to up to 5.8 IPC.

A study by Wall [1991] showed that while speedup under the the assumption of perfect knowledge could reach around 60 times, with mechanisms which are unrealistically good, but not perfect, this is reduced to below 7.

Lam and Wilson [1992] recorded similarly disappointing figures for machine organizations judged realistic at the time. However, the addition of control flow speculation allowed much

higher levels of ILP to be extracted. In some cases these approached theoretical levels proposed for VLIW architectures [Nicolau and Fisher, 1984], which had been criticized as unrealistic for using perfect branch prediction, and using numerical programs which are more amenable to extracting parallelism.

More recent studies [Lipasti and Shen, 1996] suggest that these limits can be exceeded through the use of control and data speculation, and value prediction. Postiff et al. [1999] eliminated false dependencies and utilized a large split instruction window to show speedup of over 30 may be possible for the popular SPEC suite of benchmarks. They found that employing multiple instruction windows can expose much more parallelism than previously observed. Austin and Sohi [1992] analyzed the data dependence profile for an earlier version of the SPEC suite and found similar levels of parallelism available, but also found that the parallelism is bursty in nature. This suggests that ILP architectures will need to be able to support parallel execution at higher levels than the average inherent parallelism to extract maximum performance.

The main point of argument in these limit studies is over which techniques and assumptions are realistic. Ongoing research has begun to make initially optimistic assumptions look much more reasonable. Branch prediction and out-of-order execution allow parallelism to be extracted beyond the basic block, and more recently speculative execution and data prediction, coupled with large instruction windows have further extended the reasonable limits. A large gap still exists between the limits imposed by logic and those by implementation, providing plenty of scope for improved implementation techniques.

2.2 Conservative Parallel Execution

To guarantee correct results from computation a processor must ensure that no dependencies are violated and that dependent instructions are executed in the correct relative order. This condition is known as *sequential consistency* [Lamport, 1979]. The simplest method of maintaining sequential consistency in a program is to assume that each instruction is dependent on the previous one. Only once an instruction has completed and its results written to storage can the next instruction begin retrieving its input data from storage. Of course this severely constrains the program execution speed. Since each instruction must wait for

the prior one to complete, the execution time is the cumulative time for all instructions in the program to execute.

Pipelining [Ramamoorthy and Li, 1977] is a commonly used technique to overlap instruction execution by dividing it into distinct stages. If an instruction waiting to be executed is dependent on an instruction still in the pipeline the instruction cannot be executed safely and must be stalled.

A processor that can execute instructions in parallel must still be sequentially consistent, but to extract the maximum amount of ILP it needs to identify as many independent instructions as possible. The strategies for doing this can be divided into two broad categories, conservative and speculative execution. *Conservative execution* identifies known independent instructions which can be executed in parallel, and assumes all other instructions contain dependencies. *Speculative execution* avoids executing in parallel those instructions known to be dependent. Any other instructions may be executed in parallel and if a dependency violation is later identified the processor must be able to undo the effects of the dependent instruction. In this section and the next some conservative and speculative execution techniques currently in use are discussed.

2.2.1 Vector

A limited class of programs, mainly scientific applications, perform the same operation repeatedly on a large set of data, such as a matrix. Vector processors take advantage of this characteristic by allowing instructions to operate on many values at once. Vector operations must operate independently on each data value and the programmer must be able to determine the independence statically and indicate this in the program.

Cray supercomputers [Russel, 1978] are successful examples of vector processors. Unfortunately most widely used programs do not contain significant amounts of this sort of parallelism and even those which do require a large amount of programmer time to rewrite them to exploit this parallelism. Vector parallelism is of minimal use in general purpose processors, other than some graphics and multimedia instruction set extensions, such as Intel's Streaming SIMD Extensions [Intel, 1999].

2.2.2 Dataflow

Rather than considering computation as a control driven process with operations proceeding in a defined order using data computed by earlier operations, it can be considered as a data driven process. Dataflow machines process instructions asynchronously as soon as all the input data (known as *tokens*) are available. As such there is generally no shared set of registers, or shared memory, tokens are sent directly to instructions and each token is used only once.

A dataflow program is derived from a *dataflow graph* which shows the transfer of data between instructions. Synchronization is handled simply in dataflow machines by instructions waiting for data from instructions they are dependent on, constraining the order of execution, but leaving execution of independent operations unconstrained. False dependencies are eliminated by using each token only once.

Conditional branches are implemented using dataflow instructions which send a copy of the input token to different instructions based on a control input. In this way a control flow dependence has effectively been converted to a dataflow dependence.

The first dataflow architecture—the Static Dataflow Machine—was developed at MIT [Dennis and Misunas, 1975]. Dataflow architectures that have been developed since include: the Manchester Prototype Dataflow Computer [Gurd et al., 1985], the Stateless Dataflow Architecture [Snelling, 1993], the StarT family [Nikhil et al., 1992; Ang et al., 1998] and the MIT Monsoon [Papadopoulos and Culler, 1990]. A number of extensive surveys have been published comparing these and other dataflow architectures [Veen, 1986; Srini, 1986; Treleaven et al., 1982; Snelling and Egan, 1994].

While dataflow architectures promised much as clean, scalable ILP architectures, major practical weaknesses have been discovered. In programs with a high degree of inherent parallelism the instructions available for execution can far outstrip available resources, causing large scheduling overheads. Furthermore, instructions may have long lifetimes, between the first token arriving and the instruction being ready for firing. This leads to a large number of instructions active in the system, causing difficulty in matching tokens with the respective instruction. The problems in managing massive parallelism are revisited later in this thesis.

Despite the high degree of parallelism often available, sequential bottlenecks tend to form

in the queues and stores used to communicate tokens between instructions, and contribute to long latencies. Since data flows from instruction to instruction without being stored in a common memory (conceptually at least), data used by more than one node has to be copied. This becomes a problem where large data structures are involved, adding more overheads [Gaudiot, 1986].

The dataflow programming model is quite different from control driven programming [Veen, 1986] and, combined with the problems mentioned above has, led to a poor acceptance of dataflow processors for general purpose computing. However, some of the ideas introduced by dataflow computing have had an influence on control driven processor designs, most particularly in out-of-order execution.

2.2.3 Very Long Instruction Word

The strategy for extracting parallelism employed by VLIW architectures is that the compiler forms fixed length *packets* of operations which it has determined are independent [Colwell et al., 1987; Fisher, 1984]. Operations within a packet are issued in parallel and can be executed without concern for synchronization. Typically the type of operation which can be placed in each location in the packet is constrained and function units are grouped appropriately to allow concurrent execution of all operations in the packet, while minimizing the number of function units. This technique was first seen in the CDC6600 [Thornton, 1964], which used packages of up to four instructions. Examples of a pure VLIW processor include the Trace Machine and the ELI-52 [Fisher, 1983].

VLIW puts the burden for identifying parallelism on the compiler, which can do extensive analysis and code rearrangement without adding a runtime overhead. Clearly VLIW limits the parallelism which can be extracted to the number of operations in a packet, unless superscalar techniques are used to execute more than one packet concurrently. A major advantage is that the processor can be simplified substantially, although static analysis cannot identify as much parallelism as dynamic methods.

To increase the maximum parallelism that can be extracted the size of an instruction can be increased. However, this makes it more difficult for the compiler to fill packets efficiently. Longer instructions can increase the overall fetch rate, since the atomic unit is larger, but if

packets are not efficiently filled, bandwidth is wasted and code size increases. Additionally, if the instruction size, or function unit layout changes between generations of VLIW machines then, unlike superscalar processors, they will no longer be binary code compatible. These drawbacks have prevented pure VLIW processors from taking off. However, like dataflow, the ideas have influenced commercial superscalar architectures.

The explicitly parallel instruction computing (EPIC) architecture [Schlansker and Rau, 2000] developed by Intel and Hewlett-Packard is a major commercial architecture utilizing VLIW techniques. Each *bundle* of EPIC instructions may contain up to three instructions, although bundles may be grouped to execute in parallel. In addition to the central VLIW mechanism EPIC incorporates superscalar techniques, such as predication, load speculation and a prepare-to-branch instruction to further enhance parallelism.

2.2.4 Out-of-order Execution

In the programmer's model of a processor *in-order execution* is assumed, that is instructions execute in the order in which they appear in the program. If the processor actually executes the instructions in order the burden is placed on the compiler to schedule instructions efficiently. The limited information available at compile time can cause non-optimal scheduling, which may lead to the parallelism that can be extracted being restricted by execution stalling while there are other instructions ready to be executed. If independent instructions appear later in the program order while a stalled instruction is at the head of the issue queue, they all have to be stalled until the dependent instruction can execute. By allowing instructions to be executed *out-of-order* further ILP can be extracted. Typically a limited *window* of instructions is considered for issue at any time.

Executing out-of-order requires the CPU to perform dependency analysis on instructions in the instruction window to identify those that can be safely executed. An instruction can only be executed if the processor can determine that it is not dependent on any incomplete instructions. Both data and control dependencies must be tracked.

Scoreboarding [Thornton, 1964] was the first mechanism to track data dependencies by maintaining centralized tables showing where data dependencies between incomplete instructions and the next instruction to be issued exist. Entries identify registers which will be

written to by instructions in progress, requiring instructions that read those registers to wait for the new values to become available. Outstanding register reads are also tracked so that write-back to a register can be stalled if the previous value is still required by an instruction. Scoreboarding was designed to allow a processor to issue one instruction per cycle in-order and complete instructions out-of-order where data dependencies allow.

Tomasulo's algorithm [Tomasulo, 1967] performs a similar function using a distributed mechanism which places results in temporary slots known as *reservation stations*, as well as in registers. If an operand is not available for an instruction at issue time a tag identifying the reservation station for the value is used instead of the register value. Once the value has been calculated and placed in the reservation station the instruction can proceed with execution.

The most commonly used method of removing false data dependencies is *register renaming* [Keller, 1975], which makes use of a large register set. The *architectural registers* visible to the programmer are dynamically assigned to internal *physical registers* such that each register is only written to once within any instruction window. To make this easier the set of physical registers is usually much larger than the set of architectural registers. Instructions reading from the architectural registers are mapped to the correct physical registers, removing false data dependencies.

Register renaming can also be implemented as a modification to Tomasulo's algorithm. Each instruction has a reservation station assigned for its result. Any future instruction that uses that result as input data either retrieves the value from the reservation station, or associates the reservation station tag with its input so it can execute with the value as soon as it has been computed.

No instructions beyond an unexecuted conditional branch can be executed conservatively since there is a control dependency on the branch instruction. This restricts parallel execution to the current basic block. Section 2.3 considers speculative methods commonly used in commercial processors to look beyond the basic block.

2.3 Speculative Parallel Execution

In general more parallelism can be extracted from a larger pool of instructions, since there will be more independent instructions available. However, it becomes increasingly difficult to identify independent instructions in a larger instruction pool, since each instruction must be considered against all previous instructions in the pool. At times this information is not even available, for example a memory address may have to be calculated.

Speculative execution assumes instructions are independent, and proceeds with parallel, out-of-order execution. If this assumption turns out to be false (a *mis-speculation*) the effects of the dependent instruction must be undone. The final effect of the computation must be the same as if the program had been executed sequentially.

Speculative execution implies out-of-order execution and dynamic scheduling, allowing possibly useful computation to utilize otherwise idle resources in the hope of gaining speedup. Speculative execution allows the ultimate flexibility to schedule instructions in parallel at runtime. Statically scheduled methods, such as VLIW rely on the compiler to identify parallelism at a time when useful information, such as memory addresses, may not be available. Conservative dynamic methods require a guarantee that an instruction is independent of all those prior to it. Late calculation of memory addresses may dictate substantial delay before this information is available.

Speculative execution is receiving a lot of research interest and is the primary focus of this thesis. Both data and control flow can be speculated on, although they require different coordination methods.

2.3.1 Control Speculation

In order to issue more than one instruction per cycle a superscalar processor must fetch multiple instructions per cycle. When the current instruction is a conditional branch the next instruction to be fetched is dependent on the outcome of the branch. In order to maintain maximum fetch rate *branch prediction* can be used to try to determine the next instruction [Smith, 1981; McFarling, 1993; Yeh and Patt, 1993]. This is the most common form of control speculation, and is found in most modern processors. If the branch is mispredicted

the wrong instruction will have been fetched, and perhaps executed. The effects of this must be undone and the correct instruction executed.

A branch instruction occurs, on average, approximately every seven instructions [Hennessy and Patterson, 1996]. To maintain a high issue rate, particularly in the presence of dependent instructions, it is often necessary to predict several branch outcomes. Although prediction accuracy across one branch of greater than ninety percent is possible, since each branch is dependent on the previous one in a linear instruction sequence the probability of prediction quickly decreases, limiting the effective instruction look-ahead [Yeh et al., 1993].

By recognizing control independence (see Section 2.1.2) the problem of decreasing branch prediction accuracy can be curtailed. Since the code is guaranteed to reconverge at the control independent point the branch prediction accuracy is restored. However, exploiting control independence relies on having selective mis-speculation recovery, squashing dependent instructions while preserving independent instructions.

Most conventional pipelined architectures do not attempt to take advantage of control independence. When a branch prediction error occurs they simply squash all execution beyond of the prediction error, whether it was control dependent or not. Lam and Wilson [1992], and Uht and Sindagi [1995] have shown that control independence is necessary to extract ILP on the order of 10 to 100. Rotenberg et al. [1999b] provided a more detailed study, showing that control independence can moderate the penalty of branch mispredictions using a single flow of control. Rotenberg and Smith [1999] extended this work to multiple flows of control using the trace processor. Other processors which exploit control independence by pursuing multiple flows of control include the multiscalar [Sohi et al., 1995], dynamic multithreading [Akkary and Driscoll, 1998], the superthreaded architecture [Tsai et al., 1999] and the Stampede architecture [Steffan et al., 2000].

Predication [Mahlke et al., 1995] is a technique which can be used to expose control independence in some situations. A predicate is a boolean value attached to an instruction which must be true for the instruction to be executed. A conditional branch could be used to achieve the same execution path. If two different sections of code are executed depending on a condition their predicates have complementary values. Predication essentially converts a control dependence to a data dependence, allowing code following a conditional branch to be fetched early because it no longer appears control dependent on the branch.

Predication can be combined with speculative execution [August et al., 1998] to execute conditional code early by executing both branch outcomes. If the predicate is true the instructions can be committed, if not they are discarded. The disadvantage is that more resources are consumed because both branches are always executed, but only one will ever be committed. This style of predication is used in the EPIC architecture.

Eager execution [Uht et al., 1997] also executes both branches. As with predicated execution, resources are consumed by both branch outcomes, but eager execution allows both paths to be speculatively executed across multiple branches. If the execution path does not reconverge the resources consumed by speculative execution soon explode. A modification known as Disjoint Eager Execution (DEE) [Uht and Sindagi, 1995] assigns a confidence to each speculative branch, which is cumulative across all speculative branches to that point. To regulate resource consumption only the most likely execution paths are speculatively executed.

2.3.2 Data Speculation

When an instruction is executed speculatively there is a chance that it is data dependent on an instruction which has not completed, or has been executed speculatively. This means that either the input data has not been computed, or may have been computed incorrectly.

Generally register data dependencies can be dealt with conservatively using register renaming (see Section 2.2.4). However, memory values require a different approach, since there are too many addresses to use renaming, although attempts have been made to adapt register renaming to memory operations [Tyson and Austin, 1999].

Memory addresses are often calculated dynamically, which further complicates dependence analysis. Until all earlier memory addresses have been calculated the independence of a memory instruction cannot be established, so conservative execution requires it to stall. This restriction becomes more severe with a larger instruction window because more instructions must be considered for possible dependencies.

Memory data speculation can be done naively by assuming that all instructions are independent. This can lead to a large number of mis-speculations and wasted computation in aggressively speculative systems. By predicting which accesses are dependent the instruc-

tions most likely to be committed can be speculatively executed. Selective speculation can be used to mitigate the effects of false memory dependencies at a number of different levels of prediction [Calder and Reinman, 2000].

Data dependence prediction [Gharachorloo et al., 1991; Moshovos et al., 1997; Chrysos and Emer, 1998] predicts the presence of a dependence, allowing operations likely to be independent to commence as soon as the address is available.

Address prediction [Austin and Sohi, 1995; Chen and Baer, 1995; Gonzalez and Gonzalez, 1997] goes a step further, speculating on the address, which can be used in dependence analysis and to initiate the memory operation early.

Value prediction [Lipasti and Shen, 1996, 1998; Sazeides and Smith, 1997; Gonzalez and Gonzalez, 1998; Wang and Franklin, 1997] predicts the value returned by a load, allowing instructions dependent on the load to execute early. Value prediction can be used on both memory and register values.

Moshovos and Sohi [2000] performed a study on memory dependence speculation in a continuous instruction window processor. They concluded that if addresses are calculated as early as possible and used to disambiguate memory addresses most of the speedup can be obtained without speculation. However, increasing the size of the instruction window leads to increased latency in the address disambiguation unit, which rapidly undermines the effectiveness of this technique. It is also inappropriate for split instruction window processors, since there may be memory dependencies on instructions that have not yet been fetched.

These prediction methods even have the potential for overcoming true data dependencies, allowing speedup beyond the theoretical dataflow limit. Overall performance depends on prediction accuracy and on efficiency of recovery from mispredictions. These are areas of ongoing research.

2.3.3 Architectures

The leading commercial superscalar architectures include many of the techniques described above to maintain an instruction issue rate in excess of 1 IPC. Pipelining, register renaming

and branch prediction are ubiquitous and out-of-order execution is common. For example, the Compaq Alpha 21264 [Compaq, 2000] has a 6 stage pipeline and issues up to 4 instructions per cycle out of order from an instruction window of 80 instructions, using register renaming and branch prediction. The Intel Pentium 4 [Intel, 2000] has a 20 stage pipeline and issues up to 6 instructions per cycle out of order from an instruction window of 126 instructions, using register renaming and branch prediction.

By way of example, and for comparison with the WarpEngine, as described in Chapter 3, this section outlines speculative techniques employed by some of the leading research architectures, illustrating the different approaches.

Multiscalar

The multiscalar processor [Sohi et al., 1995; Franklin, 1993] divides the control flow graph (CFG) into subgraphs known as *tasks*. Tasks are selected to be control independent as much as possible, and are speculatively executed in parallel on a ring of processing elements. If a data or control dependence between tasks is violated the speculative task is removed (*squashed*) along with all more speculative tasks. Tasks split the instruction window and may be arbitrarily far apart in the dynamic instruction stream.

The multiscalar processor does not allow for selective re-execution of mis-speculated instructions. This can be disastrous to performance if mis-speculations occur frequently since independent correct computation will have to be repeated. The multiscalar relies on good task and data dependence prediction both by the compiler and dynamically to minimize squashes [Vijaykumar and Sohi, 1998]. This prevents the multiscalar from scaling to a large number of processing elements.

Trace processor

The trace processor [Rotenberg et al., 1997] is similar to the multiscalar in using a circular queue of processing elements, and a distributed register file and instruction window. However, it uses dynamic instruction sequences (*traces*) instead of tasks. As the program executes the dynamic instructions sequences are recorded in a *trace cache*, terminating at control independence points, in the hope that they will be executed again [Rotenberg

et al., 1996, 1999a]. In effect traces are variable length, dynamically composed blocks in a block structured architecture (see Section 3.2.1). Future re-execution of traces are predicted and speculatively executed. Value prediction is performed on values being passed between traces and on memory addresses, and memory loads are performed speculatively. Mis-speculation is expected to occur regularly, so only dependent instructions are re-executed to reduce the penalty.

Dynamic multithreading

Dynamic multithreading [Akkary and Driscoll, 1998] identifies control independent regions at runtime and begins *threads* of execution at those points. Threads contribute instructions to a hierarchy of instruction windows from which instructions can be selected for execution. All threads except one execute speculatively and use data speculation to avoid stalling for inputs. This leads to frequent mis-speculation which is selectively recovered from by reissuing affected dependent instructions.

Any thread, speculative or not, may spawn a new thread. To track the causal order of threads a tree is maintained, with new threads added as the right-most child node of the creating thread. A right to left preorder traversal provides the thread order. Threads most recent in the causal order have preemptive priority for thread contexts, ensuring computation that is less speculative is done by preference.

Speculative instructions and results are stored in a *trace buffer* attached to each thread context, which operates individually as a dataflow engine. Speculative register values are retired to the architectural state in order. As they are retired the speculative values used for the following thread are validated and rolled back if necessary.

Speculative multithreading

The speculative multithreaded processor [Marcuello and Gonzlez, 1998] executes multiple loop iterations speculatively in parallel. Loops are highly predictable and each thread shares the same control flow, so the instructions only need to be fetched once. Data dependence and data value speculation are used extensively to avoid inter-thread dependencies. Extensive run-time analysis is used to predict register dependencies, and execution is stalled until the

data is available.

This technique is limited because it operates only on loop bodies, so much of the available parallelism is ignored. However, speculative multithreading is complementary to the techniques mentioned above, which generally operate at the higher level of granularity of control independent tasks.

2.4 Recovering from Mis-speculation

All speculation techniques require a way of recovering from mis-speculation to remove incorrect computation from the system. In this section the major recovery methods are reviewed. The sophistication of the recovery methods required is influenced by the speculation technique used, and how aggressively it is pursued.

2.4.1 Pipeline Flushing

Typically superscalar processors execute conservatively with respect to data dependencies, but speculate on control flow by means of branch prediction. Hence, in pipelined processors (like most modern processors) the instructions in the pipeline may represent speculative state.

Instructions are fetched and executed based on the predicted branch outcome, and may progress as far as the retirement stage of the pipeline. If the branch prediction was correct the instructions are allowed to commit their results and retire. If the prediction was incorrect the contents of the pipeline up to the mispredicted instruction furthest through execution are removed (*flushed*), along with any results computed by them, and the correct instructions are issued. This inevitably creates a pipeline bubble while the pipeline is refilled, and there will be some latency associated with flushing the pipeline.

Branch prediction within a pipeline is a modest form of speculation, so it is easily recovered from. The penalties are minimal providing the branch prediction accuracy is high, and the pipeline is short. However, not much speculation is exploited with this method.

2.4.2 Checkpoint Repair and History Buffer

A straightforward way of recovering from mis-speculation is to restore the processor to a state prior to the incorrect execution and re-execute from that point with the corrected data or branch outcome. *Checkpoint repair* [Hwu and Patt, 1987] periodically saves the complete processor state. This may include the results of some speculative or incomplete instructions, which are added to the saved state as they become available.

When a mis-speculation is detected the most recent correct checkpointed state (which may be several instructions earlier than the mis-speculation) is reloaded and all later instructions are re-executed. Checkpointed states are discarded when a later state contains only committed values, when all the results of the in-flight instructions have been added.

The overhead in checkpointing after every instruction and the resources required to store the state would be impractical. This leads to the state being restored to an earlier point in the program than necessary, and some correct computation being repeated.

A *history buffer* [Smith and Pleszkun, 1988] performs the same function, but stores the change to the state, rather than a complete copy of the state. This is done by updating the register file with speculative values and storing a history of superseded values in a last in, first out stack. When a mis-speculation occurs values are popped off the stack and inserted in the register file until a safe state is reached. Saving incremental state is much faster and more space efficient than checkpointing, but it requires extra register file ports to update the history from multiple instructions per cycle, and popping values from the history buffer requires several cycles to recover from mis-speculation.

2.4.3 Reorder Buffer

Another way to allow a processor to recover from mis-speculation is to keep speculative results in temporary storage, and only commit them to the register file when they become non-speculative. Instructions are issued in-order to the *instruction reorder buffer (ROB)* [Smith and Pleszkun, 1988], which places them in slots in order. Data can either be placed in the ROB slot, or separate reservation stations can be used. When a new instruction is placed in the ROB it uses the most recent values from the ROB for its register values, or

if none are present in the ROB, from the register file. Instructions are then dispatched to functional units on a dataflow basis, as soon as all the operands are available. The instruction result is returned to the ROB entry and stored there pending retirement. Register renaming ensures that register data dependencies are observed. Typically memory operations have their relative execution order constrained to avoid possible memory hazards.

Instructions are retired from the ROB in order, to preserve sequential consistency. If an instruction has completed when it reaches the head of the ROB its result is written to the register file and it is removed from the ROB. If it has not completed retirement stalls until it completes and the result is returned. Typically several instructions can be retired in a single cycle.

The size of the instruction window is determined by the ROB and branch prediction is used to fetch enough instructions to keep it full. If a misprediction occurs the ROB is flushed beyond the misprediction and is refilled with instructions from the correct branch outcome. This is a low latency mis-speculation recovery.

Unfortunately, even relatively large instruction windows can become exhausted. If a long latency instruction occurs, such as a cache miss, it can be at the head of the ROB and incomplete while all the other instructions have completed, leaving none to issue. Increasing the size of the ROB is a possible solution, but it leads to other problems. To find the register input value for an instruction the ROB must be associatively matched, and worse, the most recent value must be identified. This centralized mechanism scales poorly to larger ROB sizes.

Associative matching can be avoided by using a *future file* [Smith and Pleszkun, 1988], which is a duplicate register file holding the speculative register values (or reservation station tags) that apply to the most recent instructions issued from the ROB. Mis-speculation recovery becomes slightly more complicated. Instructions up to the mis-speculation point must be completed and retired to the register file. Then the future file can be flushed and the remaining ROB entries removed. Execution is then restarted by refilling the ROB. This slows recovery somewhat and requires an additional register file, with the advantage of speeding up instruction issue.

2.4.4 Split Instruction Window Squashing

All the mis-speculation recovery methods discussed to this point have been suitable for architectures using a single flow of control, and are commonly applied to superscalar architectures. Squashing all speculative instructions over multiple flows of control can have much more serious consequences because all but one of the threads will be speculative and a lot of correct computation may be rolled back. The advantage is that the squash is very simple and can be performed quickly. Much less state saving information is required than to selectively undo incorrect instructions.

In the multiscalar processor, in principle, all tasks beyond the one containing the mis-speculation are squashed, although the compiler does create tasks to minimize data and control speculation errors. Memory accesses to global variables are most likely to contain intertask dependencies, and can be explicitly synchronized by compiler intervention [Sohi et al., 1995]. If a branch misprediction occurs in a task and the following tasks are control independent they are not squashed, giving a limited degree of selective rollback. Any intertask data dependence affected by the branch misprediction causes speculative tasks to be squashed.

Register dependence violations are avoided by using reservation stations to await values in a dataflow manner. A number of different ways of synchronizing memory accesses have been proposed for multiscalar processors, these are considered in Section 2.4.6, along with other speculative memory techniques.

Other split instruction window architectures, which squash indiscriminately on similar principles to the multiscalar, include: the Stanford Hydra [Hammond et al., 1998, 2000] and the CMU Stampede [Steffan and Mowry, 1998; Steffan et al., 2000].

2.4.5 Selective Undo

It is vital in architectures where mis-speculations happen frequently to only undo the effects of the instructions dependent on the mis-speculation. If instructions independent of the mis-speculation are squashed they will be re-executed to produce the same result, wasting resources and slowing down computation. Frequent indiscriminate squashing removes

the benefit of speculative execution, while still competing for execution resources. The term *squash* is reserved for indiscriminate removal of speculative operations beyond a mis-speculation point, the term *rollback* is used for selectively undoing operations.

Where control flow speculation is concerned exploiting control independence can be regarded as a way of selectively undoing computation [Chou et al., 1999]. When a branch misprediction occurs the control independent instructions aren't rolled back because they are guaranteed to execute. However, data dependencies must still be recognized and dealt with. A data dependence may be present in only one path from the branch, so a change in control flow may result in a consequent change in data flow.

Selectively reissuing instructions due to data mis-speculation can be achieved using the same dataflow method that issued instructions out-of-order in the first place. Reservation stations are single assignment registers, placing a new value in the reservation station can be used as a cue to re-execute the instruction.

Sodani and Sohi [1997, 1998] propose reusing squashed instructions. This has the incidental advantage that instructions which execute several times with the same inputs can use the same mechanism to retrieve the correct results again without having to recalculate them. Over 50% instruction reuse was measured in some cases for a speculative superscalar processor. For a split instruction window the increased mis-speculation suggests this figure would be even higher. Roth and Sohi [2000] propose *register integration* as a mechanism for implementing squash reuse.

An alternative approach for minimizing re-execution of correct instructions is to selectively rollback the mis-speculated operation and those that depend on it. Rotenberg et al. [1999b] propose using an ROB implemented as a linked list to allow alternative, possibly larger, instruction paths to be inserted into the dynamic instruction stream of a superscalar processor to selectively rollback instructions following a branch misprediction. The trace processor [Rotenberg et al., 1997; Rotenberg and Smith, 1999] uses the same principle to selectively re-execute multiple flows of control. Chains of data dependent instructions reissue as updated values are provided to them in the ROB.

Trace buffers are also utilized by the dynamic multithreaded (DMT) [Akkary and Driscoll, 1998] and speculative multithreaded [Marcuello and Gonzlez, 1998] processors. When

mis-speculation recovery is required the trace buffer is scanned from the mis-speculation point to identify dependent instructions and they are grouped together and dispatched for re-execution.

2.4.6 Memory Consistency

Speculative memory access is the most difficult class of speculation to control. Memory dependencies are generally difficult to identify when pointer operations are involved because memory addresses are often calculated dynamically. This is a particular problem in split instruction window processors, since the instruction the memory access is dependent on may not even have been fetched yet.

Speculative memory writes are problematic because they can overwrite the non-speculative value which may have loads dependent on it that have not yet executed. However, a speculative load needs to be able to access the value from the store immediately prior to it in the program order. This value may have come from the latest non-speculative write, a speculatively executed write, or a write which has not yet been executed. If the last case has not been ruled out the load must either be conservative and stall until any stores it may be dependent on have executed, or execute speculatively and rollback and re-execute if an applicable value is written to memory. The conservative solution tends to waste cycles on false dependencies, since memory dependencies are hard to analyze. The MIT Raw machine [Waingold et al., 1997; Barua et al., 1999], a radical parallel architecture, attempts to use compiler and dynamic analysis to disambiguate memory accesses, with the option of serializing accesses where dependence may exist. The superthreaded architecture [Tsai et al., 1999] speculates on control flow, but is conservative on data dependencies.

A speculative solution requires a structure for buffering speculative memory state that further speculative reads can access. The contents of this buffer can either be rolled back if mis-speculation is identified or committed to architectural (non-speculative) memory. It is crucial that such a mechanism can identify the appropriate stored value for a load from an arbitrary point in the program order. Previously proposed approaches to controlling speculative memory accesses are discussed below.

Address Resolution Buffer

The first proposed speculative memory system was the *address resolution buffer (ARB)* [Franklin and Sohi, 1996], originally designed for the multiscalar processor. The ARB can hold multiple versions of a memory location, valid at different points in the sequential execution. Since the speculative tasks are assigned in-order to a ring of execution stages, the stage number can be recorded and used to identify the relative ordering of the stores, allowing for wrapping around.

When a speculative load is done the closest earlier store for that address is returned. If none exists then architectural memory is consulted. A flag is also set in the ARB to record the existence of a speculative load from the address by that stage. A speculative store writes the value and stage number to the ARB. If a load has occurred before (in program order) any other store a mis-speculation has occurred and tasks from the loading stage onwards are squashed and re-executed.

When a stage becomes non-speculative the values in the ARB with a matching stage number are removed and committed to architectural memory. Squashing a stage performs the same process, but does not commit the values to memory. Only addresses with currently valid speculative entries require entries in the ARB.

Speculative Versioning Cache

The major limitation of the ARB is that it is a single central buffer, hence every access incurs the latency of the interconnection network, and the ARB must support sufficient bandwidth for all the processors. The *speculative versioning cache (SVC)* [Gopal et al., 1998] addresses this limitation by using a private speculative cache for each processor, avoiding bus traffic for local accesses.

Speculative versions of memory accesses are supported by implementing a modified version of a snooping cache coherence protocol for symmetric multiprocessors. A *load bit* for each cache line is set if a task loads from a line before storing to it. This indicates a potential memory dependence violation if an earlier task stores to that line.

A linked list, the *version order list (VOL)*, is maintained by a pointer in each line identifying

the next cache containing a entry for that line. A load that misses in its private cache can be satisfied by searching the VOL in reverse order until a hit is found. The VOL has a maximum length of the number of processors, which is generally small, so it will be fast to search. If there are no hits in the SVC the value is retrieved from non-speculative memory.

A store writes the value to the processor's private cache and sends an invalidation response to the following tasks until the next store to that line is reached. If a cache receives an invalidation response and has the load bit set for that line it squashes the task and all later tasks.

On task commitment all stored values are written to non-speculative memory and all lines in the private cache are invalidated. Task squashing invalidates all lines in the cache.

Gopal et al. [1998] introduce several refinements to the SVC. *Lazy commitment* allows non-speculative values to remain in the SVC until they are superseded by a later speculative value. This smoothes the otherwise bursty writes to memory when a task commits and allows the next task to be allocated without waiting for write commitment to complete. Allowing values to persist in the SVC after commitment also allows the SVC to act as another layer of caching.

Although the ARB and the SVC were originally designed with the multiscalar in mind they are applicable to any processor with multiple speculative flows of control. They can be extended to perform selective rollback by the addition of *anti-stores* which issue when a store has mis-speculated and undo the effects of the store, forcing any dependent loads to reissue. The speculative multithreading processor [Marcuello and Gonzlez, 1998], Hydra CMP [Hammond et al., 2000, 1998] and CMU Stampede [Steffan et al., 2000; Steffan and Mowry, 1998] use speculative memory systems similar in design to the SVC.

Both the ARB and the SVC rely on allocating a *sequence number* to each task to order memory accesses at a coarse granularity. This is trivial in the multiscalar processor because tasks are allocated to processors in a fixed hardware order. A task misprediction results in squashing all tasks beyond the misprediction point, so the tasks can be reallocated in order to the processor ring. The trace processor uses a mapping table to map physical processor numbers to logical (causal order) processor numbers, with pointers to neighboring processors [Rotenberg and Smith, 1999]. Neither of these methods will scale well. Squashing

reduces the value of speculation to execute far-away control independent tasks because they will frequently be squashed even when the execution is correct. The mapping table in a trace processor is a potential serial bottleneck, and could impose substantial overhead when the number of processors is large. Methods for ordering a large number of speculative threads which potentially fork will be explored in detail throughout the rest of this thesis.

Load and store queues

In the DMT processor [Akkary and Driscoll, 1998; Akkary, 1998] speculative memory accesses are stored in fully associative load and store queues. Loads check preceding threads for any store to the same address, using the value if one exists. Stores search for a matching address in the load queues of later threads until another store to that address is found, initiating a recovery action if any matches are found.

The DMT processor uses a separate thread tree to determine the causal order of threads. This represents a substantial bottleneck each time relative thread ordering is determined since the tree must be traversed to generate an ordered list. This is feasible for a small number of threads, but will not scale well. Scanning through the load and store queues of previous or following threads also does not scale well. All of them may have to be searched in the case of no matching access from the speculative threads before the non-speculative value can be accepted for loads, or memory dependence violations can be ruled out for stores.

Sun's MAJC architecture [Tremblay et al., 2000] uses a similar concept named *space-time computing* which maintains multiple speculative versions of heap memory. Loads are performed conservatively using dynamic memory dependency checking, and speculative stores are buffered in separate memory space for each thread until they become non-speculative, or are squashed. Again this will not scale well, but it is intended for a commercial chip multiprocessor (CMP) with only two processors initially. MAJC takes advantage of Java's built in memory management features to make the speculative memory system more efficient.

2.5 Summary

The previous research discussed in this section has shown that dependencies can place serious limits on the ILP that can be extracted from general purpose workloads. Some of these dependencies are artifacts of the program implementation and can be removed by compiler and architectural techniques. Even some dependencies intrinsic to the program can be bypassed using speculative execution and prediction. Studies have shown that it is necessary to use multiple flows of control to generate an instruction window with enough independent instructions to extract large amounts of ILP.

Considerable amounts of academic and commercial research are being directed at utilizing speculation over large instruction windows. A particular problem is honoring memory dependencies, which are often unknown, and difficult to predict prior to execution. This is a characteristic which is exacerbated by a split instruction window. Speculation appears to be the most promising technique for dealing with this, since mis-speculations can be rolled back, harming performance, but not correctness. By improving the average speculation accuracy the performance can then be improved.

Methods proposed to date for controlling memory speculation are adequate for small numbers of speculative threads, or for speculating in limited situations. When the number of threads is increased to extract further parallelism the methods used for determining the causal ordering of events become impractical. Either centralized structures are used, forming a bottleneck, or a large number of distributed structures must be consulted to determine ordering or dependencies. Clearly this is an issue that must be addressed if speculative architectures are to approach theoretical limits of performance. This thesis examines several proposed methods of maintaining memory dependencies in a speculative processor with potentially thousands of threads.

Chapter 3
The WarpEngine

In order to investigate issues involved with aggressively speculative execution a theoretical architecture known as *the WarpEngine* has been conceived. It is used as a basis for explaining and testing the ideas presented in this thesis. The idea was first described by Cleary et al. [1995] and was laid out in detail by Littin [2000]. The WarpEngine uses speculative execution in an attempt to extract the maximum amount of parallelism. This chapter gives an overview of the architecture with particular reference to the issues surrounding the control of speculation and development of a speculative memory system.

Analysis of potential parallelism between instructions at compile time can be classified into one of the following three situations:

- definite *independence*
- definite *dependence*
- neither dependence nor independence can be determined statically with certainty.

The first two cases can be handled by parallelizing compilers. In the case of definite independence the instructions can be freely executed in parallel. If definite dependence has been determined, then the instructions can be prohibited from executing in parallel without missing available parallelism. As the state of the art in compilers advances, more of these situations can be identified statically. However, there are some potential dependencies that are impossible to determine before program execution, such as when a memory address

is calculated by the program. Until that calculation is completed the memory reference could potentially conflict with any other memory reference, so no later memory access can be safely executed. Conditional branches also cause problems in analyzing the dependencies, since the instruction path is unknown beyond the branch, and may include instructions which are dependent on the current one. Accurate branch predictions can be made, but still have the potential to be incorrect. Conservative execution requires that the later instructions be stalled in these cases until the dependency analysis can be performed.

This indeterminate case is quite common in programming languages, such as C, which make use of pointers. In some cases there is a large amount of available parallelism which cannot be extracted by conservative methods. The WarpEngine aims to exploit this category of parallelism by speculatively assuming that there are no data dependencies, and recovering from any mis-speculation that occurs.

The mechanism used to recover from mis-speculation is based on the Time Warp algorithm [Jefferson, 1985] used in parallel discrete event simulation to maintain causal consistency between simulated events. Time Warp is an implementation of the *virtual time* paradigm which imposes a causal ordering on distributed systems. Events are given a *virtual ordering*, which represents the order in which they would be processed in a sequential system. The parallel processing of events must remain consistent with the virtual order.

This chapter discusses how the Time Warp algorithm has been adapted for use as the speculation control mechanism in a general purpose CPU. The architectural features of the WarpEngine are described and contrasted with those in other contemporary architectures.

3.1 Time Warp

The Time Warp algorithm has been shown to extract parallelism from otherwise intractable parallel simulations [Fujimoto, 1990b]. In Time Warp, parallel discrete event simulations are represented as objects that communicate by passing messages. Each message has an associated *timestamp* corresponding to its simulated time, representing the virtual time. When an object receives a message it processes an event at that simulated time. Events may produce further messages with later timestamps which are passed to other objects. The ordering of forwarded messages, known as the *principle of causality* [Fujimoto, 1990a],

must be maintained to ensure the correct results are obtained.

Local clocks are maintained for each object in the simulation, allowing events to be executed speculatively with regard to other objects. Sometimes this results in a message being processed out-of-order (i.e. a causality violation). The object is *rolled back* to the state before the incorrect speculation and the out-of-order event is re-executed. Any flow on effects from the rolled back messages must also be undone.

A typical way of restoring the state of an object is to take frequent snapshots of the state and save it in a list of states. When a rollback occurs the most recent state before the mis-speculation is restored to the object. Saving state frequently requires less computation to be re-executed, giving better performance, but consumes more resources and increases time spent saving state.

Undoing the effect of rolled back messages is achieved by sending *anti-messages*. When an object receives an anti-message it rolls back, annihilates the anti-message and the matching message and may send further anti-messages. Good performance relies on fast processing of anti-messages so that the chain of incorrect computation does not proceed far. A number of refinements to this general principle have been suggested to improve the efficiency of rollbacks [Fujimoto, 1990b; Fujimoto et al., 1992].

3.1.1 Global Virtual Time and Fossil Collection

As the simulation proceeds, the lists of saved state grow in size, consuming valuable resources. Old state copies can be removed (*fossil collected*) when it is determined that the simulation will never be rolled back to that point or earlier. The earliest point that a simulation can roll back to is known as *global virtual time (GVT)*, and can be calculated as the minimum virtual time of any active object or message in the system.

To maintain causality an object is rolled back when a message with an earlier virtual time arrives at the object. So the earliest message in the system can never be rolled back, and all saved state prior to that virtual time can be discarded. Process termination is also detected by GVT and processes can be fossil collected when GVT has progressed past them.

While local virtual clocks may roll back, GVT is guaranteed to progress forward and ensures

that the simulation as a whole will progress.

Accurate calculation of GVT is the key to keeping the state saving lists small, but this is not easy in a distributed system. Much attention has been paid to this problem in the Time Warp literature [Bellenot, 1990; Gomes et al., 1992].

3.1.2 Cancelback

Resources for state saving are necessarily finite, and it is possible for speculative events to consume all those resources leaving none available for earlier events. This will prevent execution of those events, possibly stalling GVT and forward progress of the simulation. To guarantee program completion under Time Warp a mechanism is needed to provide some resources for those events.

The *cancelback protocol* [Jefferson, 1990] extends the Time Warp storage management mechanism to allow the most speculative events to be halted and the resources used to be recovered for use by earlier events. This means discarding potentially correct computation, and is generally used only when resources are exhausted.

Cancelback has been shown to be sufficient to guarantee that a simulation run using Time Warp will be able to complete in no more memory space than is used by a sequential simulation [Jefferson, 1990].

3.1.3 Time Warp in Computer Architecture

In seeking to design an architecture based on Time Warp it is appropriate to identify the features of Time Warp that correspond to those of contemporary computer architectures, and the additional ones needed to utilize Time Warp. A full description of terminology parallels can be found in [Pearson et al., 1997].

In Time Warp an object is the fundamental unit that can be rolled back. The corresponding unit of rollback in a speculative CPU is the instruction, or instructions grouped into basic blocks or tasks. Instruction rollback is commonly performed by the ROB in a CPU. Objects carry all the state saving information for Time Warp processing. In a microprocessor memory locations carry state information in addition to the ROB. Generally memory state

is saved using speculative write buffers, which are adequate for small amounts of speculation. Saving speculative memory state is explored throughout this thesis, particularly in Chapter 8.

In Time Warp, messages are used to transfer data between objects, hence they correspond to the transfer of data between instructions in a CPU. Branch, call and jump instructions communicate control flow information between groups of instructions. Memory write instructions are messages from a group of instructions to a memory location. Memory read instructions correspond to a pair of messages: a request from the instructions to the memory location; and a reply from memory back to the instructions.

Anti-messages do not typically exist explicitly in CPUs. Rather than identifying the cascade of instructions dependent on the one rolled back they perform a coarse granularity instruction buffer flush, sometimes called a *squash*, rolling back all speculative instructions beyond the conflict.

Fossil collection corresponds to the in order commitment of speculative results. GVT is represented by the earliest instruction in the sequential program order which hasn't completed execution, all instructions prior to that point can be retired.

A mechanism analogous to cancelback needs to be included in an architecture based on Time Warp. Conventional microprocessors with a single linear control mechanism do not require such a mechanism, since instructions are issued in order, even if they are executed out-of-order.

A hardware implementation of Time Warp, the Virtual Time Machine [Fujimoto, 1989], has been proposed, along with special purpose hardware for rollback [Fujimoto et al., 1992]. However, it is largely limited to speeding up parallel discrete event simulation, rather than general purpose processing.

3.2 Supporting Speculative Execution

Time Warp provides the WarpEngine with a distributed means for identifying and recovering from causality violations. To test this mechanism under conditions of aggressive speculation other features are needed to expose a large pool of instructions for parallel execution.

Using block structured execution allows the fetch rate to be increased and allows parallelism to be extracted at a coarser granularity, reducing some overheads. By executing blocks in a tree structure *control flow independence* can be exploited to further increase the size of the instruction pool.

3.2.1 Block Structured Execution

The WarpEngine aims to extract parallelism at two different levels of granularity. By dividing the code into blocks, parallelism can be extracted at the instruction level within a block, and also between blocks by executing them in parallel. Another way of viewing this is as a large, non-contiguous instruction window split into many blocks.

The *Block Structured Architecture (BSA)* was originally proposed as a means of increasing the instruction fetch rate beyond the size of a basic block [Melvin and Patt, 1995]. Not only can a whole basic block be fetched in one atomic operation, but *block enlargement* [Hao et al., 1998] allows the atomic unit of execution (the block) to be made up of multiple basic blocks. Familiar mechanisms from superscalar processors, such as multiporting, trace caches and pipelining can be used to further boost the fetch rate.

Data dependence information is contained within the block to remove the need for complex instruction reorder logic, meaning that the order in which the instructions are stored is unimportant. In the WarpEngine this is done using a single assignment dataflow register set. Single assignment removes the need for register renaming, usually done at run-time, replacing it with compiler determination of destination registers. Each instruction has dedicated source registers, and will send its results to the source registers of instructions in the same or immediately following blocks. This allows all instructions to be issued in parallel, provided source data is available and there are sufficient resources to execute. The tag matching problems of traditional dataflow architectures are avoided by keeping dataflow communication local and restricted to a small number of possible registers with static addresses. Global inter-block communication of data in the WarpEngine is performed via the memory system.

Within a block the WarpEngine uses a limited system of predication to dynamically disable execution of some instructions. This facilitates better packing of blocks by boosting some instructions across branches. Full scale block enlargement is not necessary in the

WarpEngine since the maximum size of the instruction blocks is limited.

The WarpEngine uses a *fixed length* BSA [Eeckhout et al., 2000], in which each block has a fixed maximum number of instructions. This simplifies fetch and issue logic [Neefs and Van Campenhout, 1996]. Where the compiler cannot pack a block with the maximum number of instructions it is padded with null operations. Studies [Eeckhout et al., 2001; Littin et al., 1998] suggest that block sizes of sixteen instructions are optimal, and this is the size used by the WarpEngine.

3.2.2 Tree Structured Execution

The aim of the WarpEngine is to investigate issues involved in extracting high levels of parallelism through aggressive speculative execution. This requires having many instructions in flight and a large instruction window. As shown by the studies described in Section 2.1.3 this necessitates a split instruction window with the ability to issue many instructions each cycle.

The WarpEngine exploits control independence, identified by the compiler, to achieve a large split instruction window. Recall that control independent sections of code are guaranteed to execute regardless of branch outcomes in the other sections. This means they can be executed in parallel, as long as data dependencies are obeyed. These data dependencies can then be dealt with by the speculation control mechanisms provided by the Time Warp algorithm. This results in a dynamic execution flow graph in the form of a tree, so we call this *tree structured execution.* Figure 3.1 shows an example of the execution tree formed from some sequential code. The virtual order of frames can be obtained by doing a pre-order left to right traversal of the execution tree. Figure 3.2 shows an execution tree that can be generated for a loop where each iteration is control independent (for example a `for` loop). The nodes in the control branch running down the right perform checks to see whether the iteration should be executed. Littin [2000] provides some optimized execution trees to increase the fanout of loop iterations.

Exploiting control independence reduces the importance of branch prediction. Branch prediction now only affects the local subtree, other branches of the execution tree are still guaranteed to execute and continue regardless.

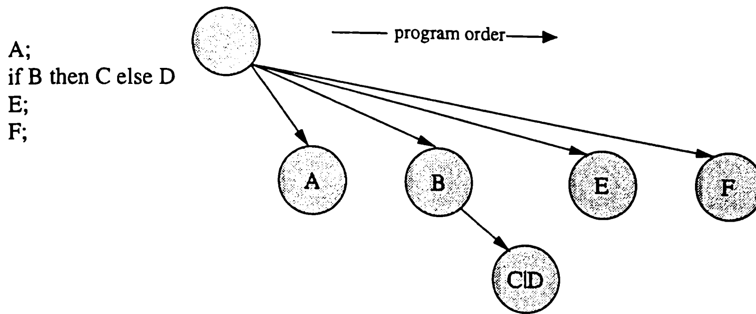


Figure 3.1: Producing tree structured execution from sequential code

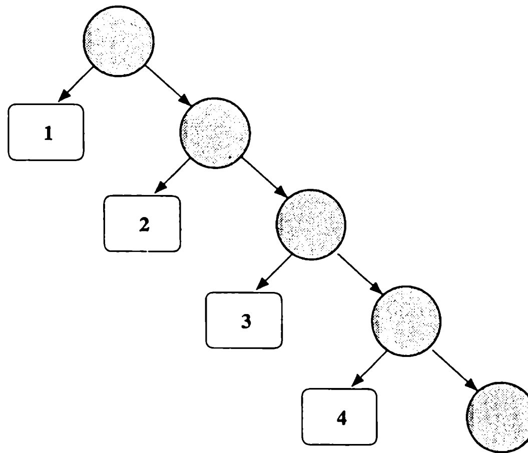


Figure 3.2: Execution tree for loop iterations

It is possible to convert some control dependencies into data dependencies and use speculative methods to exploit the parallelism. For example, in `while` loops it cannot be known whether an iteration should execute until the previous iteration has completed, or at least calculated the condition value. By making the instruction which initiates the iteration conditionally dependent on the value calculated in the previous iteration the speculative control system will allow the iteration to be fired while the condition is still speculative and then roll it back if necessary.

Throughout this thesis conceptual discussions relating to execution trees are illustrated using binary trees. All trees with a greater fan out can be decomposed to a binary tree by adding intermediate levels whose only purpose is to allow more nodes to be created. By removing these intermediate levels the principles shown for binary trees can be extended to execution trees in general. Figure 3.3 shows the four-way tree from Figure 3.1 converted to a binary tree by adding a level of intermediate nodes (unlabeled).

The CFG for tree structured execution has some important differences from the CFG for

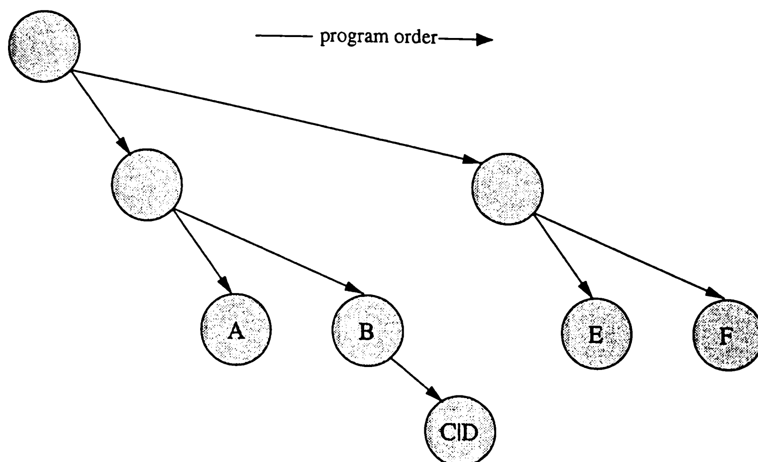


Figure 3.3: An equivalent binary execution tree

a single flow of control. A tree structured CFG allows multiple flows of control to be followed because paths to control independent nodes may be followed in parallel. Since some nodes may be conditionally executed paths to them are represented by a dashed arc, while unconditionally executed paths are represented by a solid line. Figure 3.4 shows a code excerpt containing a `for` loop and some inline code, and corresponding CFGs for a single flow of control and tree structured execution.

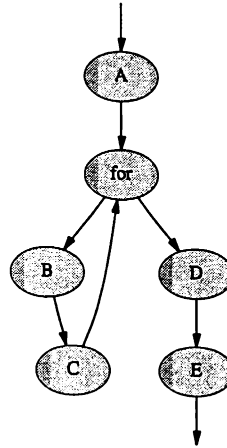
Clearly there are several possible tree structured CFGs even for a simple piece of code like this. For example, instruction A could be executed in parallel with the `for` loop and instructions D and E. The CFG design influences the obtainable parallelism and can be used to throttle execution. The most effective CFG depends on factors such as the maximum parallel execution fanout that can be sustained by the processor and data dependencies between control independent points.

3.2.3 Virtual Ordering

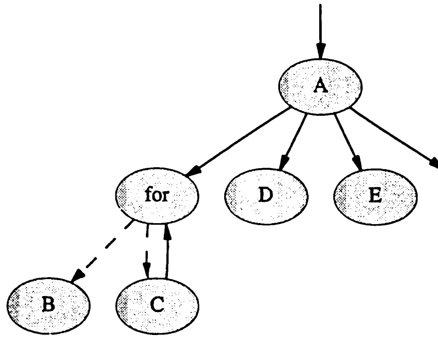
Since separate subtrees in the WarpEngine are control independent they are equivalent to threads in a multithreaded architecture. However, most speculative multithreaded architectures don't permit an arbitrary number of threads to be created between any two threads. Either they are compiler identified and statically assigned, or squash and reissue operations map the threads directly onto processing elements, as in the Multiscalar architecture [Sohi et al., 1995]. Static assignment of threads is done conservatively, because resources reserved for threads will be wasted if the number of threads required is overestimated. Con-

```
A;
for(i=0; i<5; i++){
   B;
   C;
}
D;
E;
...
```

(a) Code extract



(b) CFG for a single flow of control



(c) CFG for tree structured execution

Figure 3.4: Equivalent CFGs for single flow of control and tree structured execution

servative allocation, on the other hand, restricts the amount of speculative parallelism that is extracted. Squash and reissue allows a more flexible allocation of threads, which can be modified dynamically. Squashing limits the speculation distance that can be achieved.

The WarpEngine allows out-of-order thread allocation to occur dynamically without needing to squash and reissue. This makes it necessary to track the virtual order independently of resource allocation.

Data speculation in the WarpEngine is provided by allowing memory accesses to occur freely out-of-order. This eliminates false data dependencies because accesses are not restricted to occur in the programmed order.

True data dependencies must be detected and execution corrected where they are violated. The WarpEngine does this using a *virtually ordered memory system*, which is the focus of this thesis. Each memory access is assigned a position in the virtual order, corresponding to its order in the program when executed sequentially.

The rest of this thesis investigates ways of maintaining the virtual order. Any violation of the virtual order is detected by the memory system and the appropriate instructions are rolled back and re-executed. Chapter 8 considers a design for a memory system capable of doing this.

Register data dependencies are removed by the use of single assignment dataflow registers, since they are only read by one instruction and only written to by one instruction. If an incorrect speculative value (either from a speculative memory access, or a control speculation) was written into a register, when the correct value arrives as an update the instruction re-executes and sends updated results to further instructions in turn.

3.3 Architecture

The WarpEngine speculative architecture combines control flow speculation and data speculation with the Time Warp synchronization mechanism to form a test bench for investigating aggressively speculative, highly parallel processor techniques.

The WarpEngine architecture investigated in this thesis uses a four-way control tree with fixed size instruction blocks. Each instruction block is executed by loading it into a hardware resource known as a *frame*. Instructions in the same frame are executed in a dataflow manner, in parallel on multiple functional units.

Using a special instruction a frame may pass data to any frame it has initiated, but any further inter-frame communication must be done through the memory system. A special memory unit called the *time-space cache* stores multiple versions of speculative memory values, and tracks the virtual ordering of memory accesses, reissuing values if sequential consistency is violated. The virtual order is represented by the value of a *virtual timestamp (VTS)* assigned to each frame. By comparing the VTSs of two frames their relative position in the virtual order can be ascertained. The VTS may not necessarily be an explicit

value, it may be encoded in other ways, such as the hardware location. The progress of non-speculative execution (*GVT*) is tracked by a distributed mechanism (*fossil collection*), which commits and reclaims the resources of frames which have completed and are non-speculative.

This section describes the components of the WarpEngine important to this thesis as laid out in Figure 3.5. The WarpEngine is investigated through the use of the simulator described in Chapter 4. Littin [2000] provides a comprehensive description and investigation of the basic WarpEngine architecture and simulation methods.

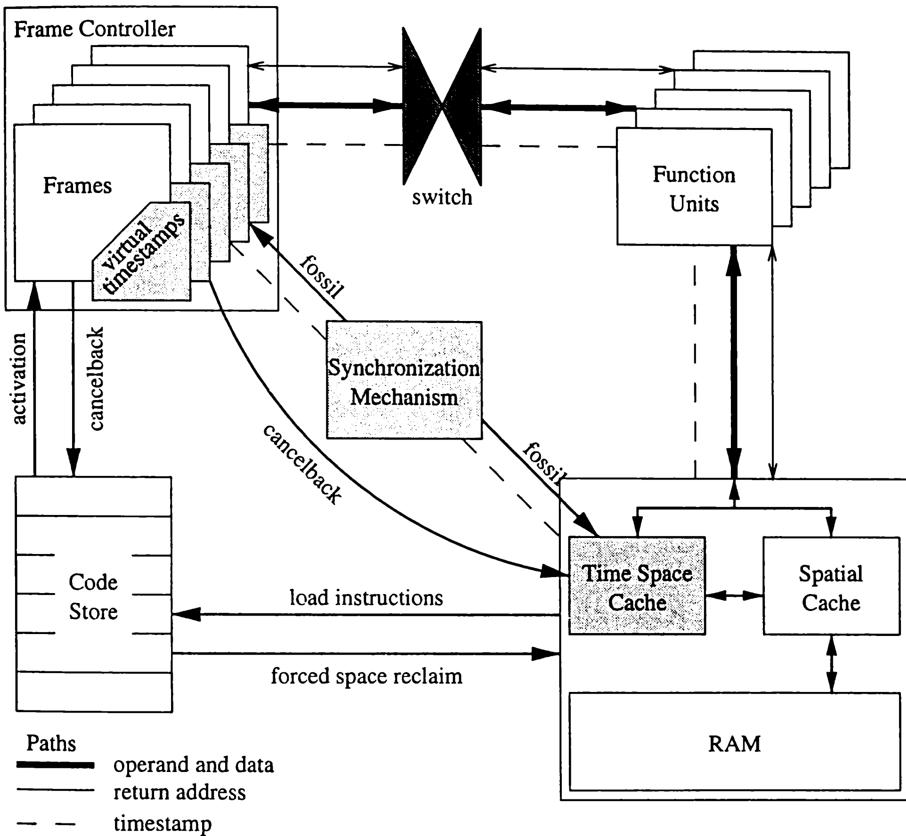


Figure 3.5: Components of the WarpEngine and their connections

3.3.1 Instruction Set

The WarpEngine instruction set is included in Appendix A. Instructions have been grouped as control, data movement, logical, floating point and integer arithmetic. The only control instruction is the *child* instruction, which generates a new node in the execution tree by loading a block into a frame and assigning it a VTS. It can be conditionally executed based

on the outcome of *cmp*, the arithmetic comparison instruction, in effect providing a conditional branch. Appendix A includes an example of a conditional branch in WarpEngine assembly code.

The *mv* (move data) instruction moves a data value directly from a register in a frame to a register in one of its child frames. To transfer a data value to another frame it is stored to memory by the source frame using the *st* (store) instruction, and then loaded into a register in the destination frame using the *ma* (move from address) instruction. These three data access instructions may also be executed conditionally, in concert with *child* instructions.

The standard arithmetic and logic instructions are provided, each sending its result to one or more destination registers.

3.3.2 Frames

Frames in the WarpEngine are analogous to the ROB in contemporary architectures. They provide physical storage for registers and instructions, a control mechanism for executing instructions, and a means of state saving. Each frame has a VTS associated with it which is used for all memory operations performed by that frame. Frames can be processed in parallel and are treated as independent speculative instruction streams. The frame controller arbitrates resource contention and manages communication between frames, function units and the code store.

Each frame comprises 16 *slots*, holding one instruction each, as laid out in Figure 3.6. The slot is made up of four fields: an op-code; execution status flags; and two operand registers. The status flags are used to determine the validity of instruction operands and to maintain the execution state of the instruction.

Calvert [1997] investigated a possible hardware implementation for a WarpEngine frame.

3.3.3 Code Store

The code store prepares blocks of instructions for execution, which are then loaded from memory into frames. The code store generates and assigns VTSs to blocks as they are allocated to frames in accordance with the program control tree.

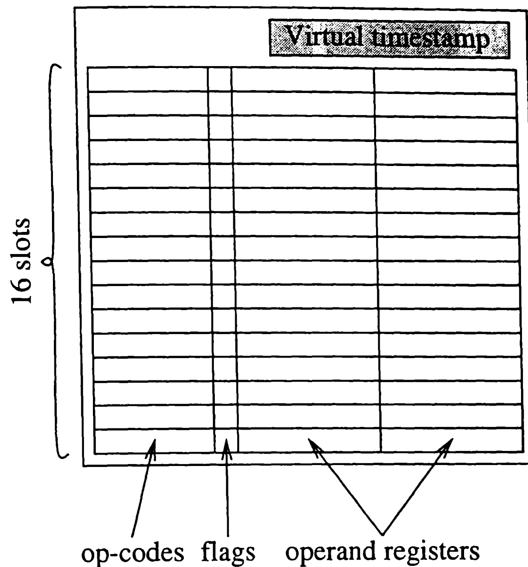


Figure 3.6: Layout of a frame

The instruction block is the atomic unit loaded from the code store. By choosing the size of an instruction cache line to accommodate a whole instruction block, stress on the instruction cache can be reduced. Since many blocks will be concurrently speculatively executed fetching from the instruction cache is a potential bottleneck.

The code store also records information about cancelled back frames, so that they can be rescheduled and resume executing when resources become available.

3.3.4 Function Units

The function units perform the same operations as any contemporary RISC architecture. Input operands and the distribution of results are handled by an external device, possibly through some form of switch.

While most operations can be satisfied within the function units, some require actions to be performed by other system components. The *child* instruction queries the frame controller for frame resource space and new VTSs, the *mv* and *ma* instructions send results to frames other than their own, which requires interaction with the frame controller, and the *ma* and *st* instructions access the memory system.

3.3.5 Synchronization Mechanism

The synchronization mechanism is used to free frame and time-space cache resources that are no longer required. This relies extensively on the VTS system to order events. Several such systems are investigated throughout this thesis.

VTS requirements for fossil collection and cancelback are addressed as the VTS schemes are proposed, but depend to an extent on the frame implementation used, which is beyond the scope of this thesis.

3.3.6 Instruction Execution

Each instruction in memory is represented by two words. The *instruction word* (I-word) contains the *op-code* and result destination information. The *constant word* (C-word) contains a single literal or constant value that can be used as one of the instruction's operands. There are a number of stages in instruction execution, shown in Figure 3.7.

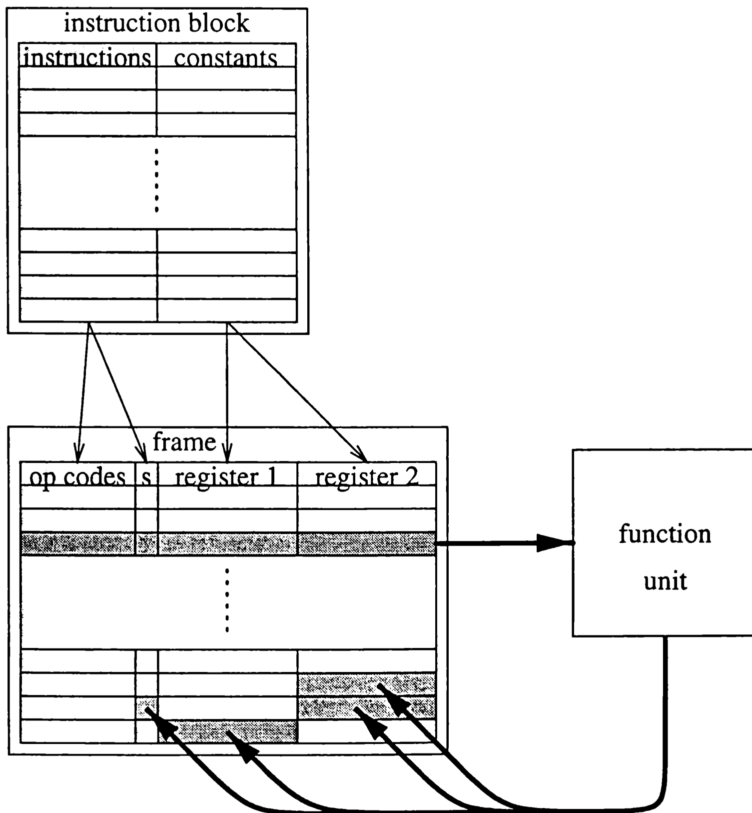


Figure 3.7: Instruction flow within a WarpEngine frame.

First, the instruction block is fetched and allocated a frame. The *op-code* and *s flag* (indicating conditional execution) are loaded directly from the instruction's I-word. Operand registers within a slot are either loaded with a constant value from the C-word, or with data that results from the execution of another instruction.

Data flow techniques are employed to execute the instructions within a frame. When the data is placed in each of an instruction's operand registers the instruction is transferred to a function unit for processing. The function unit performs the computation and returns results back to the appropriate registers in the frame.

Data flow execution of instructions within a block removes the need to issue each instruction with a unique VTS for state saving. The hardware required to perform data flow execution and the static allocation of registers to instructions provides a mechanism that executes instructions only when a change in input occurs. This is particularly useful when re-executing instructions after incorrect data value speculation has taken place.

3.3.7 Memory System

Since the WarpEngine is to be capable of storing multiple speculative memory values at each address the memory system needs to be able to select the appropriate value to return for a load. Additionally, since memory accesses may execute speculatively out-of-order, the memory system must be able to identify reads which have been incorrectly satisfied and rollback the dependent execution.

This is done using a *virtually ordered memory system*, of which the key component is the *time-space cache*. Conceptually, the time-space cache contains triples of the form *(address, VTS, value)* for writes, and *(address, VTS, destination register)* for reads. Different speculative versions of memory values are differentiated by their VTS, and incorrect speculative read replies can be reissued with the updated value. Only entries with a VTS more recent than GVT need to be stored in the time-space cache, since only they are speculative. Earlier accesses can be retired, with write values committed to RAM.

The time-space cache forms the level of the memory hierarchy closest to the frames. Beyond that is the usual cache hierarchy, denoted *spatial memory* because it contains no virtual ordering information. Only the most recently committed value is stored for each address

in spatial memory. Figure 3.8 shows the relationship between the different levels of the hierarchy.

The time-space cache can be viewed as a simple addition to the standard memory hierarchy. The shaded region in Figure 3.8 shows the components in a real ordered memory system. The difference is the time-space cache and the way it interacts with spatial memory through the memory controller.

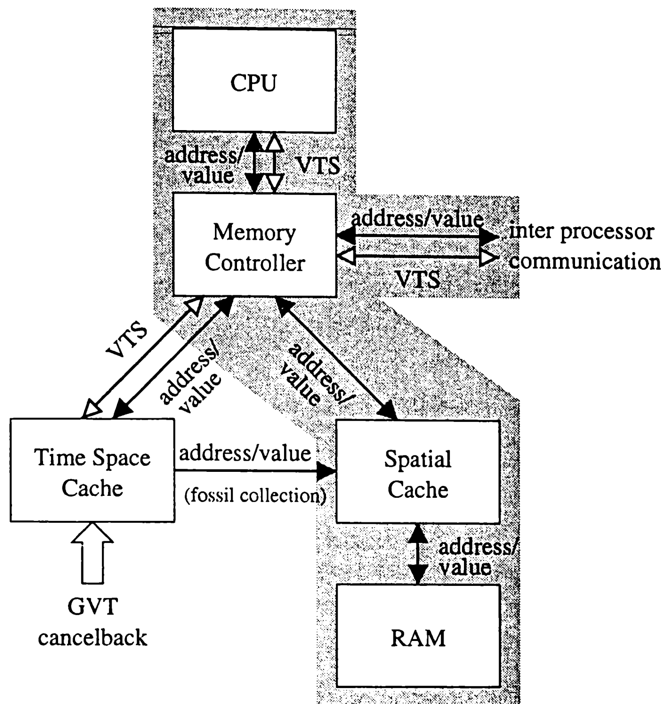


Figure 3.8: Components of the virtually ordered memory system

Six possible operations must be supported on the memory subsystem: read; write; anti-read; anti-write; fossil collection; and cancelback.

A read generates an access to the time-space cache and one to the spatial cache in parallel. The previously recorded write with the same address and the most recent VTS prior to the VTS of the read will be returned. This value will come from the time-space cache if there is a write that matches, otherwise the value most recently committed to the spatial memory hierarchy is returned.

Note that it is not essential to have parallel paths to the time-space cache and spatial memory. All memory operations could pass through the time-space cache before being transmitted to the spatial cache, and only those not satisfied in the time-space cache be passed on.

This reduces the bandwidth required to spatial memory, but it increases the latency of time-space cache misses, a major bottleneck in current architectures and likely to be so in the WarpEngine.

A write is only sent to the time-space cache, where it is recorded. This write will supersede the previous write for any read later in the virtual order than the write, but earlier than the next recorded write in the virtual order to that address. The time-space cache will re-satisfy the read and return the new value, causing dependent instructions to rollback and re-execute. In the example shown in Figure 3.9 for a single memory address the read with VTS 20 will be resatisfied with the value 5 when operation a) is processed, superseding the value of 0 previously returned. This is the only read affected, since another write has taken place with a VTS of 30.

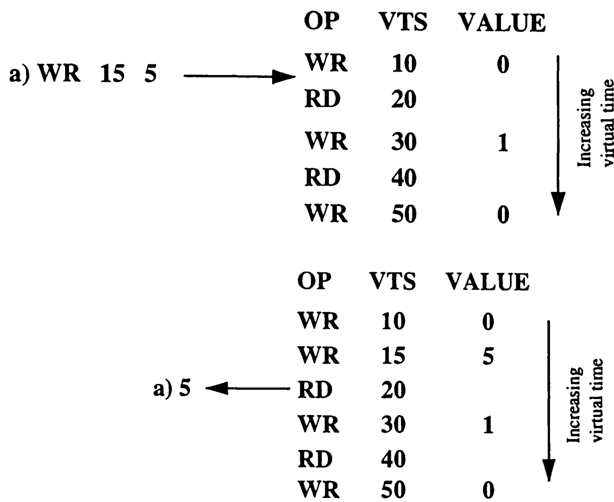


Figure 3.9: Resatisfying a read due to an out-of-order write.

Since memory operations are speculative they may need to be rolled back. The anti-read and anti-write operations undo the effects of reads and writes respectively. An anti-read removes the corresponding entry from the time-space cache. Anti-reads only occur as the result of a mis-speculated branch or a change in a computed memory address. The consequences of undoing the read are handled by the frame.

An anti-write removes the corresponding write entry from the time-space cache. It must also resatisfy any reads which returned the value from the write that has just been rolled back. This means locating all reads to that address with later VTSs up to the next write that has been recorded. The previous write value in the virtual order must be returned for each

of these, either from the time-space cache, or from spatial memory. The frame logic will then cause dependent instructions to be re-executed.

The fossil collection process commits the read and write entries earlier than GVT to spatial memory. Since GVT may progress in large increments not every write operation in the time-space cache needs to be written to spatial memory, only the latest in the virtual order needs to be recorded for each address. Once the values in spatial memory have been updated the time-space cache entries prior to GVT can be deleted.

Cancelback removes all entries in the time-space cache with a VTS greater than a given value without writing them to the spatial cache. This can be used as a method for freeing space in the time-space cache when it becomes full, although fossil collection is a preferable way of doing this since the cancelled entries will eventually be regenerated when their frames are re-executed.

Issues

The virtually ordered memory system described in this section is simplified and purely conceptual. The description implies some implementation methods which may not be optimal. It is convenient to think of VTSs as a set of explicit, ordered integers, although physical ordering in a hardware structure could also be used to determine virtual order. Alternative methods for this and other features of the virtually ordered memory system are explored throughout the rest of this thesis.

A number of factors determine the optimal design for the virtually ordered memory system. These are explored in varying levels of detail with the simulation techniques described in Chapter 4.

The latency of memory operations is an important factor in the overall performance of the WarpEngine. Littin [2000] showed that the proportion of memory instructions on the critical execution path is substantially higher than in the total dynamic instruction count, and the execution time is very sensitive to increased memory latency.

The WarpEngine is designed to minimize the effects of high instruction latency by executing instructions speculatively. This means there will be an increased number of instructions

and memory operations in-flight at any time. This means bandwidth considerations will be important, and features that reduce the number of operations should be considered carefully.

The size of the time-space cache is another important consideration. In general a small cache will be faster, particularly if associative matching is required. However, if the capacity of the time-space cache is exhausted cancelback may be required to reclaim space and allow execution to continue. Cancelback limits the amount of effective ILP that can be extracted because speculatively executed computation is being discarded. Efficient fossil collection will be important in minimizing the required size of the time-space cache.

A final factor is the complexity of the time-space cache design. A complex design may limit the number of time-space cache entries that can be implemented on the chip. This is particularly important if a multiprocessor WarpEngine is being implemented on a single chip. This question is only be answered in the most general terms in this thesis, pending gate-level designs.

3.4 Related Architectures

The WarpEngine is most suited to extracting parallelism from programs in which tasks are likely to be independent, but in which the dependence, or lack thereof, cannot be easily determined. Architectures which use static scheduling will not be able to extract this potential parallelism, but the aggressive speculation of the WarpEngine makes it at least theoretically possible.

The WarpEngine does not rely on the programmer identifying areas of parallelism, instead regulating the speculative execution through the multiple version memory system. However, the programmer's memory model is still a single linear address space, requiring no explicit synchronization.

The WarpEngine shares many features with the architectures presented in Chapter 2. This section examines the important similarities and differences.

3.4.1 Superscalar

Superscalar architectures reuse registers and memory locations, since the size of both are limited. This creates false dependencies between instructions. Register renaming can effectively eliminate false register dependencies, provided the physical register set is large enough, while some false memory dependencies can be resolved using memory reorder buffers. However, neither of these mechanisms scale well to large instruction windows. The size of the instruction window is also limited by branch prediction errors that inevitably occur.

The WarpEngine avoids false register dependencies using single assignment registers in the frames. False memory dependencies are eliminated using unrestricted speculation in combination with the virtually ordered memory system.

Superscalar machines do support out-of-order execution, but only when it can be determined that the correct data is available. The speculative execution of the WarpEngine can potentially extract much more parallelism, but extra overhead is introduced to correct misspeculation.

In contrast to the single flow of control utilized in superscalar architectures, the WarpEngine supports multiple independent flows of control scheduled in a tree structure, again allowing more parallelism to be extracted.

3.4.2 Dataflow Processors

The WarpEngine utilizes dataflow techniques in that each frame acts as a miniature dataflow engine. Instructions fire when values have been placed in their source registers, and the result is sent to one or more source registers of other instructions. However, unlike dataflow machines, the number of possible destinations is quite limited because a maximum of sixteen slots are available in each frame, and a result can only be sent directly to a slot in the same frame or a child frame, and can be specified directly. This avoids the need for the large token matching stores required in dataflow machines, which tend to be a performance bottleneck. Although the time-space cache must match VTSs in a similar way to token matching in dataflow machines, it does not require it on every instruction, only on memory

accesses. Additionally, since VTSs are assigned at the block level, less are required than in dataflow execution, which uses a token for each instruction. A complication present in the WarpEngine is that VTSs are not matched exactly, as with tokens, rather they must match with the nearest VTSs in the store.

In traditional dataflow architectures the instructions available for execution are all considered equal, which can lead to resource contention problems. In the WarpEngine this is neatly solved by prioritizing blocks based on their VTS. Instructions earlier in the virtual order receive scheduling preference since they are less speculative, and more likely to execute correctly, or be holding back GVT.

Specialized dataflow languages must be used to extract good performance from dataflow machines, and typically require the programmer to learn a new programming model. By contrast the WarpEngine uses standard imperative languages, although custom languages and constructs could be developed to further exploit the features of the WarpEngine.

3.4.3 Multiscalar

The most closely related of the architectures discussed in Chapter 2 are the multiscalar and related architectures, such as the trace processor.

The multiscalar also takes advantage of speculative execution, although it uses a fixed hardware ring structure to order tasks and memory accesses. Maintaining the order with VTSs allows the WarpEngine to use a more flexible and scalable network of processing resources, although manipulating VTSs is more complex than hardwiring the order.

When a mis-speculation occurs the multiscalar squashes all computation beyond the mis-speculation point, possibly discarding independent computation, which the WarpEngine would retain. Other architectures developed from the multiscalar model, such as the trace processor and dynamic multithreading, retain independent computation beyond the mis-speculation point.

Tree structured execution allows the WarpEngine to generate independent flows of control more flexibly than the ring structure of the multiscalar and trace processors, which results indicate are only scalable to around eight concurrent tasks [Sohi et al., 1995]. Dynamic

multithreading implements a similar tree structured hierarchy of threads, although causality violations are not detected until results are committed, delaying re-execution, but reducing the amount of transient execution from that seen in the WarpEngine[Akkary, 1998].

3.5 Summary

This chapter has discussed the conceptual approach of the WarpEngine. It aims to extract large amounts of ILP through speculation on the outcome of control decisions and data values in memory. Using tree structured execution the program is split into control independent instruction streams, which can be executed independently providing data dependencies are observed.

The Time Warp algorithm is incorporated into the WarpEngine architecture to synchronize the speculative execution. Any violation of causal dependencies is identified and the offending instructions are rolled back and re-executed to correct the computation. Determining the sequential program order, or virtual order, of the instructions is critical to this process. VTS methods used for doing this are discussed later in this thesis.

The WarpEngine architecture uses a block-based instruction set. Instructions are grouped into fixed-size blocks, which are scheduled to be processed in parallel. When blocks are invoked for execution they are placed on a frame, which controls instruction execution and provides state saving space. Instructions are fired in data flow order, extracting the largest amount of parallelism possible.

The main focus of this thesis is the virtually ordered memory system. This is a crucial component of the WarpEngine, since the memory system is used extensively to communicate values between instruction blocks, and it controls all the data speculation.

The eventual success of the WarpEngine will depend upon the ability to design key novel components of the processor. Functional simulation is used to determine the conceptual viability of the architecture, building on previous simulation work [Littin, 2000]. However, issues concerned with detailed gate level design of components are beyond the scope of this thesis.

Chapter 4
Simulation

4.1 Virtual Order Simulation

In order to simulate the WarpEngine architecture efficiently a new simulation paradigm, known as *virtual order simulation* has been developed as part of the WarpEngine project. A full description of its development can be found in Littin [2000].

The approach in virtual order simulation is to simulate events, not in the order they would actually happen, as with usual event driven or cycle by cycle simulation, but in the sequential program order that implicitly describes the instruction dependencies. Instructions executed on an in-order architecture will be simulated in the same order in real order and virtual order simulation, since instructions are always executed after those they are (or may be) dependent on. However, for out-of-order execution the real execution order may be different to the virtual order. To determine the simulated execution time each event is considered in virtual order, and the execution time calculated based on the time of execution of events earlier in the virtual order that the current event is dependent on for control, data or resources.

Virtual order simulation allows dependencies to be tracked easily because the register or memory values used in the instruction are the most recently calculated. In the presence of speculative execution this means that only the correct execution needs to be considered, since the instructions it is dependent on have already been calculated, and the earliest time the correct data is available is known. This reduces the complexity of simulation, since speculative execution and rollback do not have to be tracked.

The real time the instruction completes execution is calculated by taking the maximum of the times the input data become available and simply adding the instruction latency.

4.1.1 Assumptions

Virtual order simulation allows a speculative architecture with a large instruction window to be simulated by a single instruction thread because instructions are considered one at a time, and each instruction only has to be considered once. However, the real time at which each data value is available must be stored, requiring extra storage space.

Large amounts of control and data coherence detail have been abstracted away, leading to much faster development cycles. However, this simplification means that virtual order simulation is only suitable for high level analysis.

Virtual order simulation assumes oracular knowledge of events is available for scheduling resources. Priority is given strictly to events earliest in the virtual order, since events later in the virtual order have not been considered at that stage of the simulation, even though they may begin earlier in real time. This represents an upper bound on scheduling performance, which may not be achievable in a real architecture.

Since each instruction is processed in program order, virtual order simulation does not track the transient state of each instructions inputs which are subsequently rolled back. These transient states will often propagate through a sequence of dependent instructions, and can result in speculative speedup if the correct result is obtained from speculatively incorrect computation. Section 4.1.3 discusses how virtual order simulation can be extended to record these changes in state.

4.1.2 Limiting Resources

The basic virtual order simulation model assumes there are sufficient resources for unrestricted execution of the program. In order to obtain realistic results for a system it is necessary to restrict the resources available. These resources include storage, such as memory and state saving space, bandwidth and processing resources, such as functional units.

To simulate limited resources each resource is allocated to events in virtual order, and a

record is kept of when the resource is available. If a resource is unavailable at the desired execution time the event is stalled until the resource can be allocated. Some resources must be retired in order, such as frames, while others can be retired out-of-order, for example processing units.

This method of resource allocation simulates an architecture with oracular knowledge of resource requirements.

4.1.3 State History

The basic virtual order simulation model does not capture information about computation which executes speculatively, is rolled back and never committed. Although these transient states have no effect on the results of the computation, they can affect the execution time.

Transient computation states still consume resources, which could otherwise be used for useful computation. In the basic model it is assumed that the incorrect transient states can be identified pre-emptively, and are allocated resources with a lower priority than useful computation. This is unlikely to be possible in a real machine, so simulating the resource requirements of transient states will provide more accurate simulation.

Transient states are modelled by replacing state values with state history lists. Rather than a single value state which becomes valid after a certain real time, a state history is a succession of values which supersede the previous value in the list as real time advances.

When an instruction is processed the input history lists merge to form an output history list. The output history list records all values generated by the instruction operating on all valid combinations of input values. This may produce a smaller or larger state history list, depending on the combination of states in the input history lists.

Figure 4.1 shows two examples of history lists being processed by instructions to generate an output history list. Figure 4.1(a) shows an addition operation where the final values of the two operands are obtained at times 15 and 27 respectively. Both operands have transient speculative values prior to that. The first entry in the output list is formed at the time both input lists have a valid value, plus the operation latency. In this case the first operand receives value 0 at time 5 and the second operand receives value 100 at time 13. With an

operation latency of 1 the first entry has a time of 14 and a value of 100. For each new operand value in the input lists a new entry is placed in the output list, until the committed value is obtained at time 28.

In Figure 4.1(b) the less-than comparison evaluates to true at time 14, then again at time 16 and time 28. Since only a change in the output value will require dependent operations to re-execute these nodes are redundant and do not need to be recorded. Note that this is an example of speculative speedup beyond the "theoretical" maximum being obtained because the correct answer was obtained at time 14, even though the correct inputs were not available until time 27.

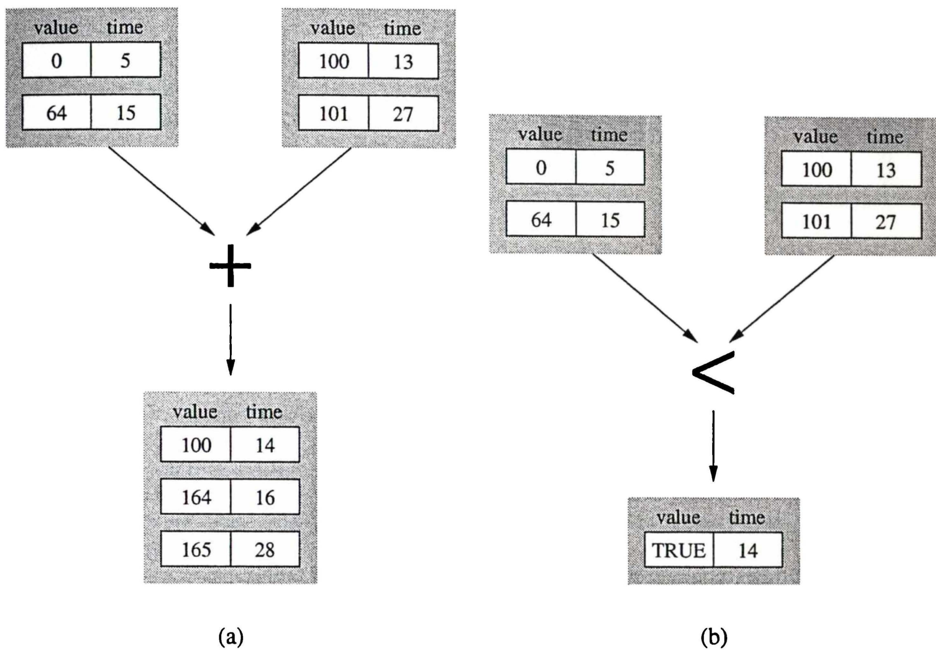


Figure 4.1: Processing input history lists to form an output history list.

4.2 Simulation Test Bench

Littin [2000] provides a detailed description of the development and operation of the Warp-Engine simulator based on the virtual order simulation paradigm. A brief description is provided here, focusing on the aspects pertinent to the work in this thesis.

The WarpEngine extracts parallelism at two levels: block level and instruction level. Similarly, the virtual order simulation model is applied at the frame level, and at the instruction

level. Frames are processed one at a time in their virtual order, as defined by the control tree. Instructions within a frame are scheduled in dataflow order, but are processed using virtual order techniques. Figure 4.2 shows the interconnection between the major components of the WarpEngine simulator.

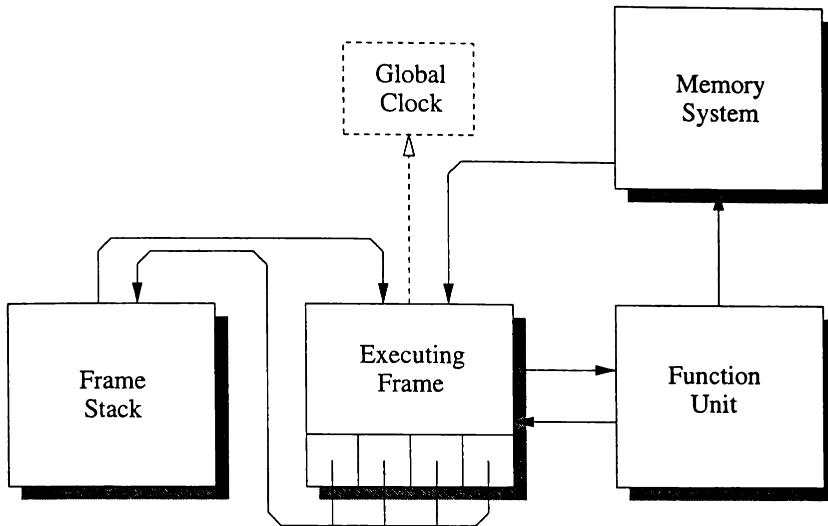


Figure 4.2: Components of the WarpEngine virtual order simulator

The *frame stack* is used in the simulator to enforce the virtual order in processing frames. Initially a single starting frame is processed and any child frames it generates are pushed onto the frame stack in reverse virtual order. When the frame has been processed the next frame is popped off the stack, and processed in the same way, pushing its child frames onto the frame stack in reverse virtual order, and the next frame is popped off the stack. In this way frames are processed in a left-to-right traversal of the control tree, maintaining the virtual order. When the frame stack is empty and the currently executing frame has been processed the program has completed.

At any point in the simulation only one frame is being processed, and it is held in the *executing frame* component. The executing frame manages the instruction processing, handling interaction between instructions, communicating with the function unit and passing data to child frames. To execute the instructions within a frame in dataflow order, each slot is checked to see if its input registers contain valid values. If all registers are valid, the instruction is sent to the function unit for execution. The slot checking process is repeated until all instructions in a frame have been executed.

The *function unit* processes individual instructions, forwarding the results to registers in the

executing frame, to registers in frames on the frame stack, or to locations in the memory system. The function unit also calculates the real time the operation completes from the known latency of the operation and the real time the operands and resources are available. All instructions except memory operations are processed by the function unit. Memory operations are forwarded to the memory system.

The basic *memory system* component abstracts away the details of the time-space cache. All stores are recorded in a list for each memory location, along with the real time they occurred, so that a value and a real time (or a list for the transient state extension) can be returned to the executing frame when a load is executed. Loads do not need to be recorded because the virtual order simulation ensures that instructions never need to be re-executed. Since the virtual order is maintained by the frame stack, explicit VTSs are not required, and the mechanism for maintaining the virtual order never constrains execution. However, VTS schemes which do constrain execution can be implemented in the memory system component and this technique is used later in this thesis to refine the memory system model.

A *global clock* is used to record the greatest end time of any frame that has been processed. When each frame is retired the global clock is updated to contain the maximum of its current value and the frame's end time. Each advance of the global clock indicates a time at which frames are retired. When all frames have been processed the value of the clock gives the program's execution time.

4.2.1 Simulation Parameters

The main performance metric used throughout this thesis is the speedup resulting from executing the test programs on the WarpEngine against the time for a sequential in-order processor modelled on the WarpEngine. Speedup is used rather than absolute execution time, since a number of assumptions have been made about the processor architecture, beyond extracting parallelism. To model execution time accurately would depend on detailed modelling of the components not available in the virtual order simulator. These simplifications were validated by Littin [2000].

The parallel execution time on the WarpEngine is given by the retirement time of the last instruction in the program. Since instructions are retired in order this will be the overall pro-

gram execution time. The sequential execution time is calculated assuming that instructions are executed in their virtual order, with each waiting for the previous instruction to complete before beginning. This means that even a sequential machine with simple pipelining would have a speedup greater than one.

While a number of prominent theoretical studies of parallelism [Lam and Wilson, 1992; Wall, 1991; Postiff et al., 1999] assume instructions with a single cycle latency, this is not representative of modern computer architectures. More typical instruction latencies have been chosen for these simulations, as shown in Table 4.1. It has been found that variations in the instruction latencies tend not to substantially affect the simulation results [Littin, 2000], and only relative performance change is studied.

instruction	*cycles*
CHILD	15
ST	2
MV MA	8
CMP	2
ADD SUB	2
MUL	7
DIV	13
SPLIT	2
AND OR XOR	2
ADDF SUBF	7
MULF	7
DIVF F2I	13
I2F	7

Table 4.1: Typical instruction latencies

Although they are as applicable to the WarpEngine as any other out-of-order architecture, no branch or value prediction is used in the simulations in this investigation. Parallel execution beyond a conditional branch is only performed where control independence exists, or where memory speculation has resulted in a subsequent branch being executed speculatively. The only value speculation that takes place is early usage of memory values, and values calculated by dependent operations. Branch and value prediction techniques are orthogonal to the speculative mechanisms used in the WarpEngine and can be added independently. Including either of these techniques will increase the speculation that can be achieved and allow more parallelism to be extracted than is shown here.

4.2.2 Virtual Order Simulation Effects on Performance

While virtual order simulation is a very effective technique for fast simulation of out-of-order speculative architectures, it does have a number of shortcomings in the features it can accurately simulate. These limitations, which are particularly manifest in complex real time interactions, and their effects on the performance measurements made here, are summarized below. They are discussed in more detail in the relevant sections.

As mentioned earlier, the virtual order simulator models perfect knowledge of events for the purposes of instruction scheduling and resource allocation. This leads to optimistic performance results since this oracular knowledge would not be available to a real processor. In the context of the WarpEngine this means that frames and instructions that would otherwise get rolled back are never issued, saving the processor from rolling them back, and the consequent performance penalty. It does, however, demonstrate an upper bound on the performance of such a processor. The performance of a real processor may approach this upper bound with sufficiently sophisticated resource usage estimation techniques.

Understanding the effects of oracular knowledge is particularly important in understanding the results presented for VTS schemes. When allocating VTSs to a frame, the simulations presented here assume that the number of VTSs reserved for subtrees earlier in the virtual order is known. This provides the lower limit of the VTS range for the current subtree, and is always known in virtual order simulation since all frames earlier in the virtual order have already been processed. In the real order execution the virtually earlier frames may not have had VTSs allocated to them when the virtually later frame is executed.

The VTS usage estimates are largely static, so the VTS range could be reserved prior to execution. However, if a frame is conditionally executed, the VTS range required cannot be calculated precisely at compile time. The maximum VTS range that would ever be required can be allocated, although this is a less efficient allocation than the one simulated, and will decrease performance. If the VTS range is calculated dynamically, for example based on a dynamic loop bound, the range required is not easily determined at compile time. In this case a speculative estimate could be used, possibly incurring the penalty of VTS allocation rollback and reallocation.

The basic model does not simulate transient states, which leads to underestimation of re-

source usage requirements and a subsequently inflated performance measurement. Conversely, it may give pessimistic performance calculations where there are sufficient resources, since there may be cases of incorrect speculative execution leading to the correct answer, which is used for further speculative computation and later validated. Comparisons of simulations under different constraints are still valid though, since all are simulated without transient states, except the virtual memory system in Chapter 8.

Adding the transient state extensions negates much of the performance benefit of virtual order simulation. This makes it impractical in terms of simulation time and memory required to use on several programs in the test suite, even though they are small programs. For these reasons the transient state extensions are used sparingly, and only on selected test programs. Transient states are only vital in measurements of resource consumption and rollback statistics.

Complex real time interactions are particularly prevalent in the memory system. Simulation of caching in general is not possible, since virtual order simulation cannot easily track the most recently used values at any point in the simulation. Any attempt to extend the paradigm to include this will further degrade the performance of the simulator, providing no advantage over traditional real order simulation. There are several other memory system features particular to the WarpEngine which cannot be simulated using virtual order simulation. These are discussed in more detail in Chapter 8 where the virtual order memory system is proposed.

4.3 Test Programs

A suite of test programs has been developed to benchmark the performance of the WarpEngine in different configurations. Littin [2000] provides a detailed analysis of the test programs, their control and data structures, and performance characteristics. A summary is provided here, the source code can be found in Appendix B.

The test suite spans the types of operations performed in many programs, including matrix and array manipulation, sorting, dynamic structure operations and recursion. Table 4.2 lists the abbreviations used for each algorithm within the test suite. These small loop oriented programs were chosen because there is no compiler for the WarpEngine instruction set, and

all programs were hand coded in WarpEngine assembly.

"Real world" applications contain more complex data and control interactions than the simple programs presented here, suggesting that the simulation results are optimistic. However, the theoretical parallelism available and algorithmic complexity of these simple test programs can be determined, providing a baseline to compare the simulated results to.

Although the programs are small, they contain varying amounts of control and data dependence. Table 4.2 shows the sequential execution time (*work*) and the *speculative parallelism* available for each algorithm. The parameter N is a measure of the problem size. The work and parallelism are given as complexity measures using $\mathcal{O}$ notation and only consider the complexity of the data dependencies in the program. Loops are assumed to parallelize perfectly and operate in $\mathcal{O}(1)$ time, meaning that their control mechanism has no impact on performance. Speedup from data value speculation is not considered in these measures, providing the possibility that a real system may exceed these values. See Littin [2000] for the derivation of these values.

algorithm	*abbr.*	*work*	*parallelism*
matrix multiplication	mat	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
transitive closure	trans	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
Gauss-Jordan elimination	gj	$\mathcal{O}(N^3)$	$\mathcal{O}(\frac{N^2}{logN})$
quicksort		$\mathcal{O}(NlogN)$	
left-to-right search	qu1		$\mathcal{O}(1)$
ends-to-middle search	qu2		$\mathcal{O}(1)$
binary tree insertion		$\mathcal{O}(NlogN)$	
naive	bin		$\mathcal{O}(N)$
AVL	avl		unknown
Fibonacci numbers	fib	$\mathcal{O}(2^N)$	$\mathcal{O}(\frac{2^N}{N})$

Table 4.2: Complexity of test algorithms

While the data set chosen does have an impact on the absolute execution time of the sorting and tree insertion routines, previous studies [Littin, 2000] showed that the data set does not affect the relative performance. For this reason, and in order to include results from long running simulations, only results from simulations of a single data set are presented.

4.3.1 Matrix Manipulation

Three matrix manipulation algorithms, matrix multiplication, transitive closure and Gauss-Jordan elimination have been included in this test suite. Matrices provide well structured data and allow operations to be performed on multiple instances of data in an orderly manner. In general this allows parallelism to be easily detected at compile time and many architectures, particularly vector machines, do a good job of extracting parallelism from these algorithms. These algorithms have been included in the test suite to determine if speculative execution can extract this statically detectable parallelism. The amount of unpredictable control flow and the arrangement of sequential and parallel components varies among the three algorithms.

Matrix multiplication

Matrix multiplication multiplies two $N \times N$ matrices of floating point numbers. All control and data dependencies can be detected at compile time, allowing parallel scheduling to take place. There are three levels of nested loops, each containing N iterations, giving a sequential execution time of $\mathcal{O}(N^3)$.

The innermost loop contains a sequential data dependence chain of length $\mathcal{O}(N)$ through the addition of values to a temporary accumulator variable. This constrains the parallel execution time, giving $\mathcal{O}(N^2)$ parallelism.

Littin [2000] investigates several different options for the control structures of the three loops. In the experiments described here only the highest fanout tree structure is used.

Transitive closure

The transitive closure algorithm [Corman et al., 1990] processes the adjacency matrix for a directed graph such that an edge directly from x to y is added if a path from x to y exists. The algorithm again contains parallelism that is easy to detect. Like matrix multiplication it has three levels of nested loops, and hence a sequential execution time of $\mathcal{O}(N^3)$. This time the dependencies are in the outermost loop. Once again the algorithmic parallelism is $\mathcal{O}(N^2)$.

Gauss-Jordan elimination

Gauss-Jordan elimination [Press, 1992] performs matrix inversion of an $N \times N$ matrix. Unlike matrix multiplication and transitive closure, control decisions are coupled to the values of the input data. Once again the sequential execution time is $\mathcal{O}(N^3)$, but the control dependencies mean the available parallelism is only $\mathcal{O}(\frac{N^2}{logN})$.

4.3.2 Sorting

Sorting algorithms are represented in the test suite by two versions of the efficient quicksort routine. As with the matrix operations the data is in well structured arrays. However, sorting is an intrinsically sequential operation because the data is unpredictable and the program control decisions are highly coupled to it. These routines have been included to show that speculation can extract parallelism from data controlled execution.

Quicksort

Quicksort [Quinn, 1987] has a worst case sequential execution time of $\mathcal{O}(N^2)$, but the expected sequential execution time is $\mathcal{O}(NlogN)$. This algorithm lends itself to sequential scheduling due to the way it subdivides the data set as sorting proceeds. The algorithm selects a pivot element which is used to sort the array into two parts. Next the pivot selection and sorting routine is called recursively on the two parts of the array, which can be operated on independently. The expected parallelism is $\mathcal{O}(logN)$ and the worst case parallelism is $\mathcal{O}(1)$.

The two versions of quicksort used here differ in the way they select the pivot. The first selects a pivot value and moves from left to right through the array until all elements less than the pivot value are to the left of it. The second version selects a pivot value and moves in towards the centre, swapping from either end until the pivot is in the correct position. These pivot selection mechanisms have no impact on parallelism estimates, but they do have a significant impact on performance in a speculative architecture. In a speculative architecture the recursive sort calls can be initiated in parallel if the data is correct. With the left-to-right pivot search subsequent left recursive calls can begin their search phases

without knowing the right hand array limit. The ends-to-middle pivot search must know both ends of the sub-arrays before searching can begin. This means mis-speculation is more likely in this selection method.

4.3.3 Dynamic structures

Two binary tree insertion routines have been included in the test suite. Items are inserted into a dynamic ordered tree structure. As with sorting, the data determines the program's control flow, although this time the data structure varies in size and shape as new items are inserted. In general it is difficult to write parallel versions of these routines because interactions between data are unpredictable and control is highly coupled to data. These routines have been included to show the ability of a speculative architecture to extract parallelism from programs with complex and unpredictable data flow and dynamic data structures.

Naive binary tree insertion

Naive binary tree insertion [Lewis and Denenberg, 1991] creates a tree without making any attempt to balance it. In the worst case pathological data makes this the equivalent of inserting into the tail of a linked list, with sequential execution time $\mathcal{O}(N^2)$. However, because the values inserted are random the expected sequential execution time is $\mathcal{O}(NlogN)$.

Nodes may be inserted in parallel, provided the earlier node in the virtual order does not fall on the path through the tree to the insertion point of the second node. Normally locking would have to be done on the nodes to ensure that parallel insertions are independent. The speculative control in the WarpEngine eliminates the need for locking by speculatively inserting nodes in parallel, and detecting any dependencies and re-executing them. This eliminates the false dependencies between insertions and gives an expected parallel execution time of $\mathcal{O}(logN)$. In the worst case the insertion pipeline would be N stages giving $\mathcal{O}(N)$ parallel execution time. Thus, in both worst case and expected case the parallelism is $\mathcal{O}(N)$.

AVL binary tree insertion

AVL binary tree insertion [Lewis and Denenberg, 1991] maintains a balanced tree. This gives amortized insertion time $\mathcal{O}(logN)$, and sequential execution time $\mathcal{O}(NlogN)$. The same problems with naive insertions arise when trying to schedule insertions in parallel. With AVL the number of possible data dependencies in any search path increases because the internal nodes get rotated to maintain a balanced tree.

Traditionally, when inserting in parallel a lock would have to be placed on the root node to allow the rotations that may occur to operate correctly. The lock is freed when the insertion is complete and any associated nodes have been processed. This locking of the root node would serialize computation removing most of the opportunities for parallelism.

However, many of the rotations that occur will not affect the nodes near the root, so some searches can be performed in parallel without being rolled back. In the WarpEngine parallelism may be obtained through speculative insertions to data independent parts of the tree. This parallelism is hard to detect without knowledge of the data inserted making complexity analysis difficult.

4.3.4 Recursion

The recursive Fibonacci number generation algorithm was originally included to show that the WarpEngine can be programmed to perform recursive routines. It is computationally inefficient, but does map well to the WarpEngine's tree-structured control mechanism.

The large amounts of available parallelism are useful in exercising the resource allocation and selective speculation techniques developed later in the thesis.

Recursive Fibonacci number generation

Recursive Fibonacci number generation calculates the Nth number in the Fibonacci sequence using a recursive process. Each call of the fib() routine sums the results of two calls to itself with successively smaller parameters, giving a sequential execution time of $\mathcal{O}(2^N)$. Like quicksort, the problem is divided into smaller independent problems at each recursive

call, suggesting that parallel scheduling can be used. The summation of the returned values introduces a sequential dependency path of length N. This gives a parallel execution time of $\mathcal{O}(N)$ and parallelism of $\mathcal{O}(\frac{2^N}{N})$.

4.4 Compilation

A compiler has not been developed for the WarpEngine instruction set at this stage, and is beyond the scope of this thesis. The block structured architecture and tree structured execution mean that substantial work would be required to modify an existing compiler, targeted at a contemporary architecture, for the WarpEngine.

Neefs et al. [1997] discuss the difficulties encountered in compiling for block structured architectures due to the constraints imposed on instruction placement in blocks. The chief difficulties include the fixed maximum size of a block and the restrictions on register usage both between blocks and within a block, caused by single assignment. The paper presents a plausible approach to compiling for fixed length block structured architectures. Many of these issues are similar to those faced in compiling for VLIW architectures [Biglari-Abhari et al., 1998].

Mapping the program onto the tree structured control mechanism allows scope for optimization, as shown in Section 3.2.2. However, achieving the most efficient tree structure in some cases requires deep knowledge of the algorithmic behaviour that would be unrealistic for a compiler. The standard loop and branch structures could be generated using standard code libraries to map them onto the tree.

Due to the experimental nature of the WarpEngine instruction set and the size of the task it was deemed impractical to build a compiler at this stage of the project. Instead the suite of test programs was manually compiled from C code. Care was taken, however, to restrict the process to operations which would be feasible for an automated compiler. The C code for the test suite can be found in Appendix B, with the corresponding assembly code available in [Littin, 1999].

Littin [2000] performed some investigation into different control tree structures for loops and found that a structure maximizing the fanout of frames generally gave the best perfor-

mance. This structure has been used in all test programs in this thesis. Like Littin's studies, the WarpEngine simulated here generates up to four children from each block. The number of instructions in each block is maximized, although no loop unrolling is done.

Not all algorithms are simulated using the virtual order control mechanisms described in subsequent chapters because some of these techniques require compiler support. It is not practical to do this manually for the entire test suite, so a subset of the test programs with the most relevant features have been selected. This process is described further in the relevant sections.

Compiler techniques for use with architectures like the WarpEngine are an interesting area of ongoing research.

4.5 Summary

This chapter has outlined the simulation test-bench used throughout this thesis. The virtual order simulation paradigm is a novel technique especially suited to high level simulation of speculative architectures. Virtual order simulation provides a computationally efficient simulation model for speculative execution by considering each instruction only once. Extensions have been introduced to extract further execution details arising from transient execution and limited resources. This simulation method has previously been used in preliminary studies of the WarpEngine architecture core and is extended in the following chapters to explore the design decisions in creating a realizable virtually ordered memory system.

A hand compiled suite of test programs has been devised which represents the different classes of computation commonly found in programs. Theoretical complexity measures have been presented for these algorithms and are used in later chapters for comparison against the simulation results obtained.

In high level studies of this nature there will inevitably be simplifications and assumptions made about some of the architectural features used. Certain common architectural features, such as pipelining and branch prediction have been omitted from both the WarpEngine simulation model and the reference sequential architecture. Perfect instruction scheduling has been assumed in order to provide an upper performance bound. This is the most obvious

example of complex real time interactions that are idealized in the virtual order simulation. To properly capture these interactions real time simulation is necessary.

The virtual order simulation model is used throughout the rest of this thesis to extend the WarpEngine model to explore different methods of controlling speculation. This culminates in the simulation of a virtually ordered memory system.

Chapter 5
Explicit Virtual Timestamps

5.1 Concept

There are a number of situations in which the WarpEngine must be able to determine the strict virtual order of frames. When a read request is made to memory the value returned should be the one most recently written to that address. When a write is made it must cause the rollback of any reads later in the virtual order and cause them to be re-executed using the newly written value. Any reads with another write between them and the new write in the virtual order should not be re-executed. When frame resources are exhausted the latest frames in the virtual order are cancelled back to create space for earlier frames.

One possible way to determine the virtual order of frames would be to order the frames in hardware, in much the same way that modern superscalar processors use a reorder buffer (see Section 2.4.3). Figure 5.1 shows a hardware ordering of memory operations in a buffer, with a read and write being inserted in their places in the virtual order. The read (a) searches to the left to find the previous write, with the value 1, which is returned. The write with value 5 in (b) supersedes the value 0, which was returned by the read in the slot to the right and must be re-executed. The write with value 1 means that no further reads are affected.

The WarpEngine may require thousands of frames to be ordered, with the capability to insert frames in the middle of the sequence, which would stretch the limits of a centralized ROB. The memory system would require a similar hardware ordering of the multiple versions stored at each address, and this is more problematic. A novel hardware structure for this purpose is proposed in Chapter 8.

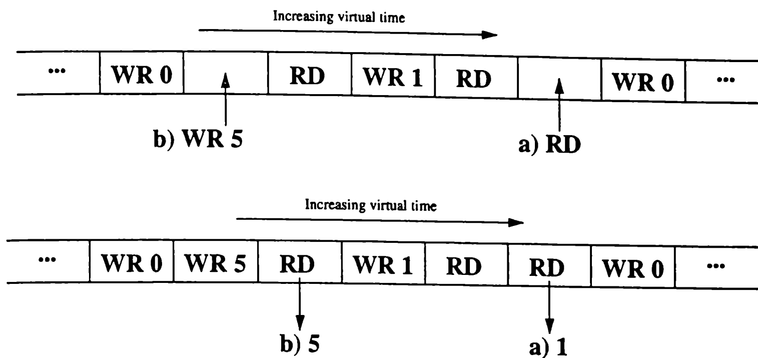


Figure 5.1: Returning the correct value from a read request and rollback due to a write

A conceptually straightforward way to approach virtual ordering is to tag each frame, and its memory operations, with a number representing its position in the virtual order. In the Time Warp literature the tag is known as a *timestamp* since it represents simulated time. In the WarpEngine it simply represents the causal ordering of the instruction, so it is called a *virtual timestamp (VTS)* to avoid confusion. The VTSs of two frames can be compared without regard for other frames to determine causal ordering. This provides a decentralized method of determining ordering between frames and memory operations, with the possibility of scaling to large numbers of frames, more akin to a software technique.

Figure 5.2 shows the same example as Figure 5.1, but storing the memory operations as a separate list, using integer VTSs to order the memory accesses, rather than position. Notice that hardware slots do not have to be reserved for the memory operations and it is not necessary to keep the list in order, although this aids efficient searching.

In the next two chapters several alternative VTS representations are described and investigated with respect to their feasibility as methods for tracking the virtual order of instructions in the WarpEngine.

Initially a naive approach is taken in order to keep the VTS allocation mechanism as simple as possible. The trade off is that a simple allocation mechanism is likely to be faster to execute, but result in poorer allocation efficiency. So while individual operations are faster overall execution may be slower because the VTS supply is exhausted more quickly and the VTSs have to be reallocated to allow execution to continue.

The length representation is a very simple method of allocating ordered bit strings to frames as VTSs. This assumes a balanced execution tree for efficient allocation of VTSs. Typically

this is not the case and many available VTSs are not allocated to frames. This requires reallocating the VTSs frequently to maintain a supply of available VTSs.

To improve allocation efficiency the exponential representation tries to more closely match the observed execution tree shape by allowing the left side of the VTS tree to be deeper than the right. The shape of the VTS tree is still fixed and relies on the execution tree matching this closely for good allocation.

In Chapter 6 more advanced analysis of the execution tree shape is introduced at compile time. A variable range of VTSs are allocated to each subtree, according to its estimated requirements. Although the exact shape of the execution tree cannot be determined until run time, greatly improved allocation efficiency is achieved by compile time analysis. Since this analysis is done at compile time the only runtime overhead introduced is in decoding and using the analysis results for VTS allocation.

Including additional instructions in the program to calculate VTS requirements dynamically for structures such as dynamically bounded loops further improves the allocation efficiency at the expense of increasing the instruction count of the program.

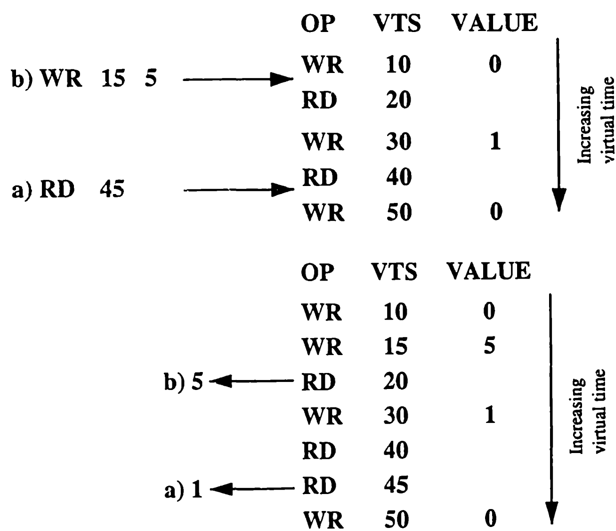


Figure 5.2: Returning the correct value from a read request and rollback due to a write using VTSs

5.1.1 Requirements

The purpose of the VTS is to provide a method of determining the program (or virtual) order of speculative memory accesses, which may occur in any real order. This is important so that the correct value is eventually returned from a read, whether it was available when the request was originally satisfied or not.

The time-space cache contains an entry for every memory access (reads and writes) together with the VTS of the frame that initiated the access. This requires many VTSs to be stored in the time-space cache, so it is important that the number of bits required is kept to a minimum. However, the situation is not as bad as it first seems because once frames become non-speculative they no longer require a VTS, since all earlier frames have completed execution. In fact, they are no longer required in the time-space cache, and can be *committed* to main memory. Access to this non-speculative data can be handled using the usual techniques if (and only if) no speculative match is found. This substantially reduces the required size of the time-space cache. The resources used by the committed frame, including its VTS, can be made available for reuse by other frames.

Timely commitment of frames once they become non-speculative minimizes the number of active VTSs in the system at any given time. However, there are certain practical difficulties in committing VTSs efficiently in the decentralized environment of the WarpEngine. A frame is only non-speculative if there are no earlier frames currently executing (although there may be earlier uncommitted frames). Determining the latest non-speculative frame (GVT) requires knowledge of the state of all frames in the system, either through a centralized record (which could form a bottleneck), or through a distributed communication system. This practicality is not examined for the software based schemes in this chapter and the next. However it has been extensively studied in the Time Warp literature [Bellenot, 1990; Fujimoto and Hybinette, 1997; Lin and Lazowska, 1990], where it is known as *fossil collection*. Preliminary discussion of commitment can be found in Chapter 8 for the twisted memory time-space cache.

Since each speculative memory access will require comparing several VTSs comparisons must be fast, although this is not necessarily crucial because, by definition, speculative operations are unlikely to be on the critical path. It is more important that the mechanism for comparison is resource efficient. There are likely to be many memory accesses in flight

in parallel, so the mechanism will have to be replicated many times.

A block cannot begin executing until it has been assigned a VTS. For this reason VTS generation needs a low latency, or it risks slowing the execution of blocks, limiting the number of instructions in flight and ultimately reducing the parallelism extracted.

An effective scheme must be able to efficiently represent a large number of VTSs. There are a potentially unbounded number of frames used in the execution of a program, since physical frames may be used many times. Added to this is the need to insert an arbitrary number of VTSs between any two adjacent VTSs. This allows dynamically bounded loop structures to be executed in parallel with following control independent code. The code in Figure 5.3 will generate the execution tree shown in Figure 5.4. The number of iterations of the while loop (shown by nodes marked A and B) is unknown at the time node C is created in parallel with the first iteration. All iterations of the while loop must have a VTS earlier than C.

```
while (A)
    B;
C;
```

Figure 5.3: Code for a while loop and later independent code

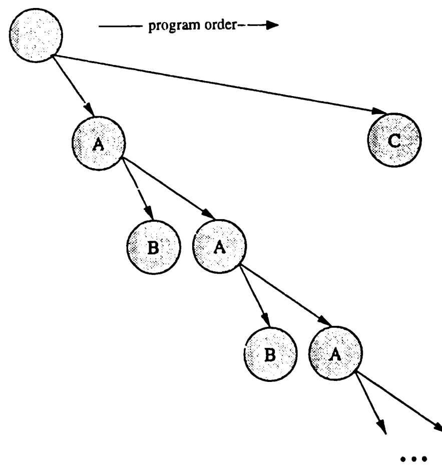


Figure 5.4: Execution tree for a while loop and later independent code

Any finite length VTS representation has the potential to be exhausted, if no VTSs are available for allocation at the appropriate place in the virtual order. There are two situations in which this can occur: at the end of the virtual sequence; and between two VTSs within the virtual order. In the former case there are simply not enough VTSs in the representation,

allowing for some inefficiency in allocation, to represent all the frames in the program. In the latter case insufficient VTS space has been reserved for a subtree when the VTS for the start of the next subtree was assigned. All practical representations must have a way to recover from the exhaustion so that execution can continue to completion. The VTSs assigned to frames that have been fossil collected are available to be reused by other frames. However, by definition they are located at the beginning of the virtual order, while the frames awaiting VTS allocation are in the middle or at the end of the virtual order.

The VTSs currently in use must be reallocated to make unused VTSs available at the right place in the virtual order for allocation to new frames. The only restriction is that all active VTSs (i.e. those which have not been fossil collected) must retain the same relative ordering. Conceptually, this means reallocating the early, unused VTSs to the active frames earliest in the virtual order, freeing their VTSs for allocation to new frames. We refer to this process as *rescaling* the VTS tree. Several methods for rescaling fixed length VTS trees are examined in Section 5.4. The performance of each method is a trade off between the speed of rescaling and the efficiency of the allocation in the rescaled tree. A more efficient allocation will require rescaling to be done less often.

Just as cancelback is used in Time Warp to free resources to allow simulation to progress, it can be used to free VTSs in the WarpEngine. If rescaling cannot supply VTSs to allow execution to progress, cancelback can be used to reclaim VTSs from frames latest in the virtual order. These can then be used for earlier frames and the cancelled frames can be restarted when there are sufficient VTSs available beyond those needed by the active subtree. Cancelback should be used sparingly because it requires discarding substantial amounts of potentially correct speculative execution.

5.2 Symbolic Representation

The greatest difficulty in generating VTSs for general purpose computation is allocating an arbitrary number of VTSs between any pair of existing VTSs. Back and Turner [1995] suggest that this can be achieved by representing the VTSs (which they refer to as time-stamps) as real numbers, since there are infinite real numbers between any arbitrary pair. An equivalent way of conceptualizing this is as a string representing the path from the root

of the tree to the node. In a binary tree each time a left branch is taken a '0' is appended to the string and a '1' is appended for a right branch. VTSs vary in length, getting longer the deeper in the tree a node is, so a terminator is needed, represented by the symbol Δ. The simple lexicographic comparison $\Delta < 0 < 1$ can be used to determine the relative ordering of any two VTSs, preserving the pre-order left to right traversal of the tree. A three level binary tree is assigned VTSs using this method in Figure 5.5.

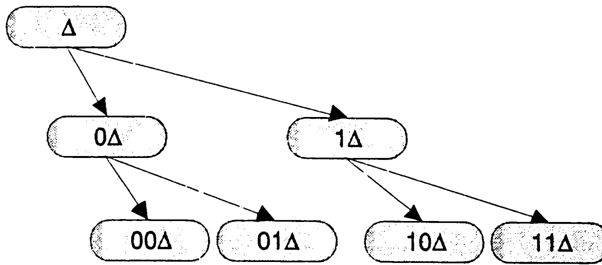


Figure 5.5: Conceptual VTSs for a binary tree

Variable length VTSs, however, would be troublesome to implement in hardware, fixing the length to a constant word size, say 32 or 64 bits, would be much more convenient. Variable length VTSs could also grow to become extremely long in the execution of a large program, which would consume an impractical amount of bandwidth and memory.

Fixing the maximum length of the VTSs effectively fixes a maximum depth on the tree. In the symbolic representation in Figure 5.5 a 32 symbol VTS would limit the tree to 32 levels (the Δ at the root, and one symbol for each successive level). If the actual execution tree is deeper than that, some VTSs must be fossil collected and reused by rescaling the VTS tree.

5.3 Fixed Length VTS Schemes

Three different VTS methods—*length*, *exponential* and *ideal* VTS representation—are proposed in the remainder of this chapter and evaluated through simulations. Although the WarpEngine uses a four-way execution tree, for simplicity the representations are described here using a binary execution tree. As shown in Section 3.2.2 a four-way tree can be trivially mapped onto a binary tree, using every second level as intermediate splitting nodes.

5.3.1 Length Representation

The main difficulty with the symbolic VTSs described in Section 5.2 is their variable length, making them awkward to implement in hardware. By representing them as a uniform length string and treating them as integers, rather than bit strings, they can be more easily manipulated for construction, comparison and rescaling.

The bit string VTS can be converted to a uniform length integer by padding the bit string to I bits with zeros and then appending the original length of the string (without the terminating Δ). The number of bits needed for the length, L, is $\lceil \log_2 I \rceil$.

Some integers remain unused in this representation (those corresponding to the contradictory long string of bits with a short length count appended). Table 5.1 shows the division of different sized length representations and the number of levels of nodes they can represent. Figure 5.6 shows the tree for a four bit length representation VTSs, with the string and length indicator separated by a comma. An unused representation in this example is '01,00'. Likewise, '01,11' is an unused representation in a four bit length representation VTS, since the maximum string length is two symbols and the length indicator '11' represents a length of three symbols. These unused representations mean that only seven out of the fifteen possible four bit integers are used. This inefficiency means that less frames can be allocated VTSs, and thus be in flight, concurrently than if the whole integer space was used.

The relative order of two VTSs can be determined by comparing the integers—the lower integer is the earlier VTS. Section 5.4 discusses some possible rescaling techniques for use with the length representation in detail.

This naive representation is attractive for its simplicity. Generation and comparison operations are simple arithmetic operations which can be done quickly and cheaply in hardware. However, it assumes the execution tree will be balanced, with frames occupying every possible position in the tree. If this is not the case the allocation will be wasteful of VTSs.

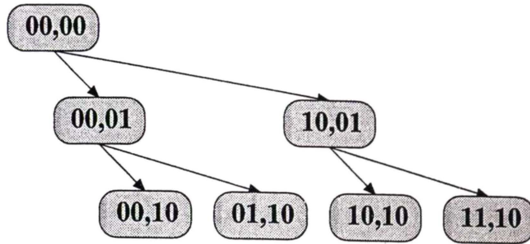


Figure 5.6: VTSs in length representation

5.3.2 Exponential Representation

VTSs can be allocated to a program more efficiently if the VTS structure maps well to the execution tree shape. This will result in less of the VTS space being wasted on tree locations which aren't filled by a frame. One approach is to allocate shorter VTSs to the positions in the tree most likely to be filled, and use longer VTSs if necessary for the rarely used locations. This allows greater tree depth before exhaustion in the most likely cases.

Examining the code for the suite of test programs, the majority of frames issued are naturally in position zero (the leftmost in the tree of the four possible children of a frame, known as *child zero*), as shown Figure 5.7. This depends to a large degree on the way loops are structured (see Chapter 3). To use child zero as much as possible the compiler can attempt to assign child zero as a frame which executes unconditionally, or with a high frequency. Sometimes this may be impossible due to semantic constraints on the virtual order of the program.

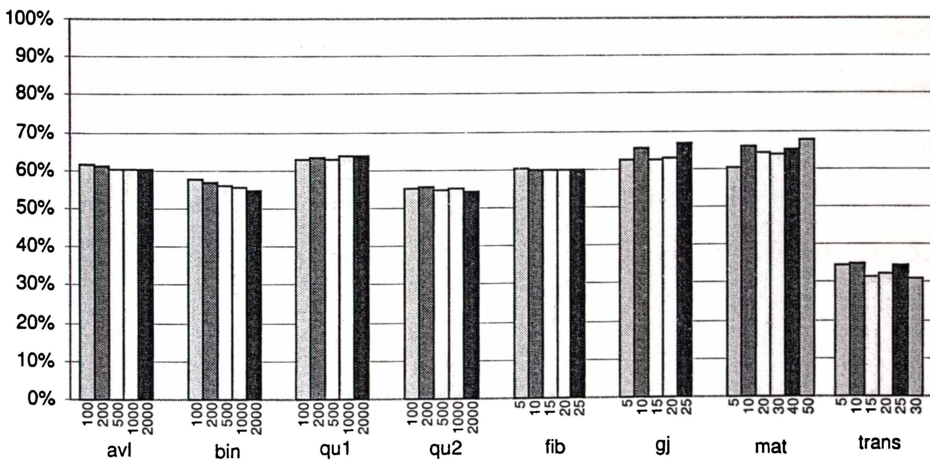


Figure 5.7: Percentage of children generated with child number zero

This suggests that there is potential for VTSs to be optimized to allocate the most frequent case, child zero, more efficiently at the expense of efficiency for other child numbers. The *exponential representation* is an attempt to do this using a scheme similar to floating point number representations to allow different parts of the tree to grow to different depths. It is comprised of two parts: a mantissa and an exponent. The exponent is the number of leading zeros in the VTS, while the mantissa is the normalized tail of the VTS. In the example in Figure 5.8 the VTS 0010Δ becomes $2, 10\Delta$, where the number to the left of the comma is the exponent and the binary string to the right of the comma is the mantissa.

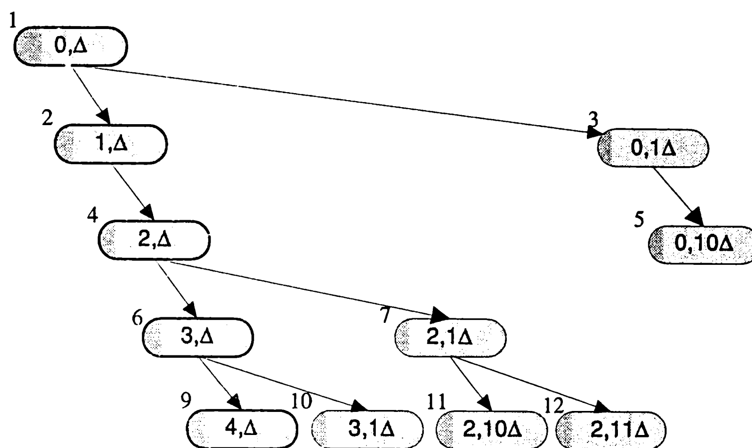


Figure 5.8: VTS in exponent representation

A complete exponential representation requires the mantissa to be coded using the length representation above. Using this representation, the left-most path from the root of the tree can grow to $(2^E + I)$ levels for a representation that uses an E bit exponent and where I is the length part of the mantissa. Moving to the right, the depth of the paths decrease until reaching the right-most path where the maximum depth is $I + 1$ levels. The proportions in which a VTS is divided into exponent and mantissa are the subject of optimization based on workload characteristics. Table 5.1 shows number of levels attainable using the exponential representation, compared with the length representation using an arbitrarily chosen exponent size. The VTS sizes span the range used in the simulations in Sections 5.3.4 and 5.4.3.

The exponential representation favors less speculative execution , since nodes that are early in the virtual sequence, on the left side of the tree, can have longer strings and so will not exhaust the maximum depth as often. Thus the number of rescale, and possibly cancelback, operations can be reduced. This has two additional advantages, first, by delaying execution

Total size (bits)	Length representation		Exponential representation		
	(I,L)	Number of levels	(E,I,L)	Number of levels	
				Max	Min
32	(27,5)	28	(16,12,4)	65548	12
64	(58,6)	59	(32,27,5)	$(2^{32}+27)$	27
96	(89,7)	90	(32,58,6)	$(2^{32}+58)$	58

Table 5.1: Number of levels which can be represented by different sizes of length and exponential representation VTS

of nodes to the right of the tree, which are more speculative, it helps balance the overall execution. Second, a compiler can take advantage of the representation by scheduling more computation to the left of the tree. Provided the compiler can schedule the critical parts of the execution to the left of the tree, execution can progress for much longer without needing to rescale using this representation. If the VTSs are exhausted by more speculative nodes, they can be stalled while the less speculative nodes continue executing. Eventually the VTSs will have to be rescaled, but the later rescale will free a larger number of VTSs and provide a more efficient reallocation because GVT has advanced further. The stalled nodes may even be rolled back before the rescale is performed.

Comparison of two VTSs is more complex for this representation. Generally a VTS with a larger exponent is earlier in the virtual sequence, but any VTS with only a zero length mantissa will be earlier than a VTS with a longer mantissa. Hardware implementations will have a greater ability to optimize this operation, but it will still be complex.

5.3.3 Ideal Representation

In order to show the restrictions placed upon execution by the VTS schemes we also simulate execution with VTSs of unbounded length, which are never exhausted and never require rescaling. Since there are always more VTSs available at any location of the tree the fossil collected VTSs never need to be reused. This is equivalent to having complete knowledge of the execution tree prior to execution and tagging each event in the virtual sequence with an integer corresponding to it's position. If perfect allocation is possible the largest of these integers corresponds to the number of VTS necessary to execute the program. The length in bits of this integer represents a minimum bound on the size of VTS required to execute the program without rescaling and can be calculated by $\lceil log_2 N \rceil$, where N blocks are executed.

5.3.4 Minimum Size VTSs

Simulations were performed to establish the minimum length required to avoid VTS exhaustion for each test algorithm. For the exponential representation an arbitrarily fixed 8 bit exponent was simulated, while the mantissa was kept to a minimum length. It isn't possible to measure the minimum necessary length of both the exponent and the mantissa, since the mantissa is used to record leading zeros once the exponent is exhausted. The total VTS lengths are compared, since bits could be taken from the exponent to use in the mantissa for the same storage and bandwidth requirements. Figures 5.9 to 5.11 and C.1 to C.3 show the length of VTS required to execute a variety of test programs without rescaling.

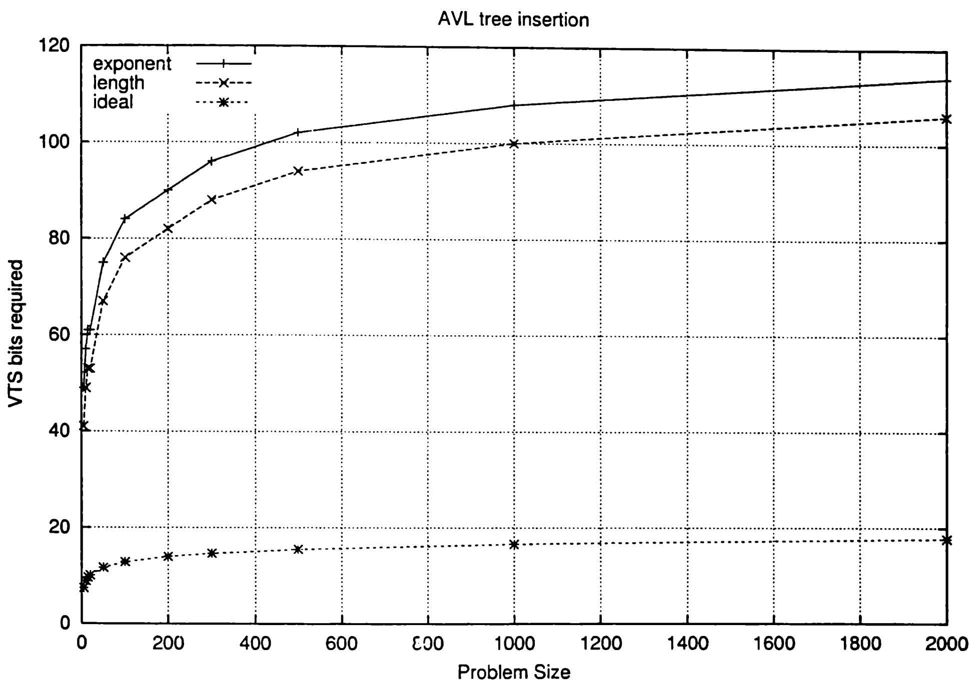


Figure 5.9: Minimum VTS length necessary to execute AVL tree insertion without rescaling

The length representation only approaches the theoretical minimum for very small problem sizes. As the problem sizes get larger they quickly diverge. The test programs used here are small by comparison with real world programs where the length representation necessary to execute without rescaling is likely to be even larger.

Fibonacci number generation (Figure 5.10) is the only algorithm which requires shorter VTSs using the exponential representation than the length representation. Even then it is only shorter by one bit in two of the problem sizes (5 and 10) and the same length in

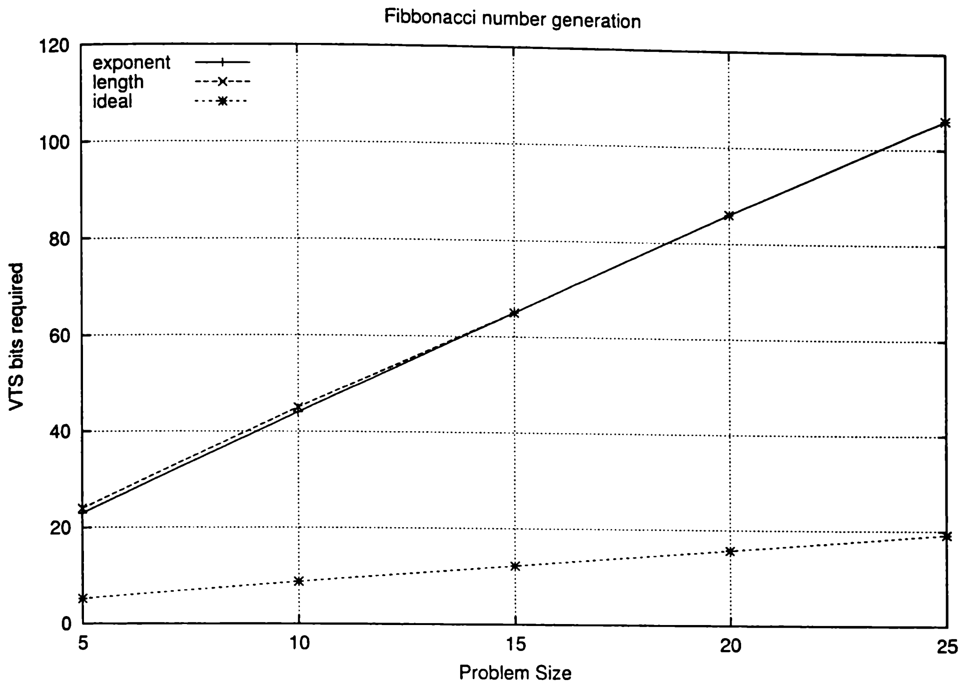


Figure 5.10: Minimum VTS length necessary to execute Fibonacci without rescaling

all others. This results from the very small and simple CFG, with only five nodes, and every non-leaf node is guaranteed to have a child in position zero. In all other cases the exponential scheme requires a VTS 8 bits longer than the length representation—the length of the exponent. The exponent has had no effect on reducing the length of VTS needed. Other exponent sizes show similar results. The minimum size exponent VTS is usually the minimum size of the length VTS plus the size of the exponent, independent of the exponent size.

The effectiveness of the exponent is reduced as soon as the longest path is not the left-most branch of the tree. The earlier it diverges from the left-most branch the less benefit is derived from the exponent. If the bits used for the exponent were transferred to the mantissa the whole tree would benefit uniformly with extra potential depth, albeit not as much as the left branch does with the exponent. However, this doesn't necessarily mean that the exponent provides no benefit. In Section 5.4.3 we compare the execution speedup obtained using the different representations.

The difference from the theoretical minimum size represents wasted VTSs. Any node with only two out of four children utilized leaves half of that subtree's VTS space permanently unused. Dynamic execution tree plots show that the execution trees tend to be sparse, with

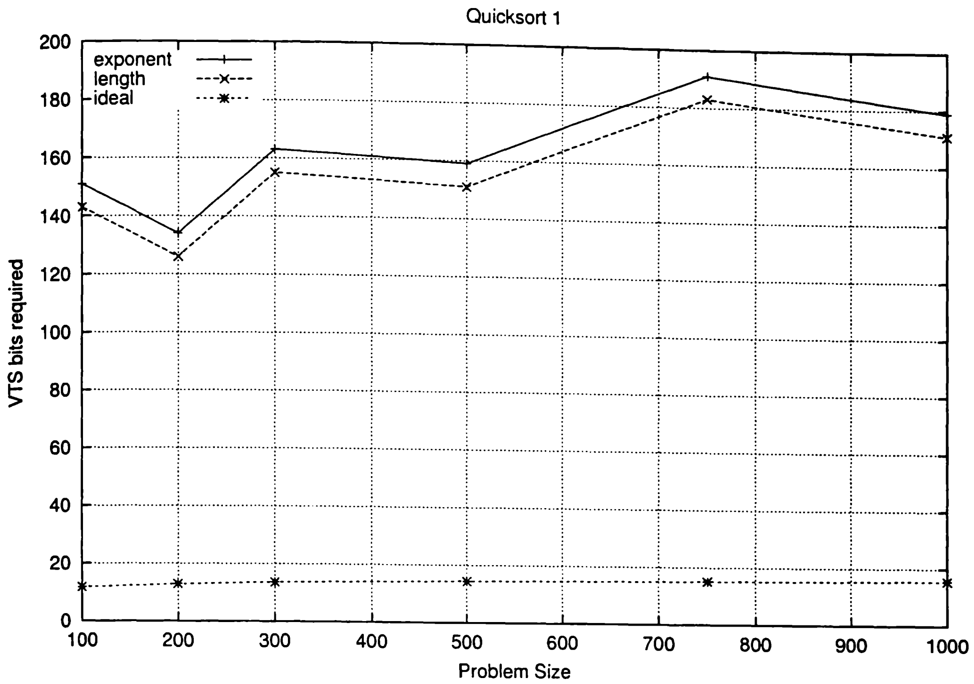


Figure 5.11: Minimum VTS length necessary to execute quicksort 1 without rescaling

a small number of long branches. The minimum VTS length required is determined by the longest branch.

5.4 Rescaling VTSs

In Section 5.1.1 we discussed the need for reusing fossil collected VTSs when the VTS space has been exhausted in some part of the tree. This process is known as rescaling the VTS tree. Implementing VTS rescaling requires consideration at two levels: global operations on the tree and local manipulation of individual VTSs. Major concerns for the global operation are the extent to which the local manipulation of VTSs can occur in parallel and whether computation can continue while rescaling is in progress. It is not only the VTS attached to the frame that must be altered in rescaling. Each time-space cache entry has a VTS associated with it which must be updated, and any memory accesses in-flight at the time of rescaling must still match with the appropriate time-space cache entries after rescaling. The time taken for local manipulation will have a major influence on total rescaling time, since it is anticipated there will be a large number of VTSs.

When the VTS space becomes exhausted the tree can be rescaled immediately, or it can

be delayed. While rescaling immediately allows the speculative execution at the exhausted point to continue as soon as possible, it may be disadvantageous in some situations. The exhaustion may be caused by a transient frame, that is one which is created speculatively, but is rolled back and ultimately does not commit. In this case the rescale is wasted and will delay other execution paths unnecessarily. In other cases a frame, particularly one which is speculatively distant, may eventually commit, but be off the critical execution path when it first exhausts the VTS space. This commonly happens with a frame which is speculatively executed, and rolled back and re-executed with new data. This frame may not be hindered by delaying rescaling since it is restarted anyway, and a more efficient rescale may be possible if GVT is allowed to progress further. These two cases suggest it would be useful to have a heuristic to delay rescaling when the exhaustion point is speculatively distant, which is also more likely to be a transient frame.

Virtual order simulation can select the optimal time to rescale the tree because it processes the frames in virtual order, and does not generate transient frames. Virtual order processing identifies exhaustion from frames early in the virtual order first and uses that as the trigger for rescaling. Exhaustion points later in virtual time will be delayed and benefit from the previous rescale, rescaling again if necessary. This gives optimistic results for rescales which will suffer under less efficient methods used in real order processing.

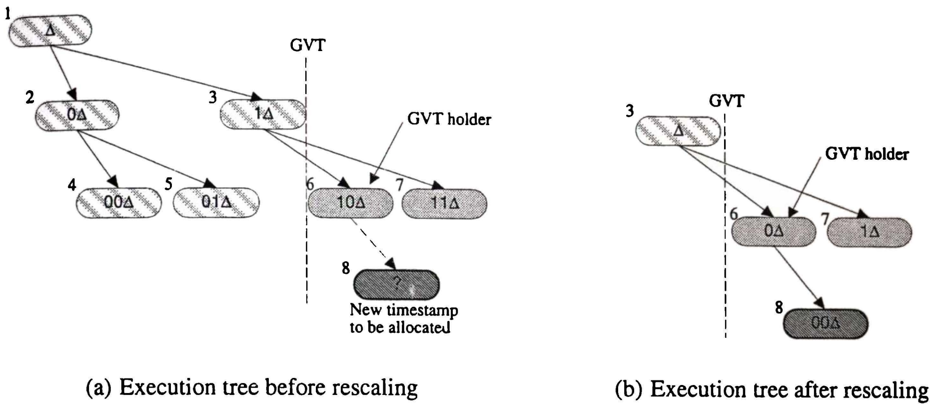
In this section we present some methods of rescaling the VTSs. Root rescaling is the simplest form of rescaling, performing the same operation on every active VTS, while moving head rescaling attempts to speed up rescaling by performing a single global operation to rescale all VTSs.

5.4.1 Root Rescaling

Root rescaling consists of finding a node whose subtree contains all the speculative and currently executing nodes, i.e. all those not fossil collected, making that node the root of the VTS tree, and adjusting the VTSs to reflect this.

Figure 5.12 shows an example of root rescaling using the symbolic VTS representation with at most three symbols (including the terminating Δ), when a new node (node 8) is to be added to the tree. In this example all nodes shaded with cross hatching have been fossil

collected, so their VTSs can be reclaimed for reuse. This can be achieved by backtracking up the tree from node 6, which is at GVT, until a node is found which has all currently active nodes in its subtree (node 3) and making this the new root of the tree. The VTSs for all the nodes remaining in the tree can then be reallocated using the original reallocation method.

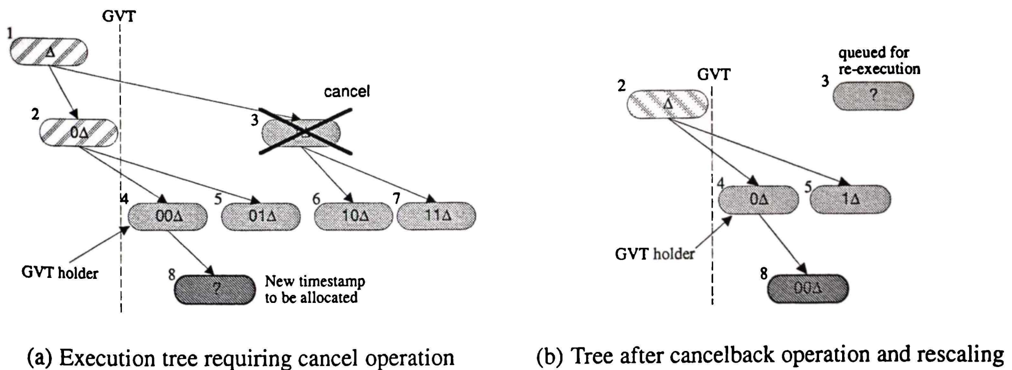


(a) Execution tree before rescaling

(b) Execution tree after rescaling

Figure 5.12: Root rescaling of an execution tree

If the root node is the only node which contains all the active nodes in its subtree the only way to release VTSs for reuse is to cancelback the right branch nodes at one or more levels and rescale the subtree containing the rest of the active nodes. For example in Figure 5.13 it is not possible to rescale the tree because the root node has a right child that does not contain GVT in its subtree. A solution is to cancel node 3, allowing rescaling to provide space to create node 8. For all finite representations investigated, rescaling with cancelback is sufficient to permit forward progress. If necessary GVT could be allowed to advance to the leaf of the tree while all other nodes are cancelled back.



(a) Execution tree requiring cancel operation

(b) Tree after cancelback operation and rescaling

Figure 5.13: Cancelback and root rescaling in an exhausted execution tree

The most problematic part of the rescale is identifying the rescale node, because the VTSs are not stored as a tree, and the frames (and associated VTSs) early in the virtual sequence

will already have been fossil collected. The appropriate time to calculate this is as part of GVT calculation, which must periodically inspect all the active VTSs in the system.

Once the rescale node has been identified each VTS in the system must be reallocated individually. VTSs may be rescaled in parallel, which will speed the operation, although it will still have to propagate throughout the system.

The local rescaling operation is straightforward for the length representation, with the two parts of the integer being operated on separately. The string part is left shifted and the length is decremented. Logic to rescale an individual VTS can be attached to each frame, and then it is a matter of propagating the number of levels to rescale to all frames. A similar operation is performed on every entry in the time-space cache to ensure the VTSs are kept synchronized with those in the frames.

For the exponential representation the global operation is the same, except that VTSs with non-zero exponents must first have the levels subtracted from the exponent. Any remaining rescaling levels after the exponent reaches zero are then removed from the mantissa in the same way as for the length representation.

Maximum level root rescaling

Maximum level rescaling reclaims the maximum number of levels possible without delaying execution. This means choosing as the rescale node the node which is the most levels down the execution tree, while still containing all the active nodes in its subtree. Only fossil collected nodes can be removed from the VTS tree during rescaling, so it is often possible to rescale further by waiting for GVT to advance. A frame to the right of the branch containing the exhausted VTS will prevent the rescaling node being selected any further down the tree without using cancelback. If the deepest possible rescale node is the root of the tree, then the node at the first level below the root on the path to the exhausted node is chosen as the rescale node and its right siblings are cancelled back. The rescale node is made the root of the tree and all the VTSs are adjusted accordingly.

N level root rescaling

If GVT progresses slowly maximum level rescaling will entail a large number of small distance rescales. This will be inefficient when rescaling costs are taken into account.

N level rescaling always reclaims a fixed *N* levels of the VTS tree, even if this involves delaying until GVT has advanced sufficiently, leaving levels of fossil collected nodes in the tree or canceling back nodes to the right. The hope is that this will allow a more efficient rescaling operation by fixing *N* to a constant, amortizing the cost of the rescale over several levels, and that the number of single level rescales can be reduced without unduly affecting performance. The optimal rescaling distance must be determined empirically. N level rescaling can also be used to force single level rescales, so that the minimum number of levels is rescaled that allows all nodes to be assigned VTSs.

When a rescale is required the simulator simply progresses N levels down the branch to the currently executing node and chooses that node as the rescale node, rescaling it to the root. This may result in a time penalty while the execution of the node at the exhaustion point is stalled until GVT reaches the rescale node. Any nodes to the right of the rescale node are cancelled back.

5.4.2 Moving Head Rescaling

A major drawback of the root rescaling scheme is that it requires every VTS in the tree to be examined and modified—a time consuming process, and liable to require a large amount of hardware to do in parallel. This is particularly problematic for VTS s attached to in-flight operations, and may require execution to be halted during root rescaling.

A technique to reduce the cost of rescaling is based on the observation that when root rescaling occurs in the length representation, the same bits are removed from the front of each VTS being rescaled. To take advantage of this a global index (the *head index*), which defines the start of all VTSs in the system, can be used. Rescaling only requires this single global variable to be altered. The bits in the VTS wrap around, so that no precision is lost when rescaling, as shown in Figure 5.14. Each VTS has its own *tail index* to terminate it because after the head has been moved no assumptions can be made about the value of the

unused bits. The tail index is an alternative way of representing the length indicator, both record the number of significant bits in the VTS.

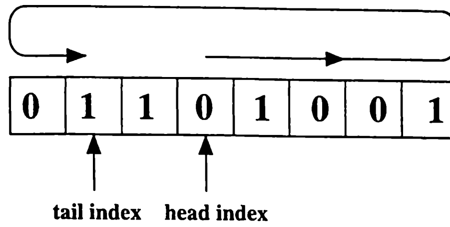
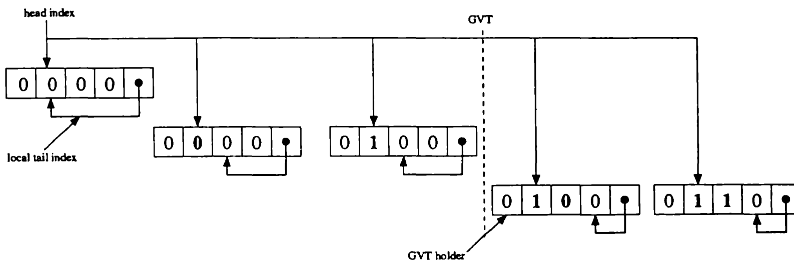
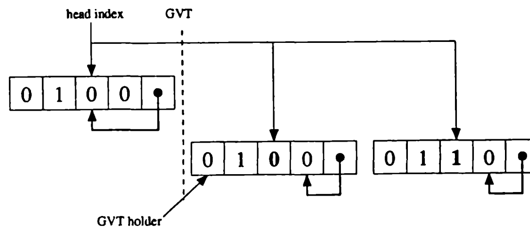


Figure 5.14: Moving head VTS representation

The local rescaling operation described for root rescaling is the same for moving head rescaling, except that it is only necessary to alter the single, global head index as shown in the example in Figure 5.15. Note that it is not necessary to perform any direct manipulation of the VTSs themselves and hence the global time taken to rescale is greatly reduced.



(a) Execution tree before rescaling



(b) Execution tree after rescaling

Figure 5.15: Rescaling the moving head representation

The main advantage of this representation is that the cost of rescaling has been significantly reduced as only the head index has to be altered. The moving head does require that the new value of the head be broadcast to all parts of the system but this can be done as part of other activities such as broadcasting values of GVT. Additionally, in-flight VTS s do not need to be individually located, since changing the head index will affect all VTS s equally.

5.4.3 Speedup

To compare the length and exponential VTS schemes four different configurations were simulated for each of the algorithms available. The configurations consisted of:

1. The length scheme with a base size of 32 bits.
2. The exponential scheme using the basic 32 bits for the mantissa and an additional 32 bits for the exponent.
3. The length scheme using 64 bits, the same total number as the exponential scheme above.
4. Ideal VTSs.

The difference between the first two configurations shows the additional parallelism extracted by adding an exponent to the length representation VTSs. The difference between the second and third configurations show the relative advantage of using the extra bits as an exponent, or increasing the bits available to the length representation. Ideal VTSs are included to show whether the VTS length is constraining the performance of the system.

Rescaling is assumed to take place instantaneously. This is an optimistic assumption designed to test the upper bound performance of the VTS scheme. The simulations in this section use the minimum amount of rescaling necessary for execution to complete with the stated VTS size and allow the earliest possible rescheduling of cancelled nodes. This corresponds to 1-level root rescaling, since rescaling is instantaneous there is no benefit in combining the cost of multiple rescales into one operation. Rescaling does delay execution when it is necessary to wait for GVT to progress to reclaim more VTSs.

Figures 5.16 to 5.18 and C.4 to C.6 shows graphs of simulated speedup over a range of problem size for each of the test programs using the four VTS configurations.

The simulator makes a number of optimistic assumptions, including zero latency rescaling, and unlimited bandwidth. Still, the speedup quickly diverges from the results for ideal VTSs and, in some cases, drops to levels comparable to current production architectures. Good speedup is achieved for some of the test programs, such as matrix multiply, Gauss-Jordan and Fibonacci, but these test programs are small and easily parallelizable.

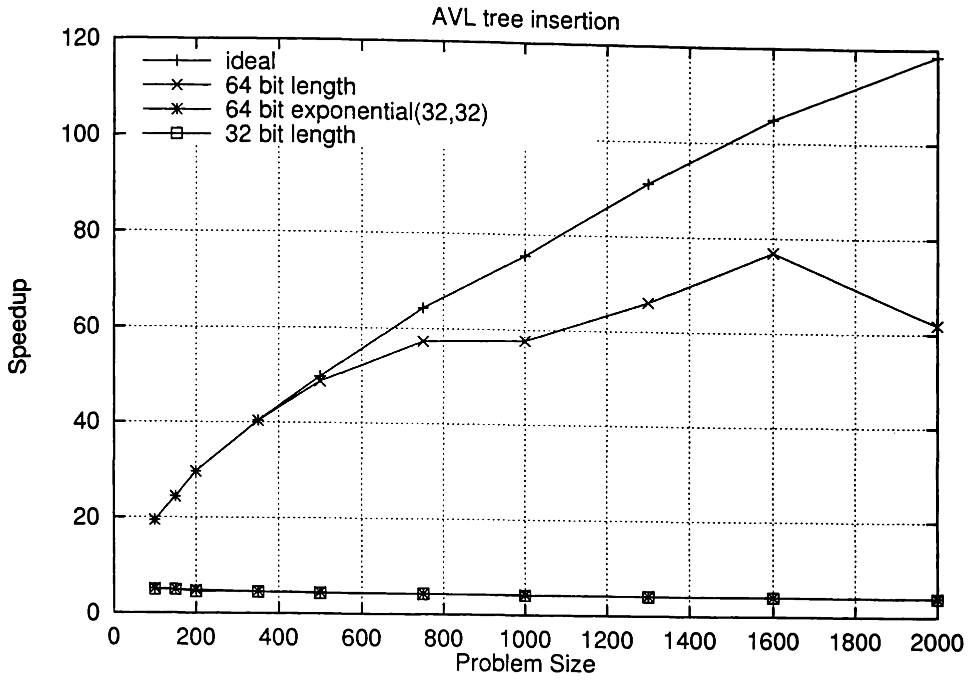


Figure 5.16: Comparison of speedup for AVL tree insertion with exponent and length VTSs

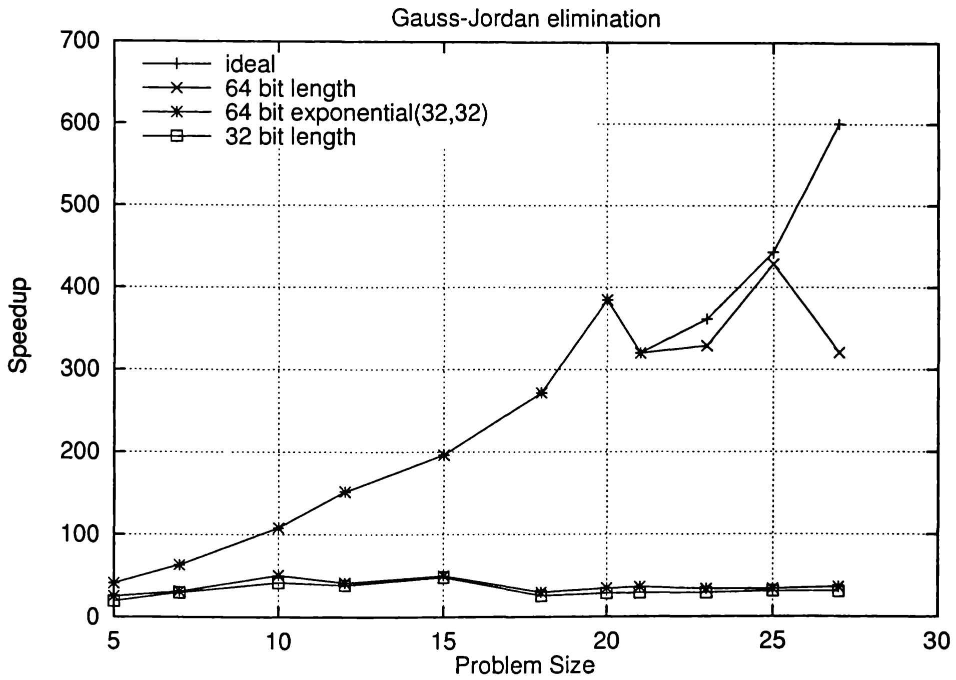


Figure 5.17: Comparison of speedup for Gauss-Jordan with exponent and length VTSs

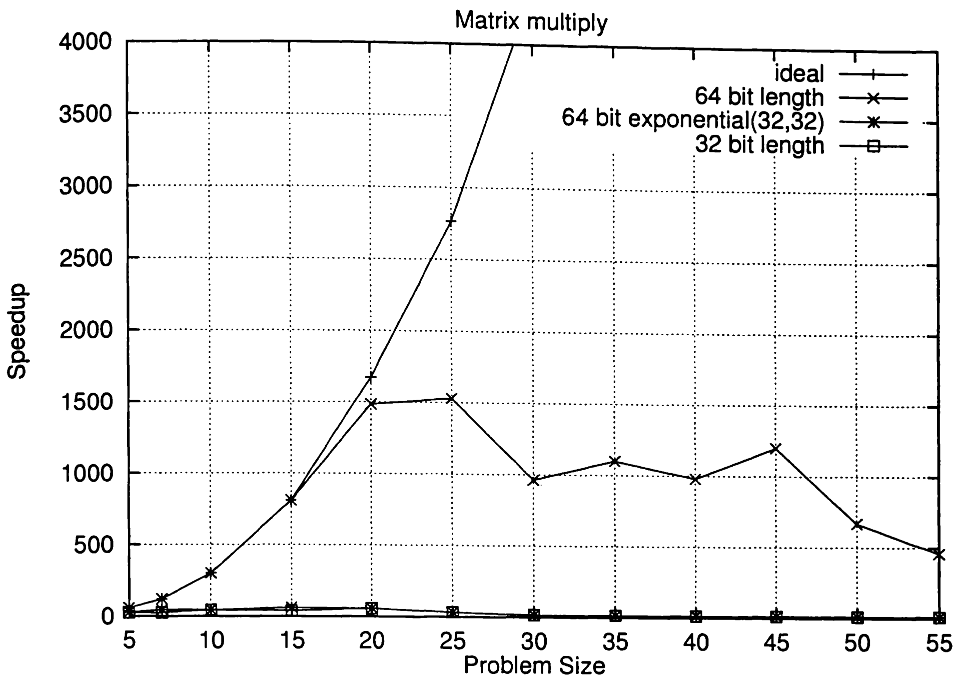


Figure 5.18: Comparison of speedup for matrix multiply with exponent and length VTSs

The 32 bit length VTSs show an unacceptable restriction in the speedup gained when compared with the ideal VTSs in all test programs. While the 64 bit length VTSs show significant improvement over the 32 bit and the exponential VTSs, only AVL and Gauss-Jordan remain close to the ideal for larger problem sizes, and even they begin to diverge. These test programs are small and it is likely that larger benchmarks would follow the trend shown by the larger problem sizes and diverge widely from the ideal.

Adding a 32 bit exponent to use the exponential scheme provides little gain over the 32 bit length representation. Despite the relatively large proportion of children (more than 60% in most cases) generated as the left-most child, there are few long chains of children on the left-most branch. Often the earliest events in the virtual sequence (and hence the furthest left) are initialization procedures, which are usually brief. Also, the top level of loop structures tend to have a high fan out in an effort to extract large amounts of parallelism. Thus, the exponent often cannot be used to replace leading zeros in the VTS, at least until rescaling is done to place the nodes on the left-most branch. The exponential representation gives inferior performance on all tested workloads compared with the same size length representation.

Using ideal VTSs the speedup generally increases with increasing problem size, as one would expect. With the other schemes, however, the speedup generally decreases as the

problem size gets larger. This is caused by increasingly long, thin branches in the execution tree of larger problems, which force delays until GVT can progress and allow fossil collection to release VTSs for rescaling to take place. This also forces more cancelback and the attendant delays.

The long thin sequential paths, which typically make up the execution tree, mean that the tree is sparsely populated and generally only a small proportion of the total number of VTSs have been used when the tree is exhausted. For a 32 bit length representation, there exist 8.95×10^7 unique VTSs without rescaling. A 100 item insertion into an AVL tree uses only 7380 VTSs, yet rescaling is done 325 times. A 20×20 matrix multiply uses 32819 VTSs, but requires 255 rescales and using quicksort to sort a 20 element list uses 7513 VTSs and 303 rescale operations. This gives an indication of how sparsely filled the tree is. When the tree layout is examined it can be seen that there are many long, thin sequential paths, causing many levels to be used up for a small number of blocks executed.

Practical rescaling

The speedup results when assuming ideal rescaling are sufficiently poor to rule out the naive length and exponential VTS representations as practical VTS schemes for the WarpEngine. For this reason simulations utilizing root and moving head rescaling with realistic latencies are not pursued for these representations. The already poor results would be further degraded.

The descriptions of the rescaling methods remain as examples of ways in which the functionality of rescaling could be implemented.

5.5 Summary

Two fixed length VTS representation schemes, length and exponent representation, were proposed to provide a distributed means of tracking the virtual order of frames. Each scheme must be able to rescale the active VTS tree to allow old VTSs no longer in use to be allocated to new frames.

The length representation is a naive scheme which represents the VTSs as integers, subdi-

viding the VTS space evenly amongst all tree locations, regardless of whether or not frames ever fill them. The exponential representation is an attempted optimization based on the observation that the majority of children appear in the leftmost position. By allowing a deeper chain of children in the leftmost path it was hoped that computation would have to halt for rescaling less often.

The minimum length VTS required to execute the test programs without rescaling was measured, and compared against the theoretical minimum length VTS. In every case, except Fibonacci, the exponential representation required a longer VTS than the length representation due to semantic constraints limiting the number of frames on the extreme left of the tree and the sensitivity of the representation to divergence from the desired pattern. The length representation required large VTSs, diverging widely from the theoretical minimum when the larger data sets for the test programs were used.

Two different rescaling methods were described: root rescaling and moving head rescaling. Root rescaling is a naive technique, simply shifting the redundant most significant bits off all the active VTSs. This requires modifying all active VTSs and can easily reach the situation at which it can reclaim no more VTSs, leading to a potentially expensive cancelback. Moving head rescaling centralizes the rescaling operation by using a global index to indicate the start of all VTSs in the system. Rescaling only requires incrementing the global head index.

Simulation results are only presented for root rescaling one level at a time. This represents the performance upper bound, since rescaling is assumed to take place instantaneously.

Adding an exponent to the length representation was shown to provide minimal performance increase, and it was always more advantageous to devote the increased VTS length to the basic length representation.

The speedup measured for the length representation VTSs was disappointing when compared to execution unconstrained by VTSs. When VTSs of a practical length are used they constrain the parallelism extracted unreasonably, to around the level of current production architectures. This is largely a result of the inefficiency of the allocation of VTSs. Many possible bit combinations are unused by the length representation. Many others remain unused during program execution because the shape of the execution tree does not match the

shape of the VTS tree.

The length representation scheme is inadequate as a means of tracking the virtual order in the WarpEngine because it forms a severe performance bottleneck.

Chapter 6

Variable Virtual Timestamp Ranges

6.1 Concept

In Chapter 5 it was shown that a large number of rescales are required to execute programs, even though the number of blocks in the program is several orders of magnitude less than the size of the VTS space. This suggests that there is scope for a dramatic decrease in the number of rescales if VTSs can be allocated more efficiently to frames.

During execution each subtree is allocated a range of VTSs. In the VTS schemes in Chapter 5 the range is fixed and implicit in the VTS. To allocate the VTSs more efficiently they can be allocated based on an estimation of the number that will be required by the subtree. This requires the ability to allocate a *variable range* of VTSs to a subtree.

The lower and upper bounds of the range need to be determined and recorded to allocate a variable range of VTSs to a subtree. As in the fixed range VTS schemes, the lower bound is also the VTS for the frame in the variable range VTS scheme. The upper bound of a fixed range VTS is implicit in the VTS, falling immediately below the next VTS at the same tree level. Figure 6.1 shows equivalent VTS trees using the symbolic representation VTSs from Section 5.2 and the same tree with the upper bound explicitly stated below the VTS. Any subtree which has upper and lower bounds the same cannot allocate any further VTSs to child frames. Attempting to do so signals that the VTS tree is exhausted at that point, and rescaling is required.

No length indicator is needed since the variable range VTS is simply a range of integers

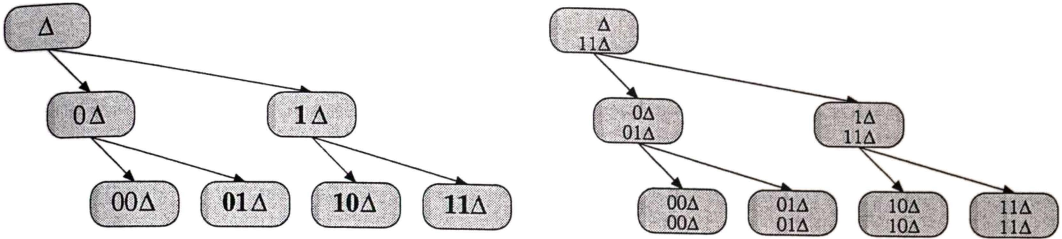


Figure 6.1: VTS trees with implicit and explicit upper bounds

represented by explicit bounds. This has the consequential advantage that, unlike length representation VTSs, all encodings form meaningful VTS, allowing more VTSs to be allocated using the same number of bits.

Explicitly stating the upper bound requires twice as much VTS storage space in the frame. However, in memory accesses and in the time-space cache only the VTS of the frame itself is needed (the lower bound of the subtree's range). Variable range VTSs will not increase the bandwidth or time-space cache storage requirements for the same size VTS. However, explicitly stating the upper bound of the VTS range to gain a more efficient allocation adds to the complexity of VTS construction and rescaling operations, as will be shown later in this chapter.

6.2 Execution Tree Size Calculation

Generally the size of a dynamic execution subtree cannot be determined statically, or even when the subtree is initiated at runtime, if it contains conditionally executed blocks or loops with a dynamic number of iterations. The key to allocating a variable range of VTSs efficiently is determining a tight upper bound on the number of frames, and hence VTSs, required in the subtrees.

If the range allocated proves to be insufficient to provide VTSs for the whole, subtree rescaling will have to be done to allocate a larger range (which may include reusing fossil collected VTSs). Since rescaling will impact performance, the WarpEngine allocates enough VTSs for the upper bound usage where possible, rather than allocating the number of VTSs most likely to be used and risking forcing rescaling. Wasting a small number of VTSs by allocating a larger than necessary range may force an early rescale because insufficient VTSs

remain for other subtrees. However, allocating too small a range will certainly cause VTS exhaustion within the current subtree and require a rescale. The tighter the upper bound can be made, the more efficient the allocation will be.

Cycles in the CFG (formed by loops) may make it impossible to determine the exact number of frames to be used. While an estimate of the number of VTSs required can be made, large loops may cause the upper bound to be exceeded, or there may not be enough VTSs in the representation, in which case the system must be able to reallocate a larger range by rescaling.

It is useful to instruct the processor to reserve a fixed number of VTSs for one subtree, and allocate all other available VTSs to another subtree. For example, in a loop such as the one shown in Figure 6.2 a subtree (iter) contains an iteration of the loop, while another subtree (cond) continues to the following iterations. If the loop is an inner loop the iteration will have an acyclic control flow, so the maximum number of frames used can be determined statically and used to allocate a fixed VTS range. All remaining VTSs in the range can be allocated to the remaining iterations, of which there may be an unknown number.

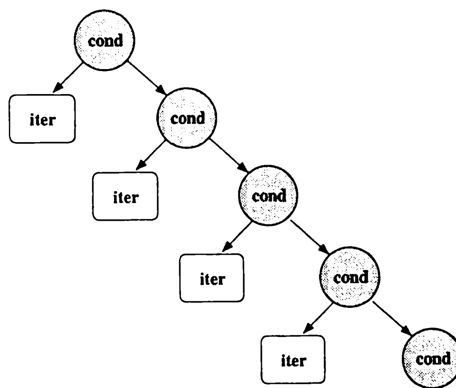


Figure 6.2: Execution tree for a dynamically bounded loop

In some cases compiler analysis of the tree size may suggest only that a very large number of frames are needed for a subtree. In this case anything more speculative than the subtree will probably be rolled back due to resource restrictions. Throttling speculation beyond this point will save the rollback overheads and reduce contention for execution resources.

6.2.1 Encoding

Predictions of subtree size in the WarpEngine can be expressed in two formats, either as a number of frames or as a proportion relative to the other subtrees initiated by the same frame, as shown in Figure 6.3. The number of VTSs allocated in a proportional split is calculated after reserving VTSs for subtrees with a fixed size estimate.

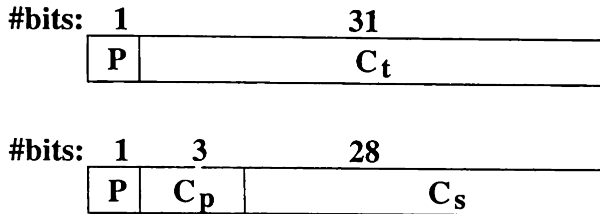


Figure 6.3: Encoding for subtree size calculations for a fixed number of VTSs (top) and proportion of available range (bottom)

Subtree size estimation is encoded in the child instruction initiating the frame at the root of the subtree. Whether the proportional or fixed range format is being used is indicated by the *proportional flag, P*, in the child instruction. The C_t field is the fixed number of VTSs to be allocated, while C_p is the proportion of the range to be allocated, and C_s id the fixed number of VTSs to be reserved before calculating the size of the proportional range.

C_p is encoded using 3 bits to allow proportions from $1/8$ to $8/8$ to be encoded. This allows an even split among two or four children, as well as the ability to give one frame a double allocation, while maximizing the number of bits available for C_s from the 32 bit total for the estimate.

Allocating zero VTSs should never be required, but can be encoded using C_t. The encoded subtree size estimate is passed to a child instruction as the second source register (a2) and can either be specified as a literal constant or calculated at run time.

6.2.2 Static Analysis

It is desirable for the compiler to perform subtree size analysis, rather than the processor, so that it doesn't impact execution time. The most basic static analysis is of linear, acyclic regions of the CFG, where blocks are counted to calculate the number of frames required. If the control flow terminates at the end of the linear sequence this count is also the number

of frames required to execute the subtree, otherwise it must be added to the frames required for the rest of the subtree.

It may not be possible to determine the exact number of frames required to execute an acyclic region of the CFG if it contains conditionally executed blocks. However, an upper bound can be established by totaling the size of all possible blocks executed. When a conditional branch is represented in a tree structured CFG there may be two mutually exclusive subtrees representing the alternative branch choices. Only the largest of the mutually exclusive subtrees needs to be added in determining the requirements for the subtree including them.

Figure 6.4 shows a sample acyclic code fragment containing `if...else` statements and the corresponding tree structured CFG. Each block of code represented by a letter is assumed to require one frame. The blocks are organized such that a left to right preorder traversal will generate the virtual order of the blocks and dotted lines represent conditionally chosen paths in the tree structured CFG. The upper bound on the number of frames required is annotated on the arc leading to each subtree. The dynamic execution tree is identical to the tree structured CFG in this case because there are no loops.

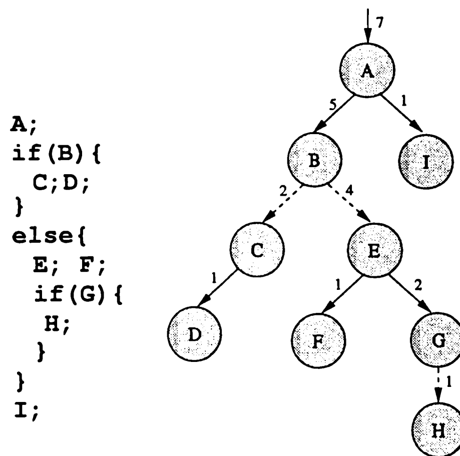


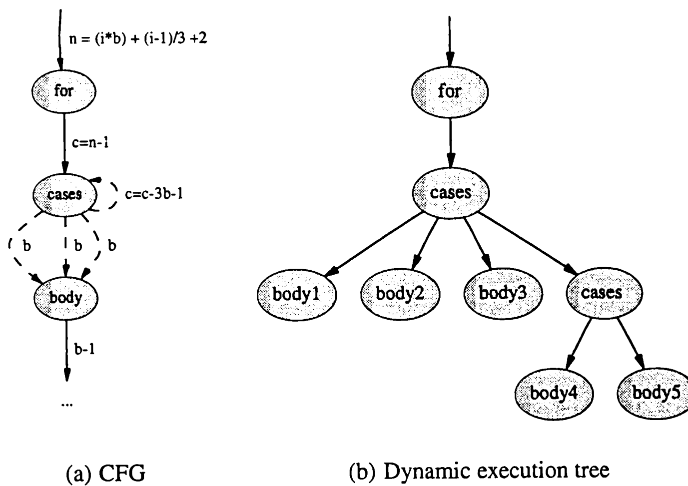
Figure 6.4: Calculating the number of frames required to execute an acyclic code region

Since C and E are mutually exclusive, the subtree rooted at B only needs one more VTS (for the node B) than the largest of the two subtrees. However, both F and G will execute if E does, so E requires the sum of F and G plus one for its own VTS. A range of 7 VTSs, or 3 bits, is sufficient to allocate a VTS to each frame. For comparison, the length representation is governed by the deepest potential path, 5 frames to node H, which requires 11 bits to

represent in the 4-way trees used in this implementation of the WarpEngine.

The size of statically bounded loops, such as most `for` loops, can also be calculated statically. The details of the calculation are dependent on the tree layout used for the `for` loop (see Littin[2000] for alternatives).

Figure 6.5 shows the tree structured CFG and the dynamic execution tree for a `for` loop using a simple linear backbone loop structure. The number of frames needed for the loop are shown on the entry arcs of each block in the CFG. The frame labeled 'for' initiates the `for` loop. The 'cases' frames form the linear backbone of the execution tree and decide whether to execute further instances of the body of the loop, labeled 'body'. Up to three 'body' nodes may be initiated by each 'cases' node until the desired number of iterations have been executed. Extra instances of the body may be speculatively issued, but will be rolled back.



(a) CFG (b) Dynamic execution tree

Figure 6.5: Calculating the number of frames required to execute a `for` loop

The total number of frames needed for the `for` loop can be calculated using integer operations by the compiler with the equation:

$$n = (i * b) + \frac{i-1}{3} + 2$$

where i is the number of iterations and b is the number of frames used by each instance of the body of the loop. The dynamic execution tree is shown for a 5 iteration loop, with a single block in the body of the loop, requiring 8 frames to execute the loop. One frame

is subtracted off this total for the subtree starting at the first *cases* frame, then four are subtracted for each successive instance of *cases* (one for the node itself and three times the loop body size for the loop iterations). This is shown in the two arc equations for the variable c.

The range can be allocated for each cases node using a proportional allocation of $8/8$ ($C_p = 7$), only reserving three VTSs ($C_s = 3$) for loop iterations. Note that the size of the tree is statically calculable for an arbitrarily complex loop body, as long as it is composed of statically analyzable subtrees. All the variables shown are replaced by constants at compile time. If constants are not available for the loop bounds the subtree cannot be analyzed statically.

6.2.3 Dynamic Analysis

A compiler cannot determine the number of frames required by dynamically bounded loops, such as `while` loops, recursion or `for` loops with non-constant bounds or data dependent increment operations. However, the size of some dynamically bounded loops can be determined at run time. For example, a `for` loop bounded by a calculated value can be treated in the same way as a statically bounded `for` loop once the loop bounding condition has been calculated. To use this information for VTS allocations the compiler inserts extra instructions that dynamically generate the subtree size estimate based on the bounding value of the loop. In the example shown in Figure 6.5 the equation on the arc leading into the *for* node would be calculated at runtime. Usually this requires only a small number of simple arithmetic operations, but can become complex if nested loops are involved. In such cases the extra instructions necessary to calculate the subtree size may cause the maximum block size to be exceeded and require a extra frame per iteration, which may have a significant impact on resource requirements and performance.

Unfortunately many `while` loops are still not amenable to this type of analysis by the compiler, requiring knowledge of the algorithm's operation and data to predict the number of iterations. For these cases a simple proportional division of the available VTS range is used in the simulations in this chapter, splitting the parent's VTS range evenly among the unanalyzable subtrees. However, this can result in a very small range of timestamps for the subtree nearest GVT, while more speculative subtrees, initiated near the root of the

execution tree, consume a large proportion of the range, and are quite likely to be ultimately rolled back.

6.3 Minimum Size VTSs

As in Chapter 5, for fixed range VTS schemes the minimum size variable range VTS required to avoid rescaling was determined from simulations. The results are presented in Figures 6.6 to 6.9, and C.7 and C.8 with the ideal representation and the length representation for comparison.

AVL (Figure 6.6), binary tree (Figure C.7) and Fibonacci (Figure 6.7) preclude easy analysis beyond the naive method of counting linear code segments and splitting the VTS range proportionally elsewhere because the `while` loops and recursion are dependent on data. Naive proportional division of the VTS range causes the VTS size to be significantly larger than the theoretical minimum because the subtrees are of uneven sizes. Fibonacci comes closer to the ideal VTS length than the other two algorithms because the two recursive subtrees at each level tend to be similar in size, so naive proportional splitting of the VTS range is more efficient.

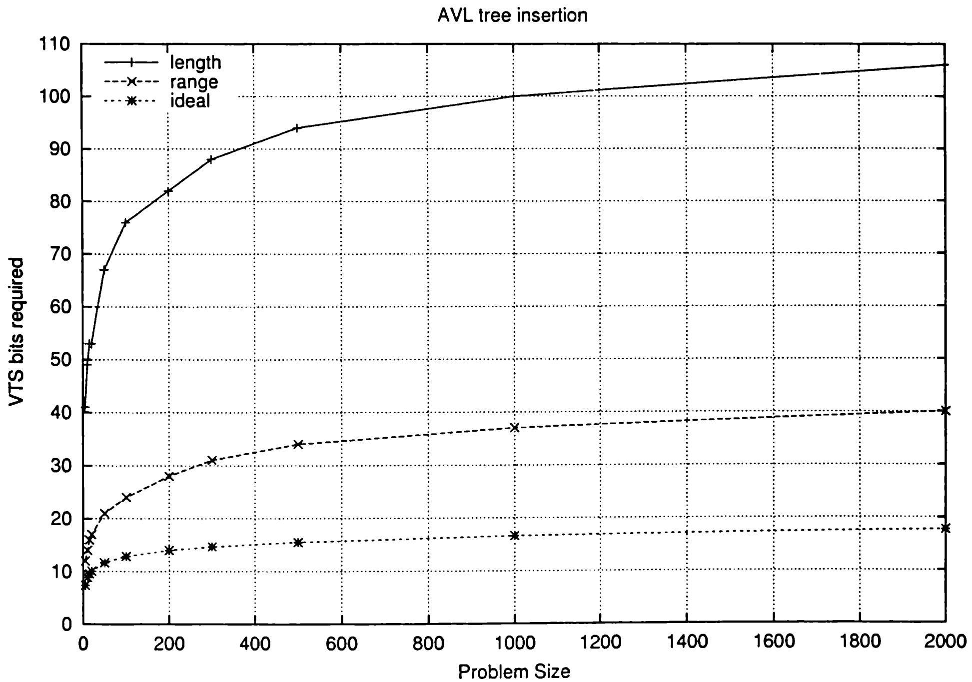


Figure 6.6: Minimum VTS length necessary to execute AVL tree insertion without rescaling

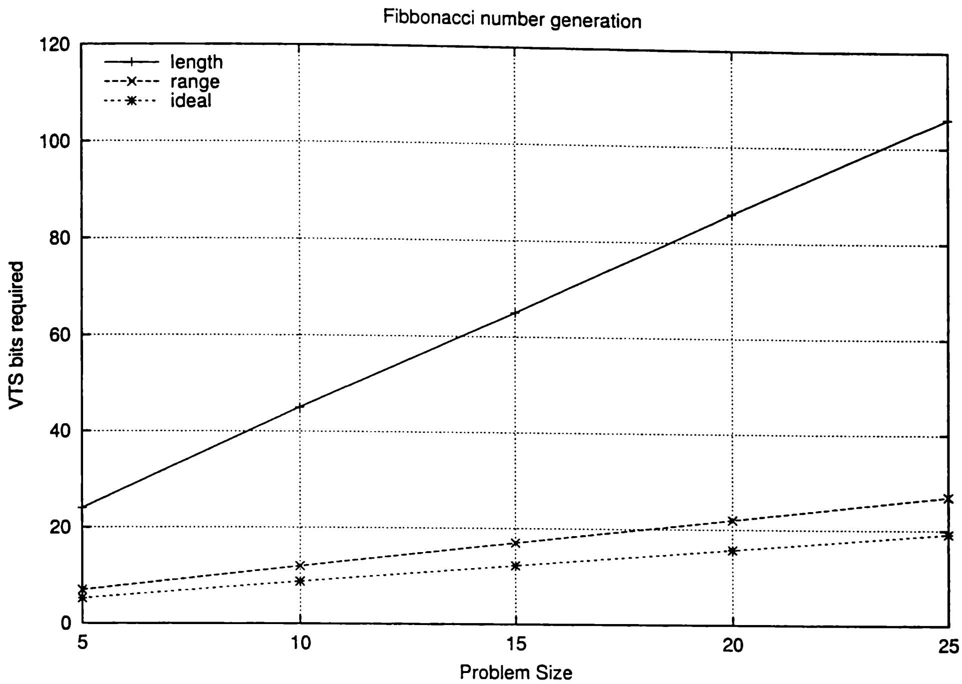


Figure 6.7: Minimum VTS length necessary to execute Fibonacci without rescaling

Gauss-Jordan (Figure 6.8) and matrix multiply (Figure C.8) consist primarily of `for` loops with static bounds, and the graph shows the speedup where iteration count is being used to calculate the number of VTSs required by the `for` loops, as well as the naive method. Naive allocation again requires a VTS substantially larger than the theoretical minimum because the subtrees are of uneven size. All of the subtrees that receive a proportional allocation of VTSs are `for` loops. Since the `for` loops are statically bounded, the VTS sizes for statically estimating the VTS requirements of the loops matches the theoretical minimum after rounding up to an integer number of bits.

Quicksort (Figure 6.9) also contains a `for` loop, however the bounds are dynamic, dependent on the way the list of items to be sorted is split. The static `for` loop estimation method assumes that the `for` loop must iterate over the original number of items, the worst case. The dynamic `for` loop analysis method calculates the maximum size of the `for` loop based on the size of the partitions created. This is a tighter upper bound than can be calculated by the compiler. A recursive loop also exists in which quicksort is called for the two partitions created from the larger partition. By nature an exact subtree size cannot be calculated for this tree split.

All of the above methods split the VTS range equally among the two procedure calls, but

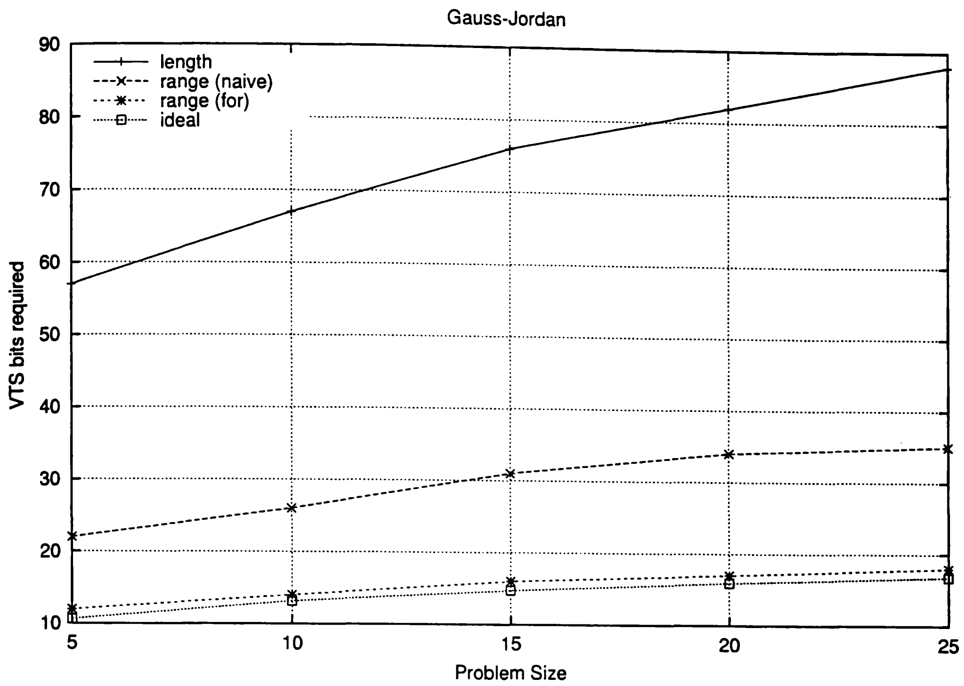


Figure 6.8: Minimum VTS length necessary to execute Gauss-Jordan without rescaling

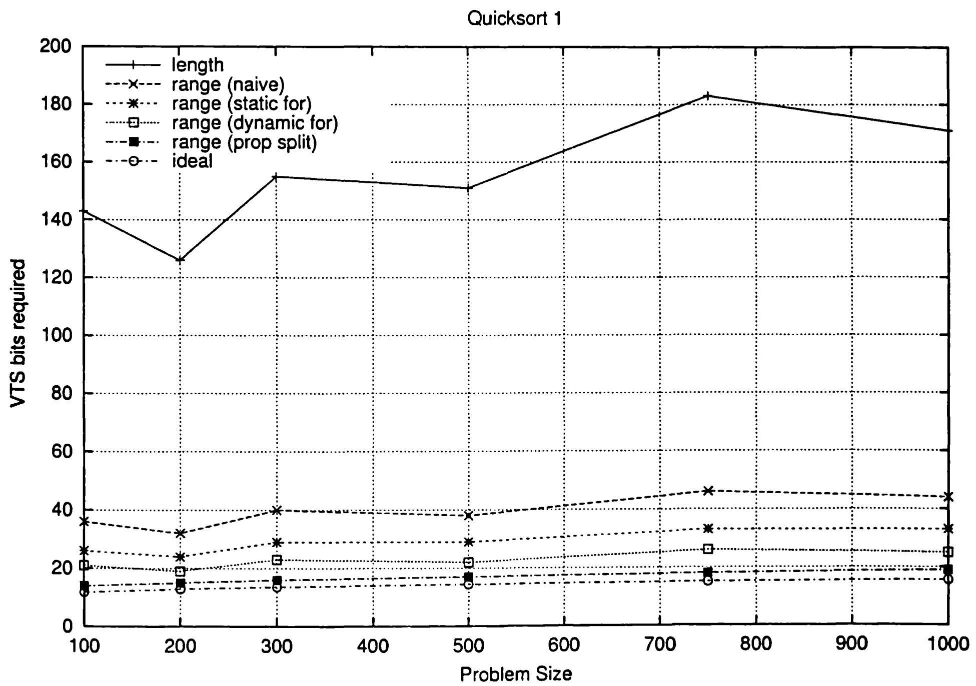


Figure 6.9: Minimum VTS length necessary to execute quicksort without rescaling

in the proportional split method the range is divided in proportion to the size of the two partitions. This is particularly beneficial where an uneven split is produced. It is questionable whether a compiler would be able to determine the relationship between the size of the partition and the VTS range required by the procedure without higher level knowledge of the algorithm. Using the proportional split method the theoretical minimum size VTS, rounded up to an integer, is sufficient to avoid rescaling.

In all cases the variable range VTSs are a great improvement on the fixed range VTSs. Using sufficiently aggressive analysis schemes all the test programs can be executed without rescaling using 40 bit VTSs, while the length representation requires in the region of 100 to 200 bits. The programs tested can be split into three broad categories:

1. Those whose tree size can be estimated well throughout the algorithm, such as matrix multiply and Gauss-Jordan elimination. VTSs can be allocated very efficiently to these algorithms using only static analysis. In the two examples shown VTSs of the theoretical minimum size are sufficient to avoid rescaling. This occurs in any program where the execution tree size is data independent and, thus, can be calculated statically.

2. Those which have few substantial subtrees whose size can be estimated easily. Examples of this are binary tree insertion and AVL tree insertion. It is difficult to allocate VTSs to these algorithms, even with dynamic analysis. The amount of computation is highly data dependent in complex ways, causing the number of bits required to be significantly higher than the theoretical minimum.

3. Those which have some large parts which can be estimated well and others which cannot be, or which have a simple relationship to data values, such as quicksort. More advanced dynamic analysis methods benefit this class of algorithm. These form an intermediate category, benefiting from variable range VTSs, but still diverging quite substantially from the theoretical minimum size when only basic analysis is used.

Real workloads are larger than the programs simulated here, and tend to be quite irregular. A typical application will consist of regions in several of the above categories. In regions which are difficult to analyze the the VTS size will limit the ILP more. However, since

these are regions where the control flow is more dependent on the data the chances of misspeculation are higher, so less aggressive speculation is appropriate.

Dynamic analysis methods require extra instructions to be added to the program to perform the calculations, sometimes forcing extra instruction blocks to be used when the frame instruction limit is exceeded. Figure 6.10 shows the percentage increase in parallel execution cycles for the different dynamic analysis methods over static analysis for quicksort. This measurement was taken with VTSs sufficiently long not to constrain execution to show the effect on execution of the additional instructions inserted for VTS allocation.

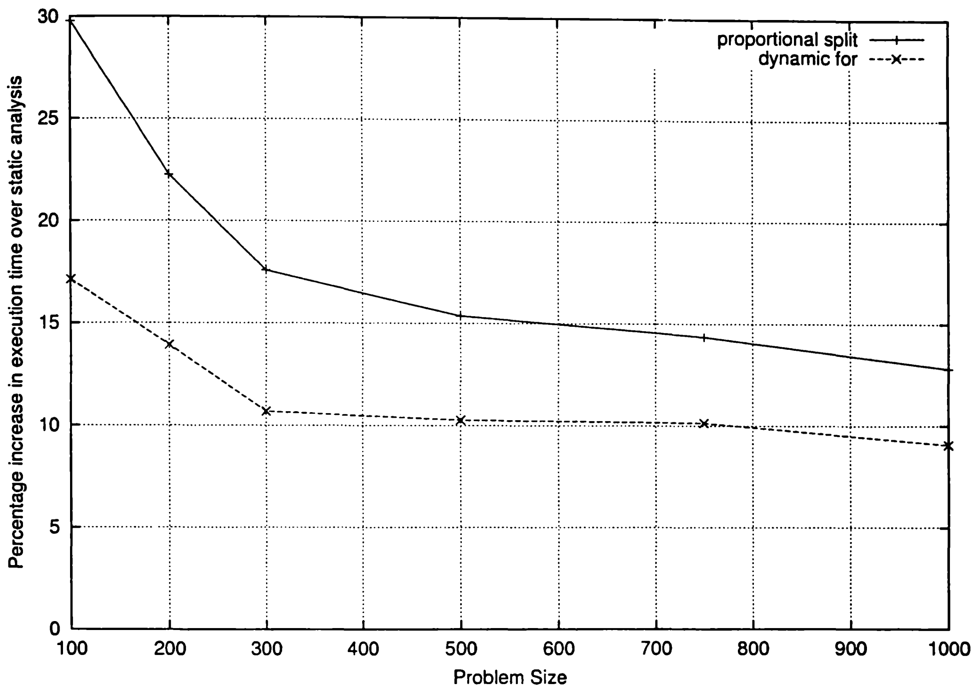


Figure 6.10: Percentage overhead of dynamic VTS analysis methods for quicksort1

The results show that both dynamic for loop and proportional split size calculations add a substantial number of additional instructions to the critical execution path. A careful evaluation for each program should be made before deciding to utilize dynamic allocation techniques. This decision can be made individually for each program at compile time.

6.4 Rescaling Variable Range VTSs

As with any VTS representation, the ability to rescale when the VTS tree is exhausted is necessary to guarantee completion of a program. In variable range schemes there is an

added complication that the explicit upper bound must be calculated, and it cannot overlap with any other subtrees (other than those which are a superset). This means that calculation of a VTS depends not only on the parent VTS, but also on the sibling VTSs earlier in the virtual order. Unfortunately this will extend the critical path of the global rescaling operation substantially. For example, child one must wait for child zero to be allocated a VTS range, rather than just their parent, before it can be allocated a VTS range. The dependency chain for allocating a VTS range to child three includes allocating a VTS range to children zero, one and two in addition to the parent.

There are two aspects which must be considered when evaluating a rescaling method. Firstly there is the computational cost of rescaling. This may include stalling computation while rescaling is performed, or stalling parts of the computation while GVT advances to allow rescaling, in addition to the actual time taken to rescale the VTSs. Secondly, the efficiency of reallocation of VTSs is important because, while a reallocation which wastes VTSs may reduce the computational cost, a careful reallocation will allow computation to continue for longer.

Rescaling variable range VTSs also introduces the added complication, and freedom, that VTS ranges may be arbitrarily changed. The rescaling methods for length representation VTSs are restricted by maintaining the frame relationship within the execution tree, and the VTSs that go with it. In this section we investigate two rescaling methods for variable VTS ranges. One which preserves the original tree, as for rescaling the length representation, and another which changes the shape of the VTS tree independently of the execution tree to attempt to achieve a more efficient VTS reallocation.

Rescaling must ensure that the exhausted subtree receives a larger range than before rescaling, or execution will remain stalled. Any subtree which is allocated a diminished range during rescaling risks not being able to allocate VTSs to all the frames that have already been started. This will force some frames to cancelback. The rescaling methods proposed here operate under the assumption that when a constant number of VTSs are requested for a subtree it is an absolute upper bound which will never be exceeded. VTS exhaustion can result from either a proportional allocation or when the requested number of VTSs cannot be allocated.

6.4.1 Preserved Tree Rescaling

A straightforward and efficient way of reallocating VTSs is to fossil collect nodes earlier than GVT and allocate the whole VTS range to the smallest tree containing all the active frames. The VTS range is then allocated to the active subtrees by regenerating the VTSs using the same process as the initial allocation. This has the advantage that the same hardware can be used for generating and rescaling VTSs, although some modification may enhance performance since rescaling must regenerate the whole tree at once. This rescaling method produces the same VTS tree as maximum level rescaling, proposed for the length representation in Section 5.4.1.

Not all the nodes prior to GVT can be removed though. In order to remain compatible with the original generation process a single connected tree must be maintained. This requires retaining nodes higher up the VTS tree which may otherwise be fossil collected, but connect the active subtrees into a single tree.

Figure 6.11(a) shows an exhausted tree with fossil collected frames represented by dotted outlines, the VTS range appearing under the frame label and the allocation information on the arc leading to the node. The connecting nodes C, E and F are retained under *preserved tree rescaling* to give the tree in Figure 6.11(b). Since the record of the VTS range and the tree shape are stored in the frame, either the frames for the connecting nodes cannot be fossil collected, or some other method of recording the tree shape and VTS range is needed. Strict adherence to the original generation method will allocate VTSs to fossil collected nodes which were originally descendants of nodes in the tree retained after rescaling, such as node D in the example, even though they are not required. In the method used here these nodes are fossil collected and their VTSs made available for reuse, preserving the minimal tree containing all active nodes.

Another drawback of preserved tree rescaling is that the proportional allocation will assign more of the reclaimed nodes to subtrees at the upper levels of the tree. For example, in Figure 6.11(b) 25 VTSs have been reclaimed, but only 4 extra are assigned to subtree H, while subtree M, which is not exhausted, receives an additional 12 VTSs. This will lead to delays while GVT advances to free sufficient VTSs for reallocation and may require the cancelback of frames.

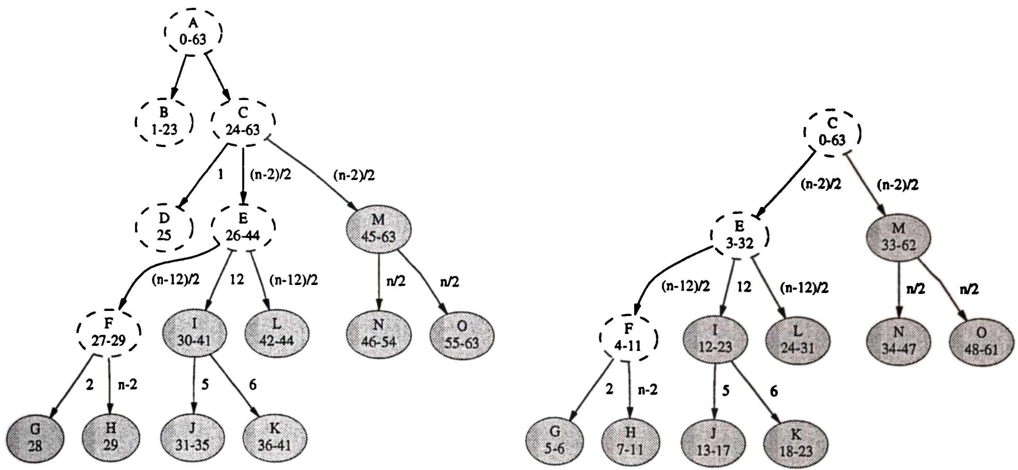
While this method achieves an efficient reallocation of VTSs, the global rescaling operation will have a long latency because potentially long tree branches have to be traversed. Calculating the new range depends both on the range allocated to the parent, since a sub-range is allocated, and on the upper bound of the preceding sibling since that determines the lower bound. This leads to a chain of dependencies running down each branch of the tree. Different subtrees can be rescaled in parallel once the root VTS of the subtree has been allocated. Since the alteration of each VTS is not systematic, updating VTSs in the time-space cache requires a global search to match each VTS.

6.4.2 Disconnected Subtree Rescaling

A minor variation on preserved tree rescaling is to consider each subtree of active nodes as a *disconnected tree* and rescale them separately, rather than preserving the original tree shape. This requires maintaining a record of the nodes at the root of each disconnected tree. However, no allocation information needs to be retained for the connecting nodes that have been fossil collected. It also requires the nodes at the head of each subtree to be allocated a VTS range before the rest of the VTSs can be regenerated.

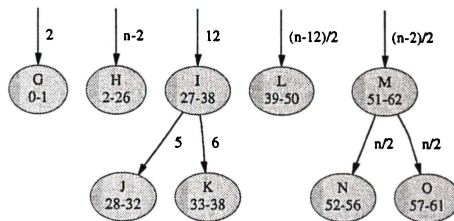
Each subtree is allocated a region of the overall VTS range in virtual order, based on the original allocation request. For a request for a constant VTS range the requested number is allocated. Subtrees requesting a proportional range are allocated VTSs from the remaining range in the ratio of the proportion requested against the proportion requested by all trees. For example, in the tree shown in Figure 6.11(c) 14 VTSs are reserved for the subtrees beginning at G and I, while the rest of the range is divided between H, L and M, with M receiving twice the allocation. Notice that M receives a smaller subrange than in the preserved tree rescale (Figure 6.11(b)), while all the other subtrees receive larger ranges.

Disconnecting the subtrees provides a more efficient reallocation because VTSs are not wasted on the intermediate frames which have already been fossil collected and only the more accurate allocation requests further down the tree are used. It also has the effect of placing all the subtrees at the same level, allocating a larger proportion to the trees further left, which are less speculative and will be deeper in the main tree. This allows more speculative execution early in the virtual order, at the expensive of throttling the speculation in the far speculative future. Since a rescale has been required, VTSs are clearly restricting



(a) Execution tree before rescaling

(b) Rescaled preserving the tree



(c) Rescaled using disconnected subtrees

Figure 6.11: Rescaling a variable range VTS tree

execution and it is likely that more rescales will be required. Proportionally reducing the allocation to more speculative subtrees tends to allow execution to progress further before the next rescale. The global rescale operation will also be faster, since removing the intermediate nodes will shorten the longest branches, and hence the critical path for rescaling.

6.4.3 Moving Head Rescaling

For a more efficient rescaling method, which doesn't require time consuming scans through the entire system, a variant of the moving head rescale from Section 5.4.2 can be used. The principle of the moving head rescale is that the most significant bit of a VTS becomes redundant when it is the same in all active VTSs in the system. In the variable range VTS scheme this means once the lower bound of GVT is past the halfway point in the range the most significant bit is redundant. By employing a global head index in the same way as in fixed range moving head rescaling, the VTS can start at an arbitrary bit position and wrap

around when the end is reached. In order to make the rescaled range as large as possible, a tail index (which can be used for both the upper and lower bounds) is kept for each VTS to indicate the end of the VTS at creation time. Any bits beyond the tail index up to the full length of the VTS are implicitly considered to be equal to zero in the lower bound and one in the upper bound. Each moving head rescale doubles the range of each subtree. Note that the range of subtrees with a constant size prediction also have their range doubled. These VTSs are wasted because a constant prediction is guaranteed to be an upper bound on the number of VTSs needed, so they will never be exhausted, providing the requested range was allocated. Only proportionally allocated subtrees will become exhausted.

Unfortunately moving head rescaling as described above will not provide more VTSs to an exhausted tree in the lower half of the range. By canceling back the frames with a VTS with a leading one, the moving head rescaling method can operate to double the range of the exhausted tree. However, since the tree is exhausted it has only one VTS in its range—its own VTS. Thus, only one VTS will be freed where it is needed, while wider ranges are made available to subtrees that don't require them, particularly those with a constant range allocation. For this reason it is important that moving head rescaling is done as early as possible. Fortunately processing doesn't have to be stalled for a moving head rescale. Generating a new VTS can be done with the new or old head index value, but a more efficient allocation is achieved if generation is done after rescaling because any constant ranges will not be doubled in size.

Additionally, cancelback discards a large amount of speculative work. Almost all the speculative events will be cancelled back if the exhausted VTS is near, but just below, the halfway point.

There is also the issue of when to restart cancelled frames. Under fixed range VTSs the strategy is to reallocate to a cancelled frame a VTS on the right edge of the tree which is never used by a frame. With variable range VTSs such a VTS never exists because allocation is done more efficiently to avoid these situations. Some heuristic must be used to decide when a sufficiently large VTS range has been freed to allow the cancelled frames to be allocated new VTSs. VTSs allocated to cancelled frames must be later in the virtual order than any currently active frame because frames are cancelled most speculative first, and restarted in last in, first out order. This requires the VTSs to be taken from the range allocated to the most speculative subtree. A simpler alternative is to wait until there are no

active speculative events, i.e. they have all completed or become non-speculative, before restarting cancelled frames. At this stage the whole VTS range is available. This effectively halts speculation at the cancelled point and forces speculation to begin again from that point.

Variable range moving head rescaling is not pursued further in this thesis. Like the fixed range moving head rescaling method, it relies on being able to complete rescaling more quickly than the other methods described. Low level implementation details are required to accurately establish this timing information, and is beyond the scope of this thesis.

6.5 Speedup

6.5.1 VTS Constrained

Figures 6.12 to 6.14 and C.9 to C.11 show a comparison of the simulated speedup delivered by 12, 16 or 24 bit variable range VTSs using either preserved or disconnected tree rescaling methods. VTSs larger than 24 bits are not shown because they are either sufficient to avoid rescaling in the test programs, or require very few rescales and do not adequately exercise the proposed techniques. For the same reason 12 bit VTSs have been used instead of 24 bit VTSs for Gauss-Jordan and matrix multiply. Instantaneous rescaling has again been assumed for both rescaling methods to provide an upper bound on performance. The only delays caused by rescaling come from waiting for GVT to advance sufficiently to perform a rescale.

The simulations show a substantial advantage for disconnected over preserved tree rescaling in all simulations for the same size VTSs, justifying the extra hardware required for the scheme. The advantage is particularly clear for AVL tree, (Figure 6.12) binary tree (Figure C.9) and quicksort (Figure 6.14). In Fibonacci (Figure 6.13) and matrix multiply (Figure C.10) the maximum speedup is achieved for the smaller problem sizes because no rescaling is required during execution. Speedup quickly drops away from the ideal once rescaling is required because the number of blocks executed grows exponentially with problem size for these algorithms, quickly requiring a large number of rescales.

As with frame restricted execution, in some cases the speedup counter-intuitively decreases with increasing problem size. This is caused by increased restriction on short range parallel

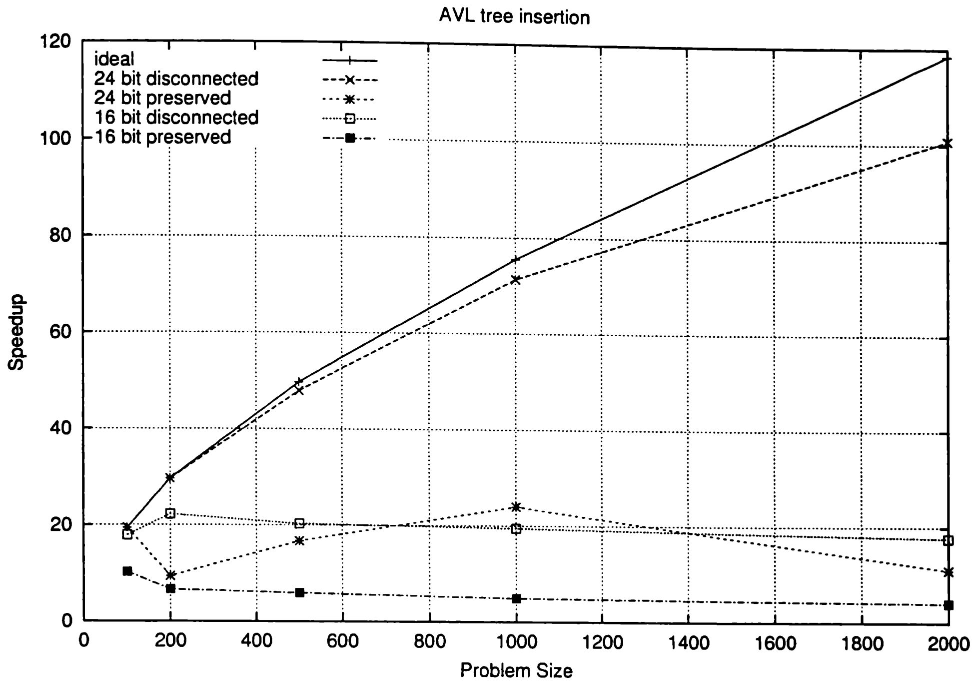


Figure 6.12: Speedup for AVL tree with variable range VTSs with rescaling

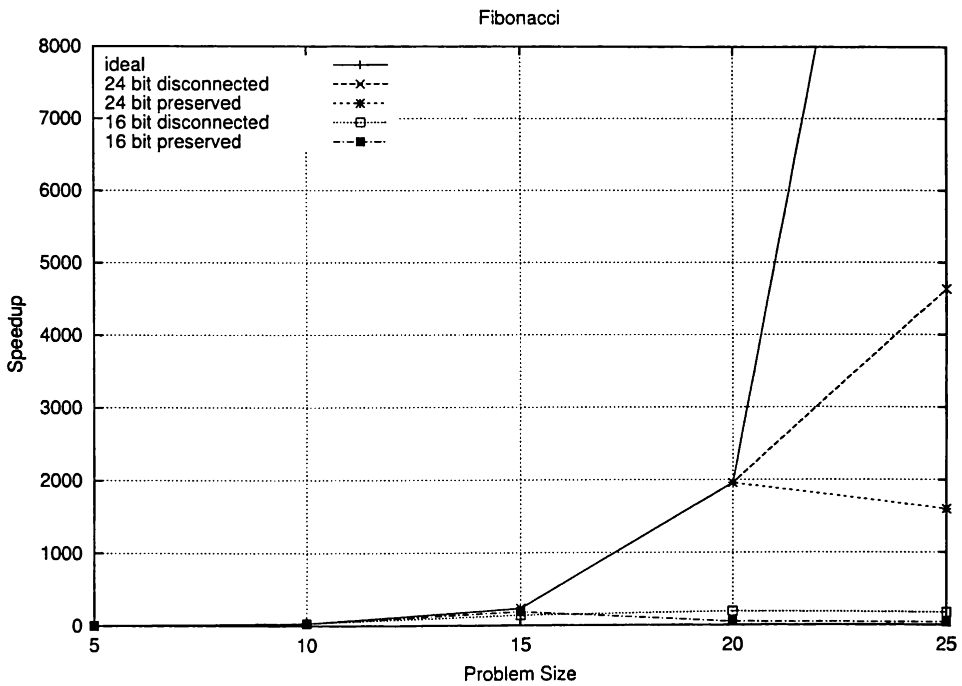


Figure 6.13: Speedup for Fibonacci with variable range VTSs with rescaling

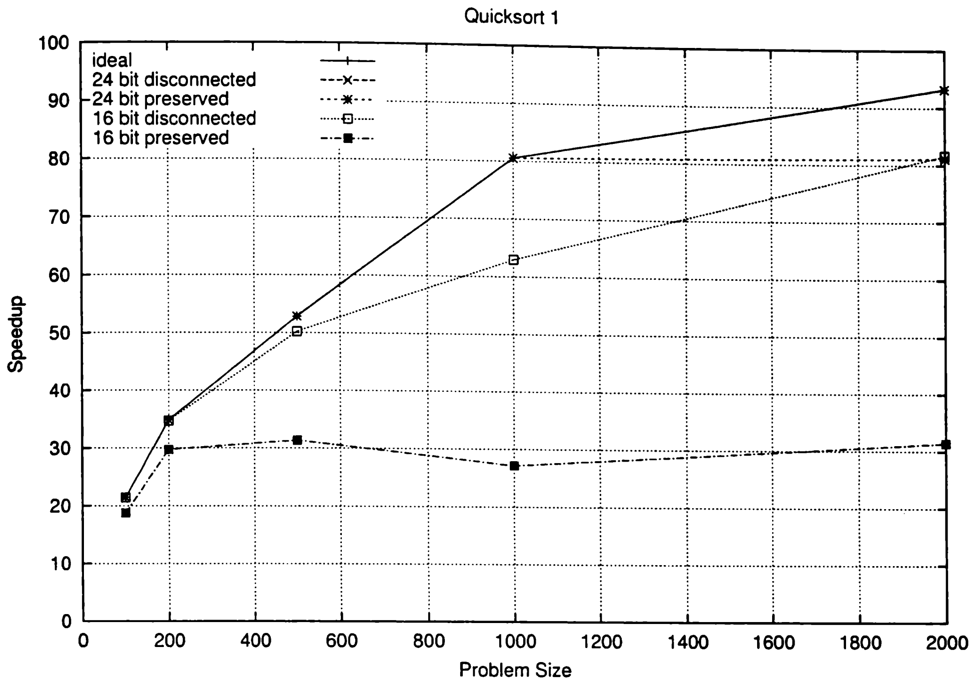


Figure 6.14: Speedup for quicksort1 with variable range VTSs with rescaling

execution due to a large proportion of the VTSs being reserved for more speculative events. This forces rescaling, which may require a delay while GVT advances. This is a greater problem for preserved tree rescaling than the disconnected tree method because a larger proportion of the VTS space is assigned to the more speculative subtrees. The short range speculation is more likely to be on the critical path and less likely to be mis-speculated, and is therefore more valuable. While there is benefit in ensuring speculation at a distance can continue, there is a cost if sufficient VTS space is not provided for shorter range speculation. While this works reasonably well for larger VTSs, when smaller VTSs are used, too much of the range is devoted to distant speculation. It is important to match the VTS size to the program.

6.5.2 Frame Limit Sensitivity

Of course, any implementation of the WarpEngine will not be limited by VTS size alone. Another important limit is frame availability, which is closely tied to VTS usage, since a VTS is assigned to each instance of frame usage. Frames are used for state saving, and in this respect are equivalent to the ROB in a superscalar processor. Chip size will limit the number of frames, but the capacity is expected to be much larger than contemporary ROBs

in order to support aggressive speculation. Calvert [1997] proposed an implementation for a WarpEngine frame, and calculated that sixteen frames could be implemented on a chip equivalent to the PowerPC 604, state of the art process technology at the time. Since there are up to sixteen instructions in each frame, this corresponds to an ROB of 256 entries, well in excess of commercial ROB sizes at the time of writing [Intel, 2000]. Littin [2000] performed simulations with constraints on the number of frames and showed that increasing the number of frames provides a sub-linear speedup. He estimated that five hundred to a thousand frames could reasonably be provided in the near future.

To examine the effects of limiting both the VTS size and the number of frames, simulations were performed using 16 bit variable range VTSs with disconnected tree rescaling and a range of frame limits up to 10,000 frames. Disconnected tree rescaling is more effective than preserved tree rescaling with no serious disadvantages and 16 bit VTSs were selected to impose moderate constraints on speculation.

Frames speculatively executed and rolled back are not considered in these virtual order simulations, only frames which commit are tracked. This means that the frame usage figures are optimistic and transient frames will consume frames above the limit imposed. However, VTSs are still reserved for transient frames because an upper bound is calculated, which reserves enough VTSs for the worst case (maximum) requirement.

Figures 6.15 to 6.17 and C.12 to C.16 show the speedup obtained from the simulations for selected data set sizes for the test suite. Each program was simulated for different frame limits with limited and ideal VTSs. Where the curves are coincident the VTSs are not constraining execution, it is solely due to the frame limitation. When the curve becomes flat and increasing the available frames does not increase the speedup, the frame limitation is not constraining execution, and any divergence from unconstrained execution is then due to VTSs alone.

The slowdown due to VTSs and frame limitation are not generally cumulative. The speedup is dictated by either the frame or VTS limitation, whichever is lower. Only when both speedup limitations are similar do both contribute to restricting parallelism. This suggests that both limits cause restriction of parallelism in the same regions of the programs.

In AVL and binary tree insertion (Figures 6.15, C.12 and C.13) the pairs of curves begin to

diverge at around 1000 frames, up to that point frame limits are more restrictive than VTSs. The larger problem sizes, as expected, benefit more from generous frame limits, and are more restricted by the 16 bit VTSs. Lengthening the VTS pushes the curves closer together. The curves for AVL are further apart because AVL VTS usage is more dependent on data. The smaller sizes (100 and 200 items) never suffer much slowdown due to the VTSs.

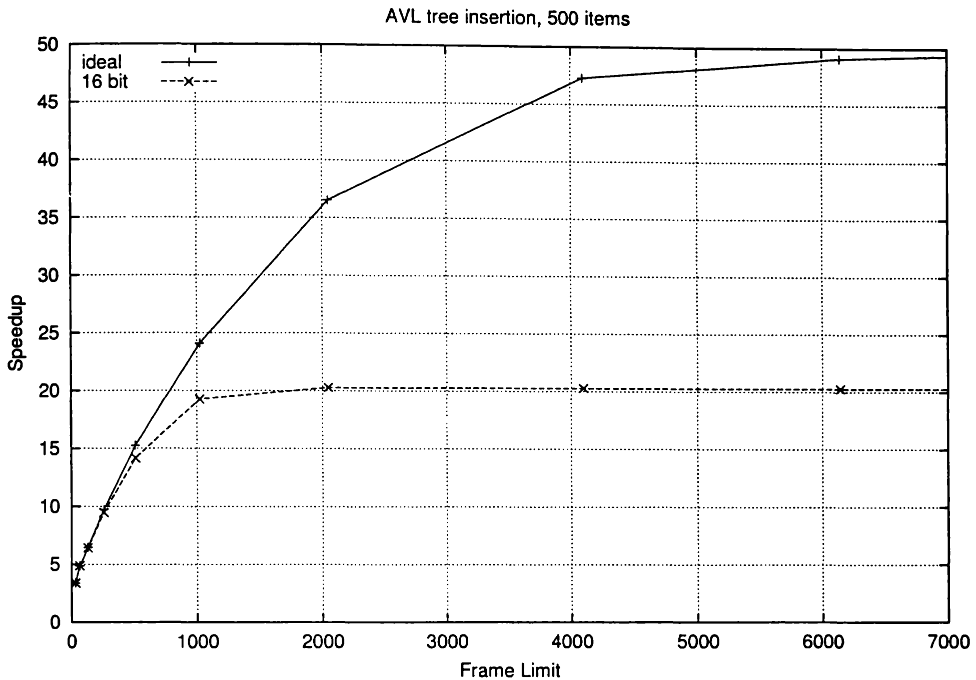


Figure 6.15: Speedup for AVL(500) with varying frame limitations and variable range VTSs

Quicksort, Gauss-Jordan and matrix multiply (Figures 6.16, C.14, C.15 and C.16) are largely unaffected by the VTSs. Curves are coincident, or very close to it for all problem sizes and all frame limitations. These algorithms have tight VTS upper bounds applied because the frame usage can be well analyzed, causing frames to be the limiting factor. 16 bit VTSs provide 65536 unique VTSs, so in cases of efficient allocation the frame limits shown here are exhausted first, even allowing for the more flexible reuse policy for frames.

Fibonacci number generation (Figure 6.17) has vast amounts of parallelism available for the larger problem sizes and the curves for 15 and 20 numbers with ideal VTSs do not begin to flatten out until large numbers of frames are made available. The 16 bit VTS only becomes the limiting factor when more than 4000 frames are provided for a data size of 15 and 2000 frames for a data size of 20.

A relatively small 16 bit VTS generally allows as much speculative parallelism to be ex-

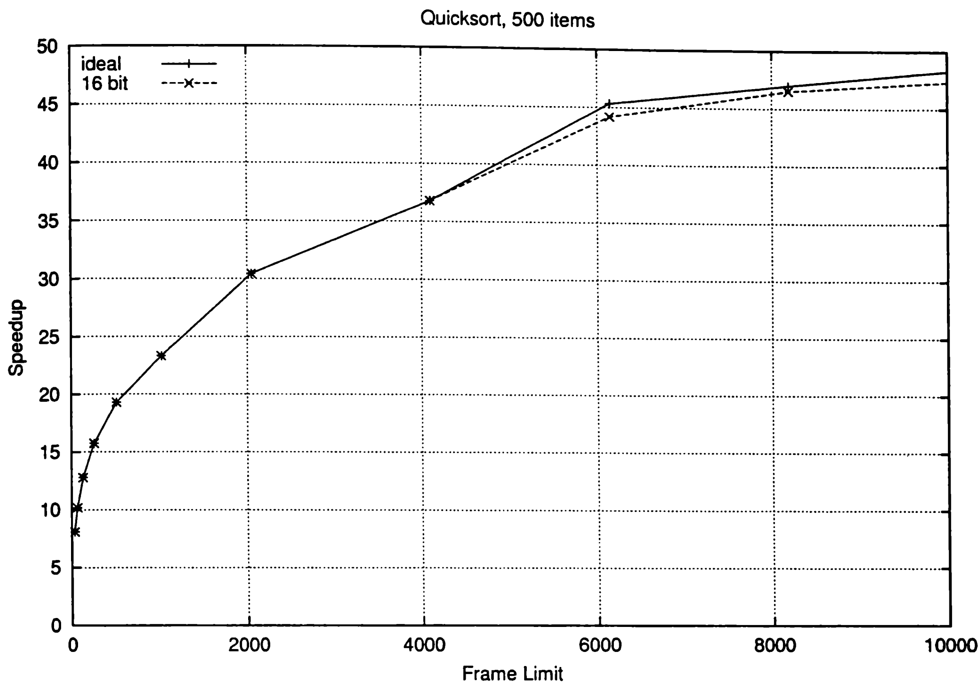


Figure 6.16: Speedup for quicksort1 (500) with varying frame limitations and variable range VTSs

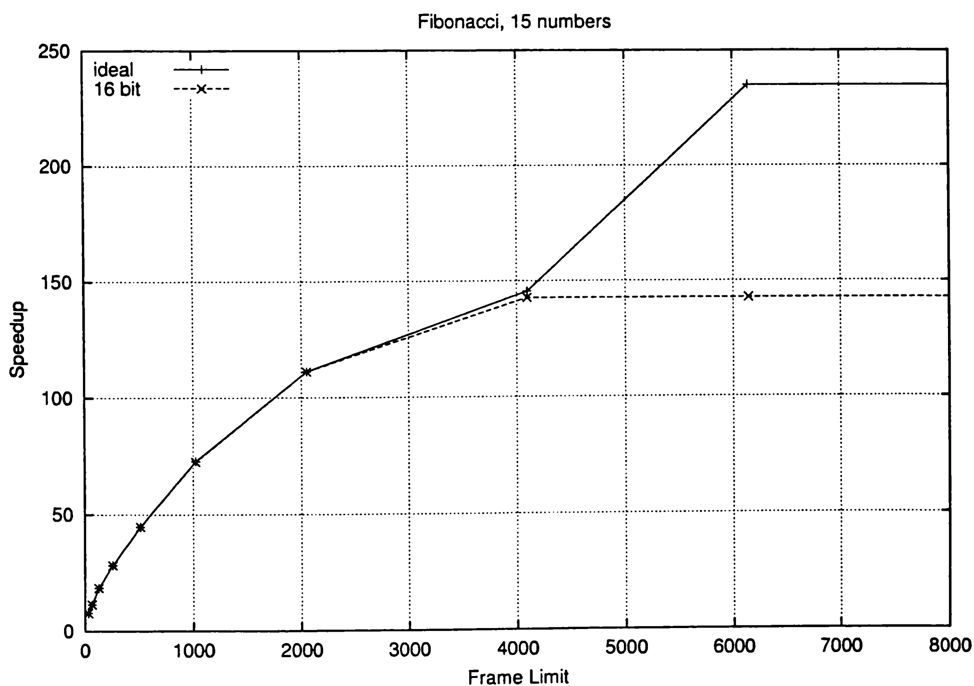


Figure 6.17: Speedup for Fibonacci (15) with varying frame limitations and variable range VTSs

tracted as up to a thousand frames can support, the limit on what will be available in the near future.

6.6 Unresolved Issues

The investigations of this chapter have focused on fundamental constraints placed on the upper bound of performance by variable range VTSs. A number of details remain to be resolved, some may improve performance, while others will degrade performance.

The largest unresolved issue is compiler interaction. Some basic analysis techniques have been discussed in this chapter and applied by hand to the assembly language programs used with the WarpEngine simulator. Beyond simple `for` loops and linear sections of code the analysis is quite primitive, the VTS range is simply split evenly amongst the possible subtrees. It is likely that a compiler could improve significantly on these techniques. Reorganizing code by moving block boundaries and modifying the loop structures could significantly affect the ease of analyzing VTS requirements.

Introducing a prediction of the most likely size of subtrees, rather than the maximum size, would provide scope for much more accurate assignment of VTSs. This would require substantial modification to the VTS allocation mechanisms, particularly rescaling methods to allow subtrees to exceed the predicted size. Particular subtrees that are often small, but are large in rare cases (for example a rarely matched `if...else` condition) would be particularly aided by such an estimation. Traditional branch prediction techniques using execution profiling could be adapted to this task.

The availability of a compiler would also allow larger and more diverse benchmarks to be simulated. This is important for testing the scalability of the VTS schemes proposed as the different workload characteristics could make a substantial difference to the effectiveness of the VTSs. Further issues in the development of a compiler are beyond the scope of this thesis.

The latency of rescaling has not been considered in detail here, but will clearly be important in determining the overall execution time. It will form a tradeoff between schemes which require rescaling to be performed more often at a lower latency and those with longer latency,

but which reallocate VTSs more efficiently, and hence less frequently. Rescaling latency can only be established with a low level hardware design, which is beyond the scope of this thesis.

6.7 Summary

Analysis of the code in a tree structured execution model, either statically or dynamically, can reveal much about the amount of work and resources required for each control independent section of the program. In this chapter the focus has been on allocating VTSs efficiently, but the same analysis can be applied to other execution resources.

After determining the maximum number of blocks in the subtree, a VTS range guaranteed to be large enough to order the whole subtree can often be allocated. In cases where such analysis is not possible the simulations presented simply divided the available range among the subtrees. More detailed analysis schemes could be devised, in particular, using a prediction of the number of VTSs likely to be used (rather than the upper bound). Such schemes remain for future development. Even with simple analysis techniques the length of VTS required to execute the test programs without rescaling was much less than for fixed range VTSs, and in some cases approached the theoretical minimum.

Two rescaling methods, necessary to guarantee execution can complete, were modelled. Results showed that rescaling does not constrain execution unduly, even when short VTSs were assumed. Rescaling was modelled as an instantaneous operation to investigate any fundamental limits involved. Rescaling latency will be important to overall performance for workloads that require frequent rescaling, but can only be established with more detailed models of the hardware.

Simulations also showed that the speedup tends to be limited by either the number of VTSs or by the number of frames available, not by both cumulatively. 16 bit VTSs constrain execution to a similar degree to 1000 to 2000 frames in most cases, the limit of what could be implemented on a single chip in the near future. Any implementation of the WarpEngine that could be manufactured today would be limited by the frames available, rather than by VTSs.

Short VTSs of 16 to 32 bits are all that are necessary to maintain the virtual order of programs in the WarpEngine using the explicit tags developed in this chapter. These techniques are a promising basis for developing a scalable virtually ordered memory system for a speculative processor.

Analysis of the execution tree also provides valuable information for use in throttling speculation. If the resources required by a subtree exceed those available then execution of anything more speculative will have to be rolled back. At best it is wasted execution, at worst the overheads involved in rollback will restrict parallel execution of nearer blocks and lengthen the critical path.

As shown in Chapter 7, the analysis developed for allocating VTSs is also useful for estimating the requirements for other linear resources. This provides a novel method for controlling speculation, by throttling speculation beyond branches that are hard to estimate resource requirements for, or that require extensive resources.

Chapter 7
Selective Speculation

In an aggressive speculative processor, such as the WarpEngine, more instructions may be available to issue than there are resources available to execute them. When speculative execution outstrips available resources some selection must be made of which code to execute first. Often the selection criteria are side effects of the instruction issue mechanism, rather than an explicit design decision.

If the selection of code to execute speculatively is done poorly it can degrade performance substantially. In the WarpEngine, when resources, such as frames or VTSs, are exhausted they are released through cancelback. Cancelback guarantees that execution can proceed to completion with the same resources as sequential execution, but there is no guarantee for performance. It is not a preemptive technique and will incur a penalty, so it would be preferable not to execute the code in the first place.

Accurate *selective speculation* is particularly important for an architecture like the WarpEngine, which speculates aggressively. It has a large instruction window, providing wide scope for speculation, and can quickly exhaust available resources. Any instructions that are issued speculatively, but not retired represent a wasted opportunity to gain speedup.

Since the analysis of resource requirements developed in Chapter 6 must be done in order to allocate variable range VTSs, it is worthwhile investigating whether this information can also be used to ensure as many speculatively issued instructions commit as possible. The resources available can be compared to the estimated requirements of the subtree under consideration. If there are insufficient resources available nothing later in virtual time than

the current subtree should be issued until fossil collection frees some resources. It is likely that anything issued will be cancelled back to free resources for earlier instructions.

By splitting the resources—frames in this case—into units called *resource blocks*, instruction issue and resource allocation can be controlled based on resource use analysis. Resource blocks are a fundamental building block of the speculative multiple version memory system developed in Chapter 8.

This chapter begins by discussing selective speculation methods used in other architectures and some alternatives that could be used in the WarpEngine. Selective speculation methods are assessed by measures seen previously, and also by the metric of *speculation effectiveness*, the ratio of instructions issued to instructions committed.

7.1 Related Work

The amount of speculation used in a CPU is tuned, by design, to the execution resources available. By speculating more aggressively the processor uses more execution resources, but can achieve more execution speedup. More aggressive speculation can be achieved by increasing the speculation distance. For example, a processor using branch prediction can increase the number of branches predicted concurrently. Of course, this will increase the number of speculatively executed instructions that have to be squashed if the prediction accuracy remains the same. The *speculation effectiveness* has been decreased since the total number of instructions committed remains constant, but more instructions have been speculatively issued and then squashed.

A speculation effectiveness plot can be drawn for a speculative execution technique as the execution resources used are varied by controlling the aggressiveness of speculation. The plot indicates how good a particular technique is at selecting instructions to execute speculatively. Figure 7.1 shows a qualitative plot of speculation effectiveness for a number of published speculative execution techniques. The speculation technique resulting in the most effective speculation varies depending on the resources available, as process technology improves the most appropriate technique to use will change. The curves shown here have been estimated from the descriptions in the relevant literature.

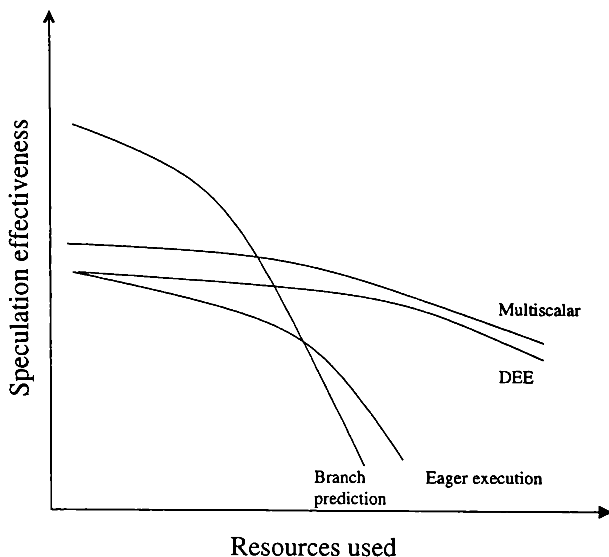


Figure 7.1: Speculation effectiveness for different speculative techniques

Branch prediction [Smith, 1981] provides effective speculation when small amounts of resources are being used, which is when the number of concurrent branch predictions are small. However, as the number of concurrent predictions increases the mis-speculation rate rapidly increases. Eager execution [Wang and Uht, 1990] is even worse at selecting instructions to speculate on than branch prediction, since it chooses to execute both sides of each branch decision. This both requires more resources to execute the same number of speculative branches concurrently and gives a lower speculation effectiveness, but the correct branch is guaranteed to be executed. Since control independence is also exploited eager execution is more effective than branch prediction for large amounts of resources.

Branch prediction and eager execution both suffer from the difficulty that they are trying to predict the instruction stream in linear order from beginning to end. This restricts the scope available for selecting instructions to execute speculatively. Split instruction techniques, such as the multiscalar [Sohi et al., 1995], disjoint eager execution (DEE) [Uht and Sindagi, 1995], the Trace processor [Rotenberg et al., 1997], dynamic multithreading [Akkary and Driscoll, 1998] and the WarpEngine, have more ability in this regard since they can choose not to execute a block of code speculatively, and instead go beyond it to a control independent point as discussed in Section 2.1.2.

Indeed, the motivation behind DEE was to increase the speculation effectiveness by exploiting control independence and calculating branch prediction confidence values [Jacobsen et al., 1996; Grunwald et al., 1998]. Branch outcomes are assigned a probability by a

predictor and as multiple branches are speculated across these probabilities are calculated cumulatively. Thus, the probability of a branch being taken reduces with depth. Branches are executed speculatively if the probability is above a threshold value. This limits speculation distance and executes the code most likely to be committed. Both branch taken and not taken targets can be executed in parallel if the probability is high enough, although one is guaranteed to be discarded. DEE can be applied to both single and multiple flows of control.

Threaded multipath execution [Wallace et al., 1998] applies the techniques used in DEE to a simultaneous multithreaded architecture. Spare thread contexts are used to pursue alternate branches in addition to the predicted one. Branches with lower confidence predictions are pursued until all the thread contexts are used. Only branches on the predicted path are considered for alternate paths. This selectively speculates on multiple branch outcomes up to one level off the predicted path.

In the multiscalar paradigm the compiler partitions sequential code into *tasks* [Vijaykumar and Sohi, 1998; Jacobsen et al., 1997] for execution as separate threads of control. Within a task instructions are executed non-speculatively relative to each other, as they would be on a superscalar processor. Tasks are selected such that they are control and data independent as much as possible. Selective speculation is implemented by speculatively executing instructions in following tasks, but not speculating on instructions in the current task, which are more likely to be control or data dependent on incomplete instructions.

The multiscalar indiscriminately squashes all computation beyond a mis-speculation, which makes it particularly important that accurate selective speculation is available, otherwise large amounts of accurate speculation will be rolled back by mis-speculation on independent code. Indiscriminate squashing also tends to limit the speculation distance because the most speculative operations will be squashed by any mis-speculation, and will have to reissue from the starting point. However, tasks split the instruction window so speculation at a distance can still be performed.

Split instruction window techniques are the reason resource usage must be used to compare speculation effectiveness, rather than speculation distance. They can achieve extremely large speculation distances while using low amounts of resources by choosing not to speculate on a large intervening section of code.

Conservative execution appears as a single point on the speculation effectiveness graph since it cannot use more resources through speculation because it does no speculation by definition. For the same reason the speculation effectiveness of conservative execution is always one.

More effective speculation for the same resource usage does imply lower execution time, although a simplification has been made in Figure 7.1 by representing resource usage by a single value. The resource usage will vary dynamically across program execution. This profile may differ considerably between the different speculation techniques, making a precise comparison on speculation effectiveness difficult. This metric also takes no account of the varying complexity of the techniques and the execution overheads involved in using them.

It is also worth noting that selective speculation could be used to throttle speculation to reduce processor energy consumption [Manne et al., 1998].

7.2 Selective Speculation in the WarpEngine

The basic WarpEngine simulator allows simulation of speculative execution limited only by the critical path of instruction issue. This allows speculation to later be limited in a variety of ways independently.

In previously published results [Littin, 2000] speculation was restricted by limiting the available execution resources, as measured by the number of frames. In the WarpEngine virtual order simulator, frames are allocated strictly in virtual order, which is equivalent to a processor using cancelback to free frames for blocks earlier in the virtual order with no time penalty. This is a very simple heuristic for selectively speculating, based on the assumption that events earlier in the virtual order are more likely to be on the critical path, and less likely to be rolled back.

In Chapters 5 and 6 speculation was restricted by limiting the number of VTSs available. Since VTSs are really just another execution resource this is similar to the first selection method, although the blocks that get stalled may be a little different. For example, fixed range VTSs allow a certain depth of speculative execution along each execution branch,

subject to rescaling. Variable range VTSs tend to provide a similar selection to frame limiting, without needing to assume zero latency cancelback to free frames. VTSs also provide scope to be much more flexible in the selection criteria applied. Instead of using the upper bound of VTSs required as the range size different heuristics could be applied, such as a best estimate of VTSs required, modified by a factor for the estimated likelihood of the instructions reaching commitment with their current source data.

The speculation selection method examined in this chapter again uses the VTS estimation information obtained in Chapter 6. This time the important factor is whether an upper bound can be placed on the number of frames (or VTSs) required by a subtree. If the size of the subtree cannot be estimated, it often means the subtree has a complicated control structure and will contain many blocks. Frequently it is not worth speculating beyond these subtrees because the frames will be cancelled back to provide resources for the inestimable subtree. By restricting the number of inestimable subtrees being speculatively executed concurrently it may be possible to better select areas for speculation. As a side effect it simplifies the hardware implementation, as we shall see.

7.3 Resource Allocation

A means of selective speculation not previously explored is to base it on the ability to allocate execution resources. The analysis described in Chapter 6 for allocating variable range VTSs predicts the number of frames and VTSs required to execute a subtree. If an upper bound can be placed on the size of the subtree then frames and VTSs can be reserved for the subtree and efficiently allocated as required. Successive subtrees can continue to be allocated in the same fashion following the reserved range. This suggests that rather than organizing frames in a structure that allows them to be freely reordered they can be arranged in a fixed virtually ordered array. This is a much simpler structure to implement, and should allow faster frame allocation.

However, an upper bound on the number of frames required cannot be determined for all subtrees. Recall that for subtrees containing dynamically bounded loops a proportion of the remaining range of VTSs is allocated because the analysis methods developed are unable to calculate an upper bound on requirements. If this proves to be insufficient rescaling

subsequently enlarges the range. A similar approach could be taken for allocating frames from the queue. However, if the estimate is insufficient copying the contents of a large number of frames would be required, and any in-flight instructions would have to have results redirected to the registers in the new frame. This would be a costly operation. The frame array could be modified by inserting additional frames at the exhaustion point, but this removes the hardware simplicity that is so attractive about the scheme. The approach investigated in this chapter is to restrict speculation from proceeding beyond the subtree until an upper bound on the frames required can be established.

The virtually ordered array of frames is termed a *resource block*. A single resource block with a finite number of frames is capable of executing a program containing many unanalyzable subtrees, albeit with substantial limiting of speculation. Reference will be made to the example tree and corresponding resource block in Figure 7.2 for illustration of frame allocation. The resource block is illustrated as an infinite length list of frames for simplicity. This can be implemented in a circular buffer by stalling when the frames are exhausted until those at the head of the queue have been fossil collected and can be reused at the tail.

Typically the total size of the execution tree will be unanalyzable, so the whole resource block is allocated to the tree (from node A). Subtrees are then allocated frames from within this range. One subtree of unbounded size may be allocated frames at each level. Subtrees B and C are allocated the requested number of frames from the start of A's range and subtree D is allocated the rest of the resource block. Since the upper bound number of frames is reserved for subtrees B and C, some reserved frames may go unused. These remain empty until those ahead of them are fossil collected and they can be reused in order.

If a second unbounded size subtree exists at a given level it is not allocated VTSs until a bound has been established on the first subtree's frame requirements. So subtree I is not allocated frames until subtree H has been allocated, and is the GVT holder. Until GVT has passed G it could still be rolled back and re-executed to fire other unbounded size subtrees. At this point I can be allocated all the frames remaining in the resource block. If a subtree has an established upper bound on its size all the subtrees within it will also be bounded.

Multiple resource blocks can be used to allow several subtrees of unpredictable size to be speculatively executed concurrently. In the example in Figure 7.2 two resource blocks would allow subtree I to be executed using the second resource block without delaying

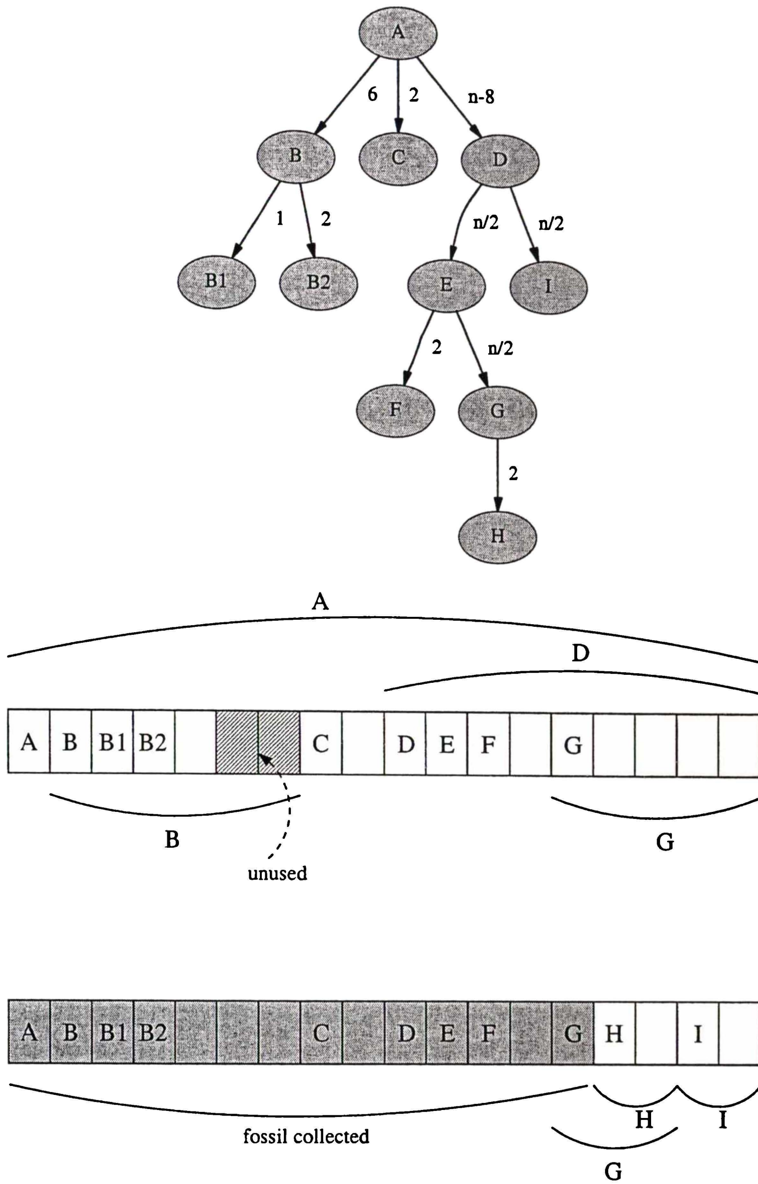


Figure 7.2: Allocating frames in a resource block

execution. Resource blocks have a virtual order and form a second layer to the virtual order hierarchy above the frame ordering within the resource blocks. Resource blocks can be viewed as a variation on the multiscalar processing units, with tasks divided by the end of unanalyzable subtrees.

Two organizations of resource blocks are simulated here. In the first resource blocks are arranged in a circular queue. Subtrees are allocated to resource blocks as described above, strictly in order. They are fossil collected for reuse in the same order from the tail of the queue. The drawback of this arrangement is that if an unanalyzable subtree splits into two unanalyzable subtrees and then the left subtree splits into two more unanalyzable subtrees

a cancelback will be forced. For example, in Figure 7.3 initially subtree B is allocated resource block 0, and subtree C is allocated resource block 1. However, when subtrees D and E are initiated subtree D is allocated to block 0 and subtree E is allocated to block 1, since it is before subtree C in the virtual order, and subtree C must be cancelled back and restarted in resource block 2.

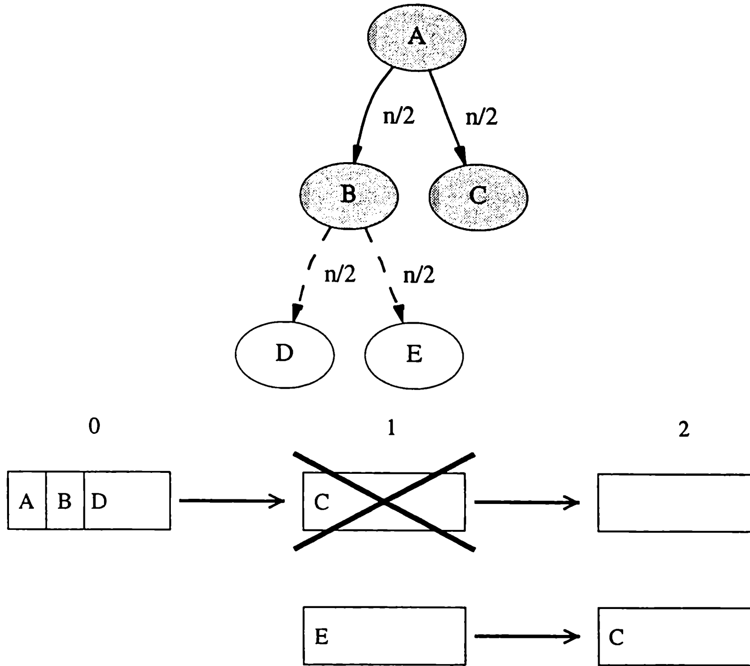


Figure 7.3: Program forcing cancelback for a circular queue of resource blocks

This problem can be avoided by allowing resource blocks to be dynamically reordered, so that in the above example resource block 2 can be placed between blocks 0 and 1 and used for subtree E. Subtree C can continue executing in block 1. Resource blocks could be reordered using a crossbar switch or an ordering lookup table. Either method is feasible for a small number of resource blocks.

In the simulations in this chapter resource blocks have been modelled with sufficient length to avoid blocking due to frames running out within a resource block.

7.3.1 Assigning Frames to Resource Blocks

A resource block is equivalent to an ROB using a queue to maintain the order of instructions (grouped in frames in the WarpEngine). Assigning a frame a position in a resource block is equivalent to assigning an instruction to the ROB. Since instructions in a typical out-of-

order superscalar processor are assigned to the ROB in virtual order, the ROB is easy to implement as a linear buffer. However, in the WarpEngine frames are allocated in the order the execution tree is generated, providing motivation for multiple resource blocks.

Real resource blocks will have a finite length, and eventually the end will be reached. A strategy must be devised for continuing to execute the tree branch in that resource block. This section considers the available options, although all the simulations in this chapter use resource blocks big enough that they are never exhausted in the course of executing the test suite.

Circular looping

One possibility is to stop issuing more frames on that execution branch until frames at the start of the resource block have been freed by fossil collection. Then the resource block can be used as a circular queue, with the frames at the beginning being reused as the latest in the virtual order.

Unfortunately this will only be of use in the resource block containing the GVT holder. All other resource blocks will be blocked until they become the GVT holder. To gain maximum advantage from speculation it should be possible to allow other execution branches to continue either in the same resource block, or in another resource block if one is available.

Compiler controlled allocation

Compiler analysis could be used to force the processor to allocate frames to a new resource block before the frames in the original resource block are exhausted. This involves altering the shape of the execution tree to start a new sub-branch at the point switching to the new resource block is desired, and indicating that the subtree size is unanalyzable for its sibling earlier in virtual time. It may, in fact, be possible to give a good estimate of size, but this would not force the branch onto a new resource block.

The problem here is that it will be difficult for the compiler to calculate a good point to switch to a new resource block. The optimal point is one that guarantees that the end of the resource block will never be passed, and also uses the resource block efficiently, allocating

instructions to as many of the frames in the resource block as possible. Note however, that this method does not require any additional ability to reorder resource blocks.

Processor controlled allocation

The most effective option is to have the processor dynamically allocate frames in a new resource block when the existing one becomes exhausted. This promotes maximum utilization of resource blocks since only full resource blocks have their execution branch transferred to a new resource block. Poorly utilized resource blocks may still result from a branch overflowing to a new resource block and finishing a few frames later, but this is unavoidable.

This method presupposes the ability to reorder resource blocks dynamically, so that a new resource block can be inserted following the exhausted one. Otherwise a cancelback or squash of all future speculation is necessary, which is potentially disastrous for performance.

Allocating a new resource block, rather than trying to reuse the existing one through fossil collection allows fossil collection to be simplified by restricting it to whole resource blocks only. The resource block is simply marked for fossil collection once there are no actively executing frames, or active resource blocks earlier in virtual time. While this makes fossil collection somewhat bursty it can occur in the background to normal computation.

As with frame allocation, cancelback of resource blocks may be required to guarantee program completion. In the simulation results below virtual order simulation again provides the equivalent of cancelback with no execution penalty by allocating resource blocks in priority of virtual order.

7.4 Speedup

A major design decision in the use of resource blocks is the number to be implemented. In this section results from the WarpEngine virtual order simulator are presented showing the effect on speedup of the test suite using between one and sixty four resource blocks.

Since the frame requirement analysis derived in Chapter 6 is required for resource block

allocation, the same programs from the test suite are simulated: AVL tree insertion; binary tree insertion; Fibonacci number generation; and quicksort. Results for Gauss-Jordan elimination and matrix multiply are not shown, since they provide ideal speedup with only one resource block because the number of frames used can be determined at compile time.

Figures 7.4 to 7.7 show results using resource blocks in a fixed order circular queue, meaning that all future speculative resource blocks must be cancelled back when a new resource block is allocated. This restricts speculation in a similar way to the multiscalar, although much more care is taken in the multiscalar to create tasks which avoid wasting speculative execution in this way.

The curve for execution with the ideal VTS representation is shown to allow comparison against the theoretical speedup limit.

The speedup of all the test programs is heavily restricted when using only one resource block. The two tree insertion programs (Figures 7.4 and 7.5) show a steady improvement as additional resource blocks are supplied. Sixty four resource blocks provide a speedup near to the theoretical limit. Each insertion forms a region of execution that is usually independent from the insertions around it, allowing each to be allocated to a resource block.

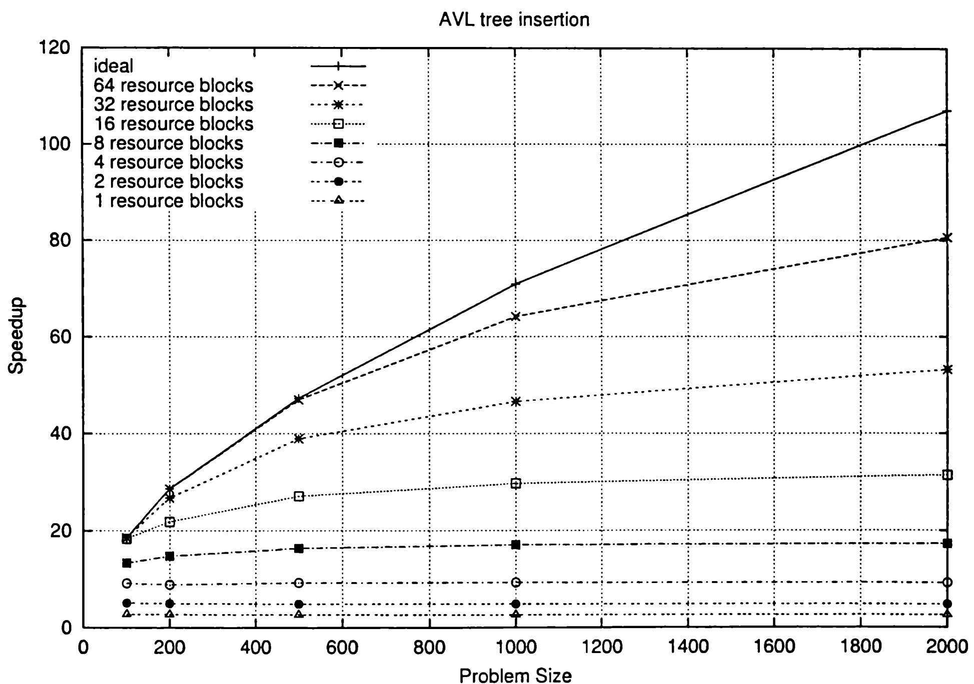


Figure 7.4: Speedup for AVL tree using resource blocks in a circular queue

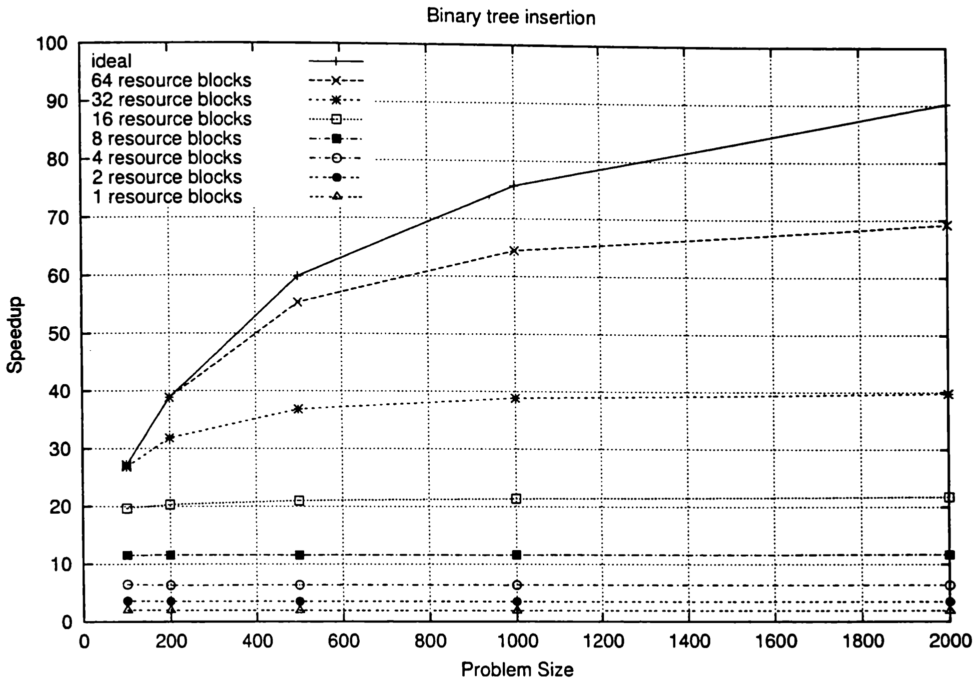


Figure 7.5: Speedup for binary tree using resource blocks in a circular queue

Fibonacci and quicksort (Figures 7.6 and 7.7), however, do not show much improvement as the number of resource blocks is increased. This is largely due to the presence of recursion. Each time another level of recursion is begun all the more speculative events must be cancelled back due to the fixed order of the resource blocks.

Fortunately recursion is relatively uncommon because it will always be difficult to allocate linear resources to. These examples and others with similar structures will be helped by estimating the number of resource blocks required by a subtree and reserving them in the linear sequence. This analysis uses similar information to analysis of VTS requirements for a subtree, but is only concerned with the presence of a subtree size bound, not the size of the bound itself.

Figures 7.8 to 7.11 show the results of the same simulation, but for resource blocks which can be dynamically reordered. This will complicate the hardware design, but should be feasible for a small number of resource blocks.

The tree insertion programs show similar results to the fixed order blocks, due to the usually independent operations mentioned above. The performance of Fibonacci and quicksort improve to acceptable levels, although still some distance from the ideal.

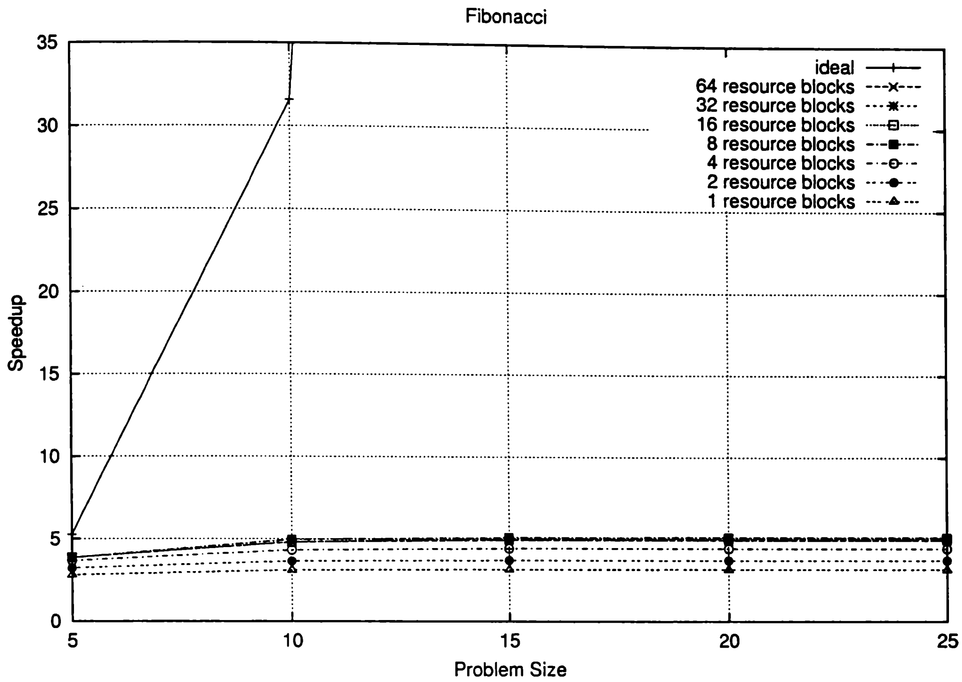


Figure 7.6: Speedup for Fibonacci using resource blocks in a circular queue

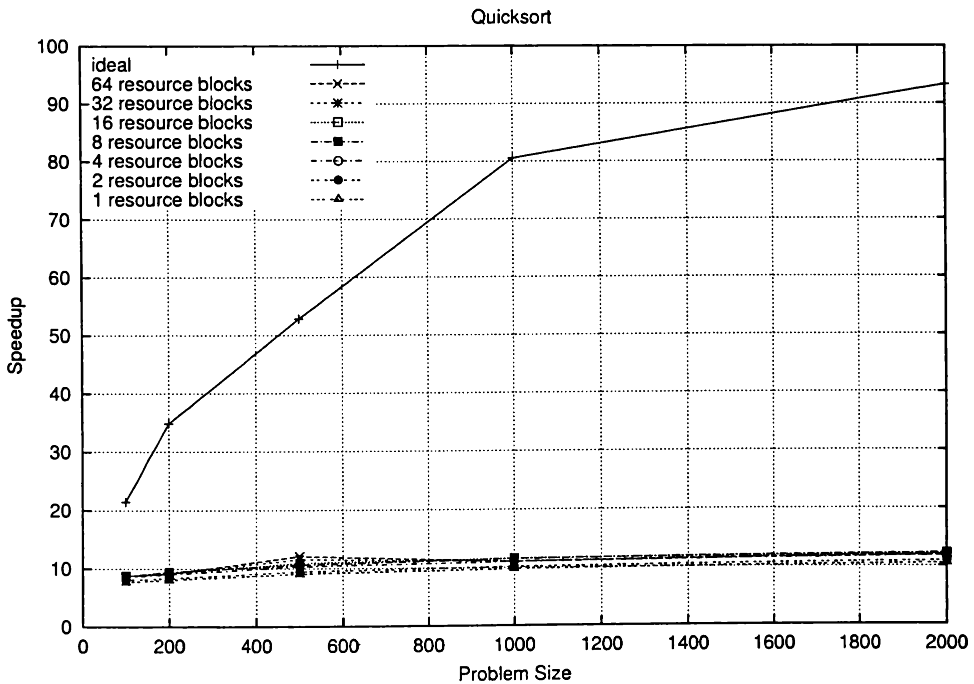


Figure 7.7: Speedup for quicksort1 using resource blocks in a circular queue

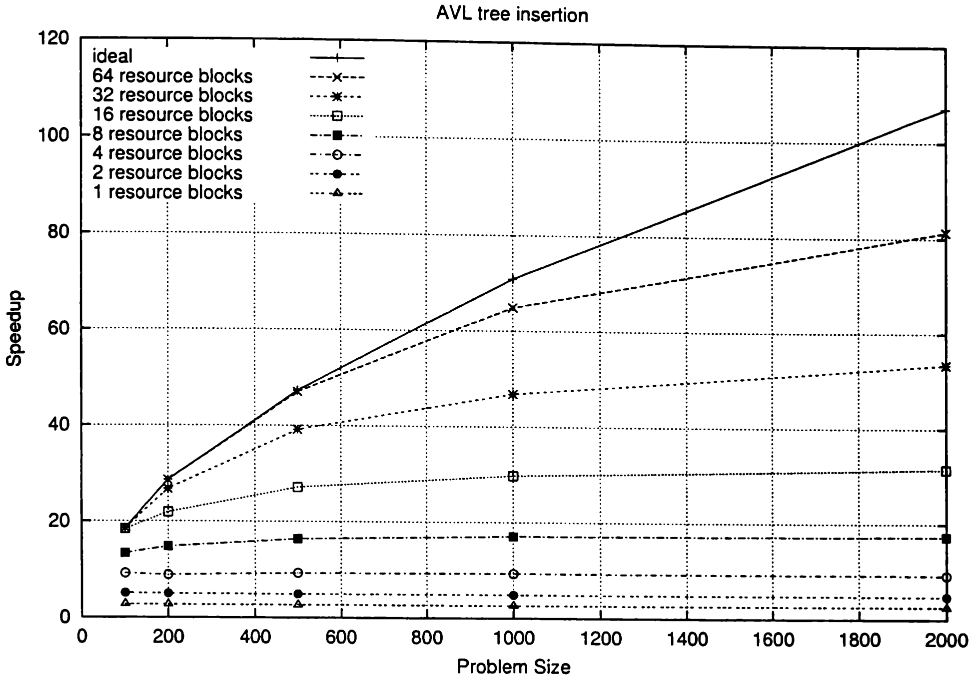


Figure 7.8: Speedup for AVL tree using reorderable resource blocks

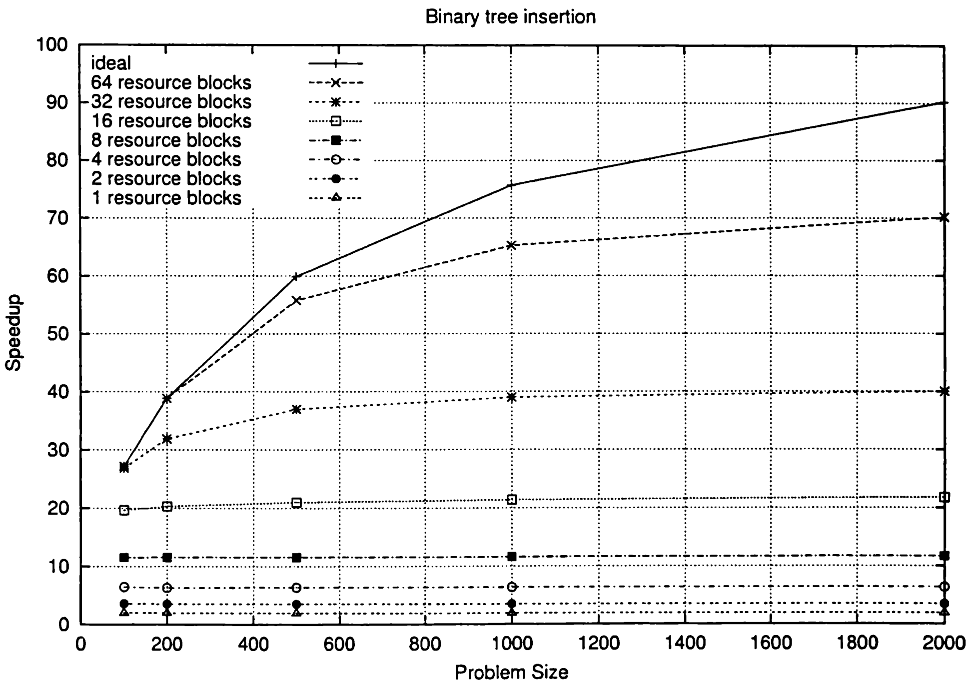


Figure 7.9: Speedup for binary tree using reorderable resource blocks

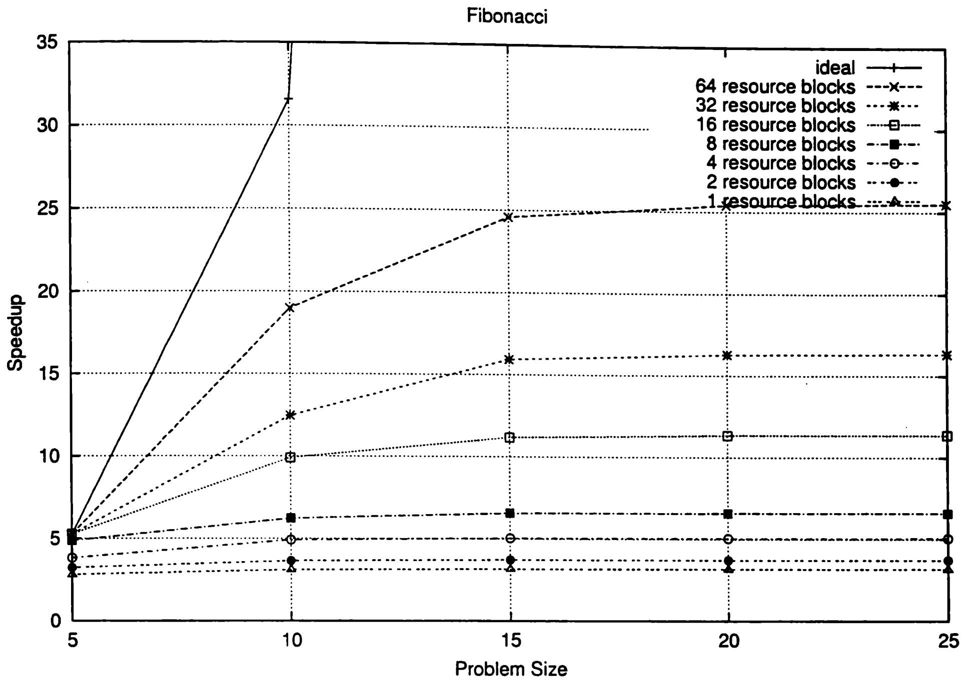


Figure 7.10: Speedup for Fibonacci using reorderable resource blocks

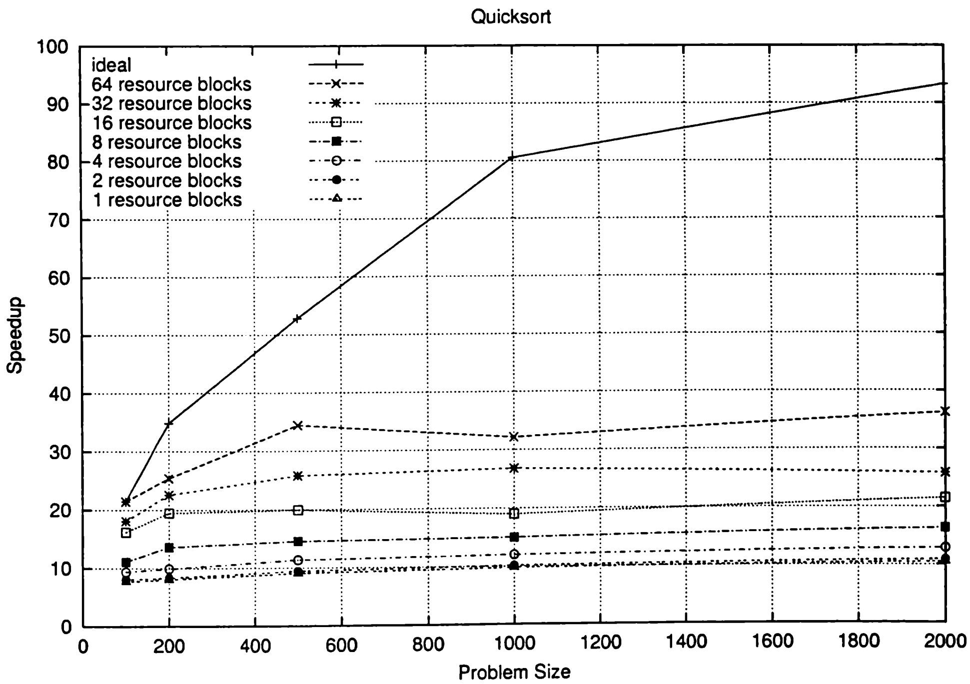


Figure 7.11: Speedup for quicksort1 using reorderable resource blocks

It must be remembered that there will be other constraints on the speedup, such as the number of frames, and resource blocks may not be the limiting factor.

7.5 Resource Blocks and Frame Limiting Compared

The graphs in Figures 7.12 to 7.15 show the speedup obtained when limiting the number of reorderable resource blocks, compared with limiting the number of frames available in a WarpEngine which does not use resource blocks.

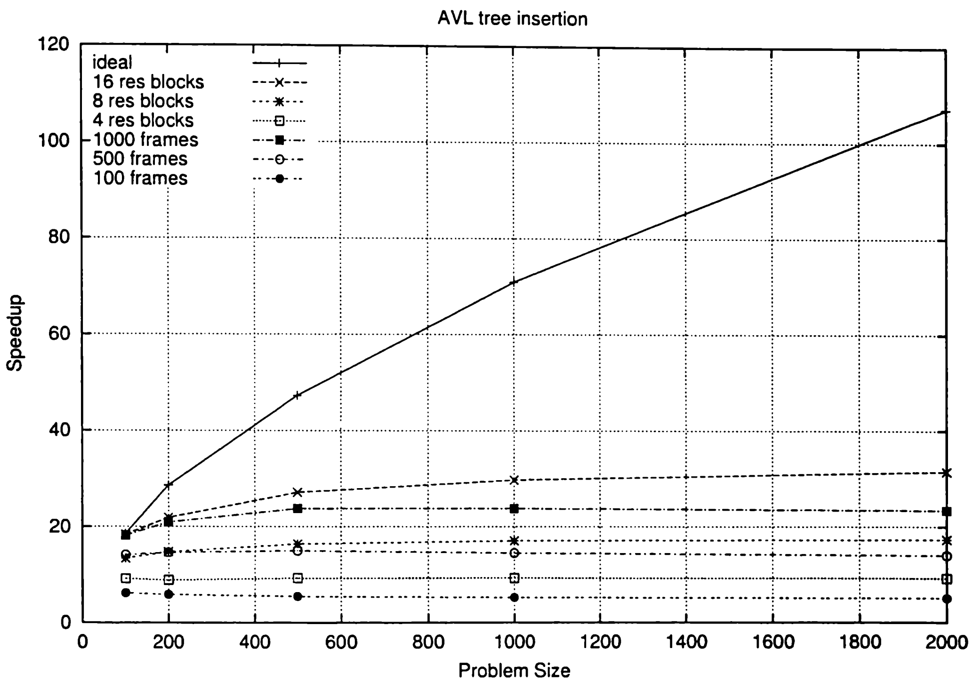


Figure 7.12: Speedup for AVL tree with frame limited and resource block limited execution

AVL tree insertion is the least limited of the test programs shown, relative to frame limitations, with sixteen resource blocks allowing greater speedup than a frame limit of a thousand frames. The restrictions of resource blocks get steadily worse relative to frame limits for quicksort, binary tree insertion and Fibonacci. This is in order of increasing available parallelism, suggesting that resource blocks tend to limit the parallelism extracted to similar absolute levels across all programs much more than frame limits do.

However, programs, such as matrix multiply and Gauss-Jordan elimination, whose frame usage can be calculated at compile time are not restricted by resource blocks, but can suffer substantial restriction by limiting the number of frames, as shown in Section 6.5.2 and in

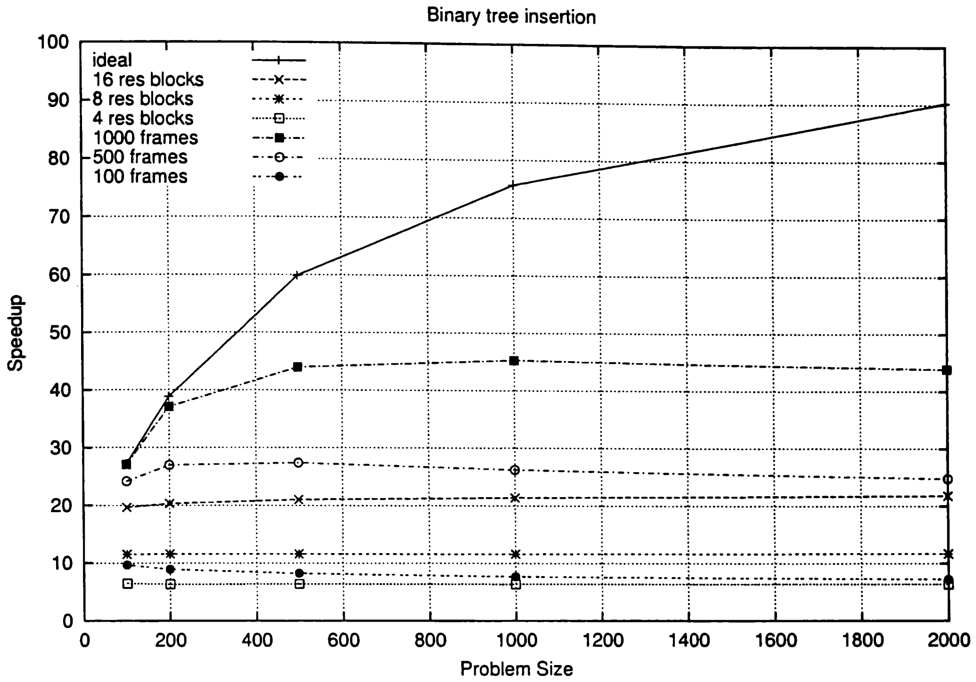


Figure 7.13: Speedup for binary tree with frame limited and resource block limited execution

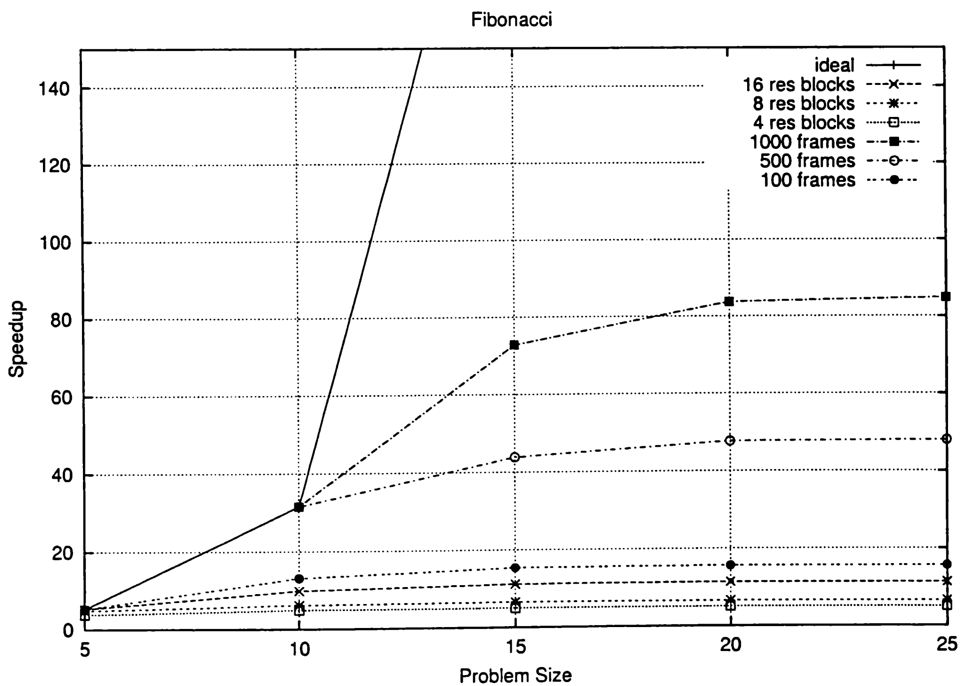


Figure 7.14: Speedup for Fibonacci with frame limited and resource block limited execution

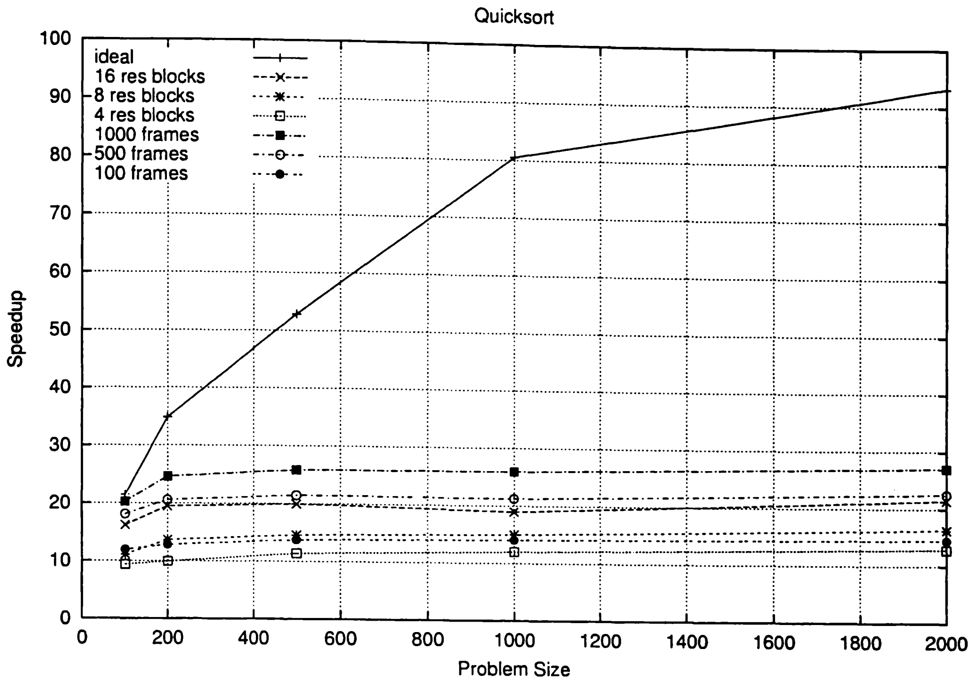


Figure 7.15: Speedup for quicksort1 with frame limited and resource block limited execution

more detail by Littin [2000].

It is estimated that five hundred to a thousand frames are as many as could reasonably be provided in the near future [Calvert, 1997; Littin, 2000]. This limit restricts speculation to a similarly to sixteen resource blocks.

7.6 Speculation Effectiveness

Using the WarpEngine virtual order simulator, speculation effectiveness can be calculated from the ratio of committed instructions to the total number of issued instructions. Committed instructions can be obtained from the basic WarpEngine simulator, without transient state information, since only instructions which are ultimately committed are simulated. The total instructions issued can be measured using the transient state extensions, since rolled back instructions are also tracked in this version of the simulator. Each instruction issued, whether it ultimately commits or not, is only counted once in the simulator, even if it is rolled back and reissued several times.

The speculation effectiveness is plotted against the resource limit enforced in Figures 7.16

to 7.18 and C.17 to C.19 for the suite of test programs. Since the resource limit is measured by the number of resource blocks available for resource block limited execution, rather than the number of frames used, the two sets of curves are not exactly equivalent. A varying number of frames will be used in each resource block. However, the figures in Section 7.5 can be referred to for the speedup relationship between frame limiting and resource block limiting, and the number of resource blocks available is the important resource limit. All programs in the test suite have been simulated with a variety of data sizes for both frame and resource block limited execution.

For AVL and binary tree (Figures 7.16 and C.17) insertion resource block limiting shows a substantially better speculation effectiveness frame limiting right across the range than. Increasing the problem size moderately decreases the speculation effectiveness for both forms of limiting, due to the larger split instruction window that is used over a longer execution time. The curves for all data sizes are quite closely grouped, however.

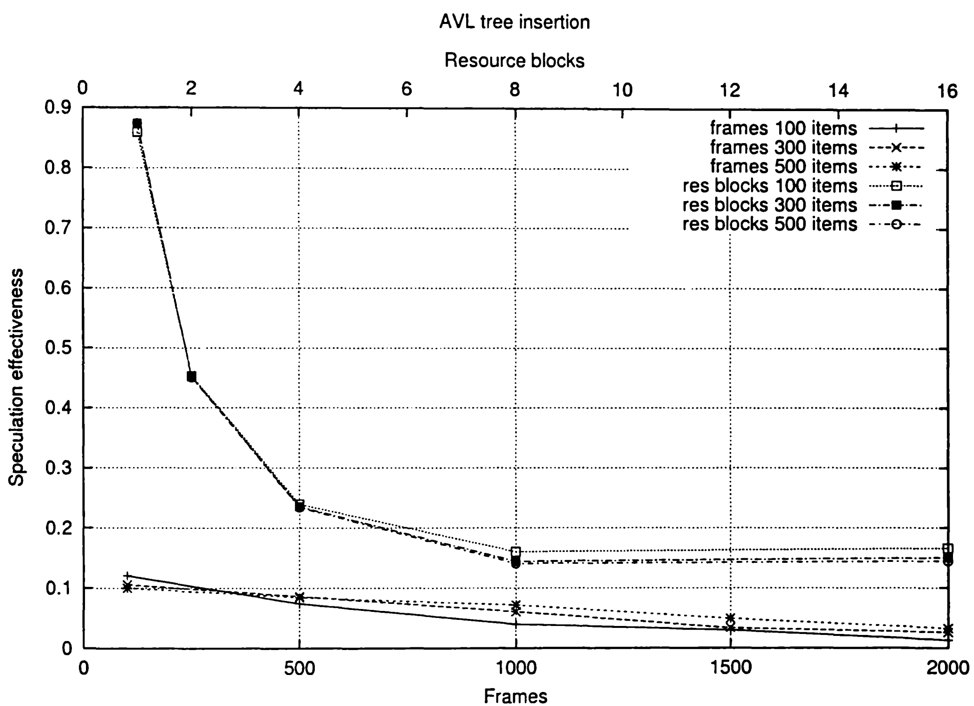


Figure 7.16: Speculation effectiveness of resource block and frame limiting for AVL tree

For quicksort (Figure 7.17), with small datasets resource block limiting allows more effective speculation. However, as the size of the data set increases deeper recursion is employed which, as previously noted, resource blocks do not cope with well. In instances of deeper recursion the effectiveness is similar to frame limiting, which does not vary much with data

set size. The situation is the same for Fibonacci, although frame limiting performs better than for quicksort.

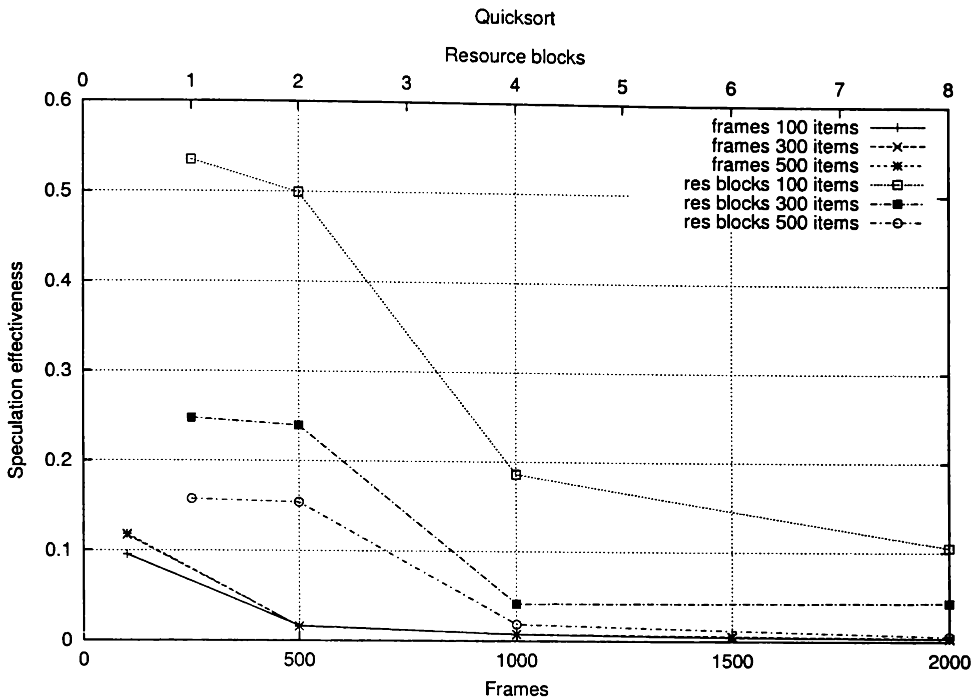


Figure 7.17: Speculation effectiveness of resource block and frame limiting for quicksort1

The frames used for both Gauss-Jordan elimination and matrix multiply (Figures 7.18 and C.19) can be calculated statically for the whole program, hence they can execute unconstrained in a single resource block. However, the analysis used in the transient state simulations has been stated such the whole program is given all the available VTS range, and the right-most branch of each subtree is given all the available range remaining after the fixed estimate has been provided to all the other branches. When there is an even division of the VTS range between two subtrees, which could have been calculated precisely, the parent's range is simply divided in half. This estimation performs suboptimally with resource blocks, since each time the right-most branch of the tree is divided equally it is treated by the simulator as two unknown size subtrees, which are allocated to separate resource blocks. However, this is a much more interesting allocation pattern to investigate, since performance will vary with the number of resource blocks. The essential pattern is a backbone down the right side of the tree with known size subtrees branching to the left, mixed in with some unknown size subtrees.

Even with artificially introduced unanalyzable subtrees the speculation effectiveness for

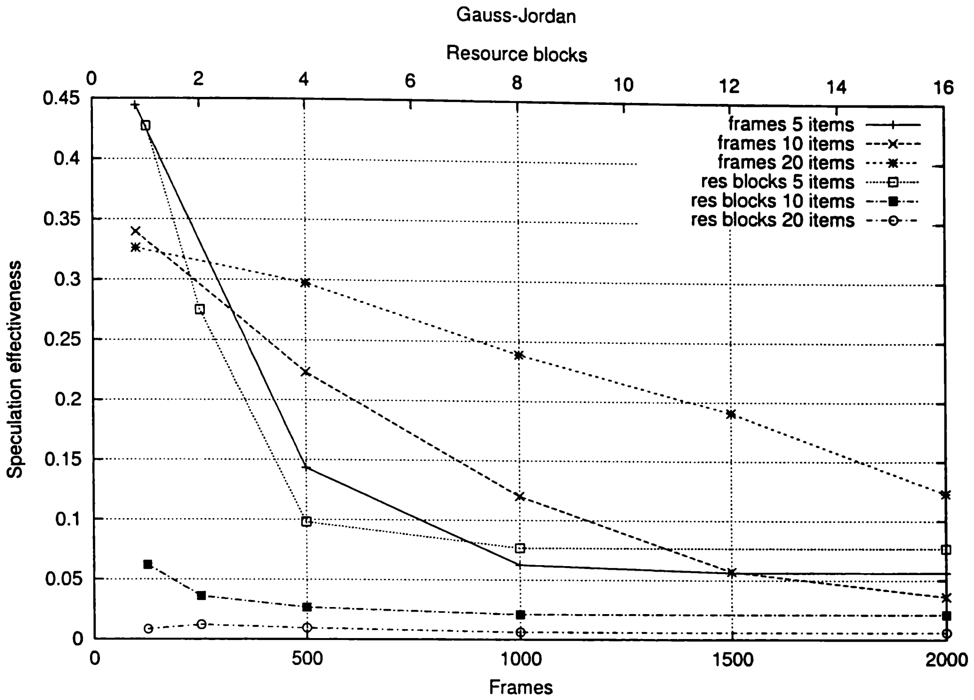


Figure 7.18: Speculation effectiveness of resource block and frame limiting for Gauss-Jordan

Gauss-Jordan elimination still levels off at eight resource blocks, not decreasing much beyond four resource blocks. This demonstrates the small number of resource blocks are actually being used. The relatively low value of the speculation effectiveness is indicative of the large speculation distance within each resource block, and consequent mis-speculation when combined with the large number of data dependencies.

Matrix multiply, on the other hand, shows perfect speculation effectiveness for all data sizes and all numbers of resource blocks, despite the introduced unanalyzable subtrees. This is caused by the completely control independent computation throughout this "embarrassingly parallel" program, which will eventually commit all instructions that are issued. They may be rolled back and re-executed several times, though, which is only recorded as a single instruction issued by the virtual order simulator. The lower speculation effectiveness for frame limited execution is caused by cancelback due to insufficient frame resources.

Interestingly, the speculation effectiveness in all the test programs does not vary much as the frame limit is increased, but is always at relatively low levels. This indicates that the basic selection heuristic in the WarpEngine, which selects frames early in the virtual order for preferential execution, is not a very accurate method, but does scale well for increased

resources. The programs for which frame limiting does achieve quite high absolute levels of speculation effectiveness, binary tree insertion, matrix multiply and Fibonacci, all have large independent code regions.

7.7 Summary

Resource blocks are a useful way of selectively speculating for a number of reasons. They allow speculation on branches of the execution tree to be selected early by using analysis of the size of the subtrees. This means speculative execution is focused where it is most useful, as shown by the speculation effectiveness results. Resource blocks use speculative execution much more effectively than the frame limits which allow the same execution speedup.

Arranging frames into resource blocks allows them to be allocated in a linear order within the resource block. This has the potential to simplify the ordering enormously when compared with the model of a pool of frames to be allocated arbitrarily. Instead of needing to assign an explicit VTS to each frame, only resource blocks need VTSs. This both reduces the necessary length of VTSs and reduces the overhead of calculating VTSs over the life of a program's execution.

Resource blocks are similar to the processing elements used in the multiscalar. Each processing element processes a task, which is a contiguous region of the instruction window. In the same way each resource block is a contiguous region of the instruction window formed by one execution subtree of unknown size and possibly several subtrees on which an upper bound of size can be placed. Resource blocks are more flexible than multiscalar tasks however, since several known size subtrees can be placed in a resource block with one unknown size subtree at runtime, whereas tasks must be determined at compile time. The multiscalar also requires that all future tasks be squashed when a task is squashed. The WarpEngine allows resource blocks to be selectively rolled back.

The unified instruction window of a standard out-of-order superscalar processor is analogous to one resource block. The issues involved in resource blocks do not become important until a split instruction is being used.

The simulation results presented in this chapter show that, even restricting resource allocation to a small number of resource blocks, on the order of sixteen, reasonable speedup can still be achieved. Recursive function calls badly affect the linear resource allocation. This is an intrinsic problem with this structure and is uncommon in typical processor loads.

For programs with loop based parallelism, such as AVL and binary tree insertion, sixteen resource blocks allow parallelism to be extracted of the same order as limiting frames to five hundred to a thousand. It is likely that this is the number of frames that could be implemented on a single chip WarpEngine with current technology. Where the parallelism is recursion based, as in Fibonacci, or a combination of the two, as in quicksort, resource block limiting is disadvantaged compared with frame limiting alone.

For the programs in the test suite used here resource blocks provide a better method of selecting instructions for speculative execution. A larger proportion of those issued are committed, compared with those executed under simple frame limits. This indicates that if the resource requirements of a region of code can be determined in advance it is more likely to contain independent instructions than those in unanalyzable regions of code.

Resource blocks form one level in a hierarchy of resource allocation which stretches from the instruction and frame level through resource blocks to multiple chips. Work can be conveniently split across chips using the process to allocate instructions to resource blocks, some of which reside on different chips. There are many other issues beyond the scope of this thesis involved in multiple chip operation, but resource blocks provide a convenient basis for dividing processing into units for migration to multiple chips.

Chapter 8 follows on from this work to develop a virtual ordered memory system which uses resource blocks as the basic level of its hierarchical organization.

Chapter 8
Twisted Memory

In the preceding chapters tracking the virtual order of memory operations has been approached through explicit VTS tags. The ultimate intention of these techniques is to design speculative memory systems for very large numbers of speculative accesses over a large speculation distance. Memory systems proposed elsewhere (discussed in Chapter 2) are restricted to a small number of speculative accesses, and typically a small speculation distance.

A conceptual view of a *virtual ordered memory system* was presented in Section 3.3.7. The key component in the virtually ordered memory system is the *time-space cache*, a multiple version memory system supporting arbitrary speculative reads and writes.

The conceptual model is essentially a list of memory accesses to each address stored with an indication of their virtual order so that the correct value can be returned for any subsequent reads. It is important to record reads as well as writes to detect when a write must cause a previously executed speculative read to be resatisfied. Rollback is achieved through anti-messages which remove the entry from the time-space cache, and which may also cause a read to be resatisfied. Cancelback also requires the entries from the cancelled frames to be removed from the time-space cache.

The time-space cache functions as a *write-back cache*, which reduces memory traffic to spatial memory by not writing every value through to memory. When fossil collection evicts entries from the time-space cache only the latest collected write to each address needs to be written to spatial memory. This caching can be made even more effective by employing *lazy*

fossil collection, which will only evict fossil collected values from the cache when space needs to be freed for new speculative entries. This has the advantage of reducing fossil collection overhead by fossil collecting less often, as well as allowing a greater caching effect. The time-space cache can also be used to cache values read from main memory in the same way as a spatial cache.

The memory model assumed so far has been a unified speculative memory large enough to hold all speculative memory references, and accessible in a constant number of cycles. This could be implemented by an arbitrarily large time-space cache without any spatial memory, simply retaining the fossil collected values in the time-space cache.

While storing fossil collected values in the time-space cache is useful when there is unused space, it is an inefficient use of memory, since memory locations in the time-space cache space record a VTS, which is unused by fossil collected entries. A better use of memory would be to have a small, fast time-space cache and the usual hierarchy of spatial memory beneath it.

It is also desirable to have multiple distributed time-space caches to support scalability across multiple processors, and also within a processor should the time-space cache prove to be a performance bottleneck.

In this chapter the issues associated with using explicit VTSs, such as those developed in Chapters 5 and 6, to implement a virtually ordered memory system are briefly discussed. However, assigning a VTS to every block requires long VTSs, even with the more efficient representations. By combining explicit VTSs with fixed hardware ordering a smaller number of explicit VTSs are required.

The main focus of this chapter, is the design and preliminary examination of a time-space cache using the resource usage analysis developed in Chapter 7. A hierarchy of speculative caches is used to order memory operations using a hybrid hardware ordered and explicit VTS mechanism. This memory system is shown to provide a scalable, distributed solution to the problem of ordering speculative memory accesses in a large instruction window.

8.1 Explicit VTSs

Several methods of assigning explicit VTSs to frames have been examined in detail in Chapters 5 and 6. Using variable range VTSs it has been shown that a 16 bit VTS is generally sufficient to support execution on the WarpEngine with over a thousand frames, using the VTS rescaling and allocation methods described in Chapter 6. However, adding 16 bits to every memory operation, and then storing it in the time-space cache until it is fossil collected adds substantial overhead. The time-space cache will be significantly bigger than spatial memory for the same number of entries in the cache.

Obtaining the virtual order of events, including memory accesses is a matter of comparing the VTSs of the events. While this only requires comparing two integers, a fast operation, accessing the time-space cache potentially requires many of these comparisons.

Satisfying a read correctly requires returning the most recent write to that address which is prior to the read in the virtual order. A read requires finding the write to that address with the greatest VTS less than that of the read. If the time-space cache is implemented as a list of memory accesses the list corresponding to that memory address must be searched and a comparison done with each existing entry. Reading could be made more efficient, for example by using a hashing algorithm, but this would add to the complexity of inserting an entry into the time-space cache, necessary for both reads and writes.

Writes and anti-writes also involve searching the time-space cache in order to identify reads which need resatisfying with the newly written value, or with the value superseding the rolled back value.

The address resolution buffer and speculative versioning cache described in 2.4.6 are restricted implementations of an explicit VTS time-space cache. They use the task sequence number in the multiscalar processor to track the virtual order. This is much simpler in the multiscalar than in the WarpEngine because tasks are allocated to processing units in a fixed hardware order, and the largest published configuration consists of only eight processors, making the sequence numbers much shorter than VTSs. This approach to time-space cache design is not pursued further here, rather an approach based on resource blocks is developed.

8.2 Twisted Memory

The resource blocks developed in Chapter 7 suggest a natural hierarchy of virtual ordering. The frames, and hence the memory accesses, are already organized in virtual order within the resource block, so only the order between the resource blocks needs to be explicitly maintained when comparing memory accesses for different resource blocks. The memory accesses from each resource block fall within a contiguous virtual order range, not overlapping with any other resource block. This means that a read should return the most recent write to that address from the same resource block, and will only need to access writes from previous resource blocks if there has been no write that is earlier in the virtual order within the same resource block. Only the latest write within a resource block for each address is ever returned to reads in other resource blocks, although that write may be superseded as speculative execution progresses.

The need to search through resource blocks to find a memory access, first in the current resource block, then through progressively earlier resource blocks until an access to that address is found, suggests a further hierarchical organization to the time-space cache. The memory architecture proposed is shown in Figure 8.1 for four resource blocks. It is known as *twisted memory* because of the way entries twist through the network as they are propagated to the different levels. The interconnections between twisted memory cells forms an *omega* [Lawrie, 1975], or *baseline* [Wu and Feng, 1980], network topology.

The resource blocks themselves, each composed of an array of frames, are shown at the top of the diagram. Each resource block is connected directly to a time-space cache cell, which all memory accesses from that resource block initially go to. Level zero time-space cache cells must record the frame number the access originated from, to track the virtual order of memory accesses, since intra-resource block ordering is important at this level. However, at lower levels only the resource block the access originated from is significant.

The principle of locality states that most reads will be satisfied by a write from the same resource block, or a nearby resource block. This is an advantage for twisted memory because most loads will be satisfied by traversing a small number of levels of the hierarchy. Each level traversed adds to the latency of the operation, so reducing the average depth of an access reduces the average latency of memory reads.

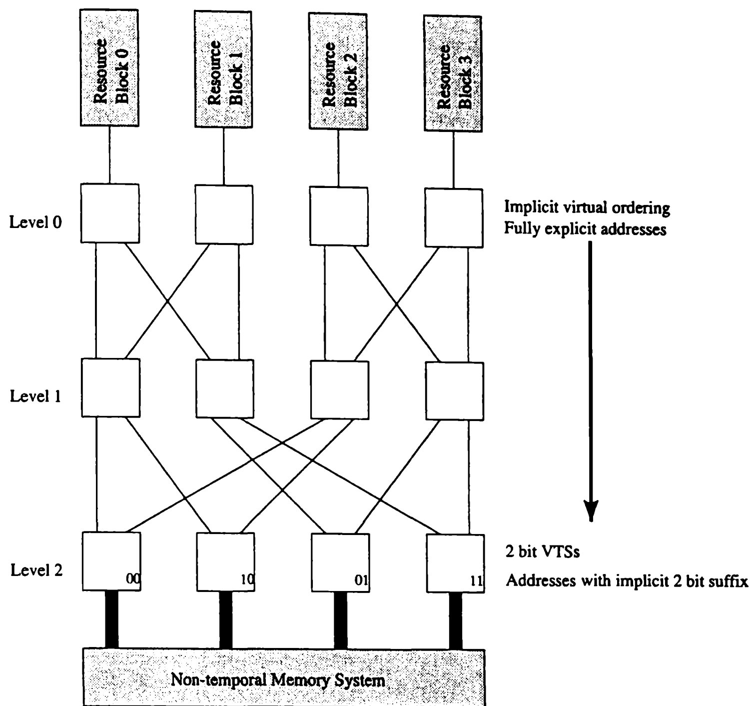


Figure 8.1: Layout of a twisted memory system with four resource blocks

At each level of twisted memory the memory accesses are sorted by the least significant bit of their addresses and propagated down either the left or right connection to the next level on that basis. The least significant bit is then redundant, since its value is implied by the location of the twisted memory cell, so it is removed. The access is then sorted again, and propagated to the next level based on the next bit of the address.

When the memory accesses have propagated to the bottom level of twisted memory all accesses from the same address have been sorted into the same cell.

While the inter-resource block virtual ordering of memory accesses at level zero can be determined purely from the twisted memory cell the access is in, this is not the case at lower levels. To track the inter-block ordering a short explicit VTS is assembled as the memory access propagates through twisted memory. One bit is added to the VTS at each level based on the connection the access *arrived* at the next level on.

Figure 8.2 shows an example of two writes and one read to address 9 (101_2) that have propagated through the twisted memory, from resource blocks one, two and three. The time-space cache entry is represented as $VTS : OP(value) : address$, where OP is the type of access, read (RD) or write (WR). The diagram is a snapshot of the memory state at a

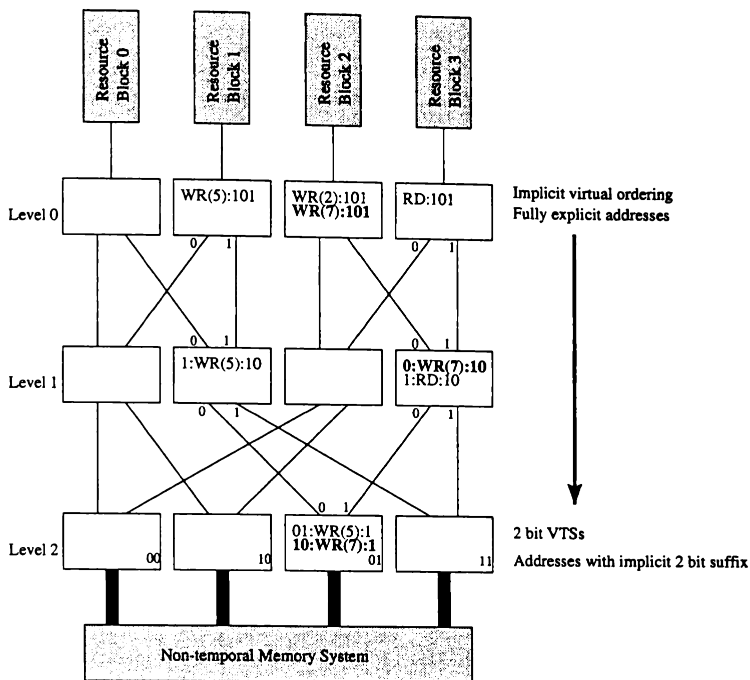


Figure 8.2: Propagation of a write through twisted memory

particular instant in real time. At some later real time an entry for an access may be inserted anywhere in the virtual order. The write from resource block 2 with value 7, shown in bold, is the latest access in real time order.

At each level one bit is also removed from the explicit address recorded, becoming implicit in the position of the twisted memory cell. An earlier write to address 9 from resource block two of the value 2 is not propagated to level one because it has been superseded by the write with value 7 later in the virtual order. The read from resource block three returns the value 7 from the write in resource block two, which it matches with at level one.

Each cell in the time-space cache is identical, apart from the level zero cells, which additionally contain the originating frame number. This contributes to a scalable design. The number of resource blocks supported can be doubled by adding an extra level of cells to the time-space cache. Of course, adding another level will increase the access latency to the bottom level of the time-space cache and to spatial memory. This will ultimately limit the size of the time-space cache. However, Chapter 7 showed that thirty two resource blocks provide reasonable performance compared to frame limited execution.

The network can be implemented with more than two input and output connections per cell. Four connections would effectively remove every second level of cells. Increasing the

number of connections between cells reduces the number of levels between the resource blocks and spatial memory, reducing the propagation time and total latency of an access to main memory. However, this makes the switching logic in each cell more complex.

As well as being scalable, twisted memory can function as a distributed design, with resource blocks being split across multiple chips, and the time-space cache could be used as part of a cross chip hierarchy. Multi-processing introduces a host of new issues which are beyond the scope of this thesis.

8.2.1 Write

When a write operation is executed the value is written to the level zero cell attached to the resource block, along with the frame number. Only the latest write in the virtual order to that address by the resource block will ever be accessed by other resource blocks, so only it must be propagated to lower levels.

If the resource block contains any reads to that address later than the currently executing write, but earlier than any other write, then a read reply must be reissued for them with the value of the new write, since they were incorrectly speculatively executed. All other writes to that address will be superseded by the propagated write, so they are not propagated and are only recorded in the level zero cell.

When a write is propagated to a lower level it is propagated left or right based on the least significant bit of the address, in Figure 8.2 a zero sends the entry left, a one sends it right. In the WarpEngine bit two is actually used to achieve a more even distribution, since all memory accesses are word aligned.

If there is already a write in the level one cell with the same VTS as the new write it is earlier in the virtual order and should be overwritten. Thus, there are a maximum of two writes to each address recorded in the level one cells.

The pseudocode for executing a write operation in each twisted memory cell is shown in Figure 8.3. The procedures within the code are described in Table 8.1.

At level one a check must again be made for any dependent reads. Reads from the same originating block as the write can be ignored, they will have been dealt with at level zero.

procedure	*description*
$InsertInList(X)$	Inserts memory access X into the list in virtual order
$Next(X)$	Returns the memory entry after X in the virtual order
$Previous(X)$	Returns the memory entry before X in the virtual order
$AccessType(X)$	Returns memory operation of the entry X, either READ or WRITE
$ReturnValue(X, Y)$	Sends the value of the write X as the result of the read Y
$LatestWrite(X)$	Returns true if X is the latest write in this memory cell
$LSB(X)$	Returns the least significant bit of the address of X
$PropagateLeft(X)$	Propagate the memory operation X to the next level of twisted memory on the left channel
$PropagateRight(X)$	Propagate the memory operation X to the next level of twisted memory on the right channel
$GetMatch(X)$	Returns the entry that matches the anti-message X
$Delete(X)$	Remove message X from the memory cell
$VTS(X)$	Return the VTS of the entry X
$Commit(X)$	Write X to spatial memory

Table 8.1: Procedure descriptions for twisted memory pseudocode

```
{ N is the new write }
{ Level is the level of twisted memory being operated on }
{ MAXLEVEL is the bottom level of twisted memory }

InsertInList(N)
E ← Next(N)
while (AccessType(E) = READ )                         { Find matching reads }
   ReturnValue(N, E)
   E ← Next(E)

if((Level > 0 OR LatestWrite(N)) AND (Level < MAXLEVEL))
   if(LSB(N) = 0)                                     { Propagate to next level }
     PropagateLeft(N)
   else
     PropagateRight(N)
```

Figure 8.3: Algorithm for a twisted memory write operation

Reads from later in the virtual order, as indicated by their VTS must be resatisfied. Any which are dependent on a different, later write which has already executed will not have propagated to this level of the time-space cache.

The write is then propagated in the same way to level two. Another bit is removed from the address and another bit is prepended to the VTS, chosen as before.

The propagation process continues until the write reaches the bottom of the time-space cache. The bottom level (level two when four resource blocks are used) is connected to the usual non-temporal memory system, although writes are not propagated to the rest of the memory system at this point because they are still speculative.

Each bottom level cell of the time-space cache contains the latest write, if any exists, from each of the resource blocks for a subset of the memory address space. Each write has a VTS indicating the resource block it originated from.

8.2.2 Read

A read operation is also issued to the level zero cell attached to its originating resource block and recorded there. If a write earlier in the virtual order can be found in the level zero cell its value is returned and the operation finishes there.

If no matching write is found, however, the read is propagated to the next level in the time-space cache based on its address in the same way as for a write. The read is recorded in this cell too, although the frame number is not required. As with a write, one bit of the address becomes implicit in the twisted memory cell, and a one bit explicit VTS is attached to the access based on the connection it arrived at the cell on. Now the comparison with writes is based on the VTS rather than the frame number. Writes with the same VTS as the read can be ignored, since they would have matched at level zero if they were earlier. If an earlier match still cannot be found the read is propagated to the next level. The most recent memory access in Figure 8.4 is a read from resource block three, shown in bold. It matches with a write from resource block two at level one and does not propagate any further.

Pseudocode for the algorithm to execute a read operation in a twisted memory cell is shown in Figure 8.5, using the procedures in Table 8.1.

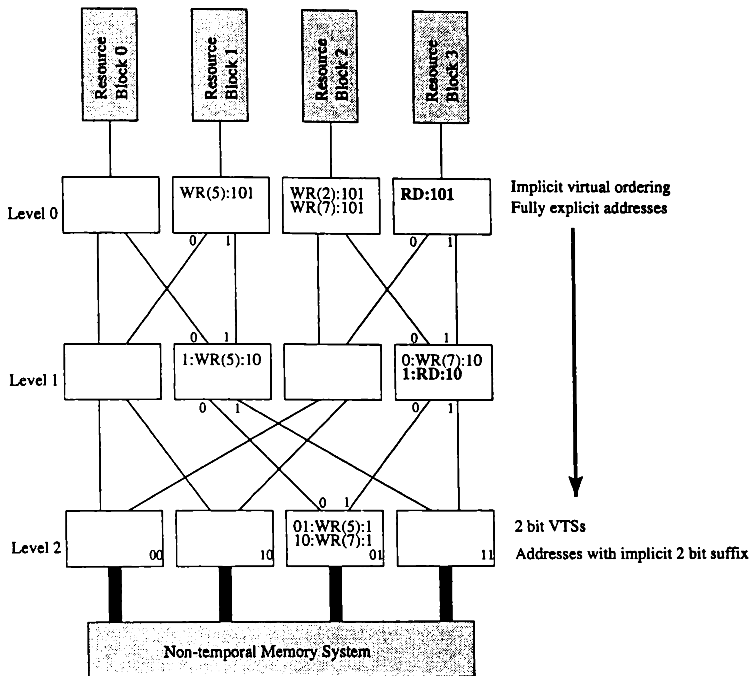


Figure 8.4: Propagation of a read through twisted memory

{ N is the new read }

$InsertInList(N)$
$E \leftarrow Previous(N)$
while($E \neq$ **null AND** $AccessType(E) \neq$ WRITE) { Find matching write }
  $E \leftarrow Previous(E)$

if($E \neq$ **null**)
  $ReturnValue(E, N)$
else
  if($LSB(N) = 0$) { Propagate to next level }
    $PropagateLeft(N)$
  else
    $PropagateRight(N)$

Figure 8.5: Algorithm for a twisted memory read operation

If a matching speculative write is not found when the bottom level has been searched then the read is issued to spatial memory in the same way as for non-speculative architectures. This means that a committed value is returned. Wherever the value is obtained it is returned through the network to the originating resource block by performing the reverse process of removing bits from the VTS and adding bits back onto the address.

8.2.3 Anti-write

An anti-write is issued when a write instruction or a frame containing a write instruction is rolled back. It propagates through the time-space cache in the same way as the original write did, but it removes the matching write from each level of the time-space cache.

The anti-write doesn't need to contain the value of the original write. Only the address and frame number is needed at level zero to match it with the appropriate write. Beyond level zero only the address and VTS is needed.

In addition to removing the write which has been rolled back, the previous write to that address from the same resource block must be propagated through the twisted memory to take its place. If there is no earlier write to that address from the same resource block, or the rolled back write has already been superseded, nothing is propagated.

It must also cause any dependent reads to be resatisfied by an earlier write. If there is an earlier write to the same address in the same cell that satisfied the read the first time it simply returns a read reply with the new value.

If the read cannot be satisfied in the same time-space cache cell it must continue to propagate through the time-space cache in the same way as a newly issued read, inserting an entry into the cells it reaches for the first time.

The read will never be resatisfied at an earlier level after the anti-write, otherwise the write being rolled back would already have been superseded for that read.

Figure 8.6 shows the algorithm for executing an anti-write operation in a twisted memory cell. The procedures used are described in Table 8.1. The example in Figure 8.7 shows a write to address 9 being rolled back in resource block two. The anti-write message (abbreviated AW) propagates through the time-space cache and annihilates with the matching

```
{ N is the new anti-write }
{ Level is the level of twisted memory being operated on }
{ MAXLEVEL is the bottom level of twisted memory }

W ← GetMatch(N)                                              { Remove matching write }
if(W ≠ null)
  Delete(W)
  P ← Previous(W)                                            { Find previous write }
  while(P ≠ null AND AccessType(P) ≠ WRITE )
     P ← Previous(P)
  E ← Next(W)
  while (AccessType(E) = READ )                              {Resatisfy reads }
     if(P ≠ null)
       ReturnValue(P, E)
     else if(LSB(E) = 0)                                     { Propagate read to next level }
       PropagateLeft(E)
     else
       PropagateRight(E)
                                                { Propagate previous write to next level }
  if((Level > 0 OR LatestWrite(W)) AND (Level < MAXLEVEL))
     if(LSB(W) = 0)
       PropagateLeft(W)
     else
       PropagateRight(W)

  if(LSB(N) = 0)                                             { Propagate anti-write to next level }
     PropagateLeft(N)
  else
     PropagateRight(N)
```

Figure 8.6: Algorithm for a twisted memory anti-write operation

write. Then the previous write to address 9 is propagated to replace the rolled back write. The read from resource block three is resatisfied with the new value of 2.

8.2.4 Anti-read

An anti-read is issued when a read instruction is rolled back. It removes the read entry from all the time-space cache cells it propagated to. The algorithm for doing this in each cell is shown in Figure 8.8.

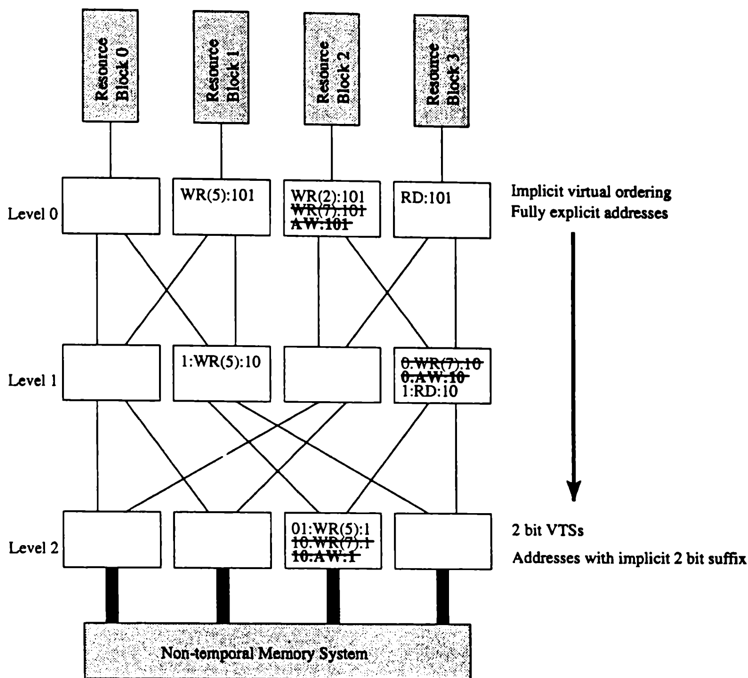
Read entries must also be removed from the lower level time-space cache cells when a write resatisfies a read at a higher level. In the example in Figure 8.9, following on from Figure 8.2, a write in resource block three resatisfies an already speculatively executed read from its own resource block, and the read had not previously matched a write until level one of the time-space cache. An anti-read must be issued to remove the read entries from all levels below level zero. If this is not done it may cause subsequent writes from other resource blocks to resatisfy the read incorrectly.

8.2.5 Fossil Collection

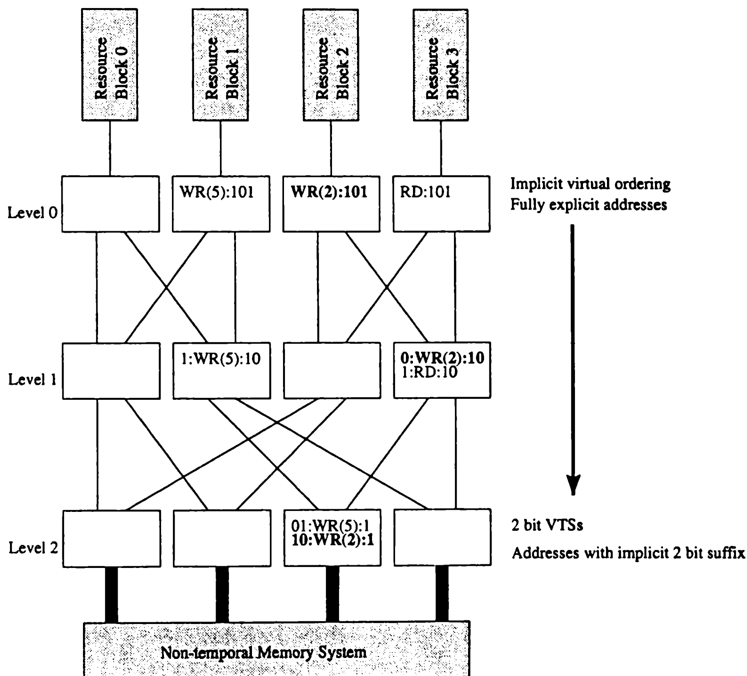
When a resource block is fossil collected the time-space cache entries generated by that resource block can also be fossil collected. This allows the VTS to be reused when the resource block is reused. It also frees space in the time-space cache occupied by memory operations which are no longer speculative.

Fossil collecting at the granularity of resource blocks allows the fossil collection message to simply propagate through the twisted memory, fossil collecting all messages with a matching VTS. As the fossil collection message propagates through the time-space cache it deletes all entries, both reads and writes, with a matching VTS. At the lowest level of the twisted memory, connected to spatial memory, writes matching the VTS are committed to spatial memory by simply writing them to spatial memory before deleting the copy in the time-space cache. This process is shown in the pseudocode in Figure 8.10.

As resource blocks are fossil collected they are available to be reused. This means that resource block zero, which started as the earliest resource block in the virtual order, will at



(a) Propagation and annihilation of an anti-write message



(b) Propagation of a replacement write

Figure 8.7: The effects of an anti-write in twisted memory

{ N is the new anti-read }

```
R ← GetMatch(N)                    { Remove matching read }
if(R ≠ null)
  Delete(R)
  if(LSB(N) = 0)                   { Propagate to next level }
    PropagateLeft(N)
  else
    PropagateRight(N)
```

Figure 8.8: Algorithm for a twisted memory anti-read operation

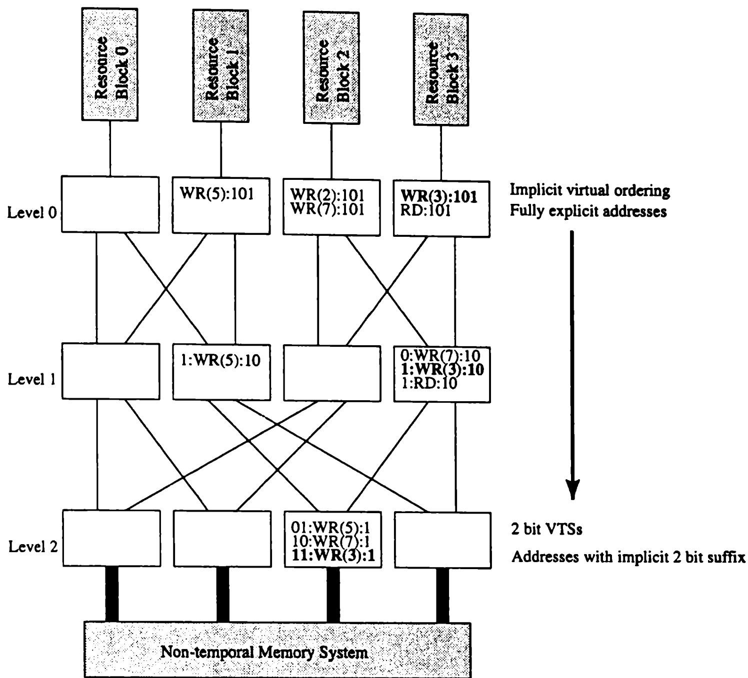
some point be the latest resource block. To ensure that memory accesses from it are treated as such in the time-space cache, an *epoch counter* is used to indicate whether the ordering of resource blocks has wrapped around. Only one bit is needed, which is incremented each time fossil collection begins at resource block zero again. If the epoch bits of the two resource blocks are different the usual ordering is reversed. The epoch bit is attached to VTSs of all time-space cache entries based on the epoch of their originating resource block.

This is also why the latest write from each resource block must be propagated right to the bottom of the time-space cache. If it could be guaranteed that resource blocks are always in a fixed relative order, only the latest write in each cell would need to be propagated to the next level.

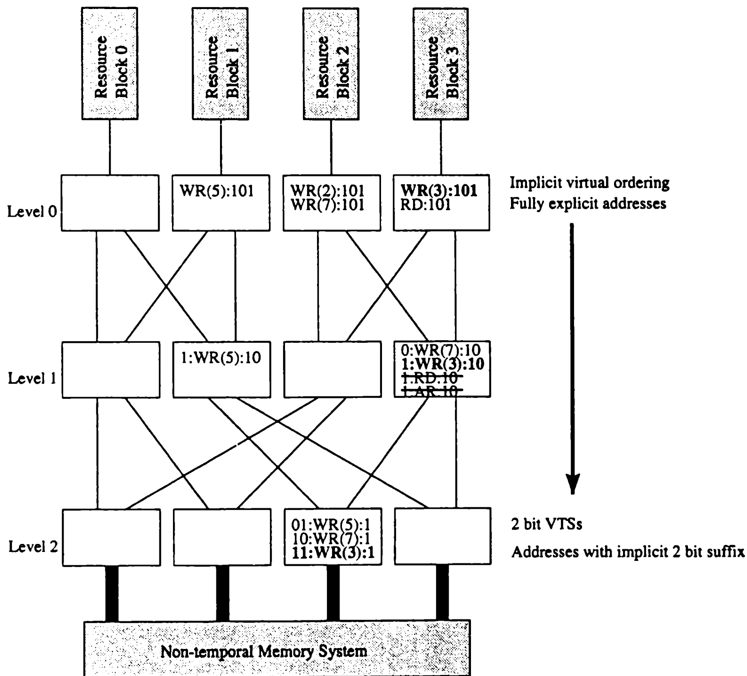
Figure 8.11 shows an example where resource block zero is the latest in the virtual order. The read in bold from resource block zero should return the value 7 from the write in resource block three, encountered at level two, not the write in resource block one with the value 5, encountered at level one, although it may be returned speculatively and rolled back. The epoch of each entry is recorded as */epoch* appended to each entry.

If the epoch bits of two memory accesses are different they should not produce a match unless the bottom level of the time-space cache has been reached, since there may be intervening accesses at a higher level. In the bottom level cells the latest accesses from each resource block have been sorted by address. All relevant speculative memory accesses to an address are recorded in a single cell, so a conclusive comparison may be made.

The fossil collection process described above commits stores to spatial memory immediately their resource block is fossil collected. In fact, it is not necessary to commit the writes



(a) Propagation of a new write



(b) Annihilation of falsely propagated read

Figure 8.9: The effects of an anti-read in twisted memory

```
{ N is the fossil collection message }
{ H is the earliest entry in the cell }
{ Level is the level of twisted memory being operated on }
{ MAXLEVEL is the bottom level of twisted memory }

while(VTS(H) ≤ VTS(N))
   if(Level = MAXLEVEL AND LatestWrite(H))        { Commit latest write }
      Commit(H)
   Delete(H)
   H ← Next(H)

if(Level < MAXLEVEL)
   PropagateLeft(N)                               { Propagate to next level }
   PropagateRight(N)
```

Figure 8.10: Algorithm for a twisted memory fossil collection

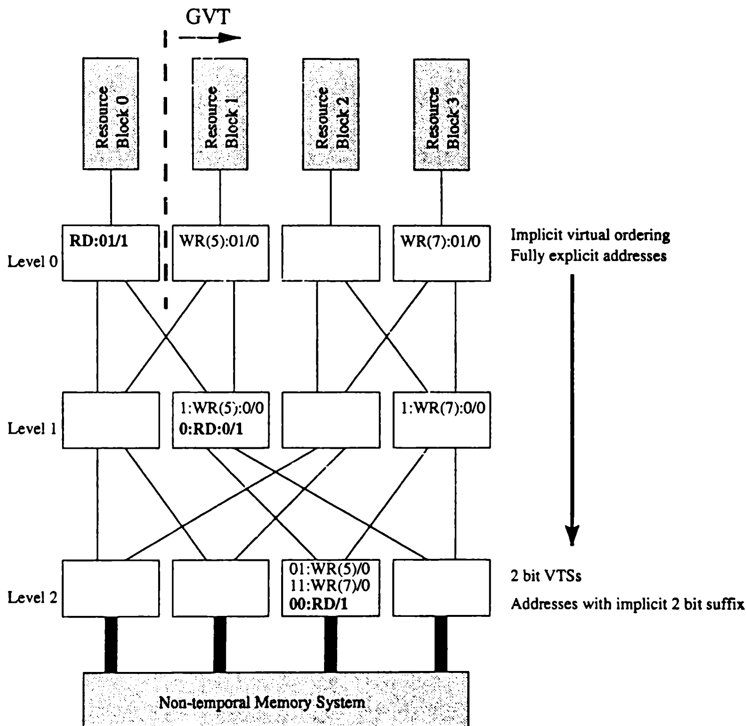


Figure 8.11: Ordering time-space cache entries using an epoch counter

until the resource block is allocated for reuse, or space needs to be freed in the time-space cache for new memory entries. This *lazy fossil collection* improves the effectiveness of the time-space cache, since more reads can be satisfied in the time-space cache rather than going to spatial memory. It also lowers the bandwidth to spatial memory consumed. If several resource blocks, or at least their memory accesses, are fossil collected at the same time, only the latest write in the virtual order to that address needs to be written to spatial memory. This has the potential to substantially lower the bandwidth required to spatial memory.

8.3 Evaluation

The remainder of this chapter presents some preliminary results, aimed at demonstrating the feasibility of the twisted memory concept. The main focus here is on the bandwidth consumed by speculative memory accesses, both within the time-space cache and from the time-space cache to the spatial memory hierarchy. This is a major area of concern for aggressively speculative execution, since the bandwidth required will be increased both by the high levels of parallelism being extracted and the increased number of accesses due to speculation. Bandwidth to spatial memory is the more important of the two, since the time-space cache is small and likely to be on the same chip as the CPU, while spatial memory will generate traffic off-chip at some level of the hierarchy.

Twisted memory is simulated using the WarpEngine virtual order simulator with transient state tracking. Twisted memory with four resource blocks, as described in Section 8.2, is used in all the results described below, both to narrow the design space, and because of simulation time limits. The time-space cache is modelled by fixed latency between levels of one cycle and an eight cycle latency to spatial memory. One cycle is the minimum possible latency between levels, and eight cycles was used for the latency to twisted memory to match the standard latency of a load, as given in Table 4.1. An arbitrary number of entries may be placed in each cell.

Lazy fossil collection is used in the simulations, although the virtual ordered simulation method requires that time-space cache entries are sometimes fossil collected earlier than absolutely necessary. The virtual order simulation paradigm also forces all writes to the time-space cache to be propagated all the way to the bottom of the twisted memory. It is not

possible to tell in advance whether any writes later in the virtual order (and hence later in the simulated order) will be written to the time-space cache at an earlier real time, meaning they would not need to be propagated.

The bandwidth measurements presented are pessimistic because of the limitations of virtual order simulation. Although no frame limits are placed on execution, it was shown in Chapter 7 that four resource blocks limit execution in similar ways to around a thousand frames. The write bandwidth to levels other than level zero and spatial memory is pessimistic because full write through to all levels of the time-space cache is forced by virtual order simulation. The number of writes at these levels will normally be substantially reduced.

For comparison, bandwidth consumed was also measured using a unified time-space cache, as has been used in the earlier parts of this thesis.

8.4 Memory Bandwidth

8.4.1 Average Memory Bandwidth

An important aspect of memory system bandwidth is the average bandwidth required. In the absence of a detailed low level model this gives an idea of whether the memory throughput can be sustained. Although simulations may show that the peak bandwidth required is beyond what could reasonably be supplied, if the average is within the capabilities execution can continue with minimal impact by delaying some accesses until the bandwidth is available.

The bandwidth used can be measured either as the bandwidth from each cell, or the total bandwidth consumed across the level. Total bandwidth is only meaningful where it is feeding into a single component, such as spatial memory, or as an aggregate where the bandwidth across each connection is similar.

The results in Figures 8.12 to 8.17 show the average read and write bandwidth required to execute the test suite using a four resource block twisted memory. The bandwidth has been aggregated over all the connections on each level for ease of presentation. This bandwidth is provided by several connections at each level, four originating from the resource blocks

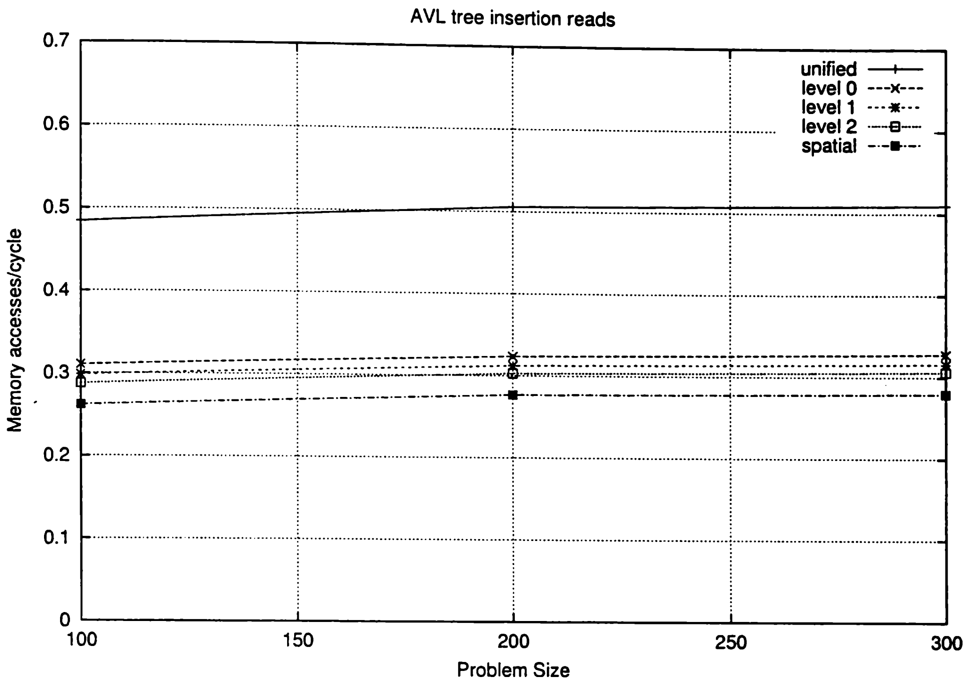
and terminating at spatial memory, and eight connections everywhere else. Many of these data paths are entirely independent, sharing neither source nor destination cell. The curves on the graphs are labeled with the level the access is being made to.

For comparison the same measurement for a unified time-space cache using four resource blocks has also been shown on the graphs. The discrepancies between accesses to the unified time-space cache and level zero of the twisted memory time-space cache are due to the change in access patterns caused by the different access latencies. The unified time-space cache was simulated with a store instruction latency of 2 cycles and a load latency of 8 cycles. An additional latency of 8 cycles was added for reading to or writing from memory to represent the long latency usually associated with a large monolithic memory. The twisted memory simulations use a cumulative latency of 1 cycle per level of twisted memory and an additional 8 cycles to spatial memory for both reads and writing of fossil collected values. The same instruction latency of 8 cycles is added for reads, but no additional latency is imposed on writes. This gives a maximum read latency of 19 cycles for values retrieved from spatial memory via twisted memory. This has little affect on the execution times for all the test programs except Gauss-Jordan and quicksort, where the times are lower for the twisted memory simulations. The total number of memory accesses for these two programs, and AVL tree insertion, are substantially reduced when twisted memory is used, while they are similar for the other programs.

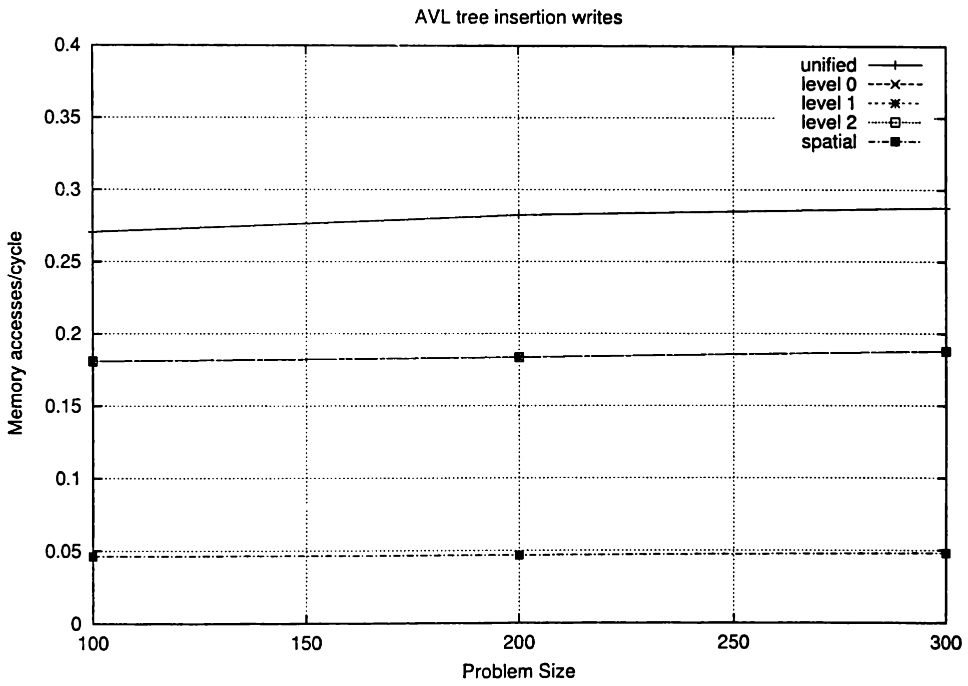
All the graphs show a drop in the bandwidth required between level zero and level one for reads, indicating that the level zero cells are performing the caching hoped for. The extent of the success of caching at level zero depends on the locality of accesses. In general the bandwidth requirements continue to decrease further from the resource blocks.

The write bandwidth required to spatial memory, caused by committing fossil collected values, is much lower than the total number of writes, showing write caching in the twisted memory again is effective. This could be improved even further by further attention to improving the simulation of lazy fossil collection.

For AVL, binary tree and Fibonacci the average number of memory accesses is well below one access initiated per cycle for both reads and writes, a level that is easily sustainable even assuming accesses have a latency of several cycles. For quicksort the averages fall in the range of several accesses initiated per cycle, except for values being retired to spatial mem-

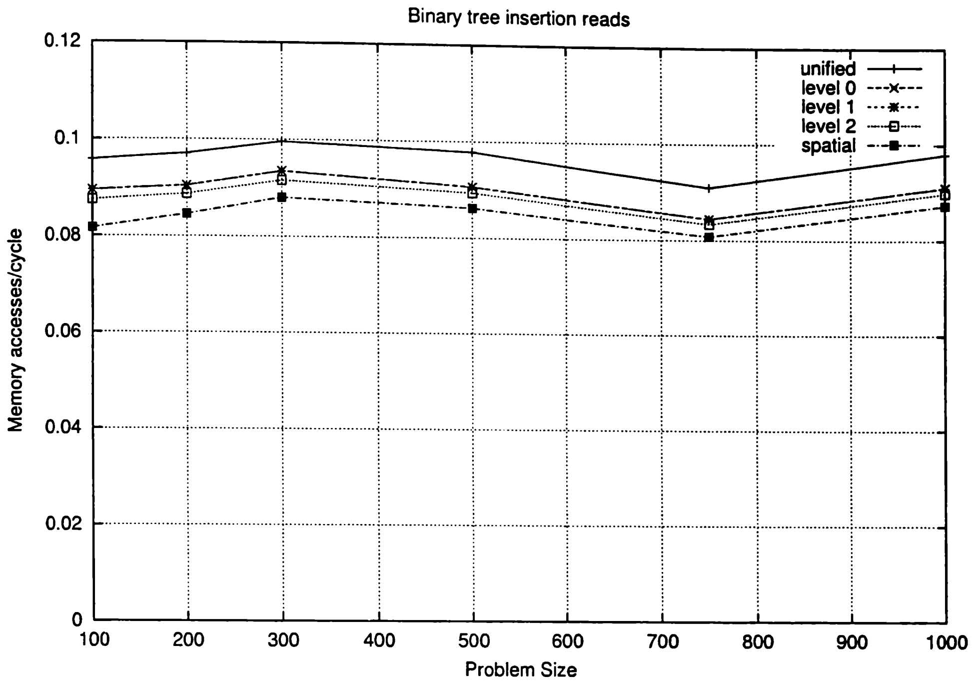


(a) Reads

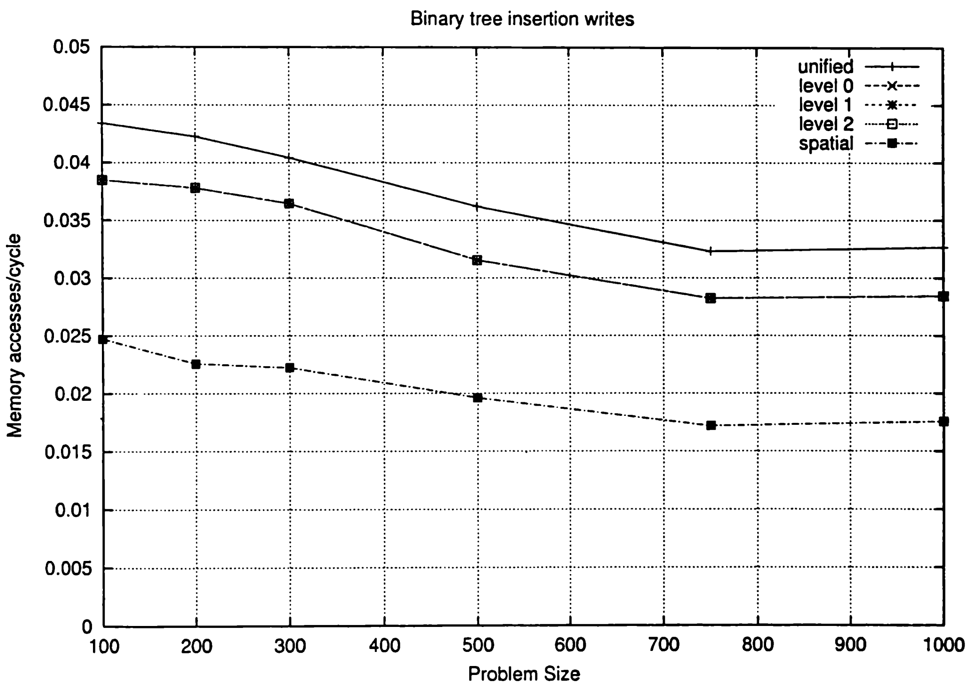


(b) Writes

Figure 8.12: AVL average memory bandwidth per cycle for twisted memory

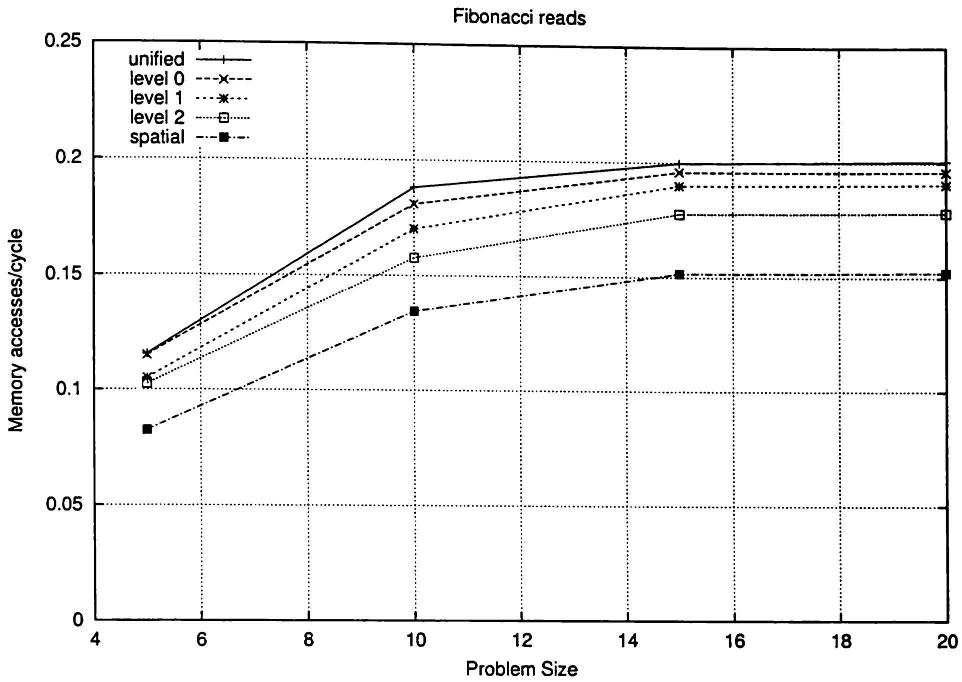


(a) Reads

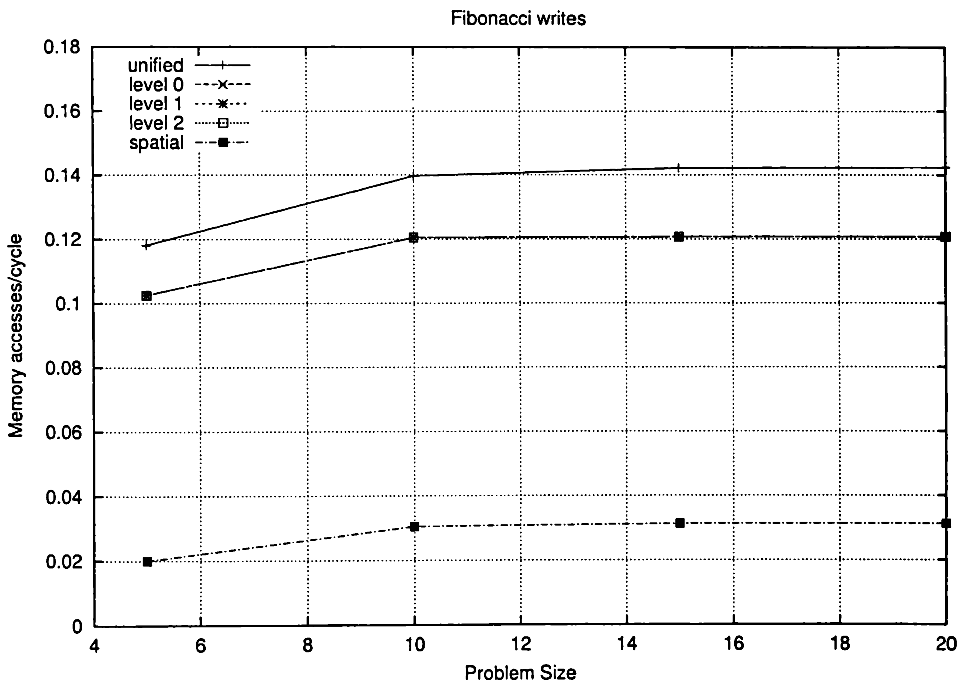


(b) Writes

Figure 8.13: Binary tree average memory bandwidth per cycle for twisted memory

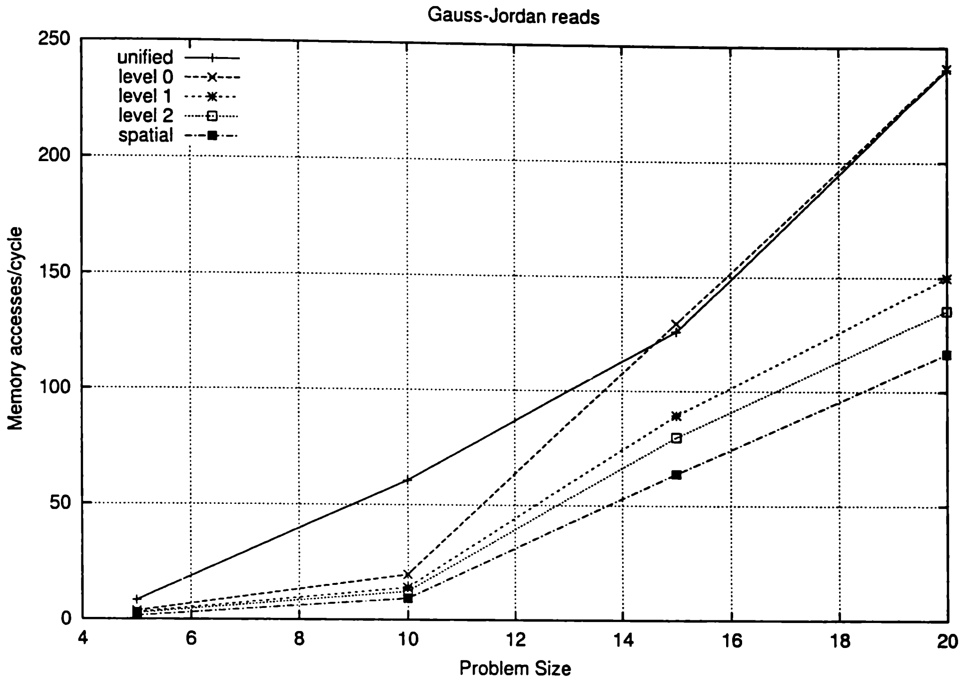


(a) Reads

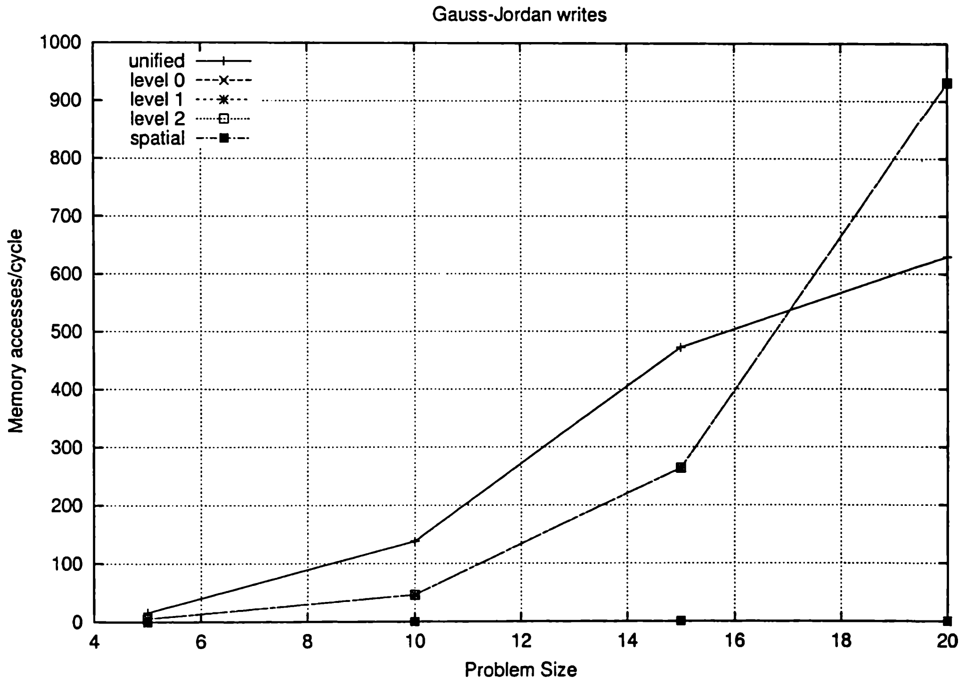


(b) Writes

Figure 8.14: Fibonacci average memory bandwidth per cycle for twisted memory

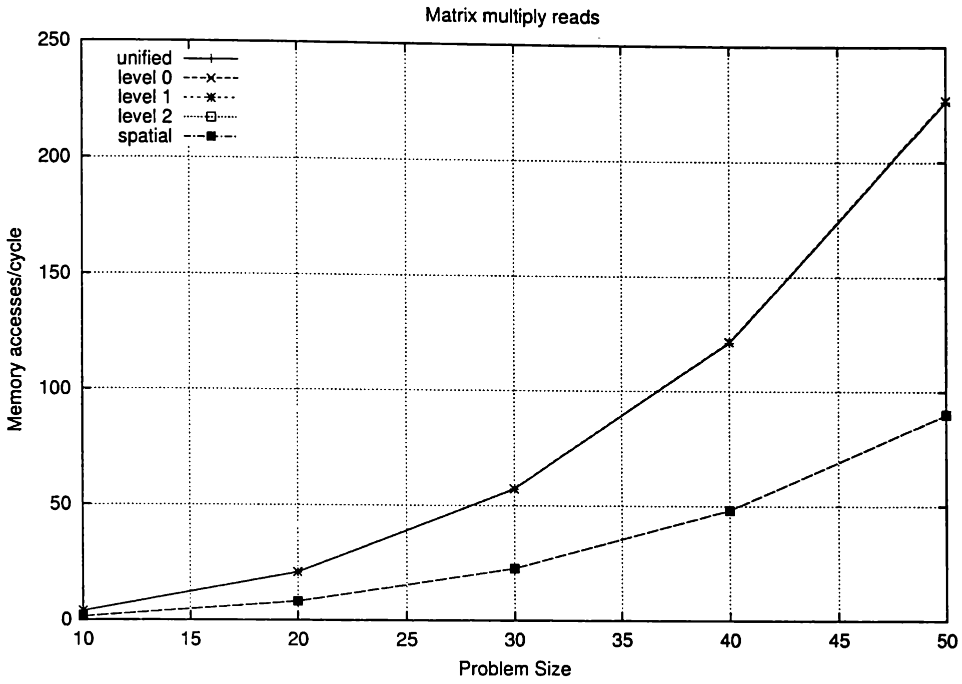


(a) Reads

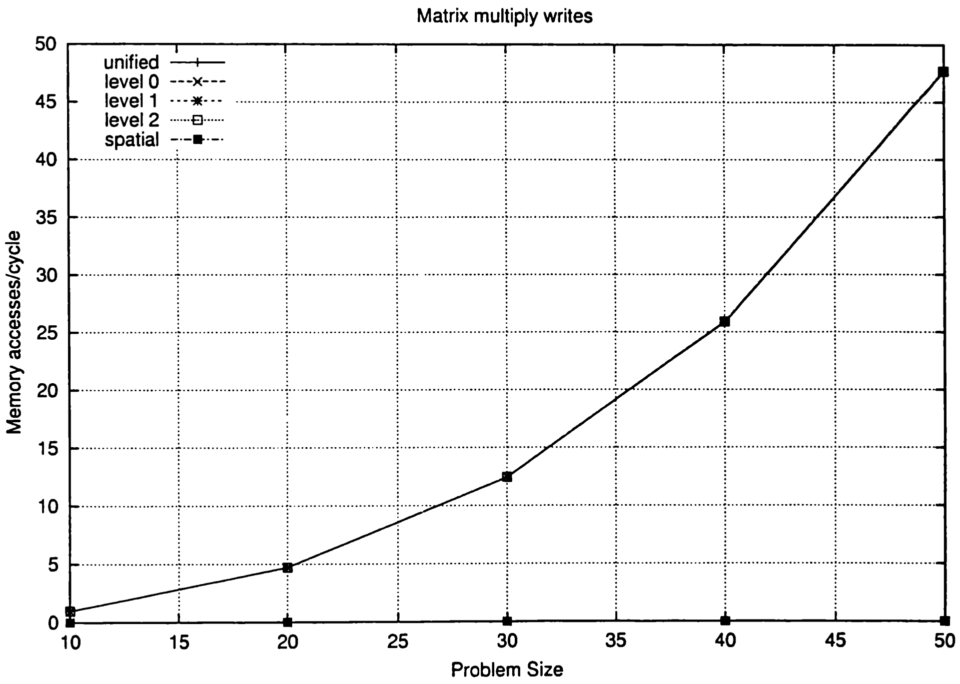


(b) Writes

Figure 8.15: Gauss-Jordan average memory bandwidth per cycle for twisted memory

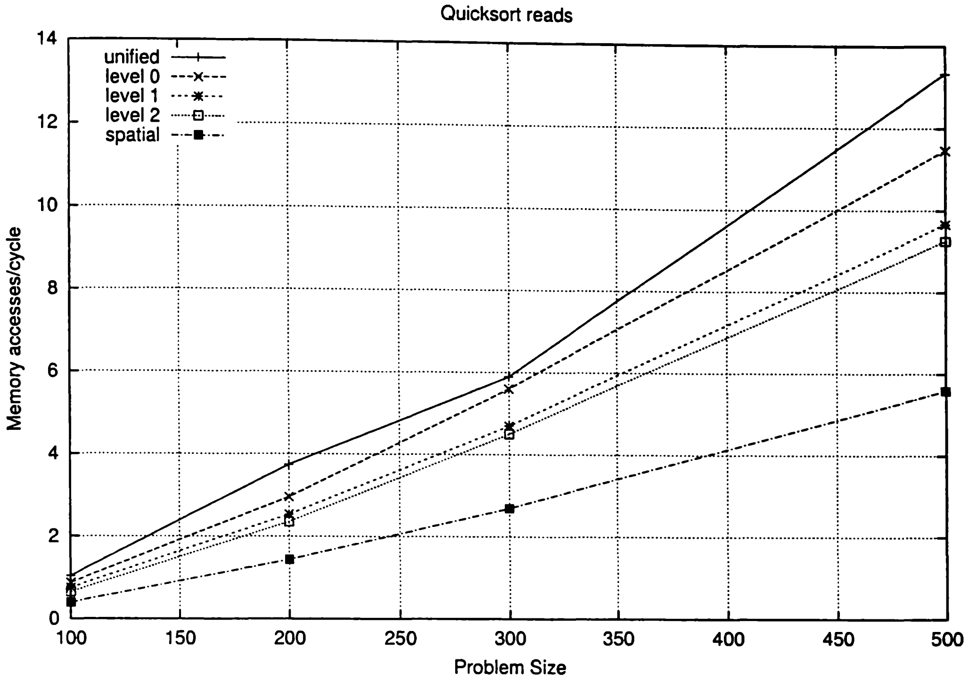


(a) Reads

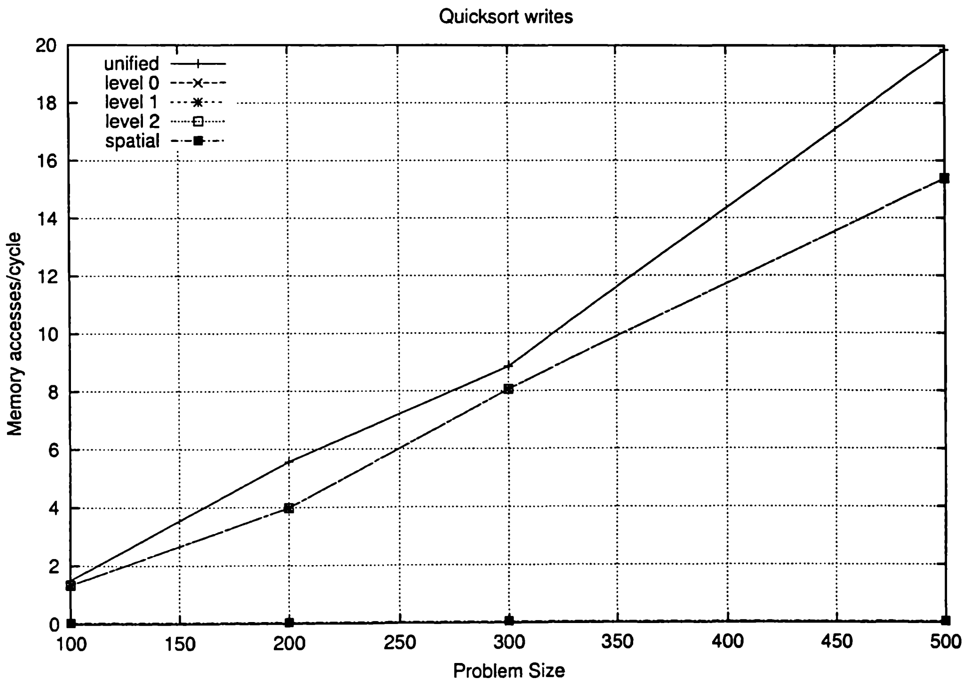


(b) Writes

Figure 8.16: Matrix multiply average memory bandwidth per cycle for twisted memory



(a) Reads



(b) Writes

Figure 8.17: Quicksort average memory bandwidth per cycle for twisted memory

ory which are well below one access initiated per cycle. This level may be more difficult to sustain, although it may be possible with careful memory design. Matrix multiply and Gauss-Jordan require very high bandwidth to memory to fully exploit the levels of parallelism achievable with resource blocks. The high memory bandwidth is caused by the large number of frames used by these programs in resource blocks because they are statically analyzable. When the frames available are limited the parallelism will be constrained to levels far below those simulated here.

8.4.2 Bandwidth Profile

Important as the average bandwidth required is, the temporal distribution of accesses is also important. In a real system the bandwidth limit will be a maximum number of concurrent accesses from each time-space cache cell, so the peak bandwidth requirements become important.

In the simulations in this chapter no bandwidth limit is applied, instead the peak bandwidth that can be utilized is measured. If this peak exceeds the bandwidth available, some of the accesses will have to be delayed. If the peaks occur sparsely and the average bandwidth requirements of the local region are low the accesses will not have to be delayed for long, and performance will not be heavily impacted. If there are sustained regions of high bandwidth requirement then memory may become the performance bottleneck.

Figures D.1 to D.36 show the number of memory reads and writes initiated on each cycle as a bar with a width of one cycle. There is a separate graph for each level of the time-space cache, with each twisted memory cell at that level (referred to as resblock 0 to resblock 3) plotted separately. Writes to spatial memory are not shown on a cell by cell basis because the bandwidth will probably constrained by spatial memory, where all the bottom level twisted memory cells are writing to a single unit. The graph of the aggregated accesses to spatial memory is also included, plotted alongside the number of frames actively executing at that time, and the graph of the access profile to a unified time-space cache. The bandwidth due to fossil collection is recorded on this graph.

The test programs used are the same ones chosen in Chapter 7, with an arbitrary data set size chosen for each available program.

Littin [2000] showed that in the count of committed instructions there are more reads than writes, usually around double the number. However, this may not necessarily be the case in the memory bandwidth profile, due to speculatively executed instructions and the differing caching characteristics of twisted memory for reads and writes.

The important feature of the bandwidth profiles is the general trend of the accesses, the peaks and the relative changes between the levels. Although the graphs appear to be densely packed with multiple accesses on every cycle this is an artifact of the graphing resolution. There are actually many interspersed cycles with no memory accesses, as can be seen from the enlarged regions of the level zero graphs in Figures D.37 to D.42. A region of five hundred cycles has been selected from each where the number of memory accesses is highest. An indication of this spacing can also been seen in the average bandwidths below one instruction per cycle for AVL, binary tree and Fibonacci seen in Section 8.4.1. With the extremely high parallelism of matrix multiply and Gauss-Jordan, a high sustained bandwidth is necessary to support the speedup gained in the simulations.

In line with the average bandwidth results reported above, the peak bandwidth required for reads consistently declines at the level zero cells, showing the effectiveness of caching at that level. Elsewhere in the twisted memory the accesses are largely being redistributed between the different cells, since almost all accesses are propagated from level one to level two. In some cases accesses do not propagate all the way to level two because they are rolled back before they get there.

In most cases the accesses are evenly distributed amongst the address sets used at level two, but AVL shows a bias towards resource block three due to the data structures used for storing the nodes of the tree. The pointers to other nodes of the tree are altered much more often than the value itself. Matrix multiply shows an excellent example of memory accesses predominantly from a single resource block being redistributed through the twisted memory. The memory accesses are mainly from resource block zero early in the program and resource block zero towards the end, but throughout the program are evenly distributed amongst all the cells at level two. The uneven distribution of memory accesses among resource blocks is caused by the large number of frames used in each resource block, and only resource block zero is being used early in the program's execution.

For AVL, binary tree and Fibonacci the bandwidth consumed to spatial memory appears

remarkably similar to, although slightly lower than, the bandwidth used by a unified time-space cache. This is a misleading impression, as shown by the significantly lower averages. The relatively high peaks in the spatial memory write bandwidth are caused by a large number of frames being fossil collected at once and the writes being committed together, with many idle cycles in between. When properly managed commitment of values is not a time critical operation, and should be easily spread across the available cycles.

8.5 Optimizations

While the descriptions of the memory operations in Section 8.2 will allow the time-space cache to function correctly, there are a number of optimizations that can be used to improve the efficiency. The evaluation in this chapter does not consider these optimizations, they are left for future investigation.

In the same way that lazy fossil collection takes advantage of unused time-space cache storage to improve performance, values can be cached in cells other than those they were originally written to. When a read is satisfied in a cell below level zero the read reply is transmitted back through cells which do not contain that value. By inserting a write entry into these cells the value can be accessed again more quickly, as the principle of locality suggests is likely to be required. Care must be taken to ensure that the entry is marked appropriately according to its place in the virtual order.

Similar to lazy fossil collection, *lazy re-execution* only re-executes instructions when the input operands change, rather than every time they are updated. In the context of memory accesses this means that a read is only resatisfied when the new write value it is dependent on is different to the previous value returned. This can be achieved by storing the value returned in each read entry in the time-space cache for comparison before the read is resatisfied.

Twisted memory, as described above, operates with resource blocks arranged in a fixed order circular queue, dictated by the fixed network of time-space cache cells. This imposes some restrictions on parallel execution, as was shown in Chapter 7. The resource blocks could be reordered, either by changing the physical connections, or by using a lookup table to redefine the order, but this detracts from the simplicity of twisted memory that is so attractive.

A more reasonable alternative is to use static analysis to estimate the number of resource blocks a subtree will require, and reserve them before allocating resource blocks to following subtrees. This is a higher level analysis of a similar nature to that used to allocate frames in resource blocks and could be performed by a compiler.

The organization within a twisted memory cell assumed so far is that reads and writes are stored in a single structure in their virtual order, but this need not be the case. Splitting the load and store lists could have performance benefits, since only the stores need to be examined to satisfy a read, and only the loads need to be examined to determine rollbacks due to a write, as long as the succeeding write is known.

By placing the time-space cache in the direct path to spatial memory the maximum latency of a memory access has been increased by the latency required to propagate through the time-space cache. The additional latency will be most noticeable for loads, and it may be worthwhile to issue loads to spatial memory in parallel with the request to the time-space cache. If the time-space cache provides a result the resource block would then ignore or roll back the value provided by spatial memory. The trade off is that additional memory accesses to spatial memory will be generated.

8.6 Summary

This chapter has proposed a hierarchical speculative memory system, known as a *twisted memory time-space cache*. The hierarchy begins with the previously developed resource blocks to partition the program into multiple instruction windows, which may still be split windows internally, and whose resource usage may be allocated linearly.

Dependent accesses in the same resource block are found using the linear frame ordering, while those in other resource blocks are found by propagating the accesses through the network of twisted memory cells, searching for matches with accesses from resource blocks progressively further apart. Only those accesses which may potentially match with accesses from other resource blocks need to be propagated through the hierarchy.

Twisted memory relies on a mixture of explicit VTSs and physical ordering to maintain the virtual order of memory accesses. This provides the flexibility and scalability of explicit

VTSs, but reduces the number of VTSs required, and provides a fast method of matching accesses in the same resource block, which the principle of locality suggests are likely to be most frequent.

The simulation results presented take a preliminary look at the bandwidth requirements in the the twisted memory network. The measurements show that the bandwidth required reduces substantially in the levels of twisted memory further away from the resource blocks. Both reads and writes benefit greatly from caching in the level zero cells and very few of the values written to the time-space cache are written out to spatial memory. Although fossil collection does tend to cause commitment to occur in bursts this can easily be smoothed by implementing lazy commitment appropriately.

Virtual order simulation forces the results presented here to be pessimistic in regard to the bandwidth required between lower levels of the time-space cache. Caching at these levels is expected to reduce the already low bandwidth even further, but can not be simulated with this methodology.

In most cases the bandwidth required between each cell is well below one access initiated per cycle and is easily achievable for programs executing under realistic resource constraints. In addition to a low average bandwidth, the bandwidth peaks are well spread, so smoothing of the access pattern by stalling excess accesses should be possible with minimal performance impact.

Twisted memory has the additional advantage that it can be implemented as a distributed memory system, spreading the bandwidth across multiple connections and memory cells. Twisted memory could be potentially used in a multiple chip implementation, connected at the lower levels of the hierarchy.

A key advantage of twisted memory is its scalable design. The modular arrangement means that extending the design to support more resource blocks is simply a matter of connecting more twisted memory cells in the standard omega network topology. The results presented here show the accesses are distributed evenly across the network even when the initial access pattern is asymmetric at the resource blocks. This also enhances scalability.

As the number of resource blocks is increased the memory bandwidth required will also increase. However, the number of physical channels per level also increases proportionally,

so the bandwidth per channel remains approximately constant. Increased bandwidth will be required to spatial memory, although this is ameliorated by the demonstrated caching effect of twisted memory.

The main obstacle to increasing the size of twisted memory is that each extra level of cells increases the latency to the bottom of the time-space cache and, ultimately to spatial memory. It is possible, though, to modify the interconnection topology to increase the number of connections between cells at each level. This decreases the number of levels required, at the expense of more complex routing logic in each cell.

A number of possible optimizations to twisted memory have also been identified, which have the potential to improve performance and reduce the bandwidth required. These will be investigated in future research.

The results obtained indicate that a twisted memory time-space cache can be used as an effective virtually ordered memory system for use with an aggressively speculative processor. The bandwidth requirements are feasible when executing with realistic levels of resources, while the parallel execution speedup is substantially improved over currently available systems.

Chapter 9

Summary and Conclusions

The aim of this thesis has been to develop a method for maintaining the dependencies between memory accesses in an aggressively speculative, out-of-order processor. The *Warp-Engine* architecture has been used as the basis for this research because it provides a speculative execution platform with very few basic restrictions on extracting parallelism.

It is important that any techniques developed are effective for much higher levels of parallelism than those currently extracted by production architectures. Only by doing this will they remain relevant as the state of art in CPU core design continues to advance.

The focus in this thesis has been on methods of tracking the *virtual order* of a large number of instructions executed speculatively and out-of-order. This is important for a speculative memory system which is required to store multiple values for each address for different periods in the virtual order, and to match each memory access with the store it depends on, or the loads that depend upon it.

9.1 Virtual Timestamps

Extensive investigation was performed into tagging blocks of instructions with explicit *virtual timestamps (VTSs)* to indicate their virtual order. By applying explicit tags to events any arbitrary pair can be compared in isolation, suggesting that the scheme will scale to large numbers of events.

The central problem is efficiently allocating VTSs, a linear resource, to out-of-order events.

The simplest schemes proposed were found to require long strings for VTSs, which place a heavy burden on the storage and communication components of the processor.

More sophisticated schemes were developed employing compiler analysis to determine the size of regions of the execution tree. At runtime the processor allocates a range of VTSs based on the maximum size of the subtree determined by the compiler.

Some program structures are more amenable to this analysis than others, however sixteen to thirty two bit VTSs were found to allow substantial execution speedup. These *variable range VTSs* show promise and are deserving of future exploration. The compile-time analysis performed here is basic and VTSs would benefit from more sophisticated analysis algorithms.

9.2 Resource Blocks

The same subtree size estimation used for allocating a variable range of VTSs was also used to partition a program into execution tasks. Since the size of the subtree is a measure of the resources required to execute that region of the program, this schedules instructions for execution based on resource requirements.

A new task is started whenever the resource requirements of a subtree cannot be determined by the compiler. This creates speculative tasks which are separated by regions of code of unknown size. Each of these tasks is assigned to a block of execution units known as a *resource block.*

Since the processor has determined an upper bound on resource requirements for all but the most speculative event in a resource block the frames in a resource block can be allocated linearly. This allows a fixed order array of frames to be used within a resource block. Thus, VTSs are only needed to maintain the ordering between resource blocks, and can be much shorter.

The novel metric of *speculation effectiveness*, the ratio of committed instructions to issued instructions, was introduced to measure the efficiency with which speculative execution techniques use execution resources. While resource blocks restrict the speculative parallelism that can be extracted, simulations show that using resource blocks gives a much

higher speculation effectiveness than frame limited execution. Speculation effectiveness is becoming increasingly important in maximizing performance as architectures make more instructions available for speculative execution with limited resources.

9.3 Virtually Ordered Memory System

The thesis culminates in a design of a virtually ordered memory system, which allows multiple values, applying at different virtual times, to be stored for each memory location. The proposed memory system, called *twisted memory*, builds on the hierarchy already established with resource blocks.

Despite the limitations of virtual order simulation, the results presented show that the bandwidth required by speculative accesses is greatly reduced beyond the first level of twisted memory. Only a small proportion of stores are ever committed to spatial memory. For realistic resource constraints the bandwidth required at all points in the memory system is within a feasible range.

These preliminary investigations show that twisted memory has significant advantages over a monolithic time-space cache. The mixture of explicit VTSs and physical ordering that twisted memory relies on also allows it to be easily scaled by increasing the size of the standard network topology used. Twisted memory is a promising approach to a distributed, scalable virtually ordered memory system for an aggressively speculative processor.

9.4 Conclusions

The results of this thesis show that explicit tags can be used to track the virtual order of memory accesses in a distributed, scalable manner. However, naive VTS allocation schemes severely constrain the parallelism extracted. Scheduling schemes which perform analysis of resource requirements are required in order to maintain a speedup of an order of magnitude over sequential execution. It has been shown that this analysis can be done at compile time.

It has also been shown that the same analysis can be used to prioritize instructions for speculative execution, leading to a more efficient use of speculative execution resources. Having a

reliable estimate of resource requirements allows instructions to be allocated linearly within the local region, simplifying the hardware without substantially constraining performance.

By combining the analysis techniques with traditionally hardware ordering techniques a distributed scalable virtual order memory system has been demonstrated which does not require unreasonable bandwidth to support the speculative execution of the WarpEngine.

The current trend in processor design is to use larger instruction windows to extract more ILP. This is being done primarily through more aggressive speculative execution, particularly exploiting control independence to create split instruction windows.

Maintaining memory dependencies in a very large instruction window will become a critical problem in the next few years, which will require new techniques, such as those presented in this thesis, to solve. Split instruction windows add to this problem because speculative memory accesses must be tracked over a larger part of the program's lifetime.

Many processors, particularly those that utilize VLIW techniques, rely heavily on compiler analysis to augment the processor's ability to extract parallelism. The instruction scheduling techniques used in twisted memory follow this trend and will be able to leverage further advances in compiler technology.

9.5 Future Work

The results in this thesis were all obtained using a virtual order simulator for the WarpEngine. While this simulation method allows fast simulation it has limited ability to model transient events, particularly transient memory accesses, which may have complex real time interactions. More detailed design and modeling of the components of the WarpEngine must be done to validate the assumptions used in the simulations.

Using a real time simulator to simulate twisted memory will allow the use of the cells as caches to be further explored, along with write through and commitment policies. Further, little attention has been paid to capacity requirements of twisted memory.

A number of refinements to twisted memory are identified in Section 8.5. Further investigation of these features has the potential to further improve the performance of the memory

system.

There is also scope for further development of the analysis techniques introduced in this thesis. For example, using a best estimate, rather than an upper bound of resource requirements is an interesting alternative. The development of a compiler is necessary to refine the analysis techniques. This will also allow "real world" problems and standard benchmarks to replace the small programs used here.

Now that the principles of twisted memory have been demonstrated using the WarpEngine as platform it can be adapted to other speculative execution architectures. Since most architectures do not support the same range of speculative execution features as the WarpEngine some adaptation will be required.

Appendix A
WarpEngine Instruction Set

This appendix contains the WarpEngine instruction set. This is version 2 (as of November 2, 1995 [Cleary, 1995]) and is the instruction set used throughout this thesis.

The labels in the *inputs* and *outputs* columns of the following table have these meanings:

`a`	32 bit address
`v`	32 bit value (uninterpreted)
`f`	32 bit floating point value
`d`	destination register number
`cop`	comparison sub-operator: $<, \leq, >, \geq, =, \neq$
`sop`	split operator
`c`	2 bit specifier of child number ranging over 0–3
`cr`	child frame address hardware reference

The control and data movement instructions are conditionally executed, with the remainder unconditionally executed. Conditional execution is achieved by setting/resetting the special *s* register that is associated with each slot in a frame. These 1-*bit s* registers can be read and written as conventional registers.

inst.	*inputs*	*outputs*	*description*
control			
`child`	`a1 a2`	`c ds`	Execute child c at address (a1). a2 contains the estimated subtree size for variable range VTS allocation. The ds field specifies a 16 bit mask, 1 bit for each slot in the frame. If a bit is on then a child will set both the S-register for that slot and send a CR word to the second register.
data movement			
`st`	`v1 a2`	`a`	Store v1 at address (a2+a×4) (just after current time).
`mv`	`v1` `cr2`	`d`	Store value v1 into destination register d of child referred to by cr2.
`ma`	`a1` `cr2`	`d a`	Load value at address (a1+a×4) at time just before execution of child referred to by cr2 into destination d of child referred to by cr2.
comparison			

`cmp`	`v1 v2`	`cop1`
		`d1...2`
		`cop2 d3`
			Compare v1 and v2 using cop. The result of cop1 is sent to d1 and its complement to d2. The result of cop2 is sent to d3.

logical			
`and`	`v1 v2`	`d1...4`	Take bitwise AND of v1 and v2 and move result.
`or`	`v1 v2`	`d1...4`	Take bitwise OR of v1 and v2 and move result.
`xor`	`v1 v2`	`d1...4`	Take bitwise XOR of v1 and v2 and move result.

inst.	*inputs*	*outputs*	*description*
arithmetic			
`add`	`v1 v2`	`d1...4`	Add v1 and v2. Move sum to d1...3 and overflow to d4.
`sub`	`v1 v2`	`d1...4`	Subtract v2 from v1. Move difference to d1...3 and overflow to d4.
`mul`	`v1 v2`	`d1...4`	Multiply v1 by v2 move the low order 32 bits of the result to d1...3 and the high order 32 bits to d4.
`div`	`v1 v2`	`d1...4`	Divide v1 by v2. Move integer part of the result to d1...3 and the remainder to d4.

`split`	`v1 v2`	`sop1 d1`
		`sop2 d2`
		`sop3 d3`
			Divide the word v1, about bit b=(v2 mod 32). Each sop contains two bits. The first says whether the left or right part of the word is being referenced, the second says whether that part is to be justified left or right in the result.

floating point			
`addf`	`f1 f2`	`d1...4`	Add f1 and f2. Move sum to d1...4.
`subf`	`f1 f2`	`d1...4`	Subtract f2 from f1. Move difference to d1...4.
`mulf`	`f1 f2`	`d1...4`	Multiply f1 by f2 move the move the result to d1...4.
`divf`	`f1 f2`	`d1...4`	Divide f1 by f2. Move the result to d1...4.
`f2i`	`f1 f2`	`d1...4`	Divide f1 by f2. Move the integer part of the result to d1...2 (as a 32-bit integer) and the fractional part (as a floating point number) to d3...4.
`i2f`	`v1 v2`	`d1...4`	Multiply the (integers) v1 and v2, convert the result to a float (avoiding loss of precision as far as possible) and send the result to d1...4.

Conditional execution

For an example of conditional execution and the semantics of time using the WarpEngine instruction set consider the code in Figure A.1. The corresponding piece of WarpEngine assembly code is given in Figure A.2, where `@label` represents a data location, `&label` defines a block boundary, `label:` is an instruction label, and `?inst` denotes a conditional instruction.

```
if (x > 100) {
    count++;
}
extra = count;
```

Figure A.1: Conditional C code.

```
&start
        0:child   &if      0        0 0
        ma        @x       0:       x 0
        1:child   &next    0        1 0
        ma        @count   1:       count 0

&if
        cmp       x        100      > 0:
        0:?child  &then    0        0 0
        ma        @count   0:       count 0

&then
        add       count    1        cp1
        st        cp1      @count   0

&next
        st        count    @extra   0
```

Figure A.2: WarpEngine assembly for the C code in Figure A.1.

In the WarpEngine assembly code the `&start` block fires 2 children. The first (`&if`) performs the comparison of x to 100. If this comparison is true the`&then` child is executed performing the conditional increment to `count`. The second child of `&start` performs the assignment of `count` to `extra`. The code blocks `&if` and `&next` start executing in parallel (if execution resources permit). If the outcome of the conditional test on `x` is true the `count` variable is re-read and the `st` in `&next` is re-executed.

Appendix B
Test Code

This appendix contains C source code for the test problem algorithms used throughout this thesis. The C code and corresponding WarpEngine assembly code can be found at [Littin, 1999].

Matrix multiplication

```
float A[N][N], B[N][N], C[N][N];

void Mult(void) {
    int i,j,k;
    float t;

    for (i=0;i<N;i++)
        for (j=0;j<N;j++) {
            t = 0;
            for (k=0;k<N;k++)
                t += A[i][k] * B[k][j];
            C[i][j] = t;
        }
}
```

Gauss-Jordan elimination [Press, 1992]

```
void gaussj(float *a, int n) {
  float big, dum, pivinv;
  long int i, icol, irow, j, k, l, ll;
  long int indxc[N], indxr[N], ipiv[N];

  for (j=0;j<n;j++) {
    indxc[j] = 0;
    indxr[j] = 0;
    ipiv[j] = 0;
  }
  for (i=0;i<n;i++) {
    big = 0.0;
    for (j=0;j<n;j++)
      if (ipiv[j] != 1)
        for (k=0;k<n;k++)
          if (ipiv[k] == 0) {
            if (ABS(a[j*N+k]) >= big) {
              big = ABS(a[j*N+k]);
              irow = j;
              icol = k;
            }
          }
          else if (ipiv[k] > 1) {
            printf("pause 1 in GAUSSJ - singular matrix\n");
            getc(stdin);
          }
    ipiv[icol] = ipiv[icol] + 1;
    if (irow != icol) {
      for (l=0;l<n;l++) {
        dum = a[irow*N+l];
        a[irow*N+l] = a[icol*N+l];
        a[icol*N+l] = dum;
      }
    }
    indxr[i] = irow;
    indxc[i] = icol;
    if (a[icol*N+icol] == 0.0) {
      printf("pause 2 in GAUSSJ - singular matrix\n");
      getc(stdin);
    }
    pivinv = 1.0 / a[icol*N+icol];
    a[icol*N+icol] = 1.0;
    for (l=0;l<n;l++) {
      a[icol*N+l] = a[icol*N+l]*pivinv;
    }
    for (ll=0;ll<n;ll++)
      if (ll != icol) {
        dum = a[ll*N+icol];
        a[ll*N+icol] = 0.0;
        for (l=0;l<n;l++) {
          a[ll*N+l] = a[ll*N+l] - a[icol*N+l] * dum;
        }
      }
  }
  for (l=n-1;l>=0;l--)
    if (indxr[l] != indxc[l])
      for (k=0;k<n;k++) {
        dum = a[k*N+indxr[l]];
        a[k*N+indxr[l]] = a[k*N+indxc[l]];
        a[k*N+indxc[l]] = dum;
      }
}
```

Naive binary tree insertion

```
typedef struct node node;

struct node {
    int key;
    node *left, *right;
};

node *New(int key) {
    node *new_node = (node *)malloc(sizeof(node));

    new_node->key = key;
    new_node->left = NULL;
    new_node->right = NULL;
    return new_node;
}

void Insert(node **root, int key) {
    if ((*root) == NULL)
        *root = New(key);
    else if (key < (*root)->key)
        Insert(&((*root)->left),key);
    else if (key > (*root)->key)
        Insert(&((*root)->right),key);
}
```

AVL binary tree insertion[Lewis and Denenberg, 1991]

```
struct node {
    int key;
    int bal;
    node *left, *right;
};

node *New(int key) {
  node *n = (node *)malloc(sizeof(node));

  n->key = key;
  n->bal = 0;
  n->left = NULL;
  n->right = NULL;
  return n;
}

void ShiftNode(int *d,node **c,int key,node **p) {
  if (key == (*p)->key) {
    *d = 0;
    *c = *p;
  }
  else if (key < (*p)->key) {
    *d = -1;
    *c = (*p)->left;
  }
  else {
    *d = 1;
    *c = (*p)->right;
  }
}

void Rotate(node **p, int d,node **par, int cd,node **root) {
  node *P = *p;

  if (d == -1) {
    *p = (*p)->right;
    P->right = (*p)->left;
    (*p)->left = P;
  }
  else {
    *p = (*p)->left;
    P->left = (*p)->right;
    (*p)->right = P;
  }

  if (cd == -1)
    (*par)->left = *p;
  else if (cd == 1)
    (*par)->right = *p;
  else
    *root = *p;
}

void AVLInsert(node **root,int key) {
  int d1, d2, d3, critnodefound = 0, critdir, dir = 0;
  node *n, *a, *b, *c, *cp, *p, *r;

  p = n = *root;

  while ((n != NULL) && (n->key != key)) {
    if (n->bal != 0) {
      c = n;
```

```
      cp = p;
      critnodefound = 1;
      critdir = dir;
    }
    if (key < n->key) {
      p = n;
      n = n->left;
      dir = -1;
    }
    else {
      p = n;
      n = n->right;
      dir = 1;
    }
  }

  if (n == NULL) {
    if (p == NULL)
      *root = New(key);
    else if (dir == -1)
      p->left = New(key);
    else
      p->right = New(key);

    if (!critnodefound)
      r = *root;
    else {
      ShiftNode(&d1,&a,key,&c);
      if (c->bal != d1) {
        c->bal = 0;
        r = a;
      }
      else {
        ShiftNode(&d2,&b,key,&a);
        if (d2 == d1) {
          c->bal = 0;
          r = b;
          Rotate(&c,-d1,&cp,critdir,root);
        }
        else {
          ShiftNode(&d3,&r,key,&b);
          if (d3 == d2) {
            c->bal = 0;
            a->bal = d1;
          }
          else if (d3 == -d2)
            c->bal = d2;
          else
            c->bal = 0;
          Rotate(&a,-d2,&c,d1,root);
          Rotate(&c,-d1,&cp,critdir,root);
        }
      }
    }
    while (r->key != key) {
      ShiftNode(&(r->bal),&r,key,&r);
    }
  }
}
```

Quicksort (version 1)[Quinn, 1987]

```
int v[ARRAY_SIZE];

void QSort(int left,int right) {
    int i,piv,temp,cmp;

    if (left >= right)
        return;
    cmp = v[left];
    piv = left;
    for (i=left+1;i<=right;i++)
        if (v[i] < cmp) {
            piv++;
            temp = v[piv]; v[piv] = v[i]; v[i] = temp;
        }
    temp = v[piv]; v[piv] = v[left]; v[left] = temp;
    QSort(left,piv-1);
    QSort(piv+1,right);
}
```

Quicksort (version 2)[Quinn, 1987]

```
int v[ARRAY_SIZE];

void QSort(int left,int right)
{
    int temp = v[left];
    int i = left, j = right;

    if (j > i) {
        while (j>i) {
            while ((j >= i) && (temp < v[j]))
                j--;
            if (j<=i)
                v[i] = temp;
            else {
                v[i] = v[j];
                i++;
                while ((i <= j) && (v[i] < temp))
                    i++;
                if (j>i) {
                    v[j] = v[i];
                    j--;
                }
                if (j<=i)
                    v[j] = temp;
            }
        }
        QSort(left,j-1);
        QSort(j+1,right);
    }
}
```

Recursive Fibonacci number generation

```
int Fib(int n) {
    if (n == 0)
        return 0;
    else if (n == 1)
        return 1;
    else
        return Fib(n-1) + Fib(n-2);
}
```

Transitive closure [Corman et al., 1990]

```
int a[N+1][N][N];

void Trans() {
    int i,j,k;
    for (k=0; k<N; k++)
        for (i=0; i<N; i++)
            for (j=0; j<N; j++)
                a[k+1][i][j] = (a[k][i][j]
                  || (a[k][i][k] && a[k][k][j]));
}
```

Appendix C

Additional Graphs

C.1 Minimum Size Fixed Length VTSs

The graphs in this section are described in Section 5.3.4. They show the minimum number of bits necessary in a length or exponential VTS to execute the test program for a variety of sizes.

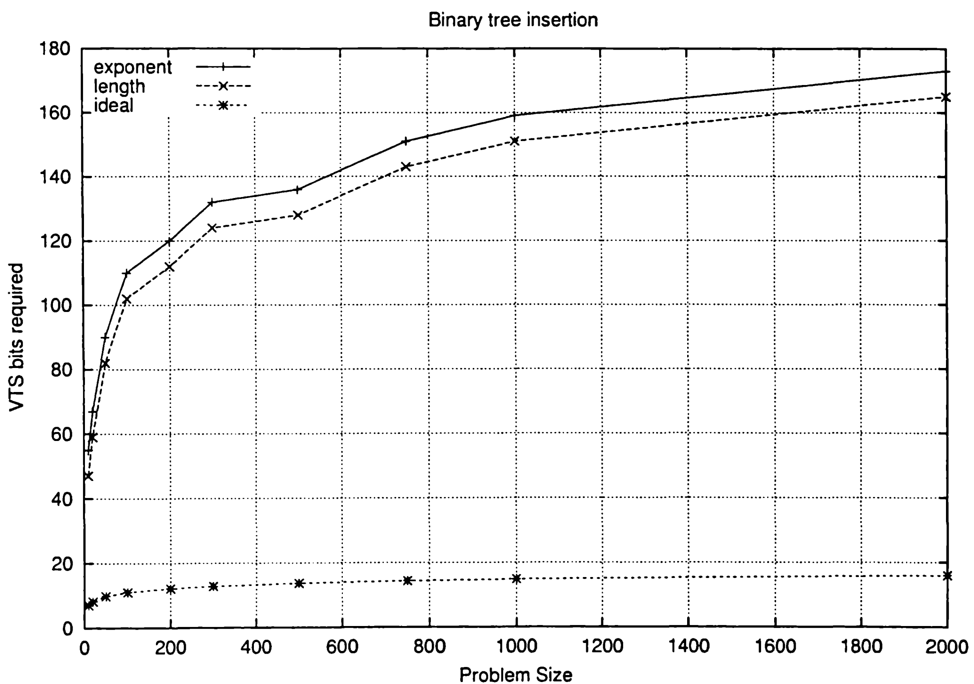


Figure C.1: Minimum VTS length necessary to execute binary tree insertion without rescaling

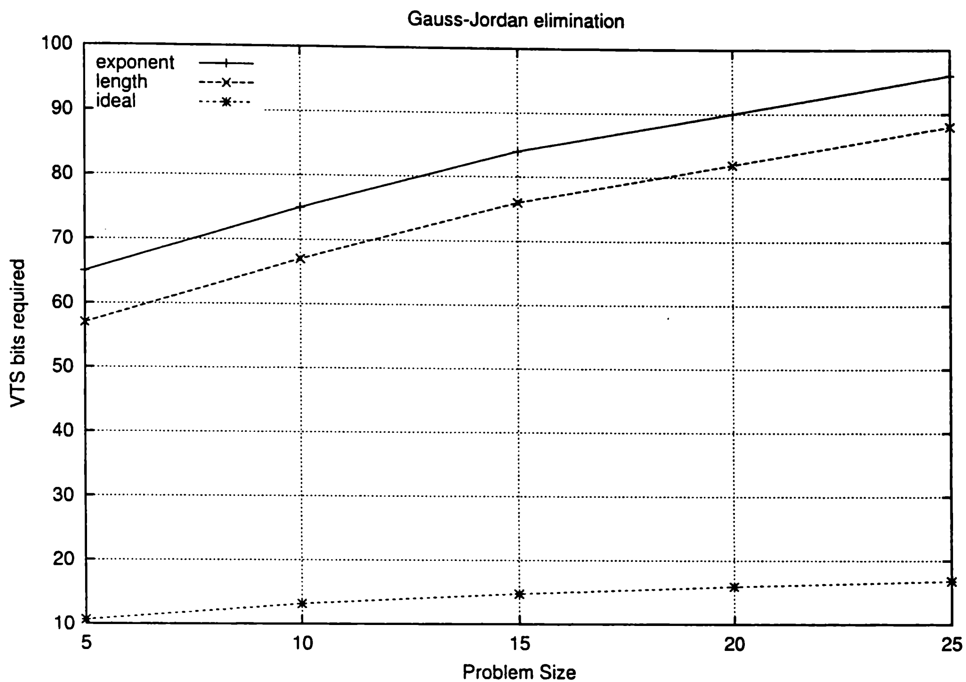


Figure C.2: Minimum VTS length necessary to execute Gauss-Jordan without rescaling

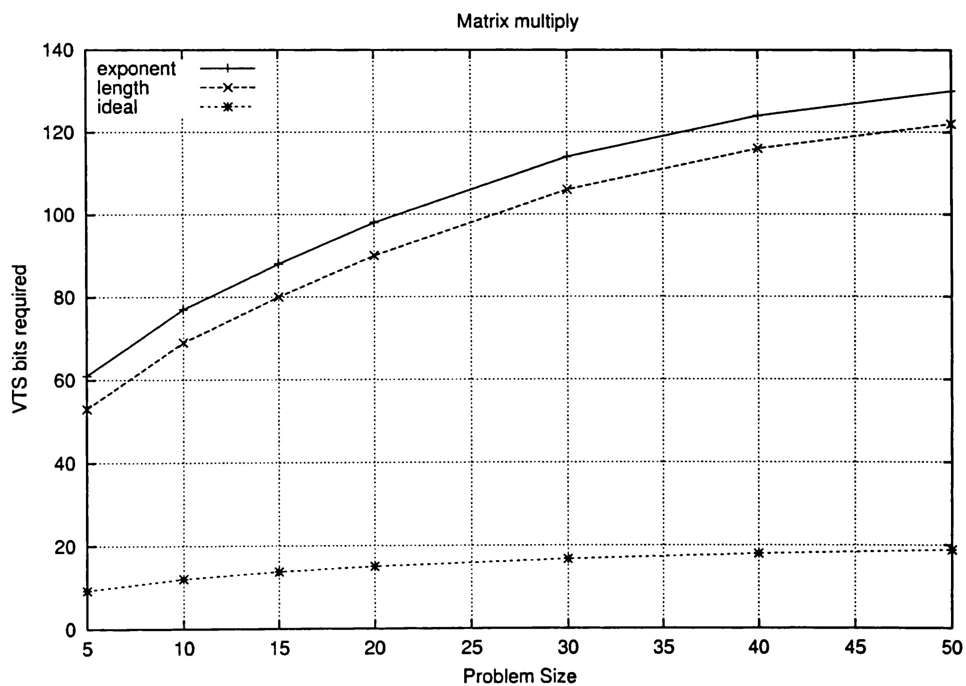


Figure C.3: Minimum VTS length necessary to execute matrix multiply without rescaling

C.2 Speedup for Length and Exponential VTSs

The graphs in this section are described in Section 5.4.3. They show the speedup over sequential execution for length or exponential VTSs executing the test programs.

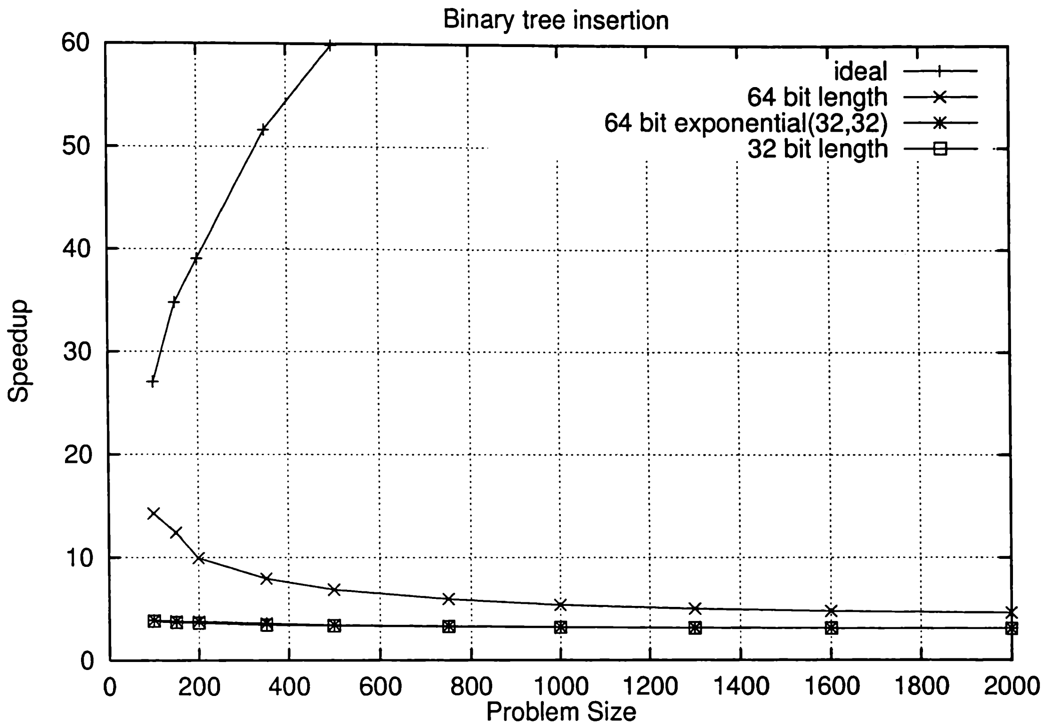


Figure C.4: Comparison of speedup for binary tree insertion with exponent and length VTSs

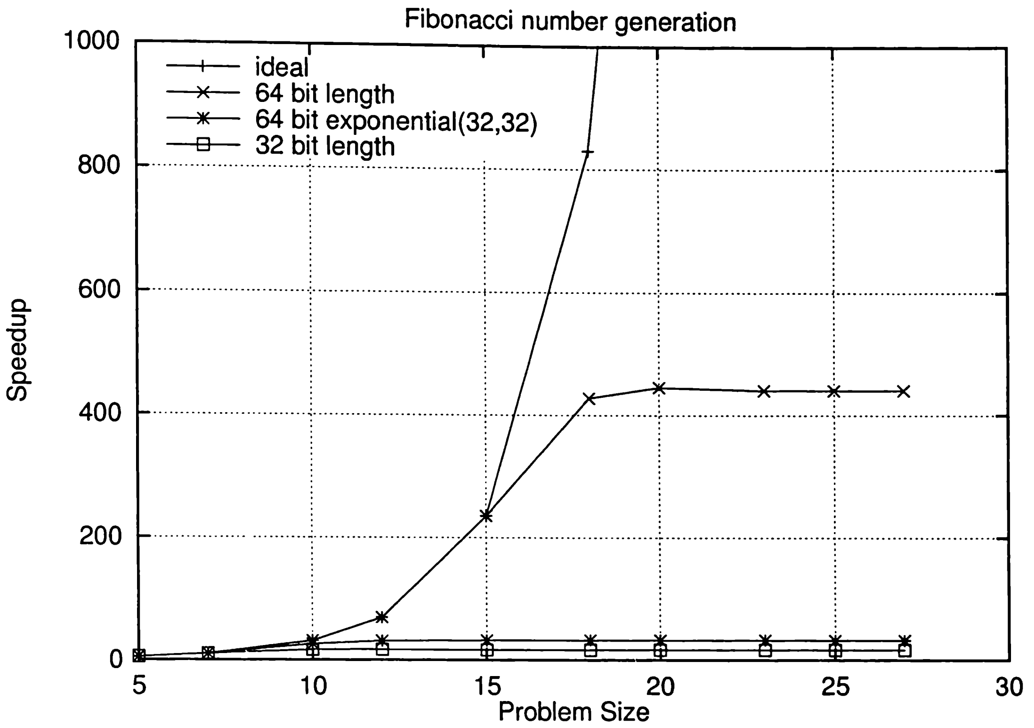


Figure C.5: Comparison of speedup for Fibonacci with exponent and length VTSs

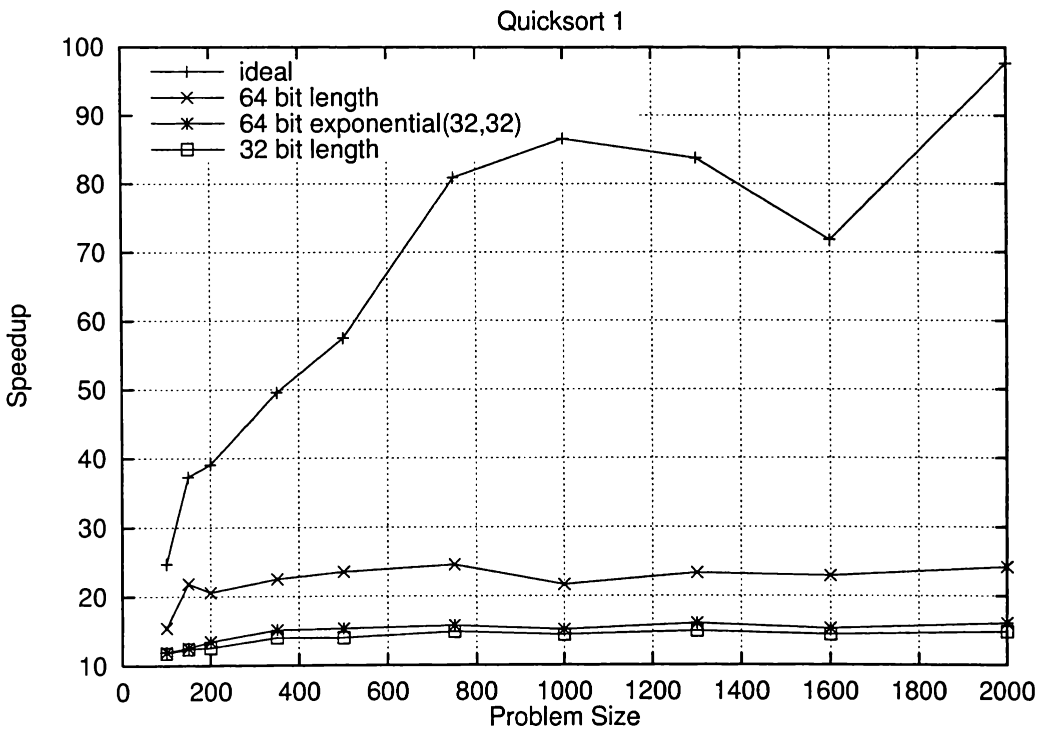


Figure C.6: Comparison of speedup for quicksort 1 with exponent and length VTSs

C.3 Minimum Size Variable Range VTSs

The graphs in this section are described in Section 6.3. They show the minimum number of bits necessary in a variable range VTS to execute the test program for a variety of sizes.

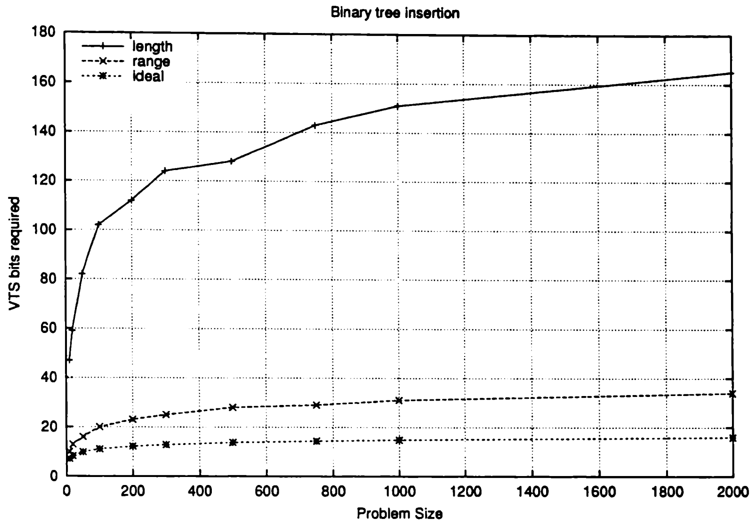


Figure C.7: Minimum VTS length necessary to execute binary tree insertion without rescaling

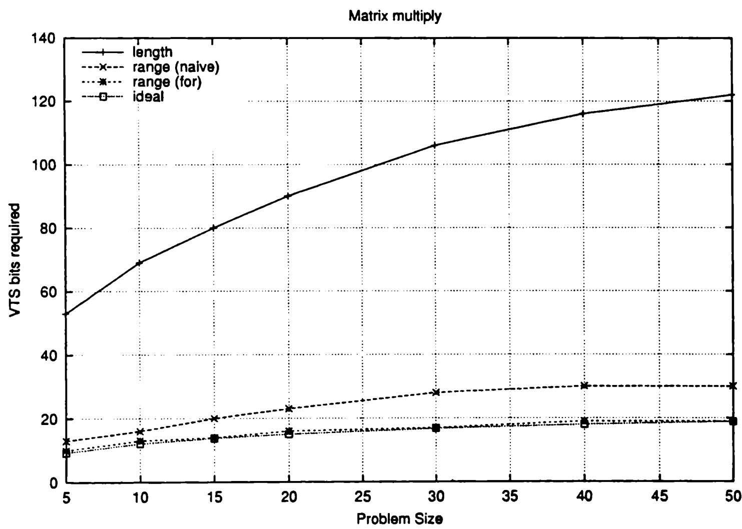


Figure C.8: Minimum VTS length necessary to execute matrix multiply without rescaling

C.4 Variable Range VTS Speedup

The graphs in this section are described in Section 6.5.1. They show the speedup over sequential execution for variable range VTSs executing the test programs.

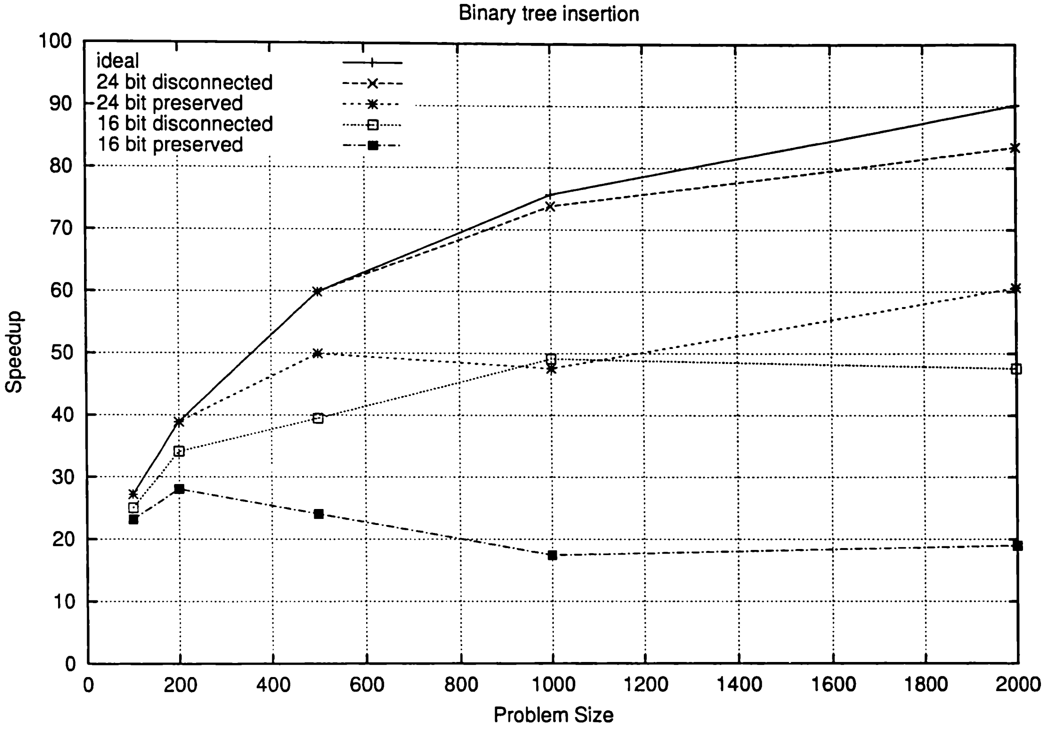


Figure C.9: Speedup for binary tree with variable range VTSs with rescaling

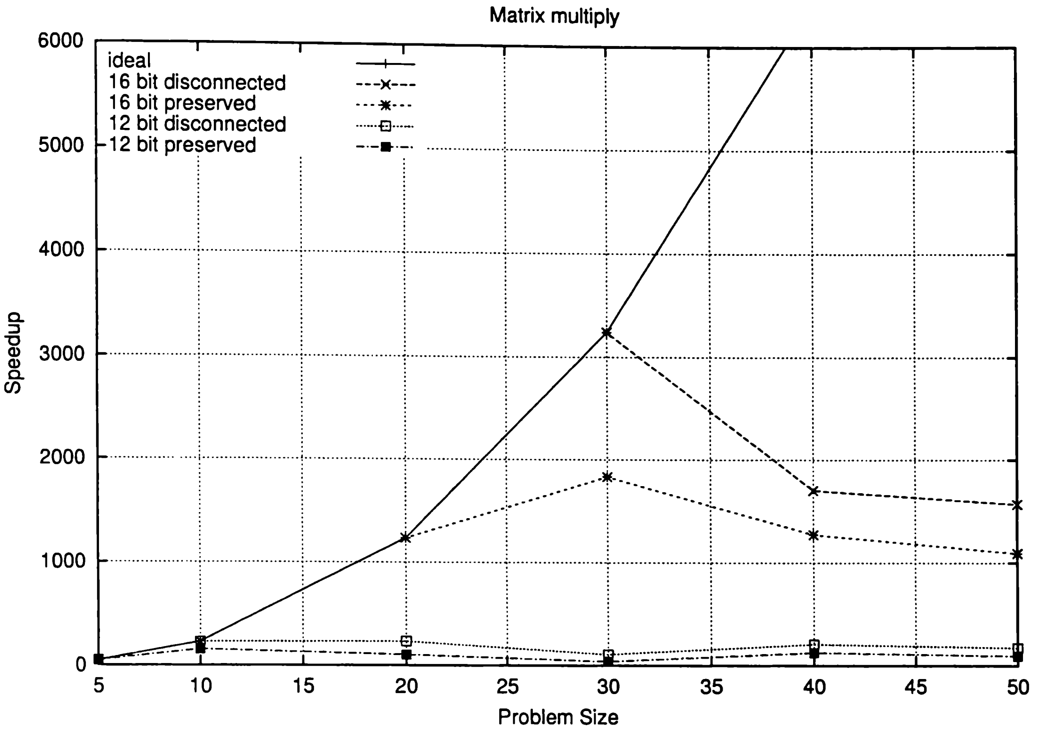


Figure C.10: Speedup for matrix multiply with variable range VTSs with rescaling

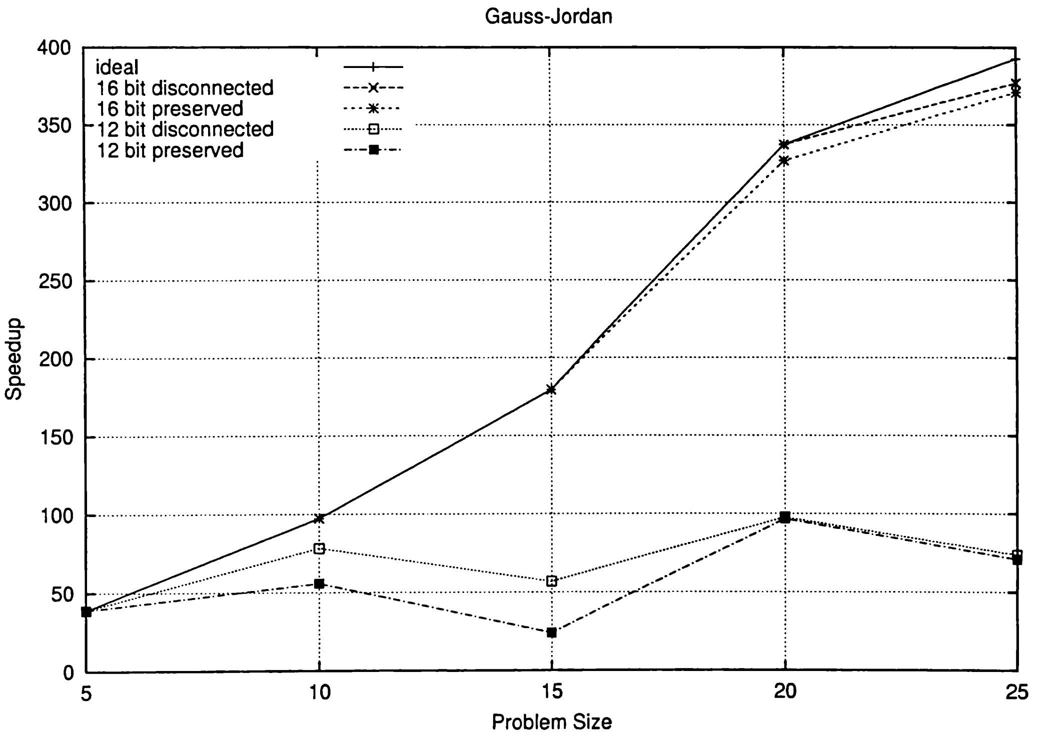


Figure C.11: Speedup for Gauss-Jordan with variable range VTSs with rescaling

C.5 Variable Range VTS Frame Limit Sensitivity

The graphs in this section are described in Section 6.5.2. They show the sensitivity to frame limiting of variable range VTSs. The programs are executed with varying frame limits and either ideal VTSs or 16 bit variable range VTSs.

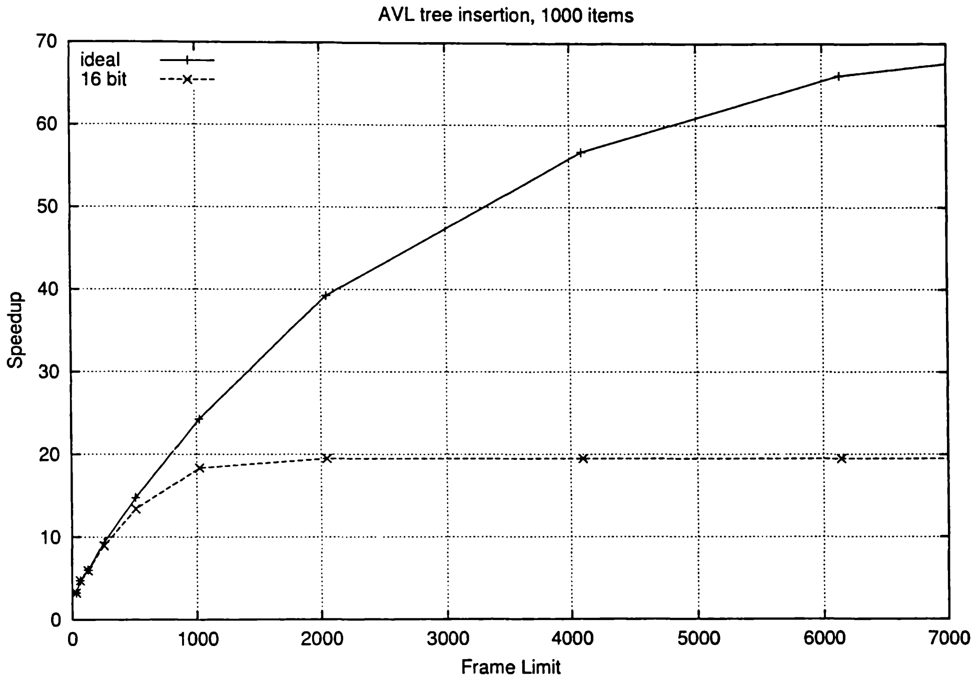


Figure C.12: Speedup for AVL(1000) with varying frame limitations and variable range VTSs

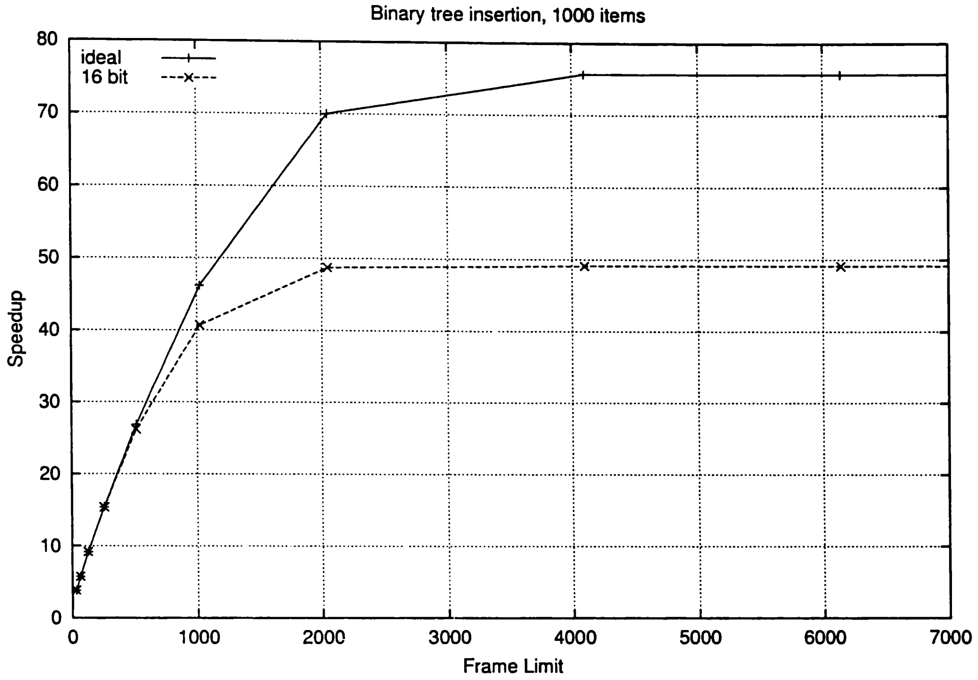


Figure C.13: Speedup for binary tree (1000) with varying frame limitations and variable range VTSs

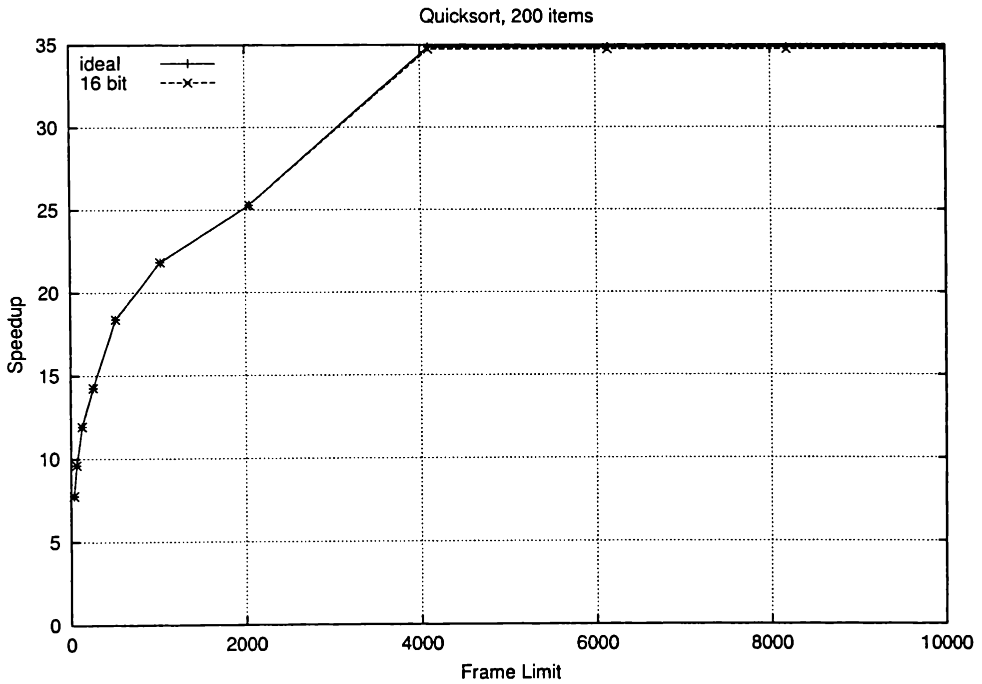


Figure C.14: Speedup for quicksort 1 (200) with varying frame limitations and variable range VTSs

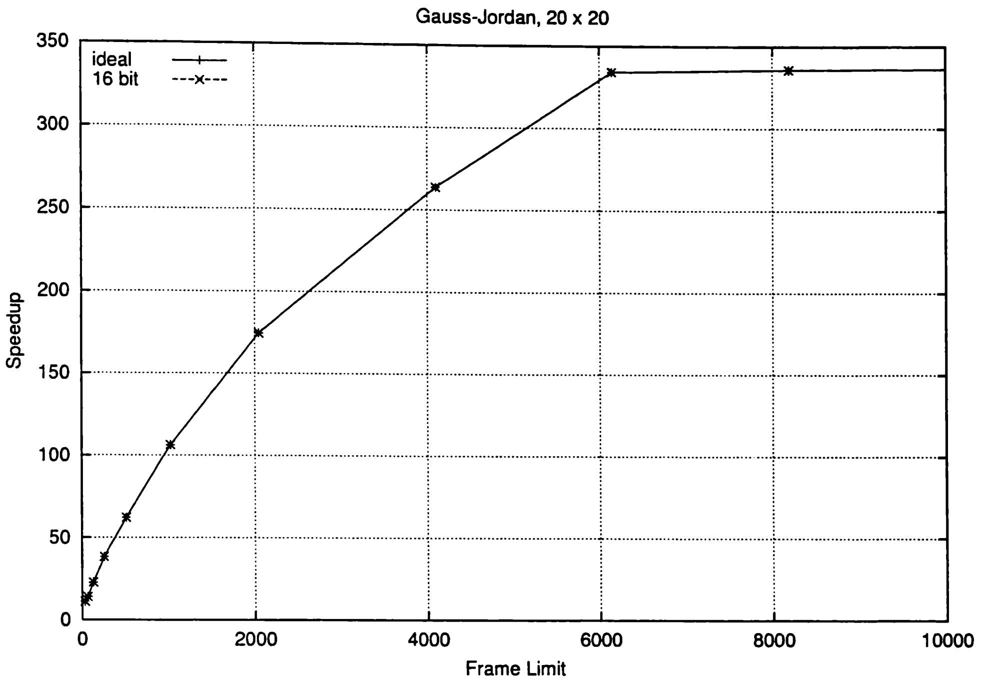


Figure C.15: Speedup for Gauss-Jordan (20) with varying frame limitations and variable range VTSs

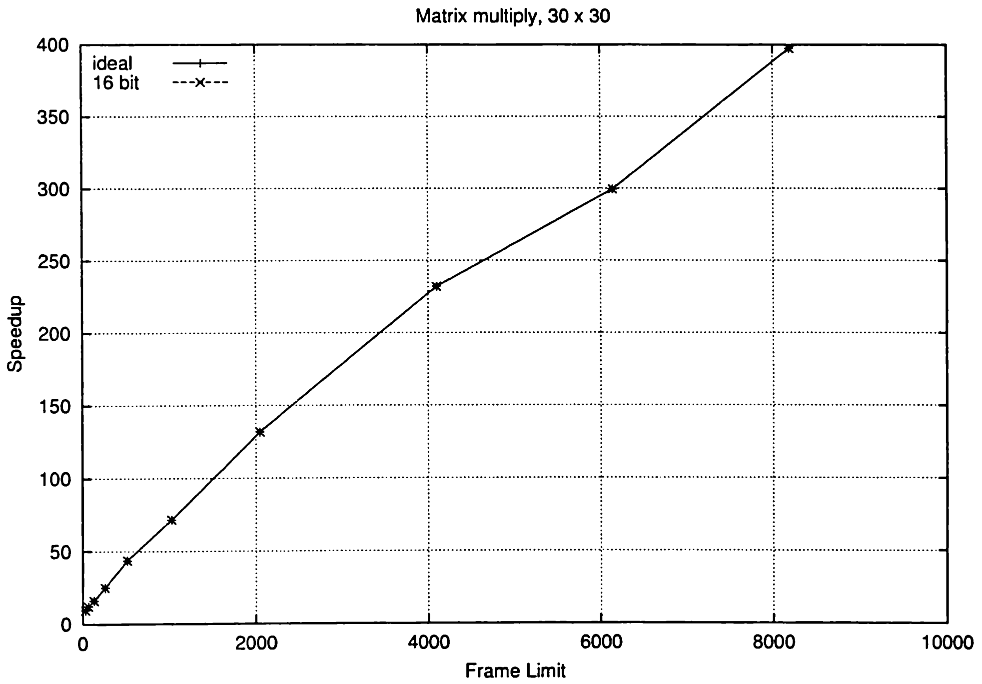


Figure C.16: Speedup for matrix multiply (30) with varying frame limitations and variable range VTSs

C.6 Speculation Effectiveness

The graphs in this section are described in Section 7.6. They show the speculation effectiveness for frame limited and resource block limited execution for a variety of data set sizes.

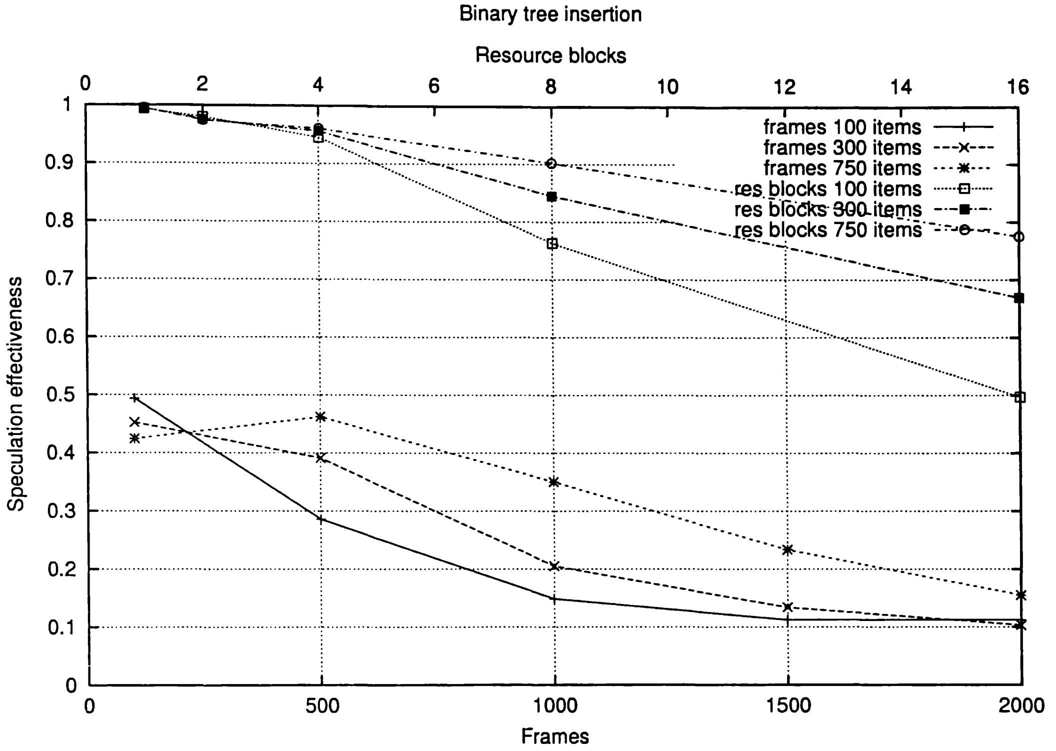


Figure C.17: Speculation effectiveness of resource block and frame limiting for binary tree

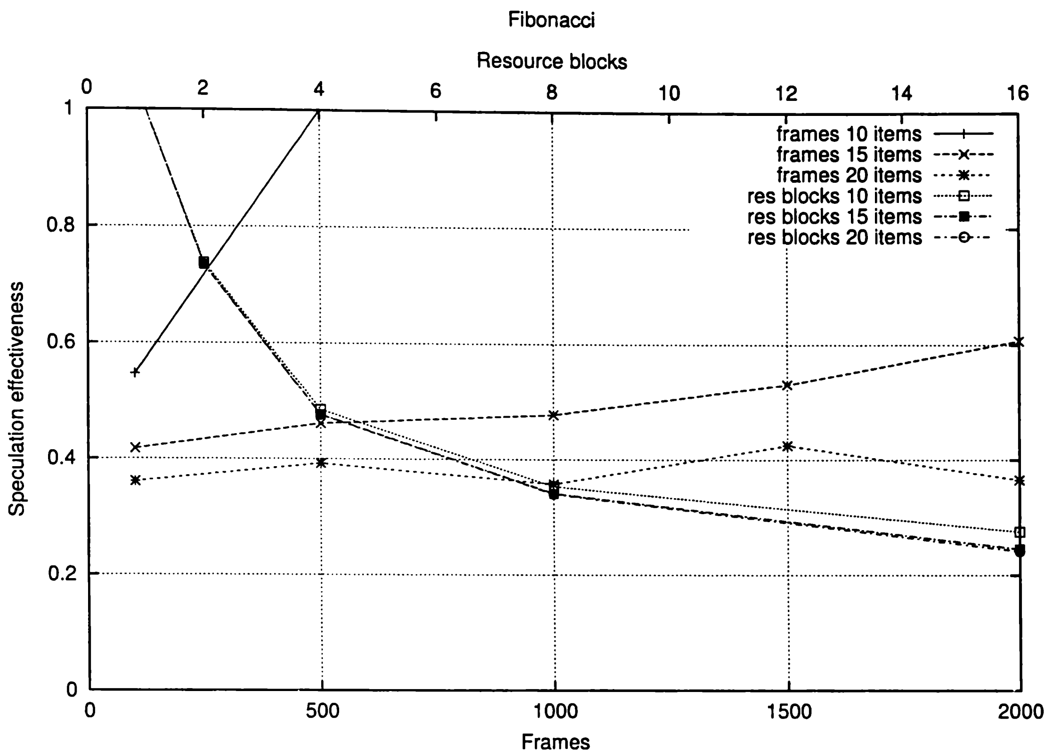


Figure C.18: Speculation effectiveness of resource block and frame limiting for Fibonacci

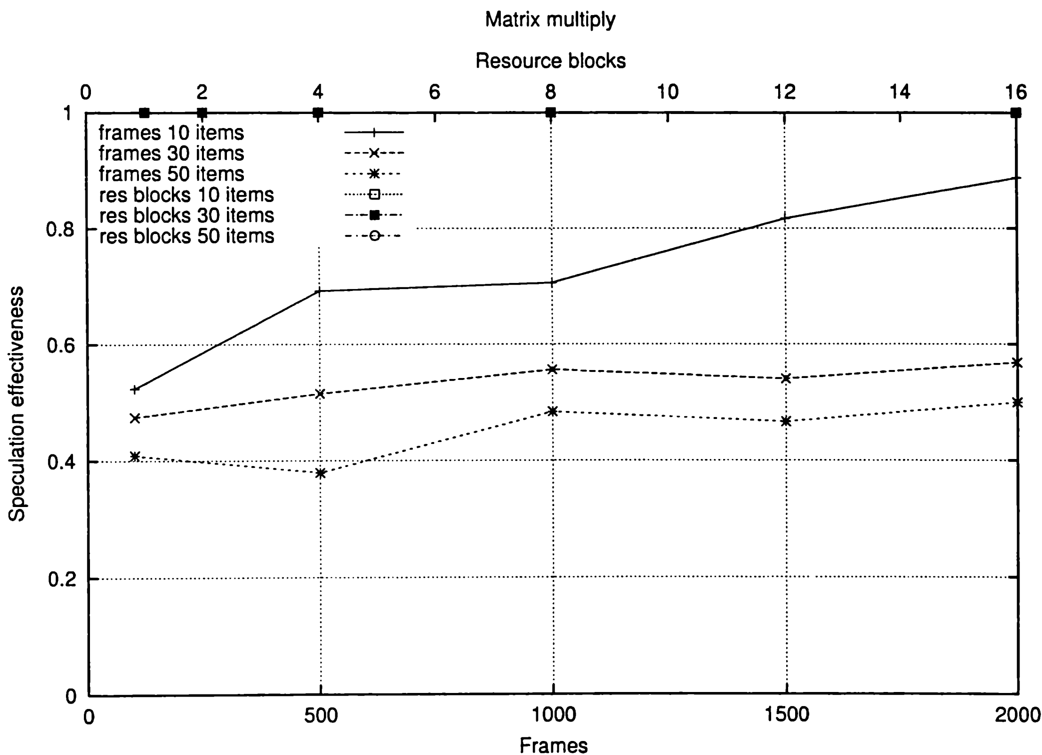


Figure C.19: Speculation effectiveness of resource block and frame limiting for matrix multiply

Appendix D
Twisted Memory Bandwidth Profile Graphs

The graphs in this appendix show the profile of memory accesses to twisted memory from a four resource block WarpEngine. For each test program there is a graph covering each level of the time-space cache, the combined bandwidth used to spatial memory, and the profile for a unified time-space cache for comparison. The profile for each cell is labeled by the resource block directly above it. An arbitrary problem size has been selected for each program. The graphs are discussed in Section 8.4.2, and are best viewed in colour.

D.1 AVL(200)

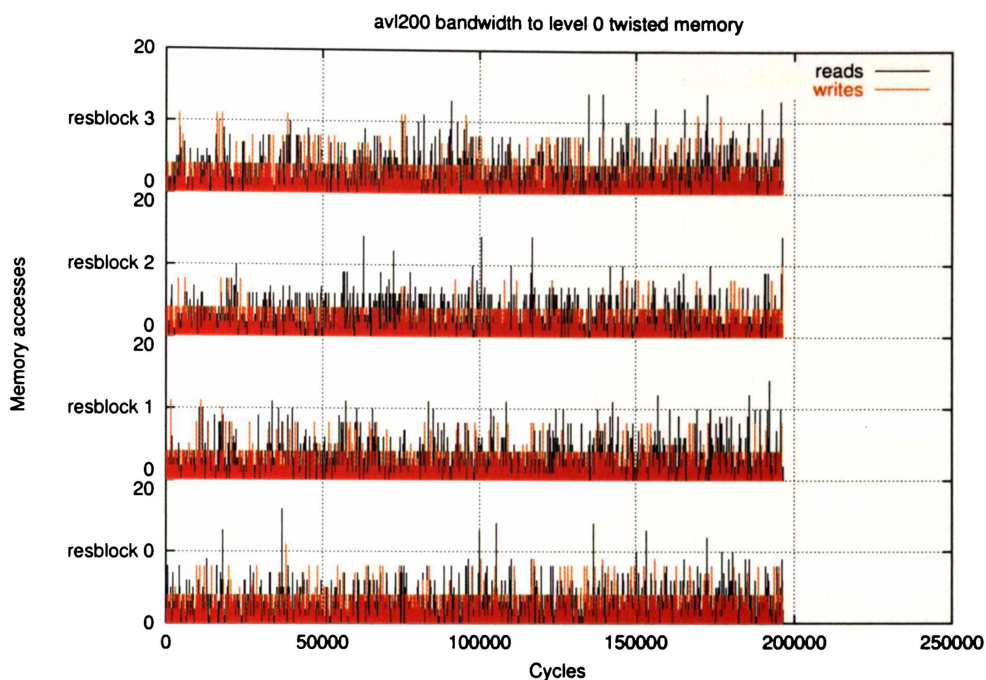


Figure D.1: Twisted memory bandwidth profile to level zero for AVL(200)

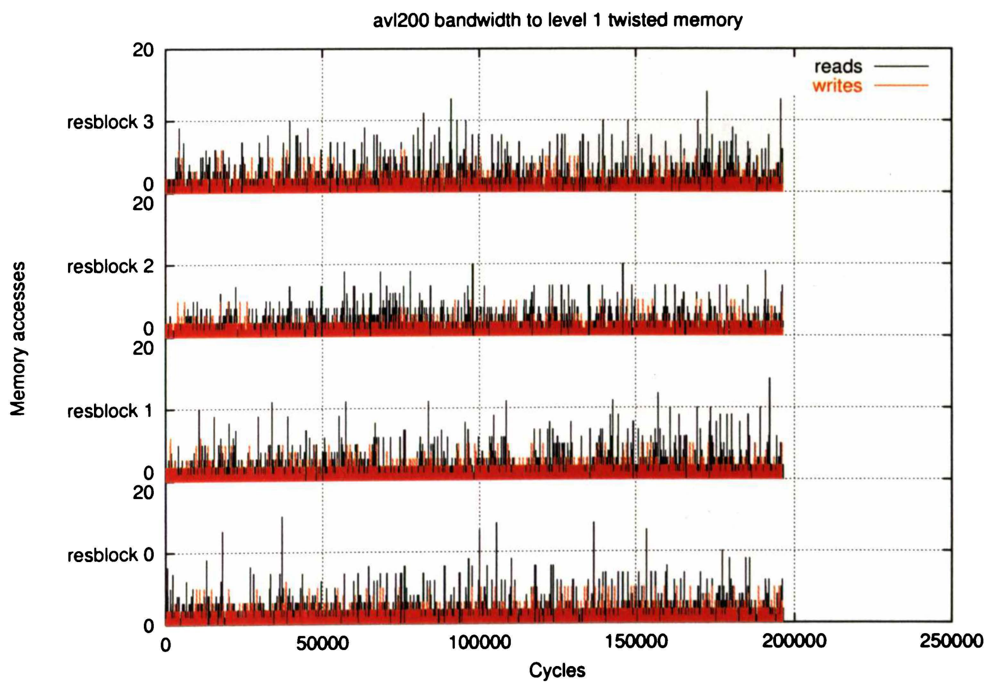


Figure D.2: Twisted memory bandwidth profile to level one for AVL(200)

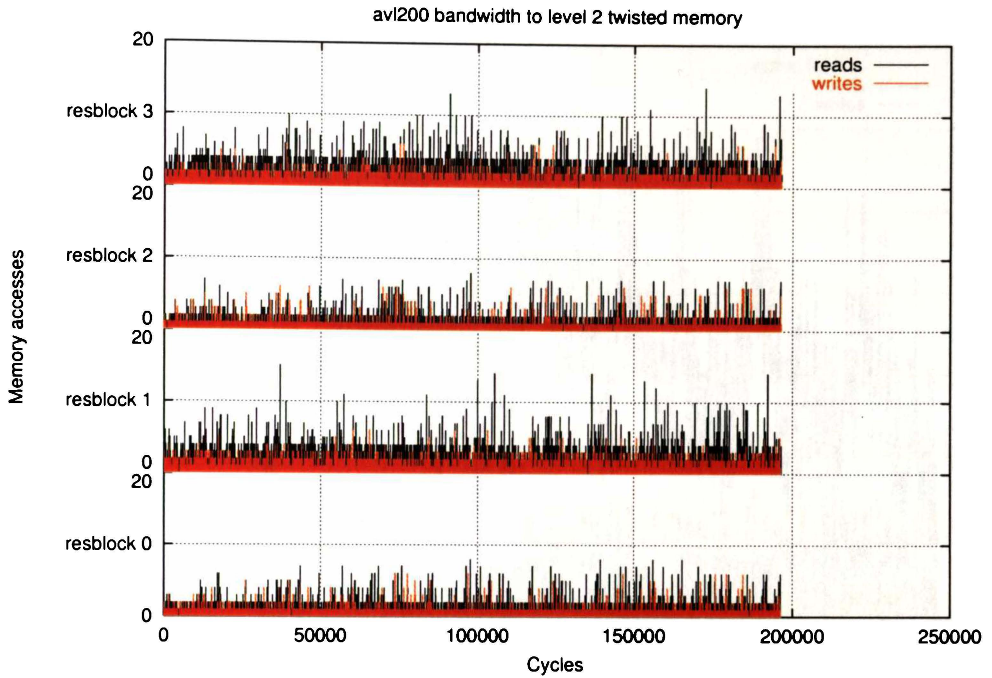


Figure D.3: Twisted memory bandwidth profile to level two for AVL(200)

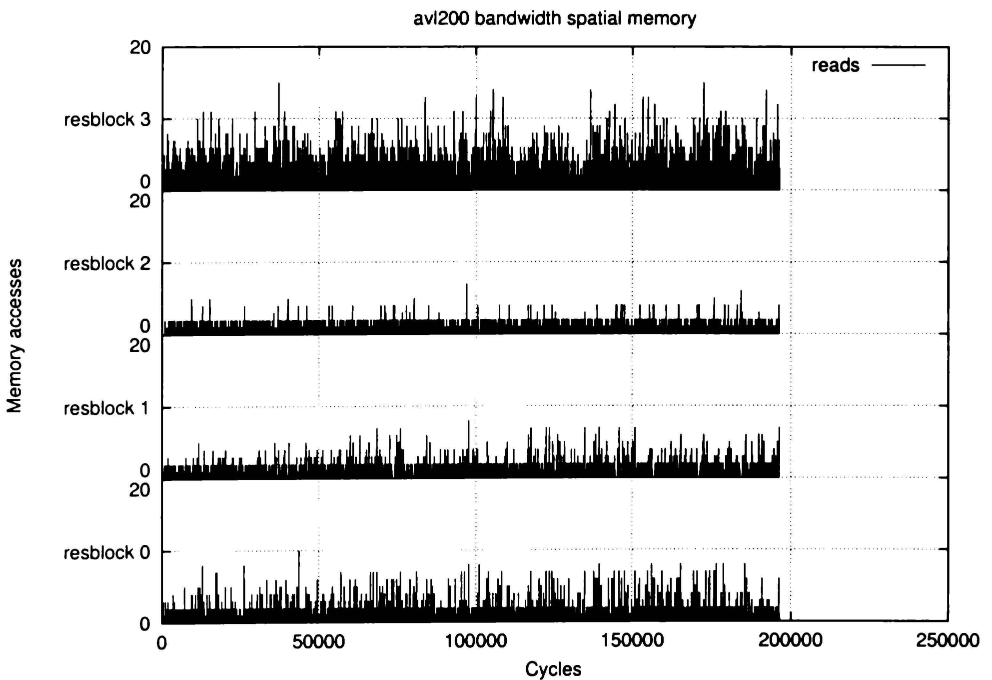


Figure D.4: Twisted memory bandwidth profile to spatial memory for AVL(200)

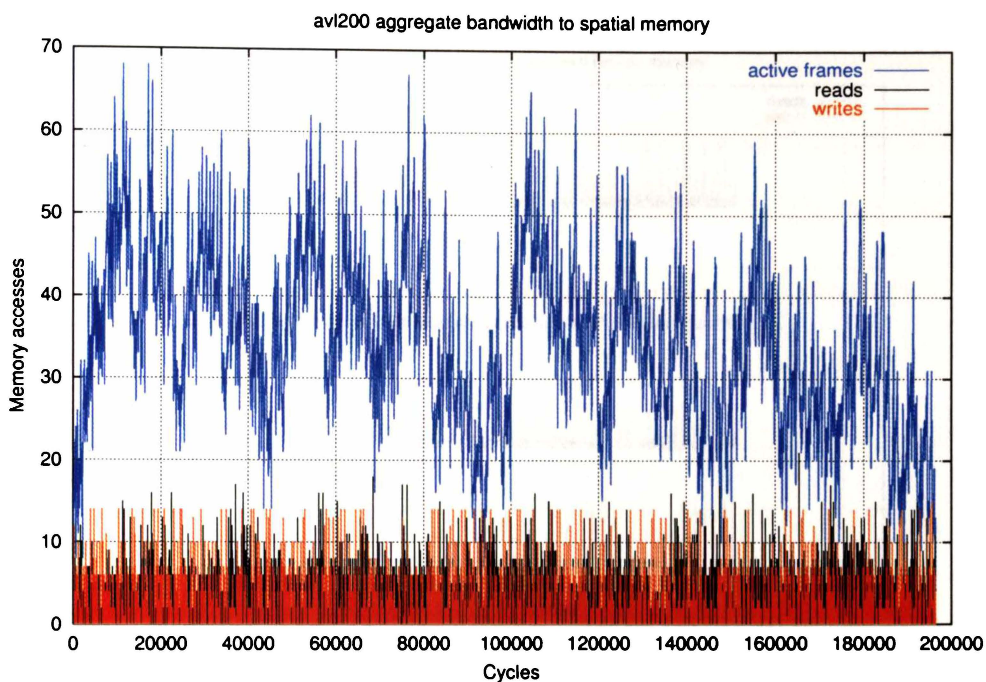


Figure D.5: Twisted memory bandwidth profile to level two for AVL(200)

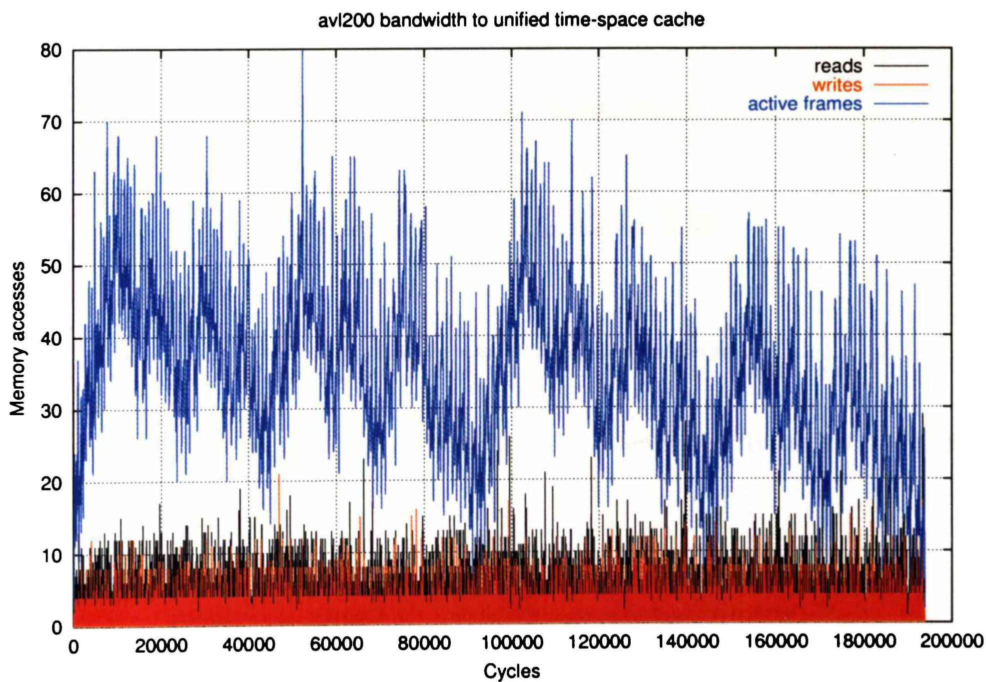


Figure D.6: Twisted memory aggregate bandwidth profile to spatial memory for AVL(200)

D.2 Binary Tree (300)

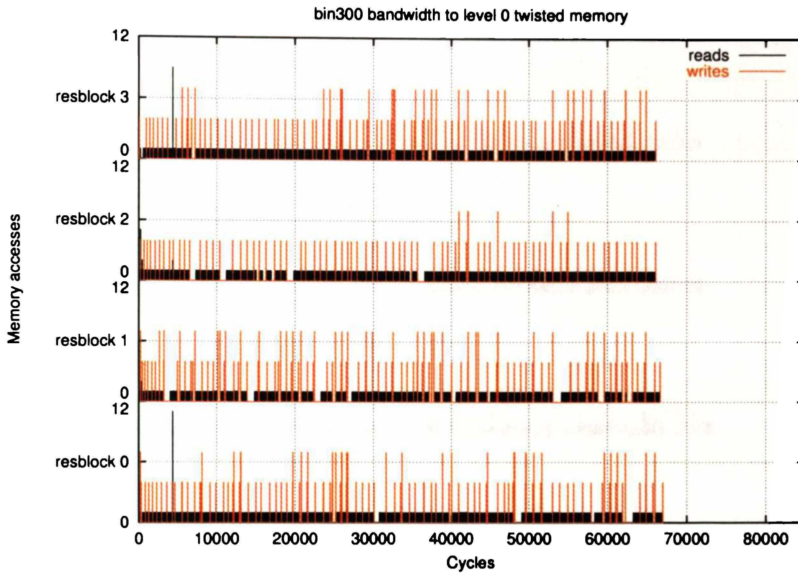


Figure D.7: Twisted memory bandwidth profile to level zero for binary tree (300)

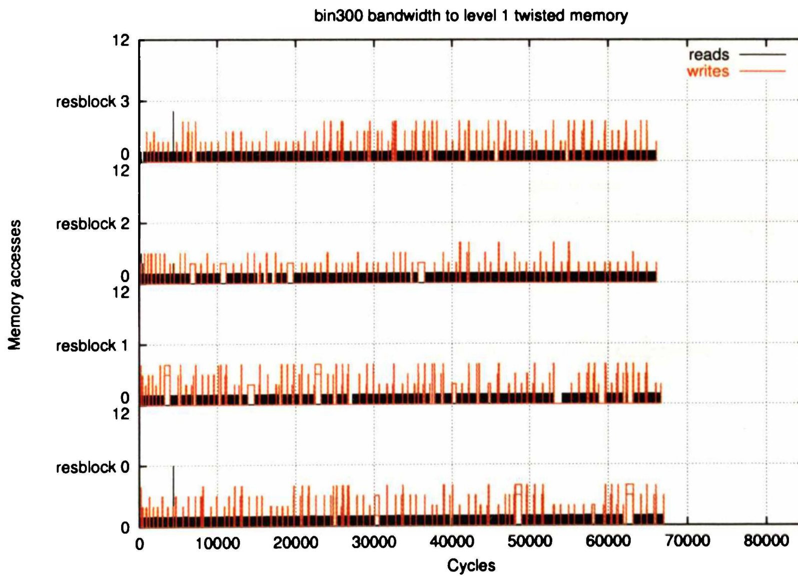


Figure D.8: Twisted memory bandwidth profile to level one for binary tree (300)

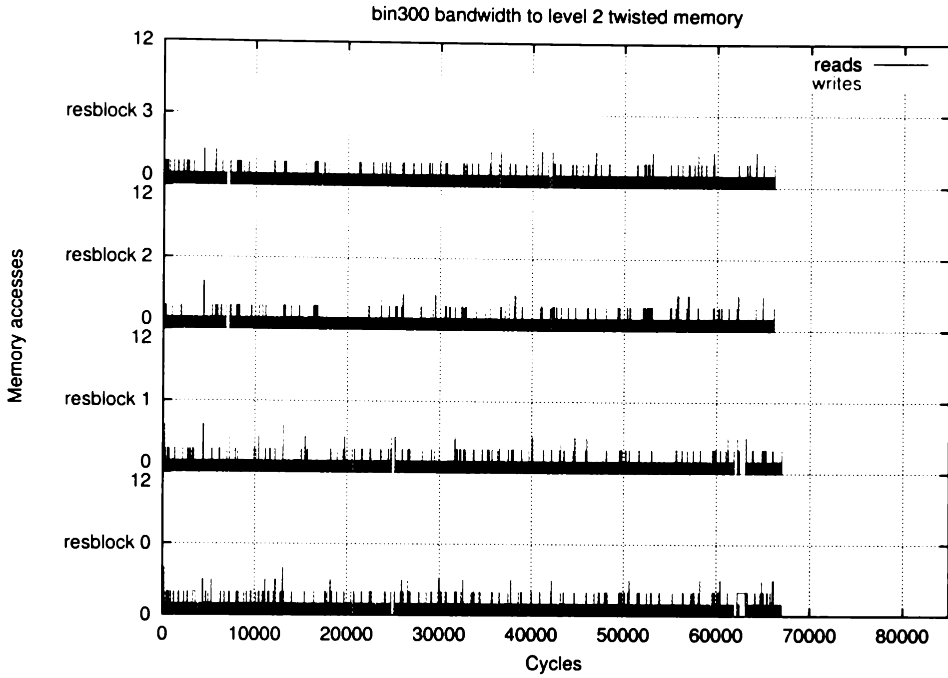


Figure D.9: Twisted memory bandwidth profile to level two for binary tree (300)

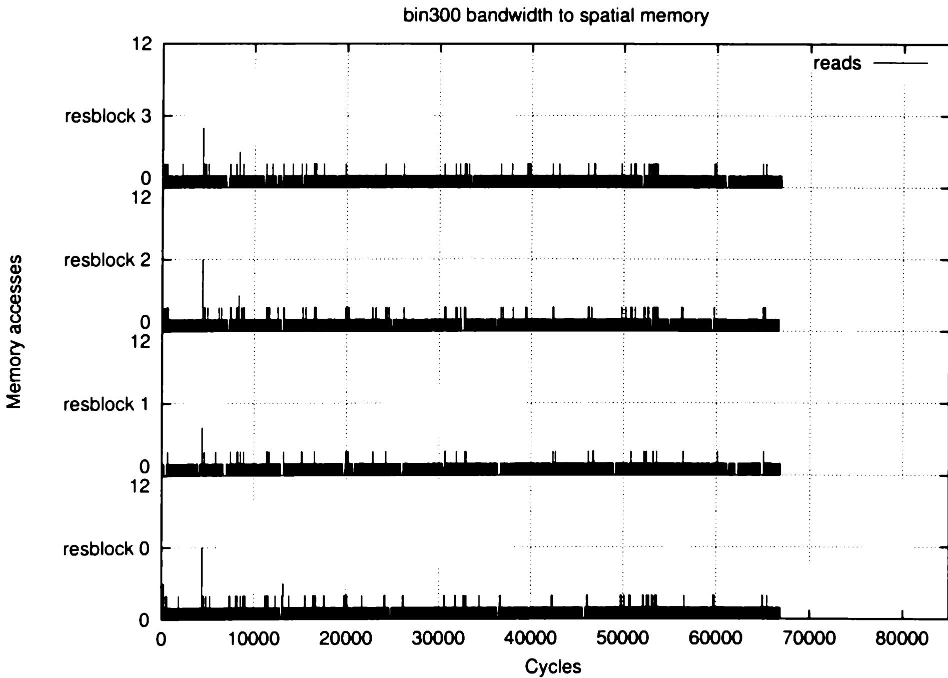


Figure D.10: Twisted memory bandwidth profile to spatial memory for binary tree (300)

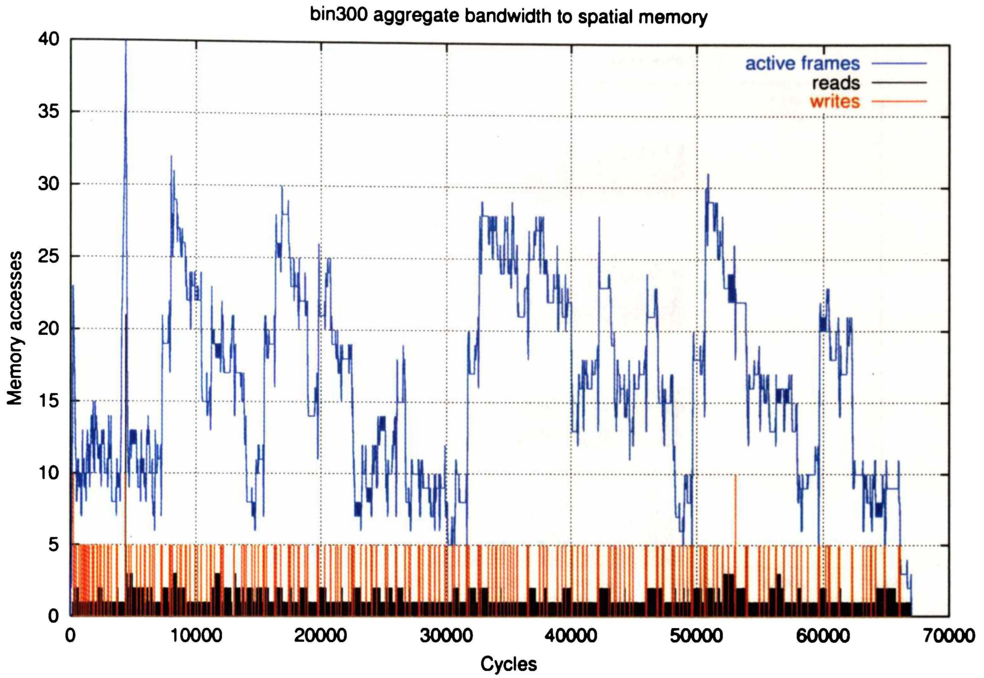


Figure D.11: Twisted memory bandwidth profile to level two for binary tree (300)

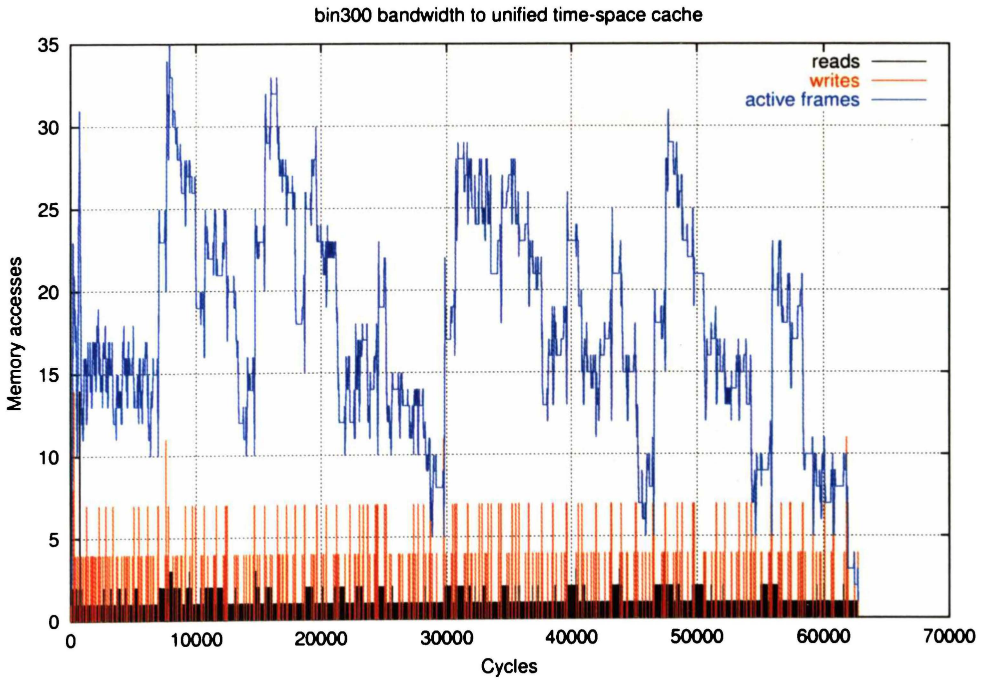


Figure D.12: Twisted memory aggregate bandwidth profile to spatial memory for binary tree (300)

D.3 Fibonacci (15)

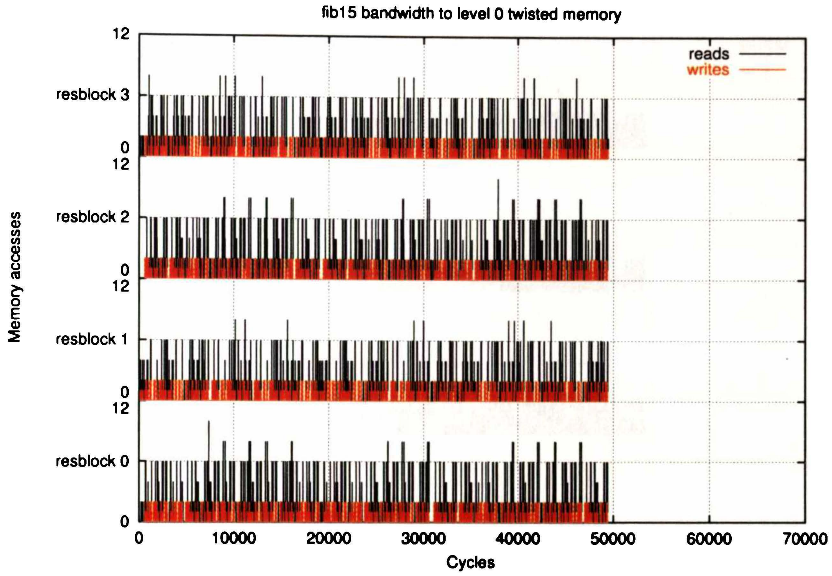


Figure D.13: Twisted memory bandwidth profile to level zero for Fibonacci (15)

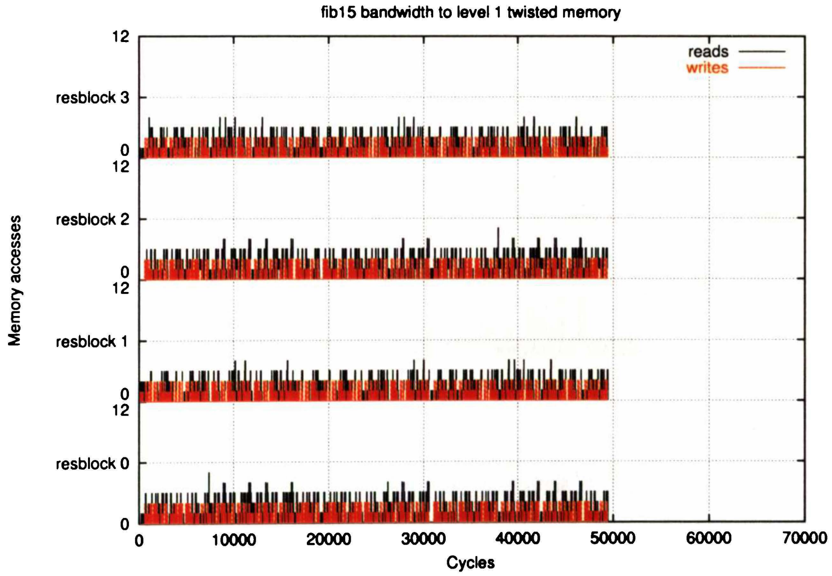


Figure D.14: Twisted memory bandwidth profile to level one for Fibonacci (15)

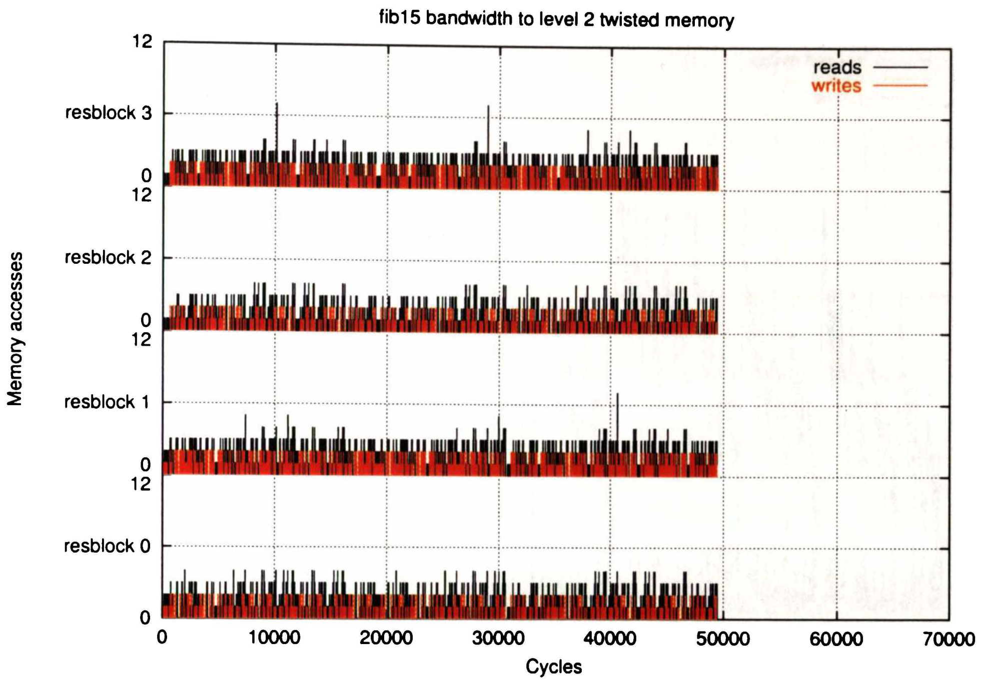


Figure D.15: Twisted memory bandwidth profile to level two for Fibonacci (15)

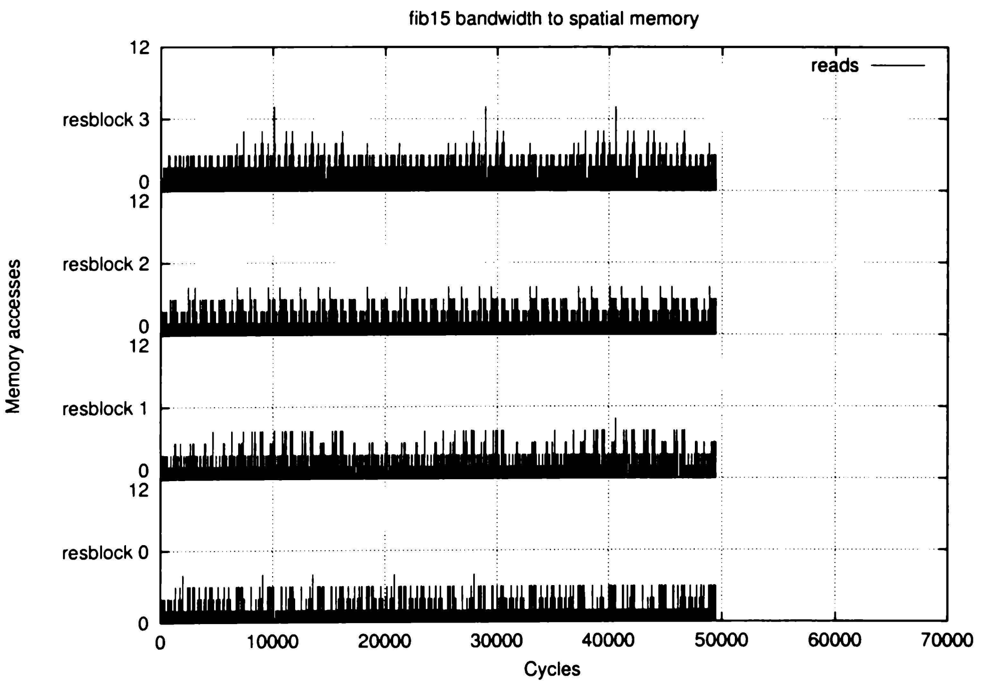


Figure D.16: Twisted memory bandwidth profile to spatial memory for Fibonacci (15)

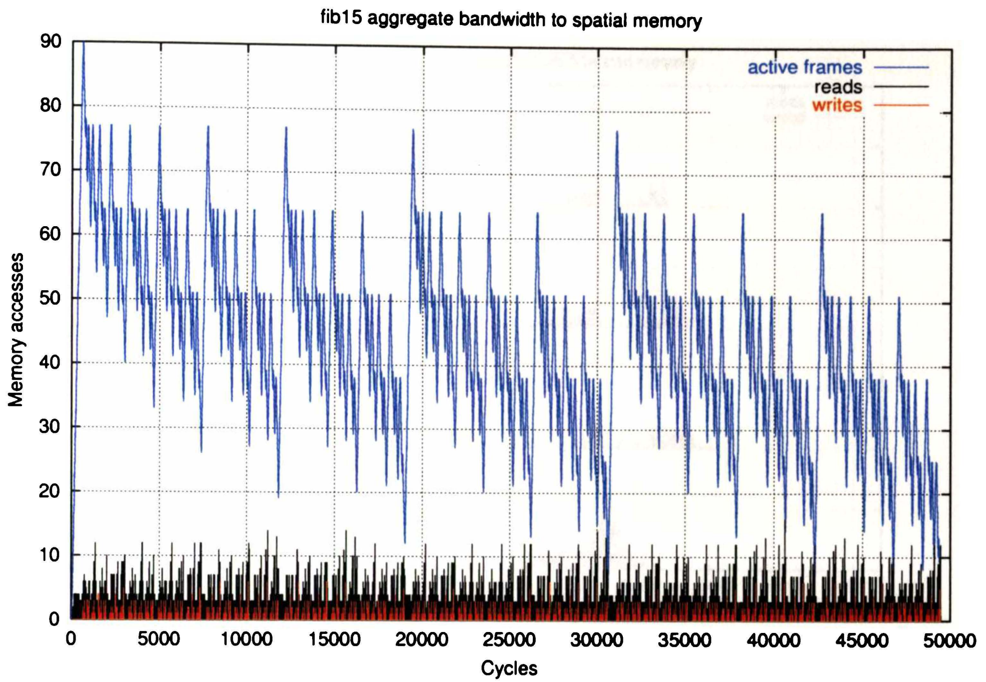


Figure D.17: Twisted memory bandwidth profile to level two for Fibonacci (15)

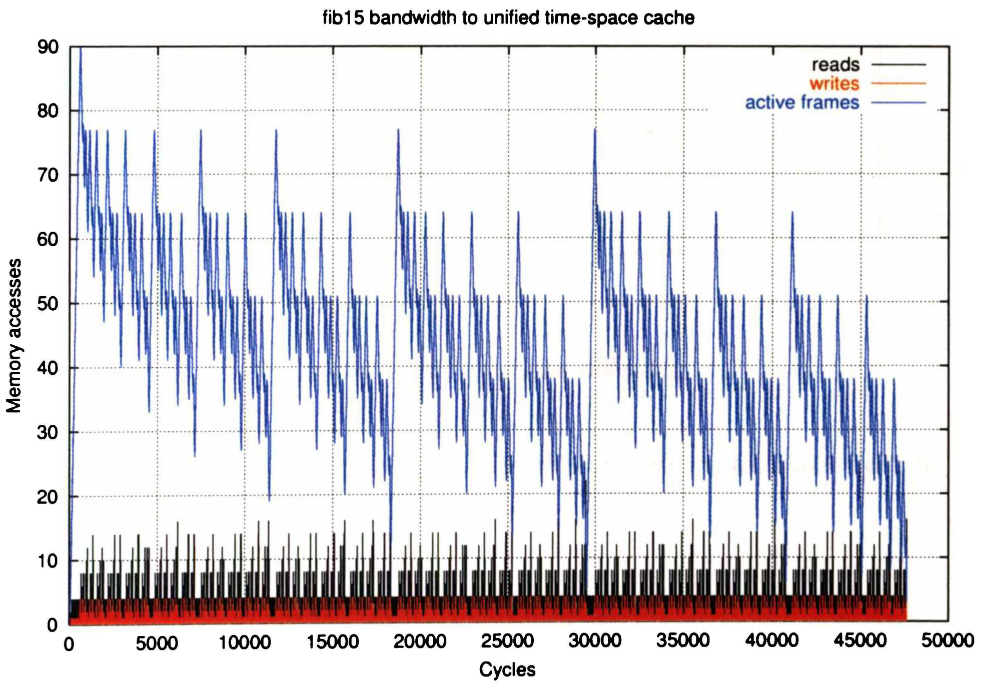


Figure D.18: Twisted memory aggregate bandwidth profile to spatial memory for Fibonacci (15)

D.4 Gauss-Jordan (10)

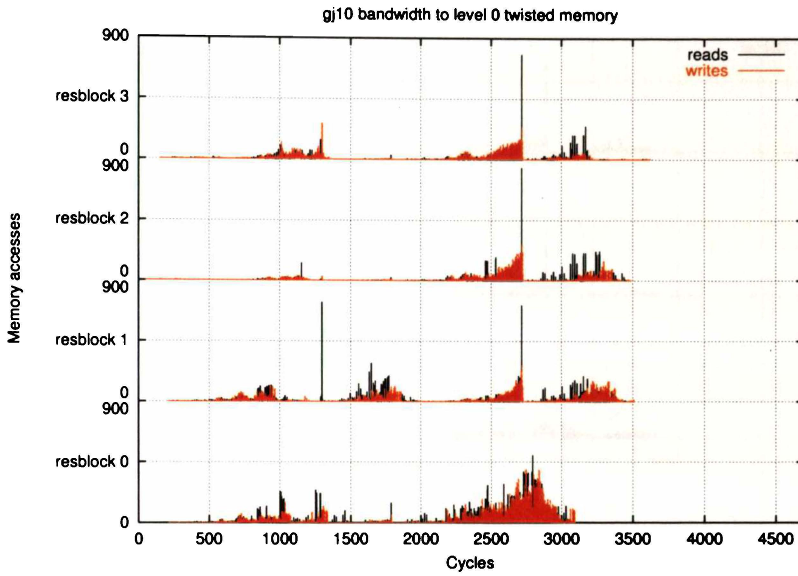


Figure D.19: Twisted memory bandwidth profile to level zero for Gauss-Jordan (10)

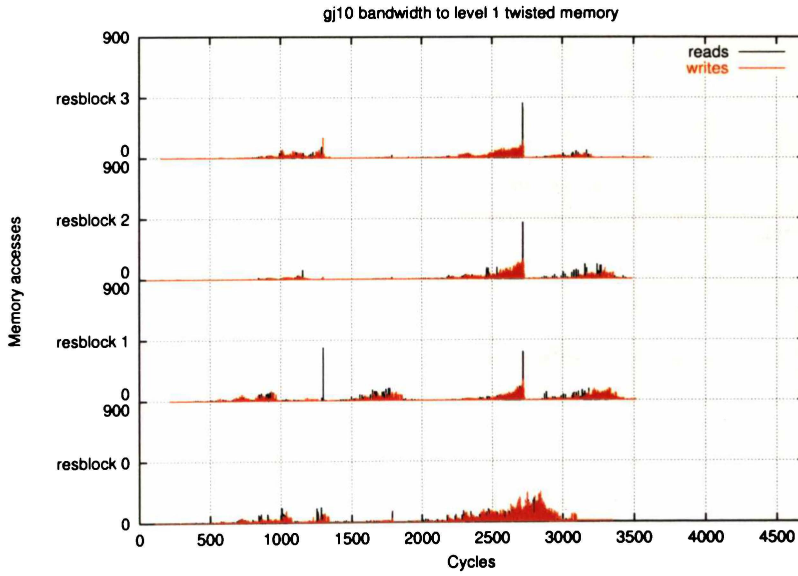


Figure D.20: Twisted memory bandwidth profile to level one for Gauss-Jordan (10)

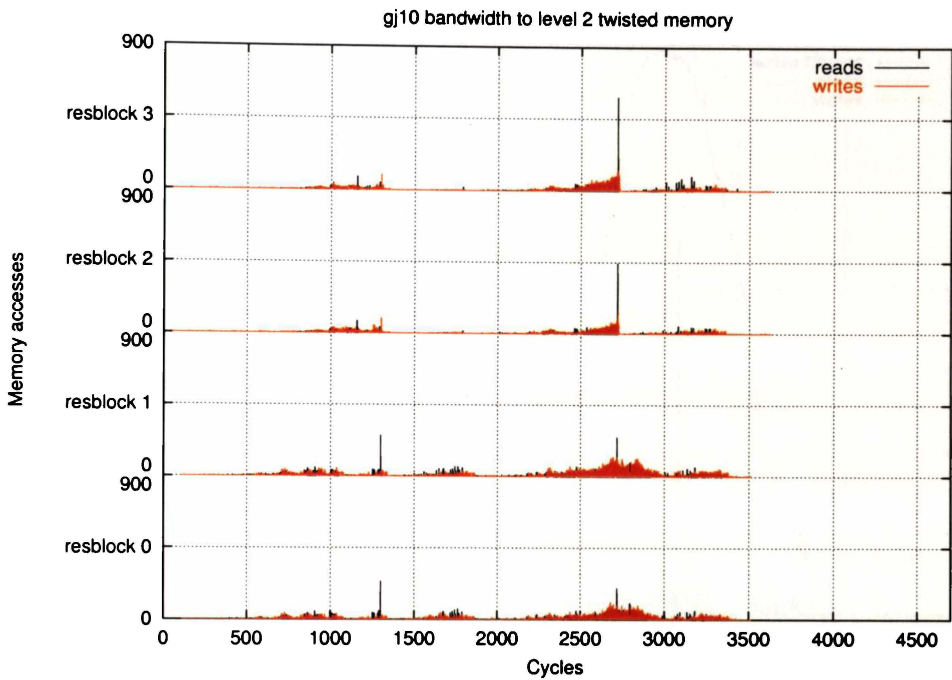


Figure D.21: Twisted memory bandwidth profile to level two for Gauss-Jordan (10)

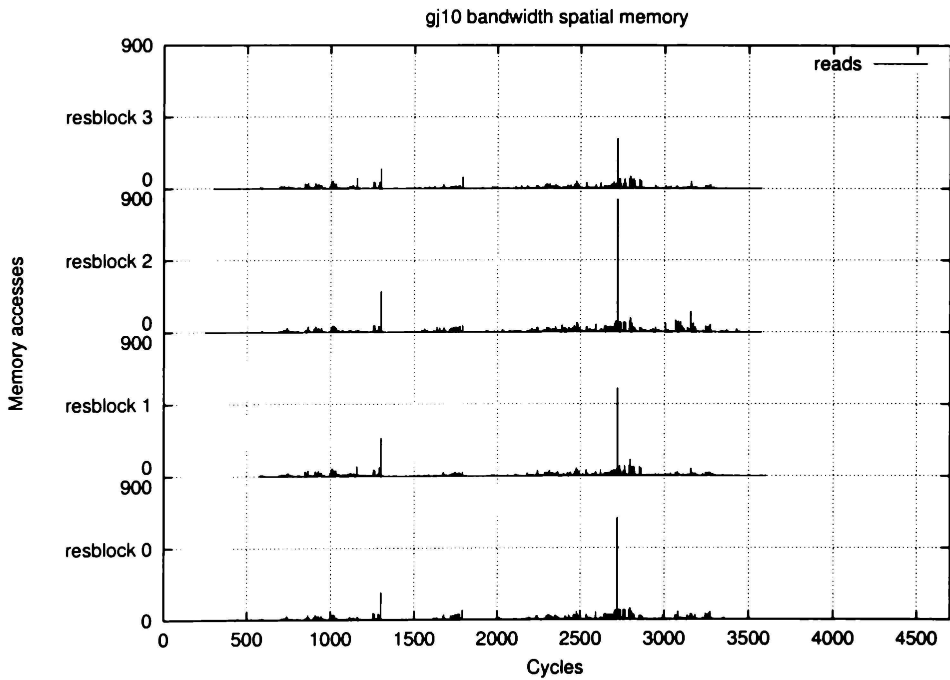


Figure D.22: Twisted memory bandwidth profile to spatial memory for Gauss-Jordan (10)

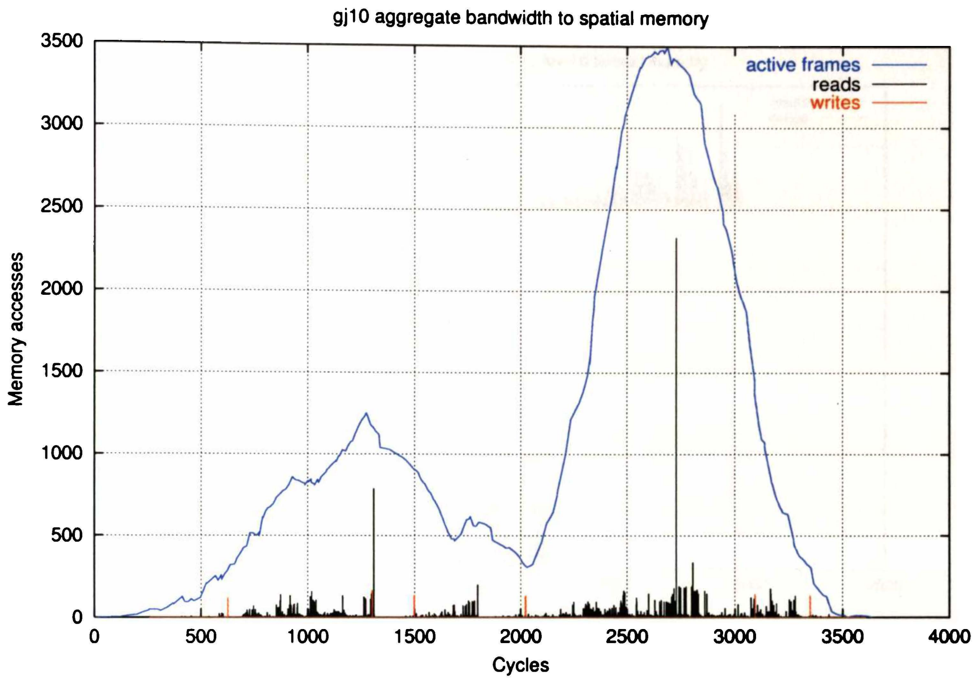


Figure D.23: Twisted memory bandwidth profile to level two for Gauss-Jordan (10)

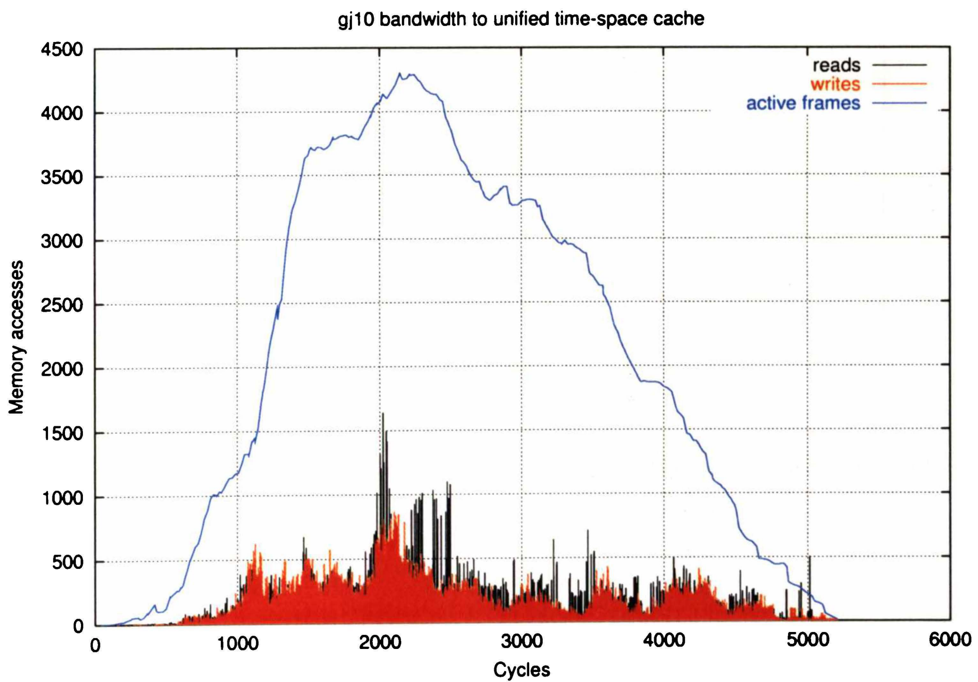


Figure D.24: Twisted memory aggregate bandwidth profile to spatial memory for Gauss-Jordan (10)

D.5 Matrix Multiply (20)

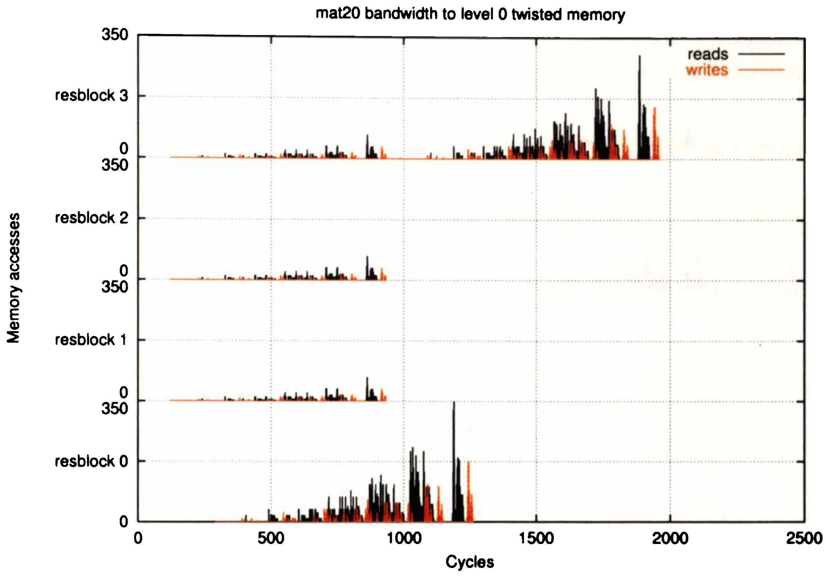


Figure D.25: Twisted memory bandwidth profile to level zero for matrix multiply (20)

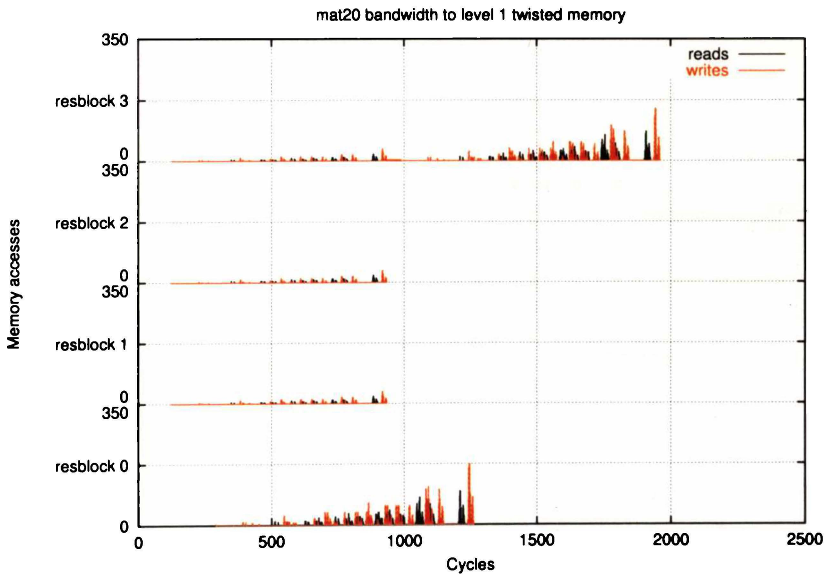


Figure D.26: Twisted memory bandwidth profile to level one for matrix multiply (20)

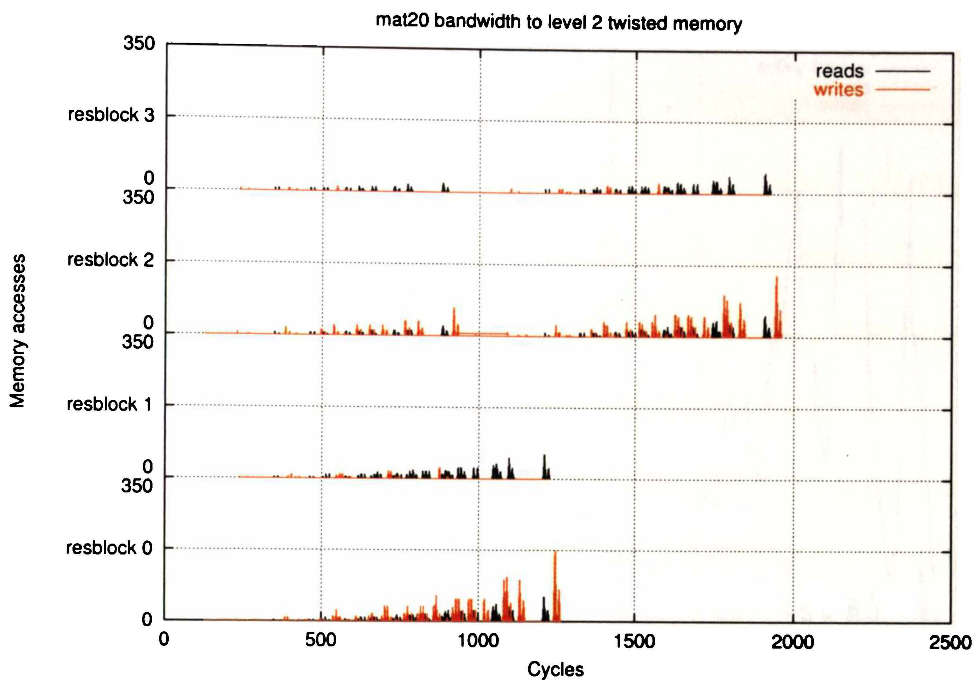


Figure D.27: Twisted memory bandwidth profile to level two for matrix multiply (20)

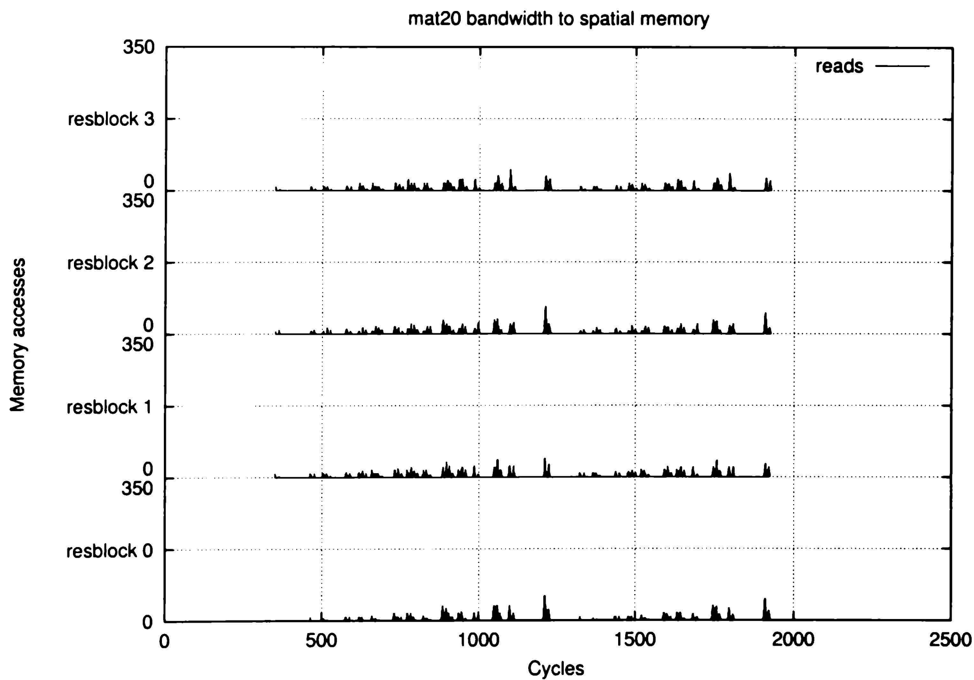


Figure D.28: Twisted memory bandwidth profile to spatial memory for matrix multiply (20)

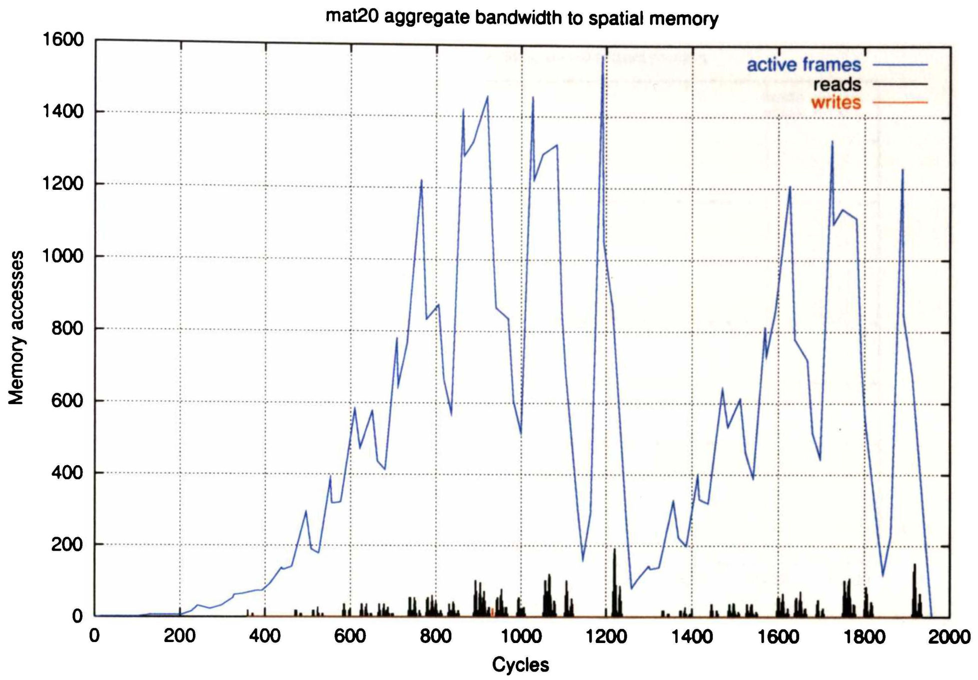


Figure D.29: Twisted memory bandwidth profile to level two for matrix multiply (20)

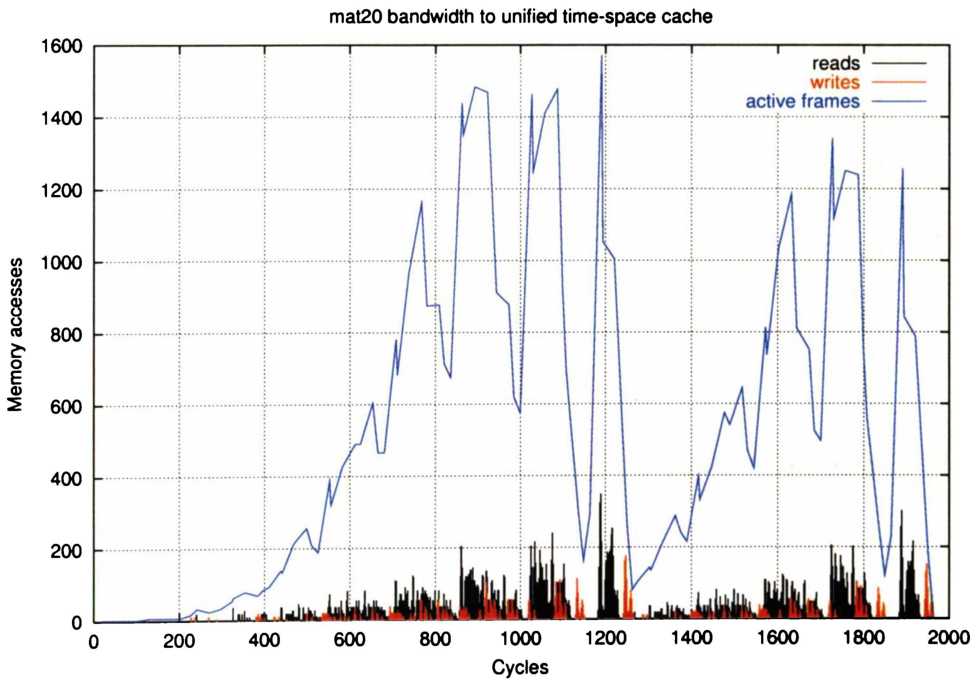


Figure D.30: Twisted memory aggregate bandwidth profile to spatial memory for matrix multiply (20)

D.6 Quicksort 1 (200)

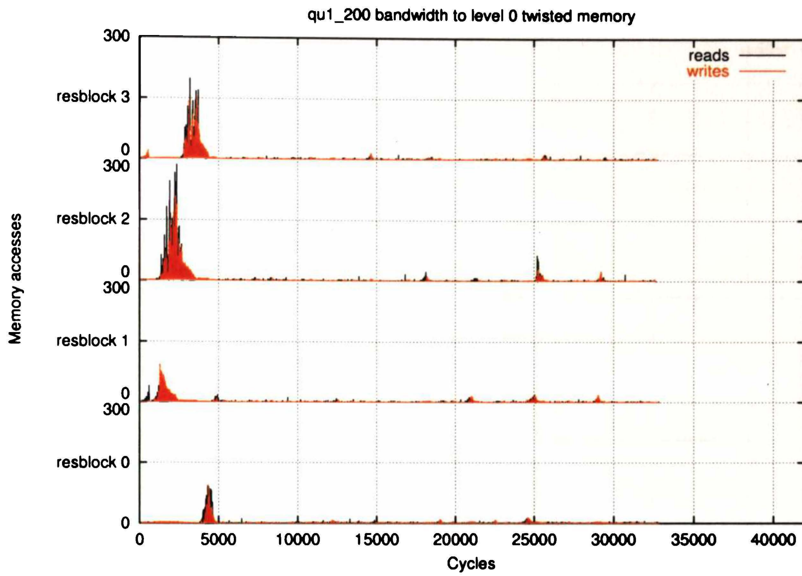


Figure D.31: Twisted memory bandwidth profile to level zero for quicksort 1 (200)

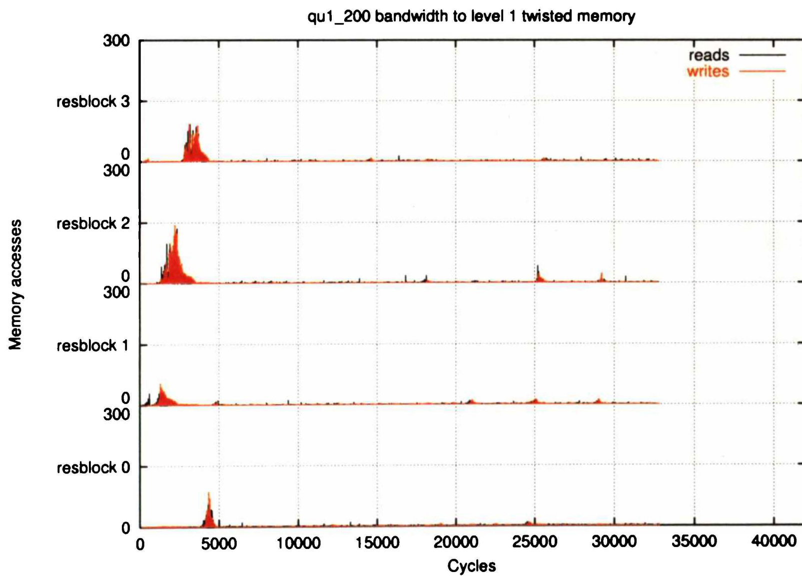


Figure D.32: Twisted memory bandwidth profile to level one for quicksort 1 (200)

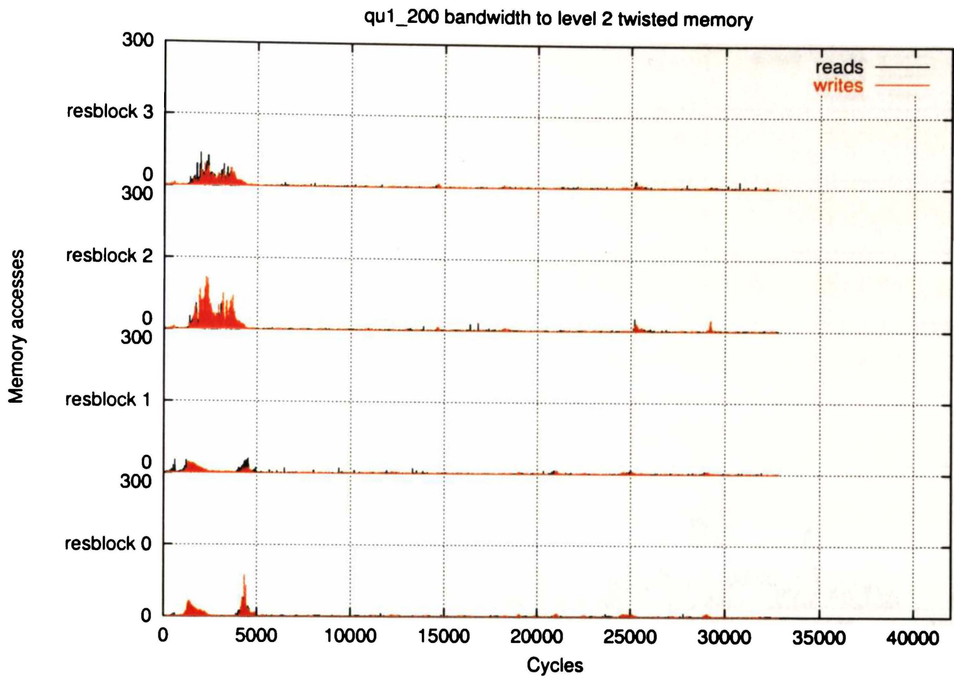


Figure D.33: Twisted memory bandwidth profile to level two for quicksort 1 (200)

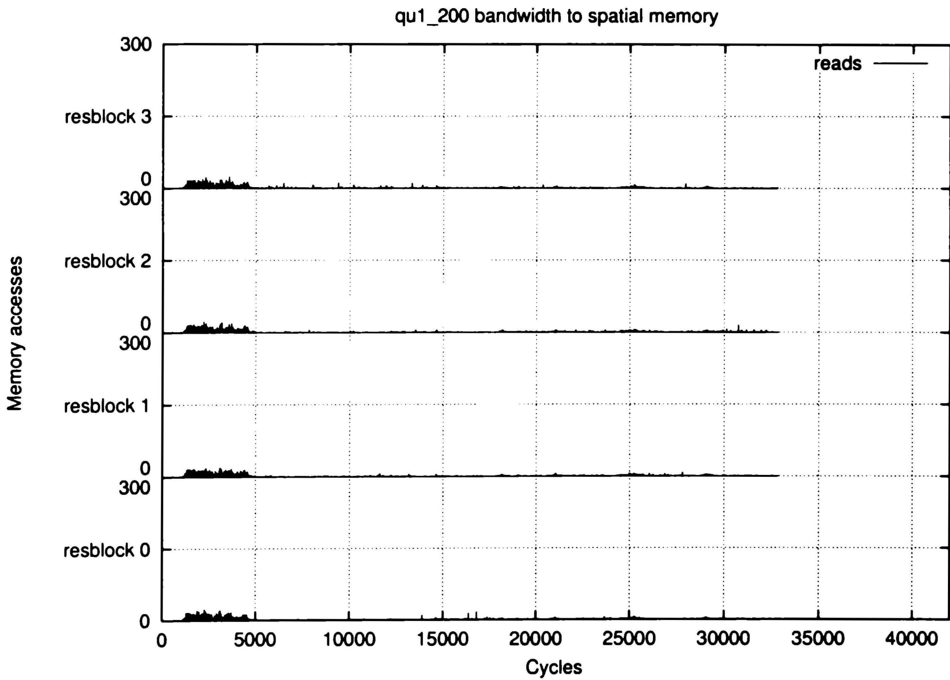


Figure D.34: Twisted memory bandwidth profile to spatial memory for quicksort 1 (200)

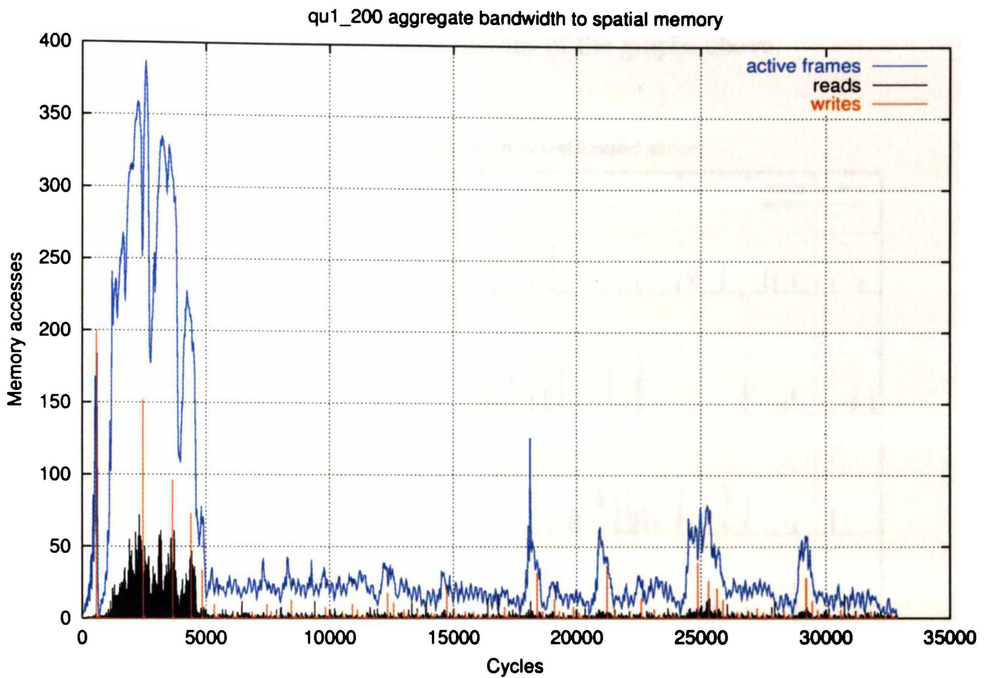


Figure D.35: Twisted memory bandwidth profile to level two for quicksort 1 (200)

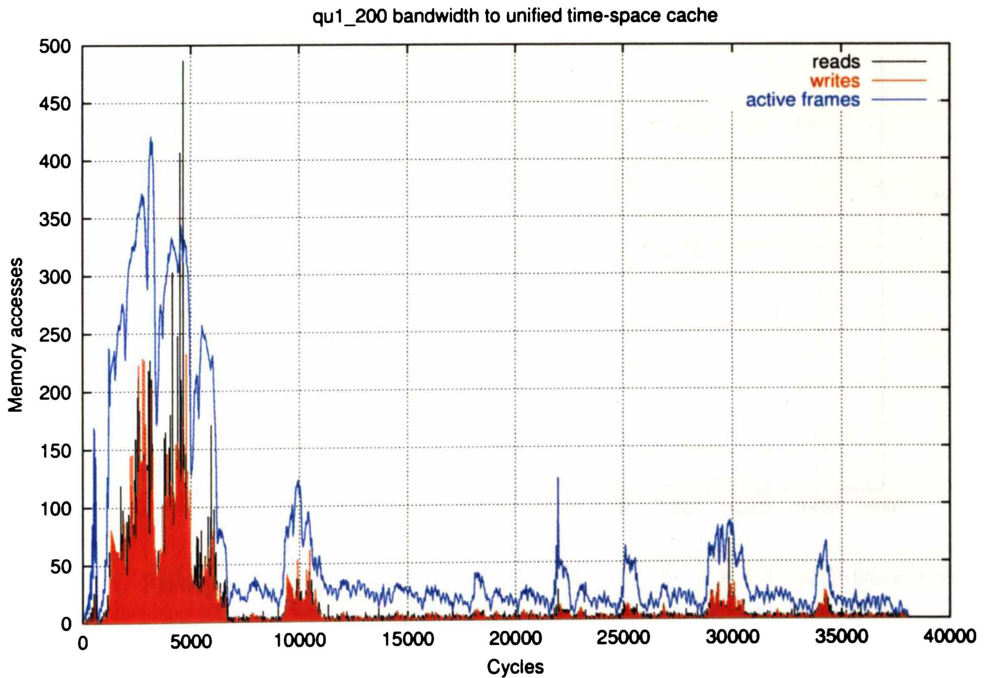


Figure D.36: Twisted memory aggregate bandwidth profile to spatial memory for quicksort 1 (200)

D.7 Enlarged Regions of Level Zero Bandwidth Profile Graphs

The graphs in this section show enlarged regions of the graphs above.

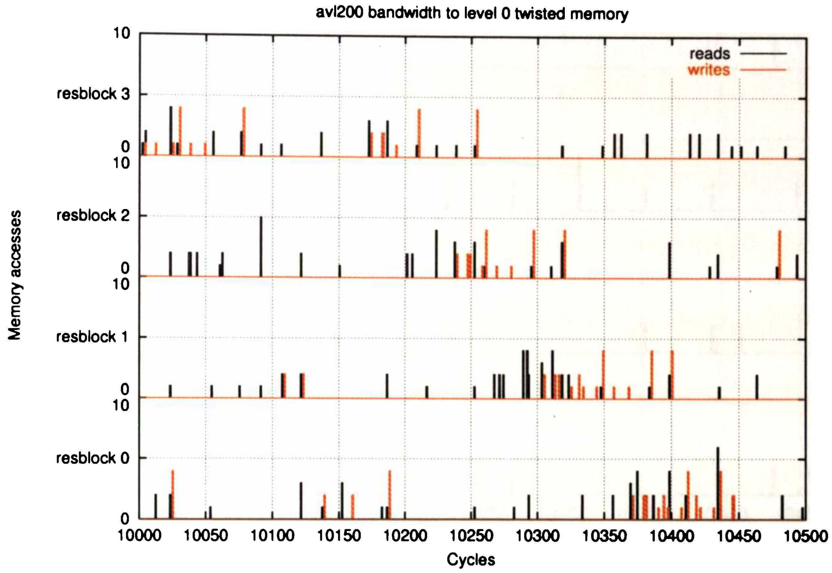


Figure D.37: Enlarged region of level zero bandwidth profile graphs for AVL(200)

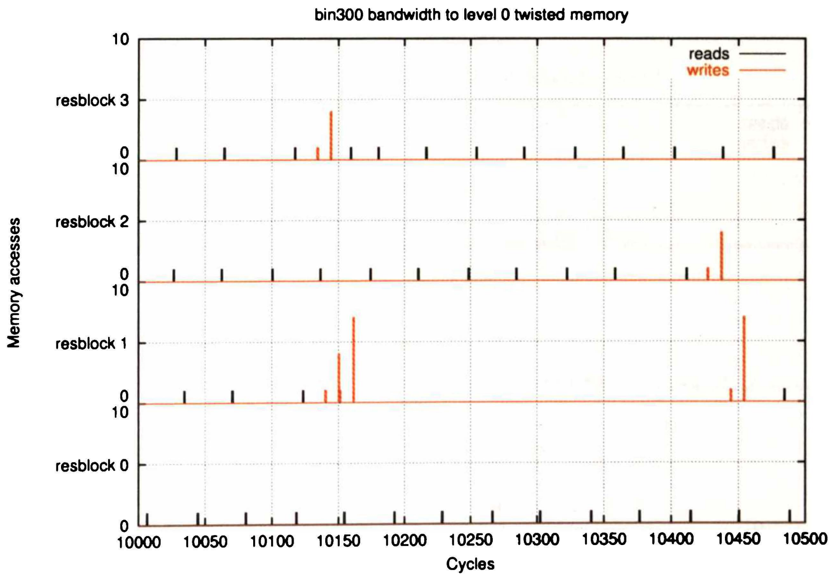


Figure D.38: Enlarged region of level zero bandwidth profile graphs for binary tree (300)

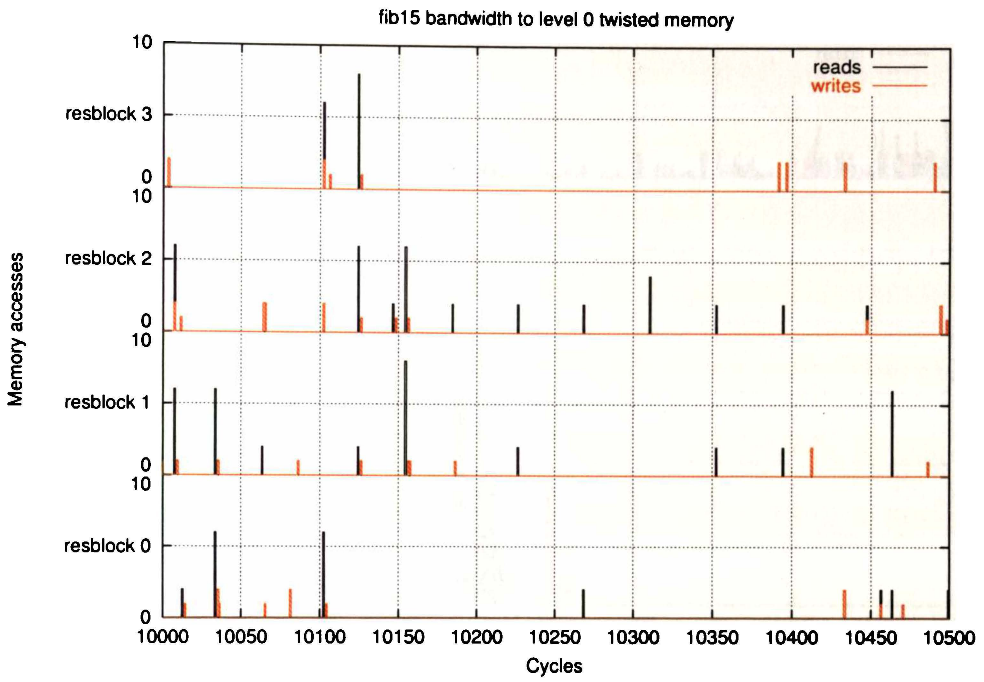


Figure D.39: Enlarged region of level zero bandwidth profile graphs for Fibonacci (15)

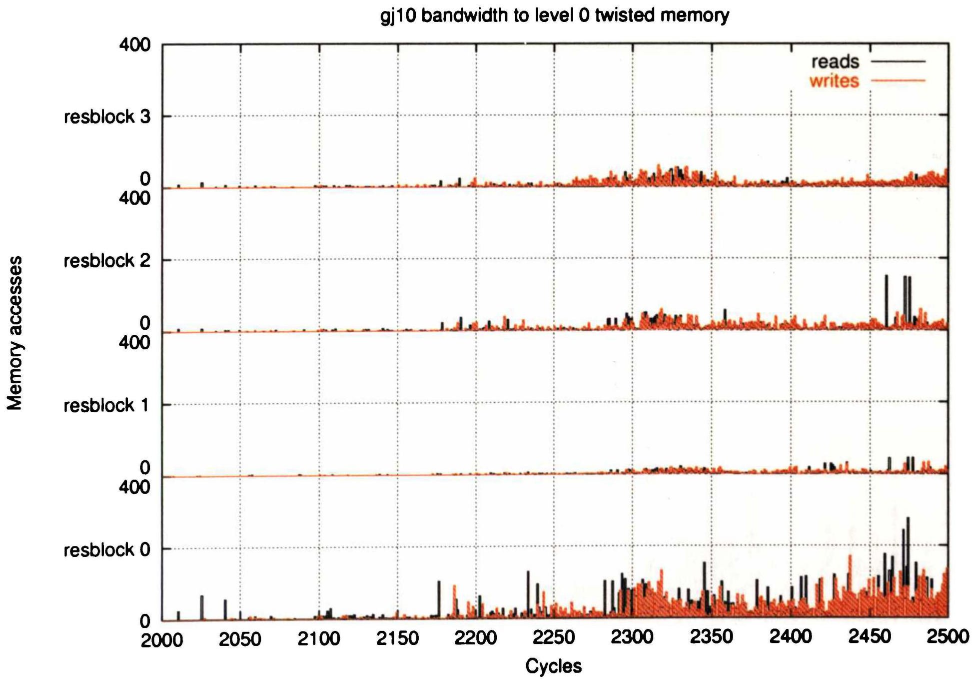


Figure D.40: Enlarged region of level zero bandwidth profile graphs for Gauss-Jordan (10)

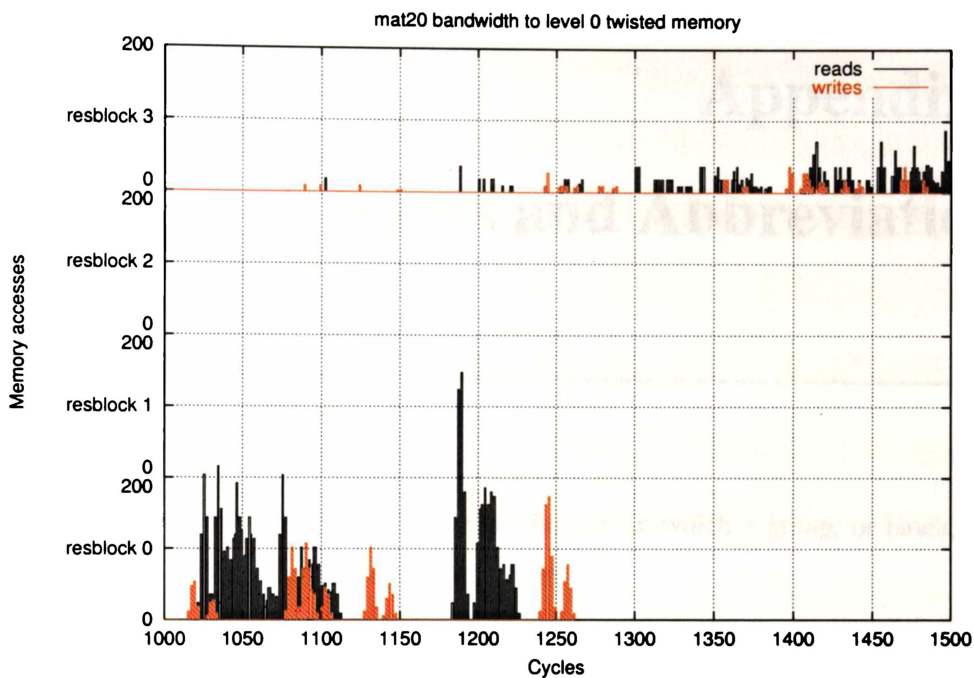


Figure D.41: Enlarged region of level zero bandwidth profile graphs for matrix multiply (20)

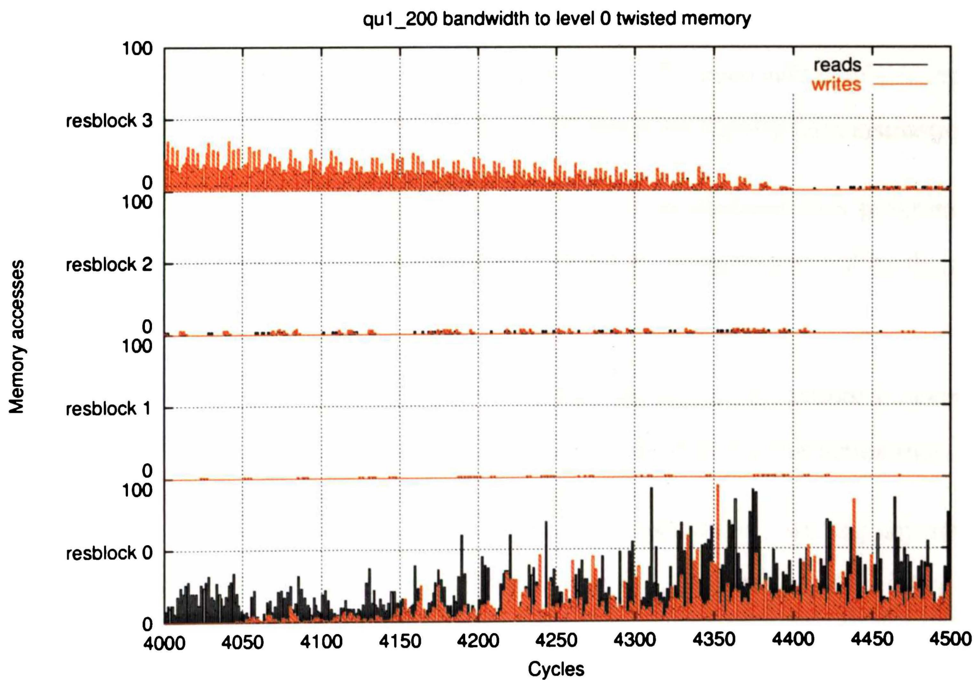


Figure D.42: Enlarged region of level zero bandwidth profile graphs for quicksort 1 (200)

Appendix E
Glossary of Terms and Abbreviations

block structured architecture (BSA) An architecture in which a group, or block, of instructions is treated as the basic unit of work.

cancelback The removal of the most speculative task to free up resources for execution of tasks earlier in the virtual order. Originally a Time Warp term.

causal order Dependent order of instructions. See virtual order.

control independence The situation in which a region of code will be executed regardless of the outcome of a conditional branch.

control flow graph (CFG) A directed graph consisting of named nodes which represent instructions, and arcs which represent control dependencies between instructions.

data dependence The situation between two sequential instructions in a program when the first instruction produces a result that is used as an input operand by the second instruction.

dataflow execution The execution model in which the issuing of instructions is determined by the flow of data. Once the input operands are available the instruction may issue.

fossil collection Retiring tasks when they become non-speculative, writing any stores to architectural memory and freeing resources for reuse. Originally a Time Warp term.

frame A hardware unit in the WarpEngine which holds up to 16 instructions with their input registers and output register destinations, and issues them for execution.

global virtual time (GVT) The earliest instruction that is actively executing, guaranteed to be non-speculative. All instructions earlier in the virtual order may be retired since they have completed and will never be rolled back. Originally a Time Warp term.

instruction block A group of up to 16 instructions, an instance of which is loaded into a frame for execution.

instruction level parallelism (ILP) Executing two or more instructions in parallel, usually taken from a sequential instruction stream.

instructions per cycle (IPC) The number of instructions executed in a clock cycle by a processor, usually an average.

mis-speculation Speculatively executing an instruction either incorrectly, or with incorrect data.

real order The order in which instructions are actually executed. This may be different from the virtual order when out-of-order execution is being used.

reorder buffer (ROB) A set of storage locations holding instructions, and sometimes result values, in program order.

rescaling Reallocating the early, unused VTSs to the active frames earliest in the virtual order, freeing their VTSs for allocation to new frames.

resource block A linear array of frames which can be allocated in virtual order.

rollback Selectively undoing incorrectly speculatively executed instructions.

selective speculation Selecting a group of instructions to execute speculatively where two or more groups are available and only one can currently be executed.

spatial memory Conventional memory, which holds only the single most recent value written to each address.

squash Undoing the effects of all speculative instructions beyond a mis-speculation point.

task A group of instructions in a contiguous instruction window executed in parallel with other groups of instructions.

thread A largely independent region of code executed in parallel with other such regions.

transient states The results of speculatively executed instructions which are never committed and will be rolled back.

tree structured execution The dynamic execution pattern which arises from a CFG where more than one flow of control may be followed in parallel from any node.

time-space cache The part of the virtually ordered memory system which holds multiple versions of each memory location, applying at different times in the virtual order.

Time Warp An parallel discrete event simulation algorithm used to impose a causal ordering on distributed systems.

twisted memory A novel speculative memory cache which uses a hierarchy of memory cells and explicit VTSs to maintain the virtual order of memory accesses

virtual order The sequential program order of instructions, which dictates the dependencies between instructions.

virtual order simulation Simulating execution of instructions in the virtual order of the instructions, rather than the real order.

virtual timestamp (VTS) An explicit tag used to indicate the position in the virtual order of a frame.

WarpEngine A theoretical computer architecture developed to investigate aggressive speculative execution and used throughout this thesis.

Bibliography

Akkary, H. [1998]. *A Dynamic Multithreading Processor*. PhD thesis, Portland State University.

Akkary, H. and Driscoll, M. [1998]. A dynamic multithreading processor. In *Proceedings of the 31st Annual International Symposium on Microarchitecture* (pp. 226–236).

Ang, B. S., Chiou, D., Rudolph, L. and Arvind [1998]. The StarT-Voyager parallel system. In *Proceedings of the International Conference on Parallel Architectures and Compilation*. Paris, France.

August, D. I., Connors, D. A., Mahlke, S. A., Sias, J. W., Crozier, K. M., Cheng, B.-C., Eaton, P. R., Olaniran, Q. B. and Hwu, W. W. [1998]. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings 25th of the International Symposium Computer Architecture* (pp. 227–237).

Austin, T. and Sohi, G. [1995]. Zero-cycle loads: Microarchitectural support for reducing load latency. In *Proceedings of the 28th Annual International Symposium on Microarchitecure* (pp. 82–92).

Austin, T. M. and Sohi, G. [1992]. Dynamic dependency analysis of ordinary programs. In *Proceedings of the 19th Annual International Symposium on on Computer Architecture*.

Back, A. and Turner, S. [1995]. Time-stamp generation for optimistic parallel computing. In *Proceedings of the 28th Annual Simulation Symposium* (pp. 144–153). Phoenix, Arizona.

Barua, R., Lee, W., Amarasinghe, S. and Agarwal, A. [1999]. Maps: A compiler-managed memory system for Raw machines. In *Proceedings of the 26th International Symposium on Computer Architecture*. Atlanta, GA.

Bellenot, S. [1990]. Global virtual time algorithms. In *Proceedings of the SCS Multiconference on Distributed Simulation*, Volume 2 (pp. 122–127).

Biglari-Abhari, M., Liebelt, M. J. and Eshraghian, K. [1998]. Implementing a VLIW compiler: Motivation and trade-offs. In Morris, J. (Ed.), *3rd Australasian Computer Architecture Conference*, Volume 20 (pp. 37–46). Perth, Australia, Springer-Verlag.

Butler, M., Yeh, T.-Y., Patt, Y., Alsup, M., Scales, H. and Shebanow, M. [1991]. Single instruction stream parallelism is greater than two. In *18th Annual International Symposium on Computer Architecture* (pp. 276–286). New York, N.Y.

Calder, B. and Reinman, G. [2000]. A comparative survey of load speculation architectures. *Journal of Instruction Level Parallelism*, *1*, 1–39.

Calvert, J. [1997]. Design of the execution control structure for the WarpEngine optimistic CPU. Master's thesis, University of Waikato.

Chen, T.-F. and Baer, J.-L. [1995]. Effective hardware-based data prefetching for high-performance processors. *IEEE Transactions on Computers*, *5*, 609–623.

Chou, Y., Fung, J. and Shen, J. P. [1999]. Reducing branch misprediction penalties via dynamic control independence detection. In *Proceedings of the International Conference on Supercomputing*. Rhodes.

Chrysos, G. Z. and Emer, J. S. [1998]. Memory dependence prediction using store sets. In *Proceedings of the 25th International Symposium on Computer Architecture* (pp. 142–153).

Cleary, J. G. [1995]. WarpEngine instruction set. Internet Web Page. URL http://www.cs.waikato.ac.nz/timewarp/wengine/instset/we_inst.nov21995.html, visited November 2001.

Cleary, J. G., Pearson, M. W. and Kinawi, H. [1995]. The architecture of an optimistic CPU: The WarpEngine. In *Proceedings of HICSS*, Volume 1 (pp. 163–172). Hawaii.

Colwell, R., Nix, R., O'Donnell, J., Papworth, D. and Rodman, P. [1987]. A VLIW architecture for a trace scheduling compiler. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (pp. 180–192). Palo Alto, California.

Compaq [2000]. *Alpha 21264/EV6 Microprocessor Hardware Reference Manual.* Compaq Computer Corporation.

Corman, T. H., Leiserson, C. E. and Rivest, R. L. [1990]. *Introduction to Algorithms.* New York: McGraw-Hill Book Company.

Dennis, J. and Misunas, D. [1975]. A preliminary architecture for a basic dataflow processor. In *Proceedings of the 2nd Annual Symposium on Computer Architecture* (pp. 126–132). Houston, Texas.

Eeckhout, L., Aa, T. V., Goeman, B., Vandierendonck, H., Lauwereins, R. and Bosschere, K. D. [2001]. Application domains for fixed-length block structured architectures. In *Proceedings of Australasian Computer Systems Architecture Conference.* Gold Coast, Australia.

Eeckhout, L., Bosschere, K. D. and Neefs, H. [2000]. On the feasibility of fixed-length block structured architectures. In Heiser, G. (Ed.), *Proceedings of the 5th Australasian Computer Architecture Conference* (pp. 17–25).

Fisher, J. [1983]. Very long instruction word architectures and the ELI-52. In *Proceedings of the 10th Annual Symposium on Computer Architecture* (pp. 140–150). Stockholm.

Fisher, J. A. [1984]. The VLIW machine: A multiprocessor for compiling scientific code. *IEEE Computer, 17(7).*

Franklin, M. [1993]. *The Multiscalar Architecture.* PhD thesis, University of Wisconsin-Madison.

Franklin, M. and Sohi, G. S. [1996]. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers, 45(5).*

Fujimoto, R. [1990a]. Parallel discrete event simulation. *Communications of the ACM, 33(10),* 30–53.

Fujimoto, R. [1990b]. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation, 6(3)*, 211–239.

Fujimoto, R. M. [1989]. The virtual time machine. In *Proceedings of the International Symposium on Parallel Algorithms and Architectures* (pp. 199–208).

Fujimoto, R. M. and Hybinette, M. [1997]. Computing global virtual time in shared-memory multiprocessors. *ACM Transactions on Modeling and Computer Simulation, 7(4)*, 425–446.

Fujimoto, R. M., Tsai, J.-J. and Gopalakrishnan, G. C. [1992]. Design and evaluation of the rollback chip: Special purpose hardware for Time Warp. *IEEE Transactions on Computers, 41(1)*, 68–82.

Gaudiot, J. [1986]. Structure handling in dataflow systems. *IEEE Transactions on Computers, C-35(6)*, 489–502.

Gharachorloo, K., Gupta, A. and Hennessey, J. [1991]. Two techniques to enhance the performance of memory consistency models. In *Proceedings of the International Conference on Parallel Processing* (pp. 245–257).

Gomes, F., Cleary, J. and Unger, B. [1992]. GVT approximation in optimistic parallel discrete event simulation: a survey. Technical report, Computer Science, University of Calgary, Calgary, Canada.

Gonzalez, J. and Gonzalez, A. [1997]. Speculative execution via address prediction and data prefetching. In *Proceedings of the 11th International Conference on Supercomputing* (pp. 547–564).

Gonzalez, J. and Gonzalez, A. [1998]. The potential of data value speculation to boost ILP. In *Proceedings of the 12th International Conference on Supercomputing.*

Gopal, S., Vijaykumar, T. N., Smith, J. E. and Sohi, G. S. [1998]. Speculative versioning cache. In *The 4th International Symposium on High Performance Computer Architecture*. Las Vegas, Nevada.

Grunwald, D., Klauser, A., Manne, S. and Pleszkun, A. [1998]. Confidence estimation for speculation control. In *Proceedings of the 25th International Symposium on Computer Architecture* (pp. 122–131).

Gurd, J. R., Kirkham, C. C. and Watson, I. [1985]. The Manchester prototype dataflow computer. In *Communications of the ACM*, Volume 28 (pp. 24–52).

Hammond, L., Hubbert, B. A., Siu, M., Prabhu, M. K., Chen, M. and Olukotun, K. [2000]. The Stanford Hydra CMP. *IEEE Micro, 20(2)*, 71–84.

Hammond, L., Willey, M. and Olukotun, K. [1998]. Data speculation support for a chip multiprocessor. In *Proceedings of the 8th International Conference on Architecture Support for Programming Languages and Operating Systems.*

Hao, E., Chang, P.-Y., Evers, M. and Patt, Y. N. [1998]. Increasing the instruction fetch rate via block-structured instruction set architectures. *International Journal of Parallel Programming, 26(4)*, 449–458.

Hennessy, J. L. and Patterson, D. A. [1996]. *Computer Architecture: A Quantitative Approach* (second Ed.). San Francisco: Morgan Kaufmann Publishers, Inc.

Hwu, W.-M. and Patt, Y. [1987]. Checkpoint repair for out-of-order execution machines. In *Proceedings of the 14th International Symposium on Computer Architecture* (pp. 297–307).

Intel [1999]. *Intel Architecture Optimisation Reference Manual.* Intel Corporation.

Intel [2000]. *A Detailed Look Inside the Intel Pentium 4 Processor.* Intel Corporation.

Jacobsen, E., Rotenberg, E. and Smith, J. E. [1996]. Assigning confidence to conditional branch predictions. In *Proceedings of the 29th International Symposium on Microarchitecture.*

Jacobsen, Q., Bennett, S., Sharma, N. and Smith, J. E. [1997]. Control flow speculation in multiscalar processors. In *Proceedings of the 3rd International Symposium on High Performance Computer Architecture.*

Jefferson, D. [1985]. Virtual time. *Transactions on Programming Languages and Systems, 7(3)*, 404–425.

Jefferson, D. [1990]. Virtual time II: Storage management in distributed simulation. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing* (pp. 75–89).

Jouppi, N. P. and Wall, D. W. [1989]. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the 3rd International Conference on Architecture Support for Programming Languages and Operating Systems* (pp. 272–282). New York, N.Y.

Keller, R. [1975]. Look-ahead processors. *ACM Computing Surveys, 7*, 66–72.

Lam, M. S. and Wilson, R. P. [1992]. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on on Computer Architecture* (pp. 46–57).

Lamport, L. [1979]. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers, C-28(9)*, 690–691.

Lawrie, D. [1975]. Access and alignment of data in an array processor. *IEEE Transactions on Computers, C-24(12)*, 1145–1155.

Lewis, H. R. and Denenberg, L. [1991]. *Data Structures and Their Algorithms.* Harper Collins.

Lin, Y.-B. and Lazowska, E. [1990]. Determining the global virtual time in a distributed simulation. In *Proceedings of the 1990 International Conference on Parallel Processing*, Volume 3 (pp. 201–209).

Lipasti, M. H. and Shen, J. P. [1996]. Exceeding the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE Symposium on Microarchitecture* (pp. 226–232).

Lipasti, M. H. and Shen, J. P. [1998]. Exploiting value locality to exceed the dataflow limit. In *International Journal of Parallel Processing*, Volume 26 (pp. 505–538).

Littin, R. H. [1999]. WarpEngine test programs. Internet Web Page. URL http://www.cs.waikato.ac.nz/timewarp/wengine/testcode/, visited November 2001.

Littin, R. H. [2000]. *Design and Evaluation of an Optimistic CPU: The WarpEngine.* PhD thesis, University of Waikato, Hamilton, New Zealand.

Littin, R. H., McWha, J. A. D., Pearson, M. W. and Cleary, J. G. [1998]. Block based execution and task level parallelism. In *Proceedings of the 3rd Australasian Computer Architecture Conference.* Perth, Australia.

Mahlke, S., Hank, R., McCormick, J., August, D. and Hwu, W.-M. [1995]. A comparison of full and partial predicated support for ILP processors. In *Proceedings of the 22nd International Symposium on Computer Architecture* (pp. 138–149). Santa Margherita Ligure.

Manne, S., Klauser, A. and Grunwald, D. [1998]. Pipeline gating: Speculation control for energy reduction. In *Proceedings of the 25th International Symposium on Computer Architecture* (pp. 132–141).

Marcuello, P. and Gonzlez, A. [1998]. Speculative multithreaded processors. In *Proceedings of the ACM International Conference on Supercomputing*. Melbourne, Australia.

McFarling, S. [1993]. Combining branch predictors. Technical Report WRL Technical Notes TN-36, Digital Western Research Laboratory.

Melvin, S. and Patt, Y. [1995]. Enhancing instruction scheduling with a block-structured ISA. *International Journal of Parallel Programming, 23(3)*, 221–243.

Moshovos, A., Breach, S., Vijaykumar, T. and Sohi, G. [1997]. Dynamic speculation and synchronization of data dependences. In *24th International Symposium on Computer Architecture*.

Moshovos, A. and Sohi, G. S. [2000]. Memory dependence speculation tradeoffs in centralized, continuous-window superscalar processors. In *Proceedings of the 6th International Symposium on High Performance Computer Architecture* (pp. 301–312). Toulouse, France.

Neefs, H., De Bosschere, K. and Van Campenhout, J. [1997]. Issues in compilation for fixed-length block structured instruction set architectures. In *Workshop on Interaction between Compilers and Computer Architectures*.

Neefs, H. and Van Campenhout, J. [1996]. A microarchitecture for a fixed length block structured instruction set architecture. In *Eighth IASTED International Conference on Parallel and Distributed Computing and Systems*. Chicago.

Nicolau, A. and Fisher, J. [1984]. Measuring the parallelism available for very long instruction word architectures. *IEEE Transactions on Computers, C-33(11)*, 968–976.

Nikhil, R., Papadopoulos, G. and Arvind [1992]. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th International Symposium on Computer ARchitecure* (pp. 156–167). Gold Coast.

Papadopoulos, G. and Culler, D. E. [1990]. Monsoon: An explicit token store architecture. In *Proceedings of the 17th International Symposium on Computer Architecture*. Seattle, Washington.

Patterson, D. A. [1985]. Reduced instruction set computers. *Communications of the ACM, 28(1)*, 8–21.

Pearson, M. W., Littin, R. H., McWha, J. A. D. and Cleary, J. G. [1997]. Applying Time Warp to CPU design. In *High Performance Computing Conference 1997*. Bangalore, India.

Postiff, M. A., Greene, D. A. and Mudge, T. N. [1999]. The limits of instruction level parallelism in SPEC95 applications. *Computer Architecture News, 27(1)*, 31–34.

Press, W. H. [1992]. *Numerical Recipes in C*. Cambridge University Press.

Quinn, M. J. [1987]. *Designing Efficient Algorithms for Parallel Computers*. McGraw–Hill.

Ramamoorthy, C. and Li, H. [1977]. Pipeline architecture. *ACM Computing Surveys, 9(1)*, 61–102.

Rotenberg, E., Bennett, S. and Smith, J. E. [1996]. Trace cache: A low latency approach to high bandwidth instruction fetching. In *Proceedings of the 29th International Symposium on Microarchitecture*. Paris, France.

Rotenberg, E., Bennett, S. and Smith, J. E. [1999a]. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers, 48(2)*, 111–120.

Rotenberg, E., Jacobsen, Q., Sazeides, Y. and Smith, J. [1997]. Trace processors. In *Proceedings of the 30th International Symposium on Microarchitecture*.

Rotenberg, E., Jacobsen, Q. and Smith, J. [1999b]. A study of control independence in superscalar processors. In *The 5th International Symposium on High Performance Computer Architecture*. Orlando, Florida.

Rotenberg, E. and Smith, J. [1999]. Control independence in trace processors. In *Proceedings of the 32nd International Symposium on Microarchitecture*.

Roth, A. and Sohi, G. S. [2000]. Register integration: A simple and efficient implementation of squash reuse. In *Proceedings of the 33rd International Symposium on Microarchitecture.*

Russel, R. [1978]. The CRAY-1 computer system. *Communications of the ACM, 21*, 63–72.

Sazeides, Y. and Smith, J. [1997]. The predictability of data values. In *Proceedings of the 30th International Symposium on Microarchitecture.*

Schlansker, M. and Rau, B. [2000]. EPIC: Explicitly parallel instruction computing. *IEEE Computer, 33(2)*, 37–45.

Smith, J. E. [1981]. A study of branch prediction strategies. In *Proceedings of the 8th Annual International Symposium on Computer Architecture* (pp. 135–148).

Smith, J. E. and Pleszkun, A. R. [1988]. Implementing precise interrupts in pipelined processors. *IEEE Transactions on Computers, 37(5)*, 562–573.

Smith, M. D., Johnson, M. and Horowitz, M. A. [1989]. Limits on multiple instruction issue. In *3rd International Conference on Architecture Support for Programming Languages and Operating Systems* (pp. 290–302). New York, N.Y.

Snelling, D. F. [1993]. *The Design and Analysis of a Stateless Data-Flow Architecture.* PhD thesis, University of Manchester.

Snelling, D. F. and Egan, G. K. [1994]. A comparative study of data-flow architectures. Technical Report UMCS-94-4-3, University of Manchester.

Sodani, A. and Sohi, G. S. [1997]. Dynamic instruction reuse. In *Proceedings of the 24th International Symposium on Computer Architecture.*

Sodani, A. and Sohi, G. S. [1998]. Understanding the differences between value prediction and instruction reuse. In *Proceedings of the 31st International Symposium on Microarchitecture.*

Sohi, G. S., Breach, S. and Vijaykumar, T. N. [1995]. Multiscalar processors. In *22nd International Symposium on Computer Architecture* (pp. 414–425).

Srini, V. P. [1986]. An architectural comparison of dataflow systems. *IEEE Computer, 19(3)*, 68–87.

Steffan, J. G., Colohan, C., Zhai, A. and Mowry, T. [2000]. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*.

Steffan, J. G. and Mowry, T. [1998]. The potential for using thread-level data speculation to facilitate automatic parallelisation. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*.

Thornton, J. E. [1964]. Parallel operation in the Control Data 6600. In *Proceedings AFIPS Fall Joint Computer Conference*, Volume 26 (pp. 33–40).

Tjaden, G. and Flynn, M. [1970]. Detection and parallel execution of parallel instructions. *IEEE Transactions on Computers, C-19(10)*, 889–895.

Tomasulo, R. M. [1967]. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal, 11*, 25–33.

Treleaven, P., Brownbridge, D. and Hopkins, R. [1982]. Data-driven and demand-driven computer architectures. *ACM Computing Surveys, 14*, 93–143.

Tremblay, M., Chan, J., Chudhry, S., Conigliaro, A. and Tse, S. S. [2000]. The MAJC architecture: a synthesis of parallelism and scalability. *IEEE Micro, 20(6)*, 12–25.

Tsai, J.-Y., Huang, J., Amlo, C., Lilja, D. and Yew, P.-C. [1999]. The superthreaded processor architecture. *IEEE Transactions on Computers, 48(9)*.

Tyson, G. and Austin, T. [1999]. Memory renaming: Fast, early and accurate processing of memory communication. *International Journal of Parallel Programming, 27(5)*.

Uht, A. K. and Sindagi, V. [1995]. Disjoint eager execution: An optimal form of speculative execution. In *Proceedings of the 28th International Symposium on Microarchitecture*.

Uht, A. K., Sindagi, V. and Somanathan, S. [1997]. Branch effect reduction techniques. *IEEE Computer, 30(5)*, 71–81.

Veen, A. H. [1986]. Dataflow machine architecture. *ACM Computing Surveys, 18(4)*, 365–396.

Vijaykumar, T. and Sohi, G. S. [1998]. Task selection for a multiscalar processor. In *Proceedings of the 31st International Symposium on Microarchitecture*.

Waingold, E., Taylor, M., Srikrishna, D., Sarkar, V., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S. and Agarwal, A. [1997]. Baring it all to the software: Raw machines. *IEEE Computer, 30(9)*, 86–93.

Wall, D. W. [1991]. Limits of instruction-level parallelism. In *4th International Conference on Architecture Support for Programming Languages and Operating Systems* (pp. 176–188). New York, N.Y.

Wallace, S., Calder, B. and Tullsen, D. M. [1998]. Threaded multiple path execution. In *Proceedings of the 25th Annual International Symposium on Computer Architecture.*

Wang, K. and Franklin, M. [1997]. Highly accurate data value prediction using hybrid predictors. In *Proceedings of the 30th Annual Symposium on Microarchitecture.*

Wang, S. and Uht, A. [1990]. Ideograph/ideogram: Framework/architecture for eager evaluation. In *Proceedings of the 23rd Annual Symposium on Microprogramming and Microarchitecture* (pp. 125–134).

Wu, C. and Feng, T. [1980]. On a class of multistage interconnection networks. *IEEE Transactions on Computers, C-29(8)*, 694–702.

Yeh, T.-Y., Marr, D. and Patt, Y. [1993]. Increasing the instruction fetch rate via multiple branch prediction and a branch address cache. In *Proceedings of the 7th International Conference on Supercomputing* (pp. 67–76).

Yeh, T.-Y. and Patt, Y. N. [1993]. A comparison of dynamic branch predictors that use two levels of branch history. In *Proceedings of the 20th Annual International Symposium on Computer Architecture* (pp. 257–266).