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:

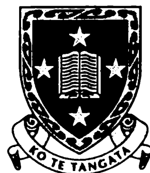
- Any use you make of these documents or images must be for research or private study purposes only, and you may not make them available to any other person.
- Authors control the copyright of their thesis. You will recognise the author's right to be identified as the author of the thesis, and due acknowledgement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

Design and Evaluation of an Optimistic CPU: The WarpEngine

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

Richard H. Littin

Department of Computer Science



The
University
of Waikato
Te Whare Wānanga
o Waikato

Hamilton, New Zealand

January 2000

© 2000 Richard H. Littin

Abstract

Instruction pipelining, out-of-order execution, and branch prediction are techniques that improve performance in processors by manipulating the flow of instructions. These control flow manipulations alone are not adequate to allow large numbers of instructions to execute in parallel because performance is limited by accesses to the relatively slow memory system. Performance can be improved by speculating on the outcomes of control decisions, and the values of data in memory, returning results early. This thesis investigates the requirements of an architecture that speculates on control flow decisions and data values to improve performance through instruction level parallelism.

A new architecture, the WarpEngine, that speculates on control flow decisions and data values is presented. This architecture is shown to have the potential to extract performance through parallelism an order of magnitude larger than that obtained by contemporary microprocessors. Control speculation is achieved using a novel tree-based mechanism that produces multiple flows of control. This scalable mechanism is shown to generate a large group of instructions that can execute in parallel. Also, it is essential that memory accesses are allowed to occur out of programmed order. This form of data speculation is shown to break false data dependencies, improving performance. The use of state saving resources is examined and the limitations of in-order retirement schemes are shown. These results indicate that the management of these resources is critical to obtaining good performance.

Virtual ordered simulation is introduced as a new simulation methodology for modelling out-of-order and speculative architectures. This novel simulation technique is unique because each instruction is only inspected and processed once, and unlike other simulation methodologies unlimited resources can be modelled. Individual components can be constrained in isolation so that their effect on performance can be examined in detail.

Investigations performed assuming unbounded resources provide new insight into the limits imposed by individual processor components.

The architecture presented shows potential for performance well beyond that of contemporary and research architectures. The insights into the limitations of processor components apply to many computer architectures.

Acknowledgements

I would like to thank my supervisors Murray Pearson and John Cleary for providing ideas, feedback and support throughout my PhD. I would also like to thank David McWha for his many ideas, suggestions, and bug reports.

Thanks to mum and dad, Nola and Ron Littin, for giving me the opportunity to attend university and for the support they've given me throughout my academic career.

Thanks to the proof readers of my thesis, Jamie Littin who got the "rough draft", and Len Trigg, Stuart Inglis, Kirsten Thompson and Nicola Inglis who reviewed the final draft during their Christmas breaks.

I would like to acknowledge the other computer science grads, who have provided great lunchtime conversations, an excellent social/sporting environment and the odd good idea from time to time. Also the staff of the Computer Science Department and the School of Computing and Mathematical Sciences who have provided space, equipment and great service.

Contents

Abstract iii

Acknowledgements v

List of Figures xi

List of Tables xvii

1 Introduction 1

1.1 Obtaining parallelism 2

1.2 Challenges 3

1.3 Problems 4

1.4 A solution 5

1.5 Thesis claims 6

1.6 Thesis layout 7

2 Current solutions 9

2.1 ILP processing 9

2.1.1 Fundamentals 10

2.1.2 Constraints 10

2.2 Extracting ILP 13

2.2.1 Software support 13

2.2.2 Hardware support 14

2.3 ILP paradigms 17

2.3.1 Vector 18

2.3.2 VLIW 19

2.3.3 Superscalar 20

2.3.4 Multiscalar 21

2.3.5 Data flow 24

2.4 Studies of ILP 25

2.5 Summary 26

3 The WarpEngine 29

3.1 Time Warp mechanism . . . 29

3.1.1 Global virtual time . . . 30

3.1.2 Cancelback . . . 31

3.1.3 Architecture . . . 32

3.2 Extracting parallelism . . . 33

3.2.1 Control flow speculation . . . 33

3.2.2 Data value speculation . . . 36

3.3 WarpEngine architecture . . . 38

3.3.1 Instruction set . . . 38

3.3.2 Code store . . . 40

3.3.3 Frames . . . 41

3.3.4 Function units . . . 42

3.3.5 Memory system . . . 42

3.3.6 Synchronisation mechanism . . . 44

3.3.7 Instruction execution . . . 45

3.4 Comparison . . . 46

3.4.1 Superscalar . . . 47

3.4.2 Multiscalar 47

3.4.3 Data flow 48

3.5 Summary . . . 49

4 Simulation 51

4.1 Background 52

4.2 Modelling execution paradigms 53

4.2.1 Sequential execution 53

4.2.2 Out-of-order execution 54

4.3 Virtual ordered simulation 56

4.3.1 Trade-offs 59

4.3.2 Limiting resources 59

4.3.3 Extensions 62

4.4 WarpEngine simulator 70

4.4.1 Components 70

4.4.2 Operation 71

4.4.3 Performance 73

4.5 Summary 73

5 Test programs and compilation 75

5.1 Test programs 75

5.1.1 Matrix manipulation 76

5.1.2 Sorting 79

5.1.3 Dynamic structures 81

5.1.4 Recursion 83

5.2 Control tree optimisations 84

5.3 Compilation 87

5.3.1 Techniques used 88

5.4 Instruction set evaluation 89

5.5 Summary 90

6 Critical instructions 91

6.1 Instruction usage 92

6.1.1 Method 92

6.1.2 Results 92

6.2 Critical paths 95

6.2.1 Method 95

6.2.2 Results 96

6.3 Summary 99

7 Performance **101**

7.1 Unlimited resources 102

7.1.1 Results 103

7.1.2 Complexity analysis 108

7.1.3 Data sets 109

7.1.4 Algorithmic decisions 112

7.2 Memory access constraints 116

7.2.1 Method 117

7.2.2 Results 119

7.2.3 Machine rankings 123

7.3 Limiting state saving resources 127

7.3.1 Method 127

7.3.2 Results 127

7.3.3 Declining performance 133

7.4 Summary 136

8 Resource usage **139**

8.1 Slot usage 139

8.2 Frame usage 140

8.2.1 Method 141

8.2.2 Results 141

8.2.3 Unlimited resources 142

8.2.4 Limited resources 149

8.2.5 Code scheduling 157

8.3 Time-space cache usage 159

8.3.1 Results 159

8.4 Possible solutions 164

8.4.1 Storing less state 165

8.4.2 Out-of-order retirement 166

8.5 Summary 167

9 **Summary & conclusions** 169
9.1 Implications for CPU design . . . 169
9.2 The WarpEngine . . . 170
9.3 Conclusions . . . 171
9.4 Future work . . . 172

Appendices

A **WarpEngine Instruction Set** 175

B **Test Code** 179

C **Extra graphs** 189
C.1 Critical path equations . . . 189
C.2 IPC with unlimited resources . . . 193
C.3 IPC with memory access constraints . . . 194
C.4 Memory constraint meshes . . . 197
C.5 IPC with frame limits . . . 200
C.6 Unlimited resource frame usage . . . 202
C.7 Limited resource frame usage . . . 204
C.8 Time-space cache entries per frame . . . 206

Bibliography 209

List of Figures

2.1 Architecture of the Multiscalar processor. 22

3.1 Message timestamp updates through a Time Warp object. 30
3.2 CFGs for a) `if-then-else`, and b) `while` statements. 34
3.3 Control tree for an if-then-else statement. 35
3.4 Control tree for loop iterations. 36
3.5 Control tree for function calls. 37
3.6 Components of the WarpEngine and their connections. 39
3.7 Layout of a frame. 41
3.8 Components of the timestamped memory system. 43
3.9 Time-space cache entries with a write occurring out-of-order. 44
3.10 Instruction flow within a WarpEngine frame. 46

4.1 Design process for a computer architecture. 51
4.2 C code for the addition of two decimal numbers stored as arrays of characters. 53
4.3 MIPS assembly for lines 8 to 10 of the C code in Figure 4.2. 54
4.4 Inter-instruction dependencies for instructions 7 to 18 of Figure 4.3. 55
4.5 Out-of-order execution for instructions 7 to 18 of Figure 4.3. 55
4.6 The flow of timestamp values during instruction processing. 57
4.7 Timestamp values during virtual ordered simulation of instructions 7 to 18 of Figure 4.3. 57
4.8 Using time-range buckets to model limited processing resources. 61
4.9 Communication between a frame and the limited frames mechanism. 62
4.10 Generating an output critical path list from two input critical path lists. . . 63
4.11 Critical path equations with variable instruction processing latencies. . . . 64
4.12 Algorithm for inserting a new path equation into the path equations list. . 67
4.13 Merging of input history lists to form an output history list. 69
4.14 Components of the WarpEngine virtual order simulator. 70

4.15 Frame stack operations during the processing of a single frame. 72

5.1 Sequential and parallel components within matrix multiplication. 77
5.2 Sequential and parallel components within transitive closure. 78
5.3 Sequential and parallel components within quicksort. 80
5.4 Multiple keys inserted into a binary tree a) with no dependencies, and b) with dependencies. 82
5.5 Control trees for a `for` loop with a tree-based control backbone. 85
5.6 Early start tree-based control backbone for a `for` loop. 85
5.7 Tree-based control backbone for a `while` loop. 86

6.1 Instruction mixes for *mat*(50), *trans*(50), *gj*(30), *fib*(30), and *fibf*(30). 93
6.2 Instruction mixes for *heap*(2000), *qs1*(2000), *qs2*(2000), *bin*(2000), and *avl*(2000). 94
6.3 Critical path equations for *gj* (30). 97
6.4 Critical path equations for *qs1* (2000). 98
6.5 Critical path equations for *avl* (2000). 99

7.1 IPC vs problem size for *mat*. 104
7.2 IPC vs problem size for *gj*. 104
7.3 IPC vs problem size for *heap*. 105
7.4 IPC vs problem size for *qs1*. 105
7.5 IPC vs problem size for *qs2*. 106
7.6 IPC vs problem size for *bin*. 106
7.7 IPC vs problem size for *avl*. 107
7.8 IPC vs problem size for *fib*. 107
7.9 Instruction count vs problem size for *qs1*. 110
7.10 Critical path length vs problem size for *qs1*. 110
7.11 Instruction count vs problem size for *bin*. 111
7.12 Critical path length vs problem size for *bin*. 111
7.13 Instruction count vs problem size for *avl*. 112
7.14 Critical path length vs problem size for the three sorting routines. 113
7.15 Instruction counts for *mat* with different loop constructs. 114
7.16 Critical path length for *mat* with different loop constructs. 115
7.17 IPC for *mat* with different loop constructs. 115
7.18 IPC for *bin* with a single and multiple free lists. 116

7.19 IPC vs problem size with memory ordering constraints in place for *mat*. . 119
7.20 IPC vs problem size with memory ordering constraints in place for *heap*. . 120
7.21 IPC vs problem size with memory ordering constraints in place for *qs1*. . . 120
7.22 IPC vs problem size with memory ordering constraints in place for *fib*. . . 121
7.23 Memory constraint ordering mesh for *mat* (50). 124
7.24 Memory constraint ordering mesh for *heap* (2000). 124
7.25 Memory constraint ordering mesh for *qs1* (2000). 125
7.26 Memory constraint ordering mesh for *fib* (30). 125
7.27 IPC vs number of frames for each problem. 128
7.28 IPC vs frame limit for *heap*. 129
7.29 IPC vs problem size with limited frames for *mat*. 131
7.30 IPC vs problem size with limited frames for *gj*. 131
7.31 IPC vs problem size with limited frames for *heap*. 132
7.32 IPC vs problem size with limited frames for *bin*. 132
7.33 IPC vs problem size with limited frames for *fib*. 133
7.34 Example of nested loops. 134
7.35 IPC vs number of loop iterations for nested loops example. 136

8.1 Frame usage over time for *mat* (20). 143
8.2 Frame usage over time for *trans* (20). 144
8.3 Frame usage over time for *gj* (15). 145
8.4 Frame usage over time for *heap* (200). 146
8.5 Frame usage over time for *qs1* (500). 146
8.6 Frame usage over time for *qs2* (200). 147
8.7 Frame usage over time for naive *bin* (500). 148
8.8 Frame usage over time for *avl* (200). 148
8.9 Frame usage over time for *fib* (15). 149
8.10 Frame usage over time for *fibf* (15). 150
8.11 Frame usage over time for *mat* (20) when limited to 1024 frames. 151
8.12 Frame usage over time for *heap* (200) when limited to 1024 frames. 152
8.13 Frame usage over time for *qs1* (500) when limited to 1024 frames. 153
8.14 Frame usage over time for *qs2* (200) when limited to 1024 frames. 153
8.15 Frame usage over time for *bin* (500) when limited to 1024 frames. 154
8.16 Frame usage over time for *fib* (15) when limited to 1024 frames. 155

8.17 Frame resource usage over time for *fib* (17) with various frame limits. . . . 155
8.18 Frame resource usage over time for Fibonacci at various problem sizes when limited to 1024 frames. 156
8.19 Code for matrix multiplication. 157
8.20 The effect of nested loop re-orderings on frame usage for *mat* (20). 158
8.21 Time-space cache entries over time for *mat* (20). 160
8.22 Time-space cache entries per frame for *mat* (20). 161
8.23 Time-space cache entries per frame for *mat* (20). 162
8.24 Time-space cache entries per frame for *qs2* (200). 163
8.25 Time-space cache entries per frame for *avl* (200). 163
8.26 Time-space cache entries per frame for *fib* (15). 164

A.1 Conditional C code. 178
A.2 WarpEngine assembly for the C code in Figure A.1. 178

C.1 Critical path equations for *mat* (50). 189
C.2 Critical path equations for *trans* (50). 190
C.3 Critical path equations for *heap* (2000). 190
C.4 Critical path equations for *qs2* (2000). 191
C.5 Critical path equations for *bin* (2000). 191
C.6 Critical path equations for *fib* (30). 192
C.7 IPC vs problem size for *trans*. 193
C.8 IPC vs problem size for *fibf*. 193
C.9 IPC vs problem size with memory ordering constraints in place for *trans*. . 194
C.10 IPC vs problem size with memory ordering constraints in place for *gj*. . . 194
C.11 IPC vs problem size with memory ordering constraints in place for *qs2*. . . 195
C.12 IPC vs problem size with memory ordering constraints in place for *bin*. . . 195
C.13 IPC vs problem size with memory ordering constraints in place for *avl*. . . 196
C.14 Memory constraint ordering mesh for *trans* (50). 197
C.15 Memory constraint ordering mesh for *gj* (30). 197
C.16 Memory constraint ordering mesh for *qs2* (2000). 198
C.17 Memory constraint ordering mesh for *bin* (2000). 198
C.18 Memory constraint ordering mesh for *avl* (2000). 199
C.19 IPC vs problem size with limited frames for *trans*. 200
C.20 IPC vs problem size with limited frames for *qs1*. 200

C.21 IPC vs problem size with limited frames for *qs2*. 201
C.22 IPC vs problem size with limited frames for *avl*. 201
C.23 Frame usage over time for *gj* (5). 202
C.24 Frame usage over time for *gj* (10). 202
C.25 Frame usage over time for *gj* (20). 203
C.26 Frame usage over time for *gj* (25). 203
C.27 Frame usage over time for *trans* (20) when limited to 1024 frames. 204
C.28 Frame usage over time for *gj* (15) when limited to 1024 frames. 204
C.29 Frame usage over time for *avl* (200) when limited to 1024 frames. 205
C.30 Frame usage over time for *fibf* (15) when limited to 1024 frames. 205
C.31 Time-space cache entries per frame for *trans* (20). 206
C.32 Time-space cache entries per frame for *gj* (15). 206
C.33 Time-space cache entries per frame for *heap* (200). 207
C.34 Time-space cache entries per frame for *qs1* (500). 207
C.35 Time-space cache entries per frame for *bin* (500). 208

List of Tables

4.1 Procedure descriptions for the algorithm of Figure 4.12. 66

4.2 Simulator run times for 50×50 matrix multiplication. 73

5.1 Algorithm abbreviations and complexity. 76

6.1 Description of keys in instruction mix graphs 93

6.2 Ratio of the highest to lowest critical path equation gradients. 100

7.1 Processing latencies assigned to instructions to give a typical distribution. 103

7.2 Abstract machine models used in memory order constraint tests. 118

7.3 IPC for each problem at a given data set sizes for each of the memory order constraint machines. 122

7.4 IPC for each problem at a given data set sizes for each of the memory order constraint machines using early address knowledge. 122

7.5 IPC for each program with frame restrictions. 129

8.1 Average number of slots used per frame. 140

8.2 Average frame utilisation for *mat* (20) over a range of frame limits. 157

8.3 Number of frames, reads, and writes used in the test programs. 161

Chapter 1
Introduction

The last 30 years has seen the emergence and evolution of the microprocessor. Performance improvements in each generation of processor have been achieved by increasing processor clock frequency and by better utilisation of chip resources. Clock frequencies and transistor densities have both provided order of magnitude improvements over each of the last three decades. However, the same cannot be said of architectural developments, which have provided only limited improvement in the last decade [Shen and Nagle, 1998].

In the 1970's and 1980's architects pushed the technology envelope generating many ideas on how to utilise processing resources, such as pipelining, out-of-order execution and branch prediction. Processor optimisations were limited by space and the main problem facing architects was deciding which of these ideas to implement to best fill available resources. Currently, chips resources are plentiful, with the main barriers being those of power consumption and architectural verification. The limit of what can be achieved with these 30 year old ideas is being reached, and they have been refined to the stage where minimal improvements are being made.

Today's state-of-the-art processors typically consist of two or more processing cores and a large portion of the chip area is dedicated to data caches. The challenge is to produce new architectural ideas that make better use of the number of transistors available in current chips [Pollock, 1999].

Architectural improvements have generally come by increasing the number of instructions that are processed per clock cycle. These improvements are gained by using more efficient algorithms for implementing operations in hardware and overlapping the execution of instructions. One way to reduce the number of processor cycles required to

perform an operation is to simplify the instruction. An architecture embracing this concept across all its instructions is known as a Reduced Instruction Set Computer (RISC) [Patterson, 1980].

Overlapping the execution of instructions also helps to increase the number of instructions processed per clock cycle. Pipelining is a technique used to break an instruction's execution into stages—in much the same way as production lines are used in heavy industry. Successive instructions can be overlapped in different stages of the pipeline. This gives an effective throughput of one instruction per clock cycle even though the average instruction takes several cycles to execute. With a single pipeline machine performance is limited to one instruction per cycle.

To increase throughput beyond one instruction per cycle multiple pipelines can operate in parallel. Analysis is performed to determine the relationships that hold between instructions in terms of data that is manipulated and processing resources available. Instructions that are independent can be scheduled to execute in separate pipelines concurrently. In this way performance is gained by exploiting the parallelism available in the instruction stream. This *fine grained* parallelism between individual instructions within a single instruction stream is known as Instruction Level Parallelism (ILP).

Processing instructions in parallel helps to reduce the effects of long latency instructions on performance. During the processing of a memory read which has to access the relatively slow memory system, for example, other short latency instructions can be processed in parallel. This makes better use of processing resources, improving performance.

1.1 Obtaining parallelism

While most programs contain operations that can be performed in parallel, it is necessary to detect the parallelism in order to extract it. Defining code parallelism could be left to the programmer but there is much legacy code that would be costly and time consuming to rewrite. It is necessary for compilers and hardware to extract the parallelism available.

Programs contain dependencies between instructions that must be maintained to ensure correct program execution. *Dependence analysis* determines the order in which instructions should be processed. Knowing an instruction's input dependencies allows it to

be scheduled to execute *out-of-order* with respect to other instructions in the instruction stream. Out-of-order execution allows instructions that would normally have to wait if processed in programmed order, to use processing resources that would otherwise be vacant.

Dependence analysis performed at compile-time can detect independent instructions in code that contains regular control flow and data structures. These instructions can then be scheduled to execute in parallel. This *static scheduling* of instructions exploits the regularity of data structures and the code that operates on them to obtain parallelism.

In most imperative programs there are complex dependencies between instructions that are caused by the data being manipulated. At compile-time many *data dependencies*, such as those caused by pointer dereferencing, are not known so instruction scheduling must be conservative to ensure they are maintained. Thus the number of instructions that can be scheduled to execute in parallel is limited. To overcome this problem *dynamic scheduling* techniques used at run-time when more is known about data dependencies due to address information being available. This information leads to better scheduling of instructions and increases parallelism. However, dynamic scheduling requires dependence analysis to be performed in hardware.

The scheduling of instructions for processing is at the heart of ILP. Static and dynamic scheduling can be used to complement each other as static scheduling obtains parallelism at a coarse granularity while dynamic scheduling obtains the fine grained parallelism between instructions.

1.2 Challenges

If maximum performance is to be obtained a pool of instructions that are ready for execution needs to be maintained so that available processing resources can be saturated. This pool of instructions can be thought of as a moving window of instructions from the instruction stream that are ready to be processed. Pending instructions are stored in hardware in an Instruction Reorder Buffer (IRB) and from there are scheduled for processing. The challenge is to enlarge the IRB to increase the probability of filling available processing resources. Further increases in performance can be made by employing more

processing resources but this in turn requires a larger IRB.

Control flow prediction is required to fill the IRB. Instructions are processed in an order defined by branching decisions that are based on values set by data and control variables. This means that instructions after a branch cannot be loaded until the branch condition has been evaluated. *Branch prediction* mechanisms have been developed to guess the outcome of branch decisions [Smith, 1981]. Instructions beyond a predicted branch can be speculatively pooled for execution. That is, they are pooled for execution on the condition that the branch outcome has been predicted correctly. Information about these speculative instructions must be maintained so that they can be removed should the branch prediction prove incorrect. This *state* needs to be retained until the predicted branch outcome is known.

1.3 Problems

There are three issues that must be resolved when filling an IRB. Firstly, branch prediction mechanisms are not accurate enough to provide an adequate number of instructions to fill larger IRBs. The control path through many branches needs to be predicted and with imperfect prediction after only a few branches the probability of a correct path is too low to be useful.

Secondly, the dependence analysis performed in hardware does not scale well. Hardware complexity is based on the product of the IRB size and number of instructions issued per cycle. Each instruction's operands are compared to the outputs of other instructions in the IRB. This level of complexity is acceptable for the small IRBs used in today's processors (up to 80 instructions [Gwennap, 1996]) but it can be restrictive when considering larger IRBs.

Thirdly, in contemporary architectures the memory system is usually considered to be a single entity and all operations on it are ordered through a single access point. This ensures correct data is used in computation but it forms false dependencies between independent operations on the memory system. These false dependencies must be removed to allow independent instructions to be scheduled in parallel.

1.4 A solution

One way to improve performance is to *speculate* on data values and the outcome of control decisions. That is, perform operations irrespective of knowing whether the operands allow instructions to be processed early. Rather than performing dependence analysis this approach assumes there are no dependencies. Any false dependencies are eliminated but true dependencies still have to be resolved. This is achieved by re-executing the instructions that violate a dependency when it is detected.

To eliminate the accuracy problems associated with branch prediction mechanisms *control independence* information can be used. In programs there are instructions that will be processed no matter what sequence of branches precede them. That is, the processing of these instructions is independent of the flow of control between them. At compile-time code can be broken into blocks between these instructions and at run-time the blocks can be scheduled to execute concurrently. This distributes the control mechanism allowing a larger IRB to be used.

Mechanisms must be in place to detect and correct incorrect data speculation, and to provide a framework for a distributed control mechanism. Assigning program ordered timestamps to, and recording, every data manipulation allows synchronisation to take place. The timestamps are used to detect when accesses to a given datum have occurred out-of-order, resulting in a data dependence violation. Correct execution is restored by re-executing those accesses that occurred out-of-order.

To manage the distributed control mechanism a tree-based framework is adopted. Each control independent block is assigned to a node in the control tree and is given a timestamp. These timestamps are used to detect and correct incorrect control speculation. When incorrect speculation is detected the code blocks in the sub-tree below that point are removed and a new control sub-tree is generated.

The speculation performed on both control decisions and data values must be verified so that correct computation is ensured. To facilitate this, all speculative actions are recorded. This gives a series of 'snapshots' of execution state that can be returned to and re-executed from.

An architecture has been developed to make use of control decision and data value spec-

ulation to extract parallelism from sequential code. This architecture, the WarpEngine, is based on the Time Warp mechanism from the parallel simulation domain. It incorporates tree-based control and timestamped state saving mechanisms in hardware.

A simulation model has been developed to examine the performance characteristics of this architecture. This simulation model processes instructions in their programmed order recording the real time that events would occur in the WarpEngine. Investigations are performed assuming unbounded resources and observing the effect of constraining individual processor components.

1.5 Thesis claims

Simulation of the WarpEngine architecture shows the level of performance that can be obtained, how instruction buffer resources are used during execution, and how constraining these and other resources affects performance. Results from these areas of investigation support the thesis claims that:

- Speculating on both the outcomes of branch decisions and data values in memory can increase performance an order of magnitude over processing instructions sequentially.
- To achieve a high level of performance it is essential that reads and writes to the memory system are allowed to occur out of programmed order.
- In-order retirement of state saving resources can have a great impact on the performance.
- The WarpEngine architecture is capable of extracting large amounts of parallelism. Incorporating the Time Warp algorithm into hardware provides a mechanism to synchronise control flow and data value speculation.

To analyse the WarpEngine a novel virtual ordered simulation model has been developed. This function level simulation model processes instructions in their programmed order and can efficiently model unbounded resources. This form of simulation provides insight into aspects of speculative systems not seen using conventional time ordered models.

A limitation of virtual ordered simulation occurs when constraints are placed on resources. The true interactions between events are not modelled precisely. Complex hardware timing information is abstracted away and replaced with simple propagation delays used to model the time taken to process higher level events. This provides a fast simulation model but it means that the results obtained are only an indication of the performance that can be obtained.

1.6 Thesis layout

Chapter 2 discusses issues related to ILP and reviews the software and hardware techniques that have been developed to extract ILP. Paradigms using various combinations of these techniques are discussed and their limitations, in terms of scalability and performance, evaluated.

Chapter 3 gives a description of the Time Warp mechanism, its history and how it can be applied to computer architecture. The WarpEngine paradigm, based on this mechanism, is described and compared to other ILP paradigms. An implementation of the WarpEngine is described so that components and the performance potential can be examined.

Chapter 4 introduces a novel simulation process that is tied to the virtual order concept present in the Time Warp mechanism. A simulation model for the WarpEngine has been implemented and is described. Chapter 5 presents a suite of test programs that are used throughout the experiments performed in the following chapters. The compilation techniques used when coding these programs are also discussed.

The following three chapters present results that show performance characteristics of the WarpEngine. Chapter 6 looks at instruction usage, comparing and contrasting instruction mixes of the program as a whole to that of the program's critical path. Chapter 7 examines the potential performance obtainable by the WarpEngine. The effects on performance of false data hazards and limited state saving resources are examined. This analysis results in a set of component requirements for a speculative architecture that extracts large amounts of parallelism. Chapter 8 investigates the use of state saving resources during execution providing insight into how they can be best used to maximise

performance.

The final chapter gives a summary of notable aspects found from the results obtained and the conclusions gained from this work.

Chapter 2
Current solutions

The need to improve microprocessor performance has spurred the development of many techniques to extract increased parallelism from programs. Reducing limitations imposed by false dependencies between instructions allows many instructions to be processed concurrently, thereby increasing parallelism.

The potential overlap among instructions is called Instruction Level Parallelism (ILP) because individual instructions can be evaluated in parallel. This chapter looks at the principles behind ILP, the software and hardware mechanisms that have been developed to extract ILP, and processor architectures based on these mechanisms.

2.1 ILP processing

Several factors motivate the design of computers with increased ILP. First, the majority of the today's computer code is written in imperative languages, and it seems inconceivable that it could all be rewritten to explicitly take advantage of parallelism. For example, in C there are no directives that allow explicit marking of parallel code. Second, it is now difficult to take advantage of and keep busy the number of transistors available in modern microprocessors without making some use of parallelism. Third, at some point in the next decade or two Moore's law will fail [Ross, 1996], and the steady exponential increase in performance of silicon based computers will cease. Extracting parallelism, and in particular ILP will then become critical to ensure to continued performance increases.

2.1.1 Fundamentals

The basic philosophy of ILP is to evaluate many instructions in parallel. In typical programs there are approximately seven instructions between branches [Hennessy and Patterson, 1996a] and these instructions form a *basic block*. The instructions in a basic block are usually dependent on one another, meaning that the amount of parallelism available is small. To obtain a substantial improvement in performance many blocks must be processed in parallel.

Blocks can be made to run in parallel by unrolling the iterations of loops and through the use of prediction across conditional branch boundaries. These techniques generate a larger pool of instructions that can then be scheduled to execute in parallel. The instructions in this pool are in an *in-flight* state. That is, they have been loaded and are waiting for their inputs to be satisfied. While instructions are in-flight some storage space is required to hold them. This storage space takes the form of a *re-order buffer*. Within an Instruction Reorder Buffer (IRB) *dependence analysis* is performed to determine which instructions have to wait for results to be returned from other instructions. When an instruction's inputs are satisfied it can be scheduled for execution, possibly *out-of-order* with respect to other in-flight instructions. This dynamic scheduling of instructions allows more ILP to be extracted.

2.1.2 Constraints

When executing a program to obtain parallelism some constraints have to be adhered to to ensure correct program execution. One constraint, *sequential consistency*, has to be maintained so that correct program execution is ensured. For a multiprocessor computer Lamport [1979] states that the system is sequentially consistent if:

> the result of any execution is the same as if the operations of all processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.

This statement can also be applied to a single instruction stream computer that processes instructions out-of-order. Instructions can be processed out-of-order provided the outcome of the program is consistent with processing instructions in-order.

An area where operations are typically processed in order is the memory system. One in five instructions is a memory access [Hennessy and Patterson, 1996a], implying that restricting memory accesses to one per cycle can only give a maximum of five times speedup through parallelism [Jouppi and Wall, 1989; Smith et al., 1989]. This means that a memory system capable of more than one access per cycle is required to extract performance beyond five Instructions Per Cycle (IPC).

A *weak ordering* programmer's model offers greater performance potential. Adve and Hill [1989] state

> A system is weakly ordered with respect to a synchronization model if and only if it appears sequentially consistent to all executions of a program that obey the synchronization model.

Other than specifying that the system executes the program is a sequentially consistent manner, this definition does not specify any explicit directives for hardware. That is, the memory system can perform operations out-of-order but the program's order has to be maintained. Using this definition for ordering allows greater freedom when designing hardware to extract ILP.

Data hazards

The constraints on memory and register access ordering are often described in terms of the *hazards* that exist. For example, if a write is logically followed by a read to the same location then they must be executed in that order. This is called a Read-After-Write (RAW) hazard. In contrast two successive reads to a location cause a Read-After-Read (RAR) hazard which can be ignored because the value returned is unchanged between reads. With a Write-After-Write (WAW) hazard the value stored has to be that of the later write to ensure later reads load the correct value. A Write-After-Read (WAR) hazard also has to be processed in order to ensure the read loads a value before the write stores a new value at that location.

When executing instructions out-of-order there are dependencies that have be maintained to ensure correct computation.

- An instruction is *data dependent* on a prior instruction if it requires a result produced by that instruction, or it requires a result from another instruction that is dependent on that instruction.

- A *control dependence* determines the ordering of instructions with respect to a branch instruction so that they are executed only if they should be. This means that an instruction that is dependent on a branch cannot be evaluated until that branch has been taken.

- A *name dependence* arises when two instructions use the same register or memory location but there is no flow of data between the two. This occurs when an instruction reads from, or writes to, a location and a later instruction writes to the same location. This is called an *anti-dependence* and this type of false dependency can be removed by renaming the location between the two instructions.

False dependencies between memory and register operations can be exploited to extract more parallelism. A RAW hazard has to be maintained, but a RAR hazard can be ignored without adverse effects on the outcome of the program. With a WAW hazard the value of the later write has to be maintained, but the operations could happen in parallel. A WAR hazard forms a name dependency that can be broken through the use of *register renaming* and memory location renaming.

Retirement

With dynamic scheduling, instructions complete their processing out-of-order. This can have adverse effects when it comes to exception handling as instructions may complete execution before an earlier instruction raises an exception. This makes the exception imprecise and hardware has to be capable of handling this situation. Rather than processing the exception when it occurs a record of it is stored with the instruction in the IRB and instructions are retired from the system in their programmed order. If the exception reaches the head of the IRB then it is no longer out-of-order so it can be processed. Solutions to precise interrupts in pipelined processors are discussed by Smith and Pleszkun [1985] and in superscalar processors by Dwyer and Torng [1992].

These issues place constraints on the order in which instructions can be executed. This

in turn limits the amount of parallelism that can be obtained. Computer architects have developed techniques to alleviate some of these problems. These are discussed in the following section.

2.2 Extracting ILP

Early architectures, such as the IBM Stretch [Bashe et al., 1986], used a pipelined function unit to overlap the execution of instructions. This led to designs with multiple pipelined function units, known as *superscalar* processors, that are popular in today's architectures. These superscalar processors are capable of scheduling more than one instruction every clock cycle. Their performance is dependent on scheduling instructions to maximise the use of the functional units in the processor. Consequently, significant effort has been invested in techniques to maximise the number of instructions scheduled for execution in each clock cycle.

Software and hardware mechanisms have been developed to minimise and possibly remove some of the constraints imposed by sequential consistency. These mechanisms generate a pool of instructions that can be scheduled to run in parallel.

2.2.1 Software support

When compiling a program optimisations can be performed to increase the potential for extracting ILP. This usually involves the rearrangement, addition and removal of instructions, the main aim of which is to remove stalls that would otherwise occur in the function unit's pipeline. These stalls originate from control (branches) and data (RAW) dependencies that must be evaluated before another instruction can enter the pipeline.

Software techniques performed at compile time include detecting and eliminating dependencies, loop unrolling, software pipelining, static branch prediction, trace scheduling, and predicated instructions. Detailed descriptions of these and other compiler based techniques used to optimise code and extract parallelism is given by Aho et al. [1986], and Smith et al. [1992].

Compile-time predictions on control and data flow are static. Code can only be rear-

ranged at this time if it can be determined that no data dependencies will be broken. Instructions can then be processed in their newly generated order with correct computation guaranteed. Compiler based scheduling works well for portions of the program where data and control is well structured, such as stepping through arrays. In this type of situation inter-instruction dependencies are determined easily. Where structures are less regular, such as the case when following pointers, dynamic scheduling must be used. This type of scheduling must be performed at runtime and requires hardware support.

2.2.2 Hardware support

Early ILP processors relied on compiler technology to generate a sequence of instructions that exhibited good run-time behaviour on the particular architecture. However the number of instructions that can be concurrently scheduled is determined by data and control dependencies. When scheduling in software it is only possible to use static information. As the resources available in a system have increased new hardware based scheduling techniques, which make use of run-time information, have been developed to allow more instructions to be scheduled every clock cycle.

Dynamic scheduling in hardware allows instructions to be executed out-of-order. This means that the IRB can be expanded, increasing the probability that functional units can be fully utilised. Additional hardware is required to perform data and control dependence analysis when the program is running. This is achieved by marking instructions when any dependencies will not be broken, to indicate that they can be executed. The scheduler then selects groups of these marked instructions to process in parallel.

Instruction re-order buffers

Instructions need to be buffered during the re-ordering phase of their execution to enable dependence analysis to be performed and to hold those instructions that are waiting for their dependencies to be met. Techniques such as *scoreboarding* and *register renaming* can be used to maintain dependencies when instructions are re-ordered.

The goal of a scoreboard is to maintain an execution rate of one instruction per cycle in a pipeline by executing an instruction as early is possible. When an instruction is

stalled, due to a dependency, another instruction behind it in programmed order that is ready to execute replaces it in the pipeline. This requires several instructions to be in an executable state simultaneously. An instruction is put into the executable state when all its input registers are valid. The scoreboard keeps track of which registers contain valid values. This requires a mechanism capable of detecting data hazards.

To remove false dependencies created by WAR and WAW hazards registers can be renamed between the write and the preceding read or write. The new register names are chosen from a larger set of virtual registers. Tomasulo [Tomasulo, 1967] devised an algorithm and hardware components to exploit the use of multiple function units. Through the use of tags, registers are renamed dynamically and these new virtual register's values are held in buffers known as *reservation stations* until dependent instructions are ready to use them.

Elaborations of Tomasulo's reservation stations are used in many contemporary production microprocessors including: the Hewlett-Packard PA-RISC processor family [Scott et al., 1997], IBM/Motorola PowerPC [Motorola, 1997], INTEL Pentium Pro [INTEL, 1995], MIPS R10000 [MIPS Technologies, Inc, 1994] [Yeager, 1996], and Sun UltraSPARC [Sun Microsystems, 1999].

The HP PA-8000 achieves out-of-order memory accesses on a small scale using a 28 entry Address Re-order Buffer [Gwennap, 1996]. This buffer allows other instructions not dependent on waiting memory accesses to continue processing. This does not help those instructions that are blocked waiting for data to be returned from memory.

Branch prediction

Speculation on the outcome of conditional branches can be used to increase the number of instructions available for re-ordering. Branch prediction involves making an educated guess as to the direction a branch will go. The prediction is typically based on previous branch decisions which are recorded in counters or tables depending on the scheme. Accurately predicting a branch direction allows instructions beyond the branch to be pre-loaded into the IRB.

Several basic branch prediction mechanisms are described by Smith [1981]. They vary

in the way in which they predict a branch and whether or not they hold branch target information. Also, they vary in the stage of the instruction pipeline where they take effect. Different forms of dynamic branch prediction are summarised in Lee and Smith [1984], and Perleberg and Smith [1993].

These mechanisms have been shown to achieve up to 93% prediction accuracy [Perleberg and Smith, 1993]. However, after only a few unconfirmed predicted branches the probability that the correct path is being followed is too low to be useful. For example, after 5 branches the prediction accuracy has reduced to 69.5% (0.93^5). Branch prediction of this form does not scale well, making it hard to fill larger IRBs. A more aggressive approach is to follow both paths through a branch. Speculating through M branches could mean that up to 2^M control paths would be predicted with $2^M - 1$ of them being incorrect [Theobald et al., 1993]. A corresponding amount of buffer space is required to store the instructions down all these paths, the majority of which are not executed. This type of branch prediction does not scale because of the inefficient use of buffer resources.

Dutta and Franklin [1995] describe a mechanism for predicting the outcome of multiple branches with a single prediction. They consider the possible paths through several branches, in their case 2, and produce a tree-like control subgraph. This means that 4 branch outcomes are possible. A *subgraph history table* and *pattern history table* are hardware mechanisms used to store previous branch outcomes and predict later ones. They claim a 50% improvement in fetch rate on a 12-way superscalar architecture.

With any form of branch prediction the system must have the ability to undo the consequences of traversing incorrect branches. In most cases this is simply a matter of *squashing* the affected instructions and removing them from the re-order buffer. Special consideration is needed for operations that store data to ensure that incorrect values are not written over correct ones. This can be achieved by performing these operations in-order.

Value speculation

Another form of speculation is to predict the value of data in the relatively slow memory system. This helps to reduce memory access latency and, indirectly, improve some branch predictions. Returning data values early allows dependent instructions to start executing earlier, increasing performance potential. Allowing instructions to execute

early can lead to improvements when predicting branches. That is, the predicted data value may in some cases be incorrect, but it could produce the correct outcome for a branch decision. Data value speculation can be applied in several ways:

Address prediction. The locations of some data can be predicted, such as the addresses of array elements, and their values can then be loaded early. This is known as *data prefetching* [Smith, 1982].

Value prediction. Hardware mechanisms predict the values of data items by observing patterns in the data. An example of where this method works well is index counter variables. Prediction accuracies of up to 80% have been observed [Lipasti and Shen, 1996] which could give performance gains of 4.5% to 23%. Research into this form of speculation is still in its infancy and as such value prediction has not been included in any delivered architecture.

Memory system optimism. In memory speculation values are returned early from reads by making use of local or current information. For example, a value might be present in a local cache but it is unclear that it is coherent or safe. That is, there might be a remote instruction which has already written to the location but the value has not yet arrived, or there may be an earlier instruction which has not yet completed execution that will write to the location. This form of value prediction is present in the WarpEngine architecture.

Again, the prediction accuracy can have a significant impact on the performance obtained. With any form of data value speculation the system will continue executing with the possibly erroneous value until a new, possibly correct, value arrives. The system must be able to undo the consequences of evaluating incorrect data and then proceed to re-execute with new data values.

2.3 ILP paradigms

With a large enough pool of pending instructions the parallelism obtained is limited to the number of instructions that can be processed concurrently. To exceed the 1 IPC barrier of a single pipeline architecture many instructions need to be issued in parallel. This

requires multiple function units, associated instruction pipelines, and hardware and software support for scheduling.

The software and hardware mechanisms for extracting ILP can be applied to computer architectures in varying combinations. There are several different paradigms that have been developed using these mechanisms to extract ILP and parallelism in general.

2.3.1 Vector

The vector paradigm was one of the earliest to exploit ILP. The main thrust of the vector paradigm is to perform the same operation across multiple instances of data. Instructions are designed to operate on arrays rather than single data elements.

Code is rearranged at compile time so that each operation can be applied to a groups of data instances. This is achieved by overlapping loop iterations, determining loop independent instructions and then forming vector instructions from them. For example, consider the code

```
for (i = 0; i < 10; i++)
    A[i] = B[i] + 1;
```

There are no data dependencies between the loop iterations so the instructions can be expressed by the single vector statement

```
A[0:9] = B[0:9] + 1
```

This produces a sequence of vector instructions load, add, and store, which operate on 10 instances of data in parallel.

A vector machine gains performance over a scalar processor in that each instruction specifies a great deal of work. Processing one vector instruction is equivalent to executing an entire loop. Since the data is regular it can be arranged in an interleaved memory system so that access times can be reduced. An entire vector of data is loaded with a single instruction. The associated single memory access instruction's overheads are amortised across the entire vector producing a smaller average latency. Also, the control hazards from loop branches are non-existent because loops are converted into vector instructions.

In conclusion, vector operations can be made faster than the equivalent sequence of scalar operations.

A description of a basic vector architecture is given by Hennessy and Patterson [1996a]. An example of a successful vector machine is the early Cray series [Russell, 1978]. These machines work well on scientific code which is has well structured control flow and data structures. The problem is that most code does not display these regular characteristics. This is one reason why the vector paradigm has not flourished as a general purpose ILP paradigm.

2.3.2 VLIW

The basis of the Very Long Instruction Word (VLIW) paradigm is to issue a fixed number of operations formatted as one larger instruction. A VLIW machine has a fixed arrangement of multiple independent function units, each of which is accessed by a corresponding operation in the long instruction. These function units fix the number and type of operations that can be put into each long instruction. Instructions are processed as in other paradigms, using pipelines, with individual operations executed in lock-step. That is, all operations in an instruction are processed at the same rate.

Special purpose function units can be built to operate faster than general purpose ones. They don't have to perform any instruction decoding and can be optimised to a particular operation, or sub-set of operations. The operation decoding is performed by the compiler which places them in the appropriate slot in the instruction word. This combination of features allows a faster clock to be used in the system because each function unit is doing less work than one general purpose function unit would have.

Operations are scheduled statically at compile-time generating long instructions. Techniques such as loop unrolling and scheduling across multiple basic blocks are used to fill instructions with independent operations. Other techniques such as trace scheduling and code compaction were developed to optimise instruction usage in VLIW machines [Fisher, 1981]. These ideas have been incorporated into the compilers used with contemporary ILP microprocessors [Biglari-Abhari et al., 1998].

The late 1970s and early 1980s saw the development of VLIW machines. The AP-120B and

FPS-164 families of processors from Floating Point Systems [Charlesworth, 1981] were some of the first wide-instruction processors. In these machines each instruction word contained multiple *parcels* each of which controlled the operation of floating point adder, multiplier, memory, and register control units. Another example of a VLIW machine is the Cydrome Cydra 5 [Rau et al., 1989].

As with the vector paradigm, VLIW machines have not prospered. VLIW machines suffer from increased code size because a significant amount of loop unrolling is performed and some operations are not used in the instructions wasting space. Also, there is a lack of binary compatibility between generations of a VLIW processor family. If the function unit configuration is altered then old program executables will no longer work.

Another problem inherent in VLIW machines is that a stall, such as a cache miss, in one function unit causes a stall in the whole processor. This must happen because all instructions are processed in lock-step. Stalls of this type are eliminated in the superscalar paradigm.

2.3.3 Superscalar

The superscalar paradigm, like the VLIW paradigm, uses multiple independent function units to obtain ILP. Associated with each function unit is an instruction pipeline and decode logic. Code may be scheduled statically by compiler, dynamically by the hardware at run-time, or by a combination of both. This paradigm is similar to VLIW in that multiple instructions are issued in parallel, but the use of dynamic scheduling facilitates better utilisation of function units.

When scheduling instructions, techniques similar to those used in vector and VLIW compilers can be applied. Static scheduling has the advantage that it simplifies hardware allowing for a higher clock rate. This is evident with the statically-scheduled DEC Alpha 21164 which has significantly higher clock rates than other dynamically scheduled microprocessors of the same era [Hennessy and Patterson, 1996b]. But in general the dynamically scheduled microprocessors are capable of extracting larger amounts of ILP.

In 1955, design of the IBM Stretch [Bashe et al., 1986] computer system began. It included many aggressive organisation techniques including: instruction pipelining, out-of-order

execution, speculative execution based on branch prediction, branch mis-prediction recovery and precise interrupts. Although not successful in its day it was a precursor to many of today's superscalar processors. Current production superscalar microprocessors include: the DEC Alpha 21264 [Gwennap, 1996], the Hewlett-Packard PA-RISC family [Scott et al., 1997], the IBM/Motorola PowerPC family [Motorola, 1997], the INTEL Pentium Series [INTEL, 1995], the MIPS R10000 [MIPS Technologies, Inc, 1994; Yeager, 1996], and the Sun UltraSPARC [Sun Microsystems, 1999].

The superscalar paradigm places less reliance on good compiler scheduling than vector and VLIW paradigms. Parallelism can be extracted using run-time scheduling with hardware in place to assist. More information about the arrangement of irregular data structures is known at run-time allowing accurate dependence analysis to be performed. Mechanisms based on Tomasulo's reservation stations and re-order buffers are in common use.

One problem with extending this paradigm to extract more parallelism is that dependence analysis logic does not scale well. The complexity of the logic is a product of the IRB size and instruction issue width. Both of these values must be increased to obtain more parallelism. This level of complexity is acceptable for the small IRBs used in modern processors (56 (2×28) instructions in the HP PA-8000 [Hunt, 1997] and 80 instructions in the DEC Alpha 21264 [Gwennap, 1996]) but it can be restrictive when considering larger instruction windows.

2.3.4 Multiscalar

One solution to the problems encountered when increasing the size of the IRB is to run several streams of instructions in parallel. The Multiscalar paradigm [Franklin, 1993; Sohi et al., 1995] evolved from the Expandable Split Window for re-order buffers concept proposed by Franklin and Sohi [1992]. Annotated code is broken into tasks that are executed as though they are independent programs with their own thread of control. A large instruction window is produced by speculatively executing multiple tasks in parallel, each with their own sub-window of pending instructions. The concurrent execution of tasks allows many instructions to be executed in parallel.

The Multiscalar architecture consists of processing units linked via a unidirectional com-

munications ring as shown in Figure 2.1. Within each processing unit superscalar techniques are used to execute instructions gaining some ILP. A global sequencer is used to speculatively assign tasks that are predicted to be contiguous into successive processing units. Tasks execute to completion as though they are independent of one another and are retired in the order they were assigned. The sequencer also maintains head and tail pointers to mark the earliest and latest tasks, respectively. When the earliest task completes, its associated processing unit is freed and the head pointer is moved to the next task. The freed processing unit then has a speculative task allocated to it and the tail pointer is set to that task.

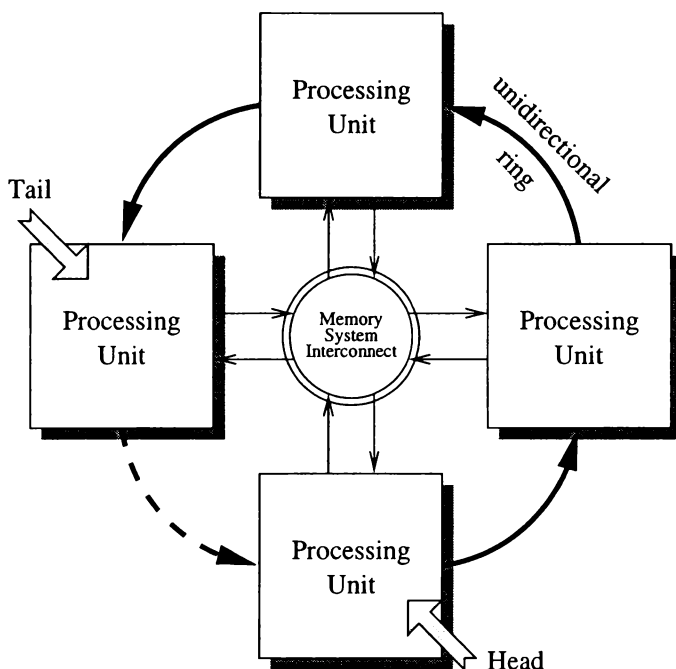


Figure 2.1: Architecture of the Multiscalar processor.

The sequencer ensures control dependencies between tasks are maintained by placing a programmed defined order on them. If an incorrect prediction is made the speculative tasks, those in front of the head task up to and including the tail task, are *squashed* and the correct task sequence is assigned to processing units.

Data dependencies between tasks are maintained through memory system via an Address Resolution Buffer (ARB) [Franklin and Sohi, 1996]. The ARB, an extension to the cache hierarchy, maintains a record of the reads performed by each task and forwards writes to future tasks. If a forwarded write meets a read at the same address then the task associated with the read is squashed and re-executed. In this way read operations are

guaranteed to receive correct values. Writes issued by the trailing task are committed to the memory system while writes issued by other tasks are retained in the ARB until such time that the associated task becomes the head task.

In the Multiscalar architecture performance is improved over superscalar by speculating on control decisions and data values. A loose sequential order between tasks is maintained by the sequencer. This allows instructions to be processed out-of-order across a large instruction window. Speculating on data values by allowing memory accesses to occur out-of-order across tasks eliminates false data dependencies. This allows independent tasks to process concurrently.

The Multiscalar architecture has shown a performance improvement of up to 6.28 times for an 8-Unit machine when compared to a single-unit scalar machine running optimised code [Sohi et al., 1995]. It gains parallelism in the range of 4 to 8 IPC. A microprocessor architecture based on the Multiscalar paradigm, the Trace Processor, has shown similar levels of performance with IPC in the range of 2 to 5 on an 8 processor machine [Rotenberg et al., 1997].

The Stanford Hydra [Oplinger et al., 1997] takes some ideas from the Multiscalar but treats multiple threads as an extension of a multiprocessor that supports fine grained data sharing. Like the Multiscalar the multiprocessing paradigm requires programmer intervention to locate parallelism. Special annotations to instructions are used to indicate where instructions can be safely executed in parallel and where storage locations are read from the last in a given thread.

In the Multiscalar architecture branch prediction is present at the task level. Tasks are invoked speculatively based on the predicted outcome of control decisions. While this produces good parallelism potential across small numbers of tasks, it does not scale well. There is a similar drop off in prediction accuracy across the speculative tasks as occurs with branch prediction. Also, if an early speculative task is squashed then all following tasks are squashed. In some cases tasks may be unnecessarily squashed and re-executed wasting prior computation performed on that task.

As with any speculative architecture a record of the state of speculative events must be retained until they have been verified correct. The ARB records speculative memory accesses. When each task is assigned to a processing unit the value of its program counter

and the state of the register file are recorded. This instruction initialisation record allows squashed tasks to be re-executed from their starting state. This level of state saving may reduce performance because independent flows of instructions within a task may get squashed unnecessarily. This can occur if a speculative read uses incorrect data. In this situation the entire task is squashed and re-executed not just those instructions effected by the read value.

2.3.5 Data flow

The data flow paradigm was derived from observations that programs are limited by data dependence. It is conceptually the purest of all the ILP paradigms. The processing model is data driven rather than control driven as in other paradigms. Data flows from instruction to instruction as described by the program's Control Flow Graph (CFG). Instructions are executed when all their inputs are valid. This in turn produces more data which flows onto other instructions. Performance is only limited by the parallelism inherent in the program provided enough processing resources are available.

The Manchester Prototype Dataflow Computer [Gurd et al., 1985] provides a good model for data flow execution. Data is placed in a packet, called a token, which is tagged so that it can be matched to a destination instruction. Each token is placed in a token queue where it waits to be matched with an instruction in the matching unit. When an instruction obtains all its inputs it becomes an executable node and is placed in a node store where it waits for a free processing resource. The executed instruction may produce more data which is tokenised, tagged and placed in the token queue. This cycle continues until there is no more data flowing in the system.

Data flow machines that have been developed include: the Manchester Prototype Dataflow Computer [Gurd et al., 1985], the EM-4 [Yasugi et al., 1992], the StarT family [Ang et al., 1995], the MIT Monsoon [MIT, 1995], the MIT J-Machine [Lethin, 1994] and M-Machine [Fillo et al., 1995], the Bristol Data Diffusion Machine [Stallard et al., 1993], the MIT Alewife [Agarwal et al., 1991, 1995], LAU, Mandala, and the Earth Multiprocessor [Hum et al., 1995]. Dennis [1980] discusses the principles behind data flow processors and Culler [1993] describes a stateless data flow architecture. A comparative study of these and other data flow architectures is given by Snelling and Egan [1994].

While the program control mechanism is distributed, which removes linear control constraints, there are major weaknesses in the data flow paradigm. These include the high cost of synchronising data and long instruction life cycles. The hardware overheads associated with global centralised matching stores and queues are large. Any data token could potentially be associated with any instruction so hardware must be able to handle this. Reductions in token store sizes can be achieved through dynamic token tagging, but the scale of the matching problem is still large.

The various stores and queues in the system form an inherently long instruction and data pipeline. This produces a bottleneck for sequential execution because tokens must traverse the entire pipeline before successive dependent instructions can execute. Also, there is no ordering placed on tokens within the system which can aggravate this problem because successive instructions may get delayed by other instructions using resources. The long delays also make faults and exceptions difficult to deal with.

The programming environments for data flow machines are radically different from those of conventional control driven architectures with different programming models required to exploit parallelism. The performance bottlenecks and programming environments have meant that the acceptance of data flow architectures as general purpose processors has been low. The weaknesses of data flow machines are discussed in more detail by Snelling [1993].

2.4 Studies of ILP

Many studies have been conducted evaluate the parallelism that can be gained in sequential code [Jouppi and Wall, 1989; Smith et al., 1989; Butler et al., 1991; Wall, 1991; Lam and Wilson, 1992]. In most cases the major emphasis has been on the effects that control flow has on parallelism, particularly the performance issues associated with different forms of branch and control prediction.

Wall [1991] showed how different levels of branch prediction, register renaming and alias analysis affect the amount of parallelism that is available. Lam and Wilson [1992] examined the effects of allowing varying amounts of control dependence resolution on parallelism. Memory disambiguation and register renaming were assumed to be perfect. The

performance obtained through parallelism was shown to be around 2 IPC. The results showed that as the complexity of the control dependence resolution increases so does the amount of parallelism obtained. This indicates that extraction of large amounts of parallelism requires a machine with perfect disambiguation capabilities.

These early studies were conservative in that only control dependence resolution was considered, no data value speculation was included. Work by Postiff et al. [1998] shows that large amounts of ILP, much greater than 2 IPC, is available in SPEC95 benchmark applications. To achieve this level of performance all false dependencies are removed through the use of register and memory renaming. Traditional out-of-order superscalar processors cannot exploit this parallelism because the distance between parallel instructions is too great to discover. They propose that a processor capable of scheduling instructions from multiple streams is required to uncover the parallelism. Suggestions on the requirements of an architecture are discussed but no implementations are given.

The sustainable throughput of instructions in superscalar processors is claimed to be around 4 IPC [Gwennap, 1996]. This level of performance is not obtained because cache misses and long latency instructions reduce the average throughput. Oehring et al. [1999] demonstrate that IPC of 1.60 can be obtained using a single threaded 8-issue superscalar architecture in a multimedia environment. An 8-threaded 8-issue superscalar architecture can obtain 6.07 IPC, an approximate four-fold increase in performance. This is comparable to the performance characteristics of the Multiscalar machine and its derivative the Trace Processor. These levels of performance are far below the amount of parallelism that is available in programs [Postiff et al., 1998].

2.5 Summary

The key to extracting ILP is to have a large group of instructions ready to be executed in parallel. Software techniques, such as loop unrolling and branch prediction, can be applied alone or in combination to help increase the number of ready instructions. Dependence analysis hardware is used to schedule instructions to be executed concurrently.

The problem is that these techniques do not scale well. Current branch prediction mechanisms are not 100% accurate and suffer a significant drop-off in accuracy after only a

few branches. Dependence analysis hardware has to check the inputs of each instruction against the output of all other instructions in the IRB. Also, with a large enough group of instructions scaling performance beyond about 5 IPC is not possible if memory operations are constrained to occur in order.

One solution is to split the IRB into many smaller independent IRBs which process instructions in parallel. To fill these distributed IRBs sequential control flow must be broken and memory accesses must be allowed to occur out-of-order. The Multiscalar machine achieves this with a 2 level control mechanism, multiple processing elements connected in an ordered ring structure, and an address resolution buffer. The program is broken into tasks which are processed in parallel. This architecture is still limited, although on a larger scale, by the accuracy of task predictors and the sequential communications ring.

The data flow paradigm shows that distributing the control flow in programs can improve performance and the program's CFG can be used to generate a better control mechanism. Processing memory operations out-of-order eliminates false dependencies between instructions allowing independent instructions to be processed in parallel. The problem with pure data flow processing is that global token matching is a costly exercise.

The strengths and weaknesses of ILP paradigms have been considered when developing a new ILP paradigm, the WarpEngine, which is described in Chapter 3. The WarpEngine uses: control decision and data value speculation to eliminate false dependencies; a tree-based mechanism derived from control independent points in the program's CFG to distribute control in a scalable manner; and data flow processing of instruction blocks is performed on a small scale with the control mechanism retaining order between blocks.

Chapter 3
The WarpEngine

A new architecture, the WarpEngine, extracts parallelism from sequential code by speculating on the outcomes of control decisions and data values. The Time Warp mechanism is implemented directly in hardware to synchronise speculation and coordinate multiple processors to perform a single task.

Time Warp is an algorithm for distributing and parallelising discrete event simulations originally proposed by Jefferson [1985]. It is one implementation for the *virtual time* paradigm which is used to organise distributed systems by imposing a temporal ordering on events. Parallel events are given a *virtual ordering* that represents the order they would be executed in a sequential environment.

This chapter discusses the principles behind the WarpEngine and how the Time Warp mechanism can be incorporated into hardware to realise this architecture. A comparison to other ILP paradigms is given to highlight features of the WarpEngine.

3.1 Time Warp mechanism

With the Time Warp mechanism it is possible to extract parallelism from simulations that are otherwise intractable [Fujimoto, 1990b]. A parallel discrete event simulation is decomposed into objects that communicate via messages. Each message has an associated *timestamp*, and when an object receives the message an event is executed at that time. Each event may generate further messages with later timestamps that are propagated to other objects. This process can be seen in Figure 3.1 where three incoming messages, at times 5, 6, and 3, produce three out going messages at times 8, 9, and 6, respectively. The ordering of forwarded messages, known as the *principle of causality* [Fujimoto, 1990a], has

to be maintained to ensure correct computation.

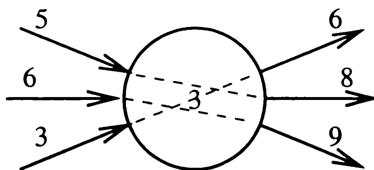


Figure 3.1: Message timestamp updates through a Time Warp object.

The Time Warp algorithm speculatively executes events with their own local clocks. Sometimes this will result in an object processing a message out-of-order. The object is then *rolled back* to the correct point by undoing all the incorrect speculative events, and the out-of-order event is then re-executed. As well as being units of parallel execution, objects carry all the state information in Time Warp. To rollback an object to a particular event two things must be accomplished:

1. restore the state of the object to what it was just prior to the event, and
2. undo the effect of any messages sent by events after the rollback point.

A common technique for restoring state is to take a copy of an object's *state* each time it receives a message, and build a list of snapshots of the object's execution. When a rollback occurs, the appropriate earlier state can be restored.

Time Warp uses *anti-messages* to undo the effect of messages sent by events which have been rolled back. When an anti-message is received it causes the object to be rolled back to a prior state, and the anti-message annihilates the message. This may cause further anti-messages to be sent.

Good performance using Time Warp depends on fine grained state saving and quick processing of anti-messages. That is, each change in state is recorded meaning that only those objects with incorrect data are rolled back and re-executed. The trade-off is that more space is required to store the state of objects.

3.1.1 Global virtual time

One side effect of objects periodically saving their state is that the object builds up a queue of old state copies. Hence, it is necessary to eventually remove these old state copies. This

process, known as *fossil collection*, can only be performed when it is known that a state copy will never be required for a rollback. This is done by periodically estimating a value called *Global Virtual Time (GVT)*. This value is equal to the minimum time of any currently active object or message between objects.

To maintain causality an object can only be rolled back by a message from an object at an earlier time. Therefore no object can possibly be rolled back to a time before GVT, and it is possible to fossil collect all state copies with timestamps prior to GVT.

GVT ensures that the system as a whole makes forward progress as it has the property that it never decreases in value. Local virtual clocks may rollback due to incorrect speculation, but GVT will always progress forward. Process termination is also detected by GVT allowing system termination to be detected when all processes have finished.

Much attention has been paid to algorithms to compute GVT in the Time Warp literature [Bellenot, 1990; Gomes, 1992; Gomes et al., 1992].

3.1.2 Cancelback

Each event in the system consumes storage space for state saving purposes until it is freed by the GVT mechanism. One problem that can occur with speculation is that events further into the future can take up all state saving resources preventing execution of earlier events. A mechanism is required to avoid this and ensure program completion.

The *cancelback* protocol is an extension of the Time Warp mechanism that handles storage management. Cancelback provides a way of halting events that are far in the future and recovering the storage they are using. This freed space can then be used for more immediate purposes. It has been proven that simulations using Time Warp with the cancelback protocol in place can complete execution in the same amount of space that is required for sequential execution [Jefferson, 1990]. This means that cancelback allows execution to complete when state saving space is limited.

3.1.3 Architecture

Pearson et al. [1997] describes the terminology parallels between the use of Time Warp in the discrete event simulation and computer architecture domains. When applying Time Warp to computer architecture the first step is to identify the components that correspond to objects, events and messages. Objects correspond to instructions and memory locations. Instruction state saving may be performed at the single instruction, basic block, or larger task level. This space is equivalent to the IRB in contemporary microprocessors. As with instructions memory location state saving can be performed at various levels, individual words, lines, or arbitrary blocks.

An event equates to the process of executing an instruction or instruction group, and messages correspond to the transfer of data between instructions. Given the object classifications there are a number of possible message types. Branch, call and jump instructions are control messages between instruction groups that communicate conditional and unconditional transfer of control. Write instructions are messages from an instruction group to a memory location and read instructions correspond to two messages: a request from the instructions to the memory location; and a reply from the memory back to the instructions.

A rollback is the squashing of a speculative instruction and re-executing it with correct operands. GVT is analogous to the necessary in-order retirement of instructions from the IRB that may have been executed out-of-order. Cancelback does not have an equivalent in contemporary microprocessors because they have a single linear control mechanism. Cancelback is necessary when a distributed control mechanism is used.

A hardware implementation of Time Warp, the Virtual Time Machine, has been proposed by Fujimoto [1989]. However, it is an implementation for the simulation of large parallel discrete event simulation problems rather than a general purpose processor. To date, no detailed analysis of the Virtual Time Machine architecture has been undertaken.

3.2 Extracting parallelism

Time Warp provides a synchronisation mechanism ensuring that correct computation is performed in a speculative system. Mechanisms are also needed to produce an environment where speculation can take place. That is, generate a large pool of instructions from which instructions can be selected to execute in parallel. The WarpEngine performs control flow speculation through a tree-based control mechanism that is generated using *control independence* information within programs. Speculation on data values is achieved through a timestamped memory system that allows accesses to occur out-of-order.

3.2.1 Control flow speculation

One way to reduce the adverse effects of long branch prediction chains, and bad predictions in general, is to use *control independence* information [Rotenberg et al., 1998]. The control flow constructs for many statements have a useful property that there is a single point at which control enters and a single point at which control exits. The CFGs for `if-then-else` and `while` statements are displayed in Figure 3.2. Ignoring any data dependencies, in both cases it does not matter which outcome of the branch decision is taken control will always end up at code block D. At D the control paths converge independent of the branching decisions that precede them. These control independent points exist for many control constructs including sequential statements, loops, conditional statements and function calls.

Using control independent points eliminates the need to predict long chains of conditional branches. Code at the start of each of these points is guaranteed to execute regardless of the combination of branches precede them. Any branch prediction mechanism need only predict branch decisions between control independent points because the code at these points has a 100% probability of executing.

Knowledge of the positions of control independent points can be used advantageously to execute code blocks in parallel. Rather than allocating resources to single highly speculative stream of speculative instructions they can be better used to overlap many smaller streams of instructions that are guaranteed to execute. Separate threads of control can be optimistically scheduled at control independent points with little fear of them being

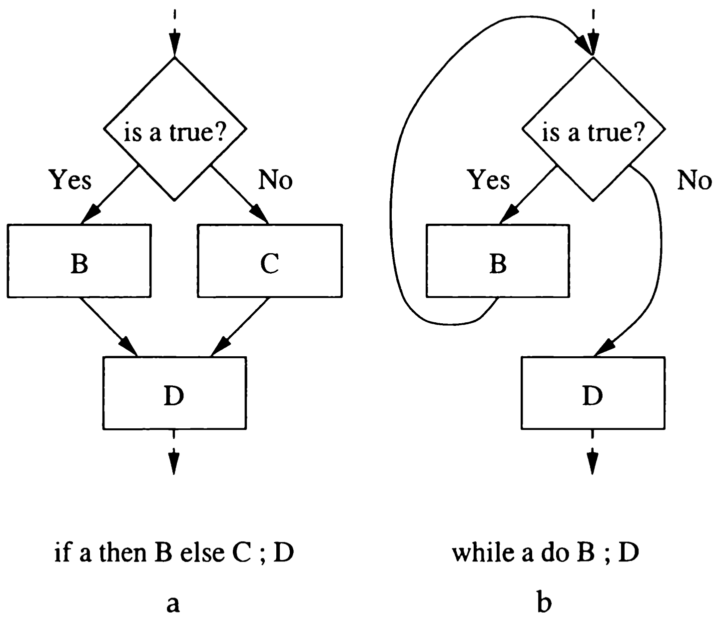


Figure 3.2: CFGs for a) `if-then-else`, and b) `while` statements.

rolled back. A linear path of instructions can be distributed across parallel processing resources by breaking it up at these points.

Tree-based control mechanism

Control independence information is used when performing data flow analysis at compile-time [Aho et al., 1986]. Internally the compiler holds the flow of control of a program in a tree structure. The same tree structure can also be used for a speculative dynamic control mechanism. A program's CFG is mapped onto a dynamic control-tree with each control independent point in the program creating a new node in the tree.

For example, the CFG for the `if-then-else` statement in Figure 3.2 a) can be mapped onto a 2-*way* control tree shown in Figure 3.3. The filled circles represent the control independent points which are linked to form a *backbone* down the right hand edge of the tree. The outgoing links from B and C to D have been broken and replaced with a link from the control-independent point associated with D. The thick dotted line indicates a continuation of the control independent path, whereas the thin dashed arrows represent conditional execution.

The sequential path through the instructions is obtained by performing a left-to-right preorder traversal of the control tree. Parallelism is gained by performing breadth first

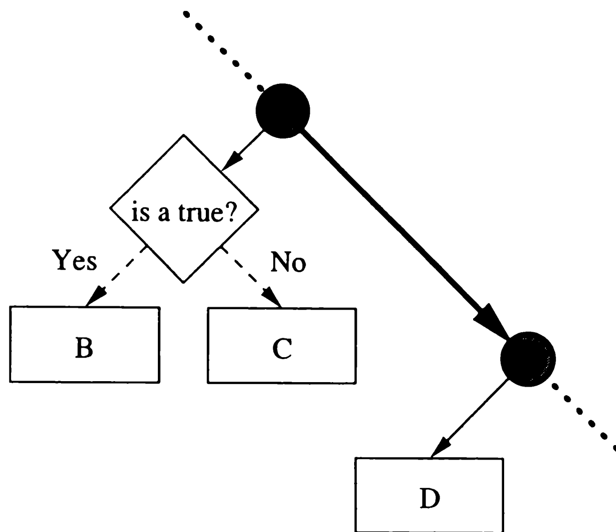


Figure 3.3: Control tree for an if-then-else statement.

traversal of the control tree processing instructions as they are met.

In this example only control dependencies have been considered. No consideration has been given to data dependencies that may hold between control independent blocks. There may be a data dependence between B and D, or C and D that requires the code processed in order. Maintaining true data dependencies is discussed in the following subsection.

Similar rearrangements of control independent points can be used to turn looping constructs into trees. If each iteration of a loop starts at a control independent point then the control tree of Figure 3.4 can be generated. At each node in the right hand control branch a check is performed to see if the next iteration should be executed. If so an iteration and another control node are invoked.

Function and procedure calls map nicely onto a tree-based control mechanism. Each function call forms another control sub-tree because it represents the start of a control independent section of code. A tree based control backbone for a series of function calls is shown in Figure 3.5. The continuation point from the return of a function call also forms a control independent point. This allows consecutive function calls to be strung together using the right branches of the tree to form a control backbone. Recursive function calls are a natural extension of this control subtree idea. Each function simply calls itself at a lower level in the control tree.

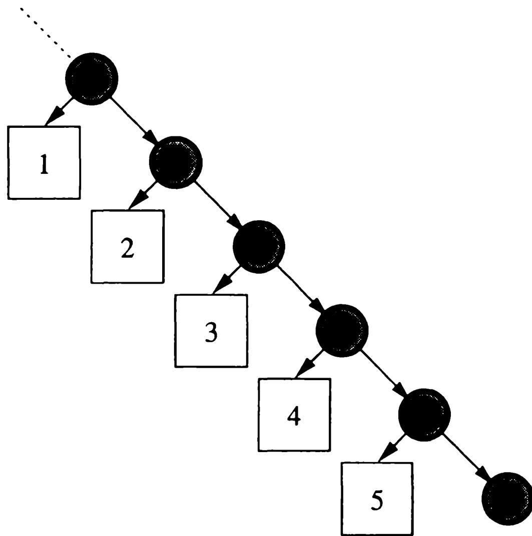


Figure 3.4: Control tree for loop iterations.

More efficient tree structures can be generated by merging nodes and fanning out loop control structures. Control tree optimisations will be discussed in Chapter 5.

3.2.2 Data value speculation

In modern processors the most common forms of speculation are those that predict the direction in which program control will proceed. The types of data speculation described in Chapter 2, address prediction, value prediction, and memory system optimism can also be used. Speculated values can allow execution to proceed at a faster rate than would otherwise be possible. Performance improvements are achieved by obtaining and processing values early.

Performance can also be improved by doing 'the right execution for the wrong reasons'. Here the data being processed may be wrong but the outcome of some computation performed on it is speculatively correct. For example, in the comparison statement

```
if x < 100 then ...
```

x can take on many values and still produce a `true` answer. The execution of conditional statement(s) can proceed even though an incorrect value for x may have been used to evaluate the condition.

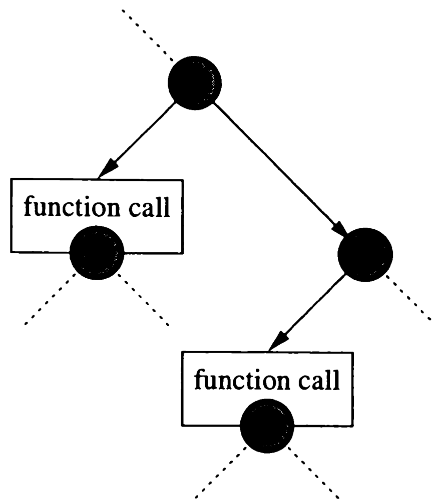


Figure 3.5: Control tree for function calls.

An example showing how speculation can improve performance for the addition of two Binary Coded Decimal (BCD) numbers is given by Littin [1999a]. The BCD arithmetic example shows that speculation on the value of a variable across loop iterations can improve performance.

Data value speculation is achieved by allowing memory operations to occur out-of-order. This eliminates false data dependencies because accesses are not restricted to occur in their programmed order. True data dependencies are speculated on by returning values early.

Timestamps

To detect data dependence violations all memory accesses are *timestamped*. The timestamps impose a single, linear, temporal order on all the memory operations. These timestamps are checked to determine the correct ordering of memory location accesses. An example of their use is given in Section 3.3.5.

The problem is that an unknown number of timestamps are needed during execution. Variable length timestamps have been proposed for parallel discrete event simulation [Back and Turner, 1994]. The problem with representing these timestamps in hardware is that the space to store them is of fixed size. Some more practical fixed length schemes that reuse discarded timestamps are described by Cleary el al. [1997] and McWha [1999]. This is an area of investigation that is beyond the scope of this thesis.

3.3 WarpEngine architecture

Both control flow and data value speculation allow instructions to be processed in parallel. Time Warp provides a mechanism to synchronise interactions between instructions to ensure sequential consistency is maintained. These ideas are combined in the design of a speculative architecture called the WarpEngine.

The WarpEngine architecture described here uses a 4-way control tree, an extension of the binary control tree, with a fixed size block of instructions at each node. When a code block becomes active it is placed in a hardware resource called a *frame*. Within each block instructions are processed in a data flow manner allowing concurrent execution using multiple *function units*.

Special instructions allow data to be passed directly between a parent and its child blocks. Other inter-block communication is handled through reads and writes to a conventional shared memory system. A distributed *time-space cache*, incorporated into the memory system, allows out-of-order memory accesses and is used to ensure sequential consistency. A GVT mechanism interacts with the frames and time-space cache to free resources when they are no longer required for state saving. The components of the WarpEngine and the connections between them are illustrated in Figure 3.6 with a description of them given in the following sub-sections.

3.3.1 Instruction set

The instruction set for the WarpEngine is given in Appendix A. Instructions can be grouped as control, data movement, comparison, logical, integer arithmetic, and floating point. There is a single control instruction, *child*, which is used to generate new nodes in the control tree. It can be conditionally executed and along with the *cmp* instruction, which incorporates all arithmetic comparison possibilities, providing conditional branching functionality. An example of conditional execution in WarpEngine assembly is given at the end of Appendix A.

The only other conditionally executable instructions are the memory access and data movement operations. These are tied to the *child* instruction allowing data to be passed between blocks. The *mv* (move data) instruction moves data directly from parent to child

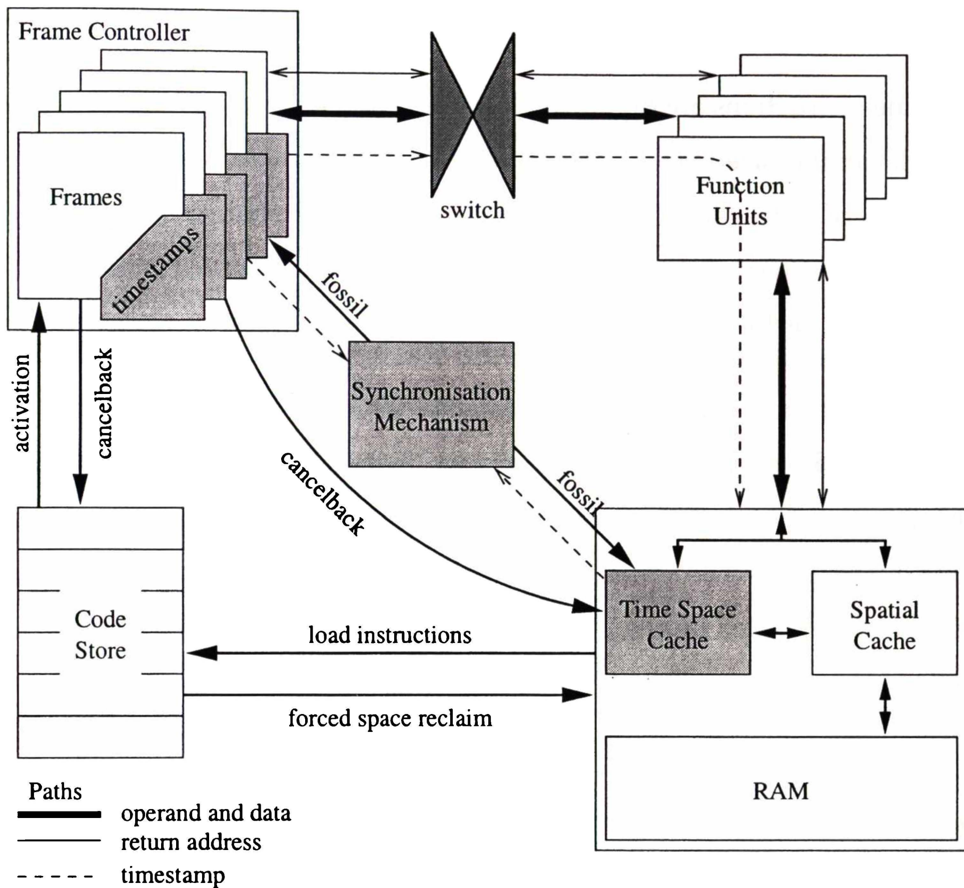


Figure 3.6: Components of the WarpEngine and their connections.

block registers. Memory data storage is handled by the *st* (store) instruction. Data retrieval is performed by the *ma* (load, or move from address) instruction which forwards the value read to a child block register.

The logic and arithmetic instructions are standard, operating on 32-bit operands. Each of these instructions sends results back to registers. With the logic instructions a single result is produced. In other instructions where overflow or results with several parts might be obtained, such as the quotient and remainder of integer division, several values are returned. This removes the need for flags that are traditionally used to signal overflow and other exceptions. Any instruction that depends on these flags can obtain the flag values from registers.

A description of the instruction set and the bitwise representation of instructions used throughout this thesis is given by Cleary et al. [1995b].

Instruction blocks

WarpEngine instructions are organized into fixed-sized blocks of 16 fixed-width instructions. With variable sized blocks, the amount of information to save state is unknown. By fixing the size of instruction blocks the problem of efficient state saving becomes tractable in hardware. Another advantage is that the IRB can be configured statically at compile-time rather than performing dependence analysis at run-time.

Within a block registers are given unique names which are determined compile-time, removing the need for dynamic renaming in hardware. Instructions each have two fixed registers associated with them from which operands are obtained. Only destination registers for results to be sent to are specified in the instruction itself. That is, the operand registers are determined by the placement of the instruction in the block. This style of instruction set provides a data forwarding mechanism that allows data flow techniques to be used when scheduling individual instructions for execution.

A fixed-length block-structured instruction set architecture that removes the register renaming and dispatch stages from an out-of-order function unit pipeline is described by Neefs and Van Campenhout [1996]. A description of the advantages and disadvantages of block structured architectures can be found in Neefs [1996]. In the WarpEngine, Neefs' fixed sized block architecture is taken one step further by removing restrictions on the number of each type of instruction in a block, allowing better filling of blocks. It has been shown that a block-based architecture that incorporates a tree control mechanism can extract large amounts of parallelism [Littin et al., 1998].

3.3.2 Code store

The code store is a mechanism that prepares blocks of instructions for execution. Blocks are loaded from memory and assigned to frames. The program's control tree is maintained through this mechanism which is consulted when child frames are to be allocated. Timestamps for instructions are generated and assigned to blocks when they are allocated frames.

The code store also has the job of maintaining a list of cancelled back frames. When a frame is cancelled a record of its state must be kept so that it can continue executing

when it is rescheduled. This record could be kept in fast special purpose hardware or in data structures in memory.

3.3.3 Frames

Frames perform a similar function to the IRBs of contemporary architectures. They provide the physical locations for registers, a control mechanism to execute instructions, and are used for state saving. Associated with each frame is a timestamp that is assigned to any memory operations performed by instructions in the frame. Frames are processed in parallel and are treated as independent instruction streams. A frame controller is used to arbitrate resource contention and to manage communications between frames, function units and the code store.

Each frame consists of 16 *slots*, one for each instruction, the layout of which is shown in Figure 3.7. A slot contains four fields: an op-code, execution status flags and the instruction's two operand registers. The status flags are used to determine the validity of instruction operands and to maintain the execution state of each instruction.

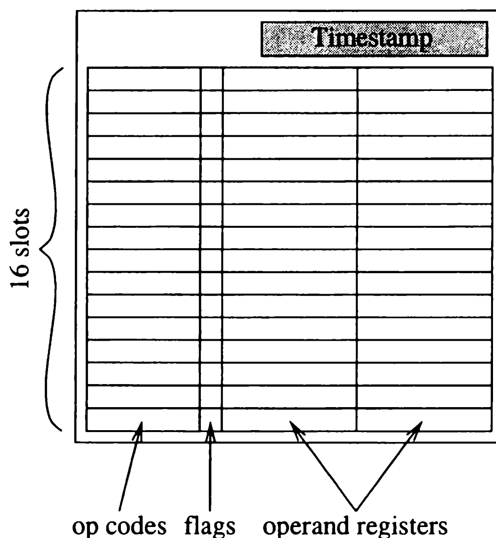


Figure 3.7: Layout of a frame.

A possible hardware implementation for a WarpEngine frame is described by Calvert [1997]. To ensure good performance the execution of instructions must be efficient. A description of an efficient mapping routine for fixed sized instruction blocks onto reconfigurable hardware is given by Siemers and Möller [1998]. The exact imple-

mentation to be used in the WarpEngine is not known, but the ideas put forward by Siemers and Möller could be considered.

3.3.4 Function units

The function units perform the same operations as any contemporary 32-bit 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 timestamps, 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 Memory system

The memory system of the WarpEngine is a three level hierarchy. At the level closest to the frames there is an associative memory cache, the *time-space cache*. This cache records speculative memory operations and detects data dependency violations, re-issuing reads to correct the violations. It contains triples of the form *(address, timestamp, value)* for all recent writes, and *(address, timestamp, destination register)* for all reads. Only entries with timestamps greater than GVT need to be stored because they are speculative. Those prior to GVT can no longer affect computation and are retired with writes committed to RAM.

The next two levels of the memory hierarchy are a standard cache hierarchy, denoted *spatial cache*, and local dynamic RAM. At these levels memory is referenced only by address, timestamps are not involved. Values stored in this part of the system are committed and are guaranteed never to be rolled back. Figure 3.8 shows a diagram of the memory subsystem's major components and the connections between them.

Six operations are possible on the memory subsystem read, write, anti-read, anti-write, fossil collection and cancelback.

A read will generate two parallel accesses: one to the time-space cache and another to

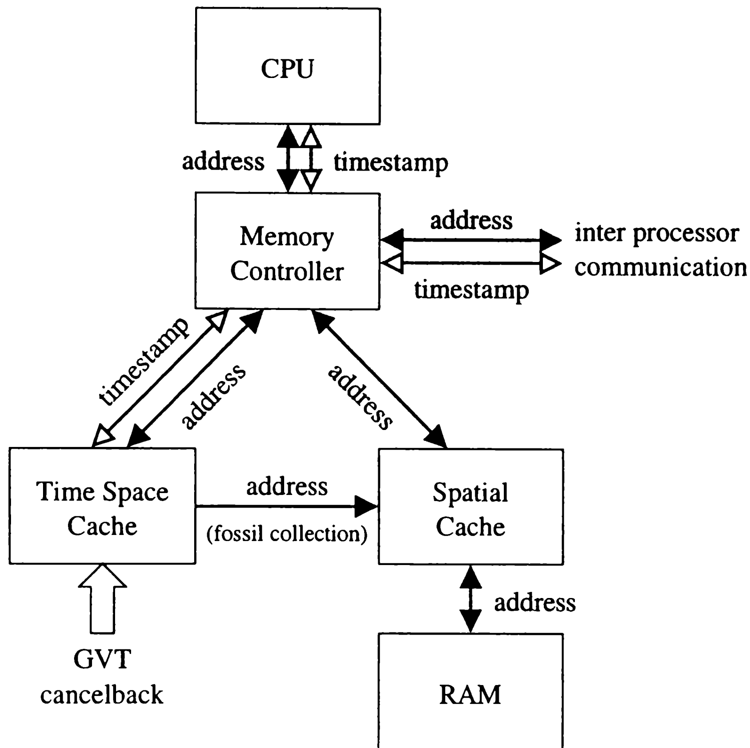


Figure 3.8: Components of the timestamped memory system.

the spatial cache. All recorded writes with the same address will be matched and the one which is most recent but prior to the timestamp on the read will be fetched. There are two possible results to this operation: 1) there may be a tuple that matches the read in which case the associated value is returned, or 2) if there is no match then a check is made to see if the spatial cache returned a result. If it did then that is used as the result. If there is still no value available then the read request is passed down the memory hierarchy to Random Access Memory (RAM).

A write is sent only to the time-space cache where it is recorded. All recorded reads for the same address with timestamps greater than that of the write but less than the timestamp of the next recorded write to the same address are retrieved. The value of the new write is sent to the destinations of each of the reads. This process is illustrated in Figure 3.9 where a write with a timestamp of 106 is performed. The reads with timestamps of 109 and 110 originally returned a value of 234 but after the write occurs a new value of 567 is returned. Any reads with timestamps less than 106 and greater than 120 are not affected. Each time a read is re-satisfied with a new value a rollback occurs and dependent instructions are re-executed.

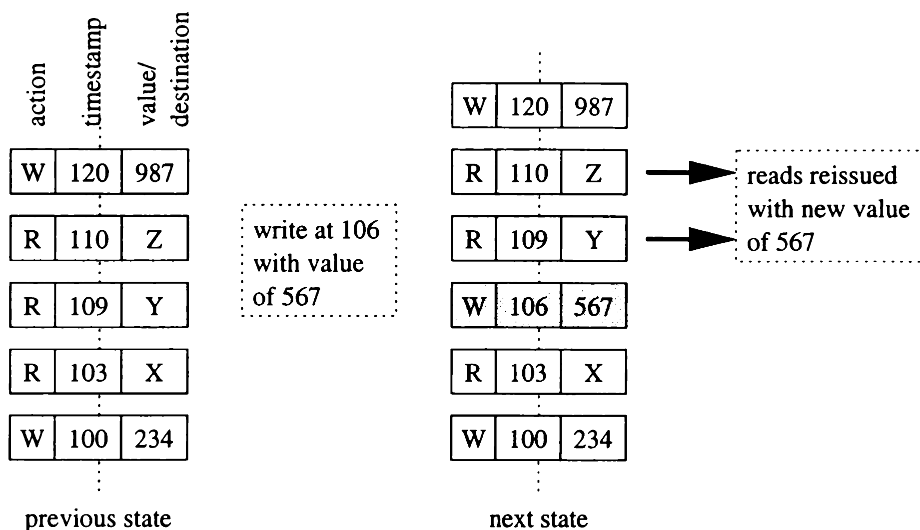


Figure 3.9: Time-space cache entries with a write occurring out-of-order.

An anti-read removes the corresponding read entry from the time-space cache. Anti-reads only occur as a result of cancelback or a misspeculated branch with any consequences of the cancelled read handled by the cancelback or control mechanism. An anti-write first locates its corresponding write and deletes it. Then all recorded reads after the deleted write and before the next recorded write to the same address are found. For each of these reads a reply is generated giving the value of the earlier write, or if there is no earlier write tuple the conventional memory system is consulted to find a value.

The retirement process deletes both read and write entries. All reads with timestamps earlier than the current value of GVT are deleted. Each write entry earlier than GVT is retrieved. For each unique address of these writes only the value with the largest timestamp is written directly to the spatial cache. The write entries are then deleted.

Cancelback removes all entries greater than a given timestamp value. No writes to the spatial cache are made. Their entries will eventually be regenerated when the corresponding frame is re-executed. Further details of a timestamped memory system are given by Cleary et al. [1995].

3.3.6 Synchronisation mechanism

The synchronisation mechanism is used to free frame and time-space cache resources that are no longer required. The synchronisation mechanism makes extensive use of

timestamps to order events in the system. An effective timestamp system must therefore permit simple comparison, be easy to generate, have unbounded range, and be of fixed size.

In the WarpEngine a suitable timestamp scheme is assumed to exist. The simulation model, described in Chapter 4, does not require timestamps to perform its simulation runs, so the implementation of timestamps is not important as long as it meets the requirements for an effective timestamp scheme. Work on potential timestamp schemes is being performed as part of the WarpEngine project. Results obtained so far are reported in [Cleary et al., 1997] and [McWha, 1999].

In a similar vein the exact implementations of the fossil collection and cancelback mechanisms have not been developed. These mechanisms will be tied to the timestamp implementation and will depend on the tree control mechanism. While investigations into these mechanisms have been performed, hardware implementations have not been proposed at this time.

3.3.7 Instruction execution

Each instruction in memory is represented by two words. The first, an *instruction-word* (I-word) contains the *op-code* and result destination information. The second, a *constant-word* (C-word), contains a single literal or constant value that can be used as one of the instruction's operands. An instruction goes through a number of stages in its execution. These are illustrated in Figure 3.10.

First, the instruction block is initiated and allocated a frame. The *op code* and *s* flag, which indicates 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 of an instruction, or with data that results from the execution of another instruction.

Data flow techniques are employed to execute the instructions within a frame. When 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 instruc-

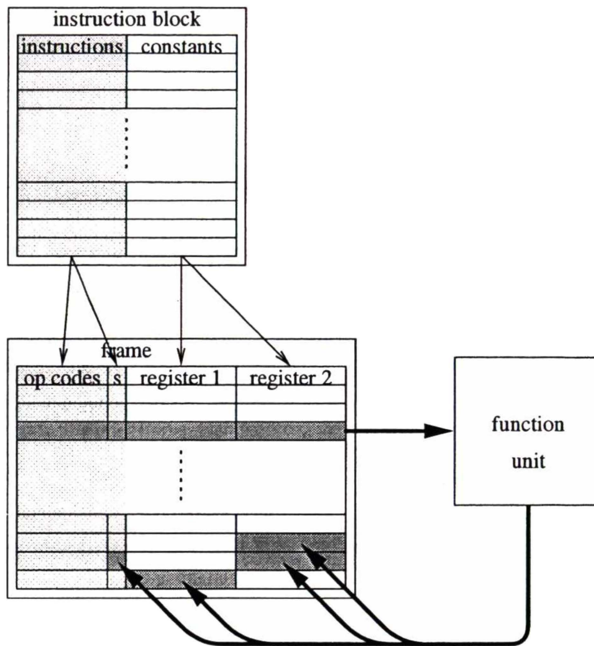


Figure 3.10: Instruction flow within a WarpEngine frame.

tion with a unique timestamp 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.

3.4 Comparison

The WarpEngine is suited to extracting parallelism from problems with tasks, such as loops and function calls, that have a high probability of being independent. That is, there are only a few potential dependencies between tasks, but their presence means that static parallel scheduling cannot be used. The WarpEngine executes these tasks as though they are independent, with dependencies maintained by code re-execution that is regulated by the memory system. The memory model seen by a programmer is a single linear address space that is modified by a single thread of control. Thus, there is no need for locks or other explicit synchronising actions when accessing memory.

In this section the WarpEngine is compared and contrasted to the dynamic scheduling ILP paradigms discussed in Section 2.3.

3.4.1 Superscalar

In superscalar architectures the single access point to the memory system and the reuse of registers creates false dependencies between instructions. False register dependencies are resolved through the use of register renaming at run-time, while false memory access dependencies a dealt with on a small scale with memory re-order buffers. The size of instruction and memory re-order buffers are limited by non-perfect branch prediction mechanisms which cannot fill large buffers.

The WarpEngine eliminates these false dependencies by allowing memory accesses to occur in any order and through the single use of registers within code blocks. By default data dependencies are assumed not to exist with any true dependencies maintained using timestamps and the time-space cache.

Superscalar machines execute instructions out-of-order with respect to their programmed order, but only when correct data is available. That is they only ever perform correct execution. The WarpEngine is more aggressive in that it processes instructions out-of-order possibly with incorrect operands. The timestamped memory system and statically arranged fixed sized code blocks of the WarpEngine allow incorrect data to be detected and affected instructions to be re-executed.

The WarpEngine processes many streams of control independent instructions in parallel. This allows memory accesses to be generated out of programmed order. These instruction streams are scheduled by a tree-based control mechanism that breaks the linear control flow that is inherent in the program counters of superscalar machines.

3.4.2 Multiscalar

The WarpEngine is similar to the Multiscalar machine and Trace Processor in many aspects. The major difference is that those machines use a fixed hardware based communications ring to order memory accesses and tasks while the WarpEngine assigns timestamps to order instructions and memory accesses. Timestamps free the hardware from directly imposing an ordering on data and tasks. This allows the WarpEngine to used more flexible and scalable networks to connect processing resources, rather than the ring structure used in the Multiscalar machines.

In the Multiscalar machine tasks are squashed and re-executed from the beginning whenever an incorrect memory read is detected. In contrast, like the Trace Processor the WarpEngine re-satisfies the affected read which then forces only those instructions dependent on the read to be re-executed. The finer grained state saving used in the WarpEngine means that only incorrect computation is re-executed.

The WarpEngine's tree-based control mechanism schedules independent instruction streams from control independent points in the program. This is similar to the Multiscalar machine's task predictor and the Trace Processor's trace predictor but differs in that mis-predicted control does not force all speculative tasks to be squashed. As with data value mis-speculation, control mis-prediction does not affect truly independent instruction streams.

3.4.3 Data flow

As with data flow, the WarpEngine pools a large group of instructions for execution. The control driven nature of the WarpEngine and its data value speculation removes the need for the large token matching stores seen in data flow machines. The fixed size and relatively small instruction blocks allow the WarpEngine to efficiently execute instructions in a data flow manner. Local function units can be used rather than the global token match and processing loop of data flow machines allowing fast processing of sequential instructions.

Traditional data flow architectures consider all instructions equal. In the WarpEngine the order imposed by timestamps can be used to prioritise instructions when resource contention becomes an issue. Events with smaller timestamps are given preference when queuing for resources. This means that the instructions most likely to be delaying program execution are given processing resources in preference to the more speculative instructions.

To get good performance from data flow machines programs are written in special data flow languages which require non standard programming models and methodologies. The WarpEngine operates on programs written in popular programming languages, although special languages and language constructs could be developed to exploit features of the WarpEngine.

3.5 Summary

This chapter has discussed concepts used in the WarpEngine. Speculation on the outcome of control decisions and data values in memory is used to improve performance. Rather than trying to predict control flow paths on known correct data the WarpEngine speculatively executes multiple streams of control on *potentially* correct data. Code is broken into streams at points that will execute independent of the branching decisions prior to them. These instruction streams are then mapped onto a distributed tree-based control mechanism.

To synchronise speculative events the Time Warp mechanism has been incorporated into hardware. Instructions and data accesses are tagged with timestamps defining a single, linear, temporal order on events. This virtual ordering is used to ensure true data dependencies are maintained.

The WarpEngine architecture described uses a block-based instruction set. Instructions are grouped into fixed-sized 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 success of the WarpEngine is dependent on being able to design a number of critical components. As the WarpEngine is a new architecture that contains new architectural ideas, many of its components have been specified and but they have not been designed in detail. Part of the design of these components will require finding solutions to problems of power distribution, power consumption and verifying chip layout. Gate level design and analysis of components is beyond the scope of this thesis which is concerned with high level performance and concept validation. The functional level simulation used here models components without knowing their exact implementation.

When developing the simulation model many architectural issues were considered. For example the simulation model closely models the required structure necessary to monitor and flag the validity of data within registers in a frame. However, issues associated with the memory system require more thought. For efficient processing, high bandwidth to instruction memory is required to allow all the instructions in a block to be loaded in par-

allel. Conceptually this is just a matter of using wider memory buses turning the problem into a resource allocation trade-off. The time-space cache has some special considerations to take into account. It has to have comparable speed to traditional cache requiring timestamp comparisons to be performed in parallel. This may require some form of content addressable memory capable of performing hundreds of timestamp comparisons in parallel. Careful consideration will need to be given to the design of the time-space cache to ensure a melt-down of silicon does not occur.

The exact mechanics of the GVT, fossil collection, and cancelback mechanisms, which depend on the timestamp scheme used have not been implemented although their design and requirements have. The simulation methodology described in Chapter 4 does not rely on timestamps to model the WarpEngine. This means that analysis can be performed even though timestamp dependent components have not been developed.

Chapter 4
Simulation

Shriver and Smith [1998] describe the design process for the development of a computer architecture, and provide a diagram for this large and involved process. The top part, describing early development stages, is shown in Figure 4.1. This thesis investigates pieces within the area indicated by the shaded region, specifically developing the instruction set, evaluating performance and resource costs, and modelling the architecture through behavioural and functional simulation and testing.

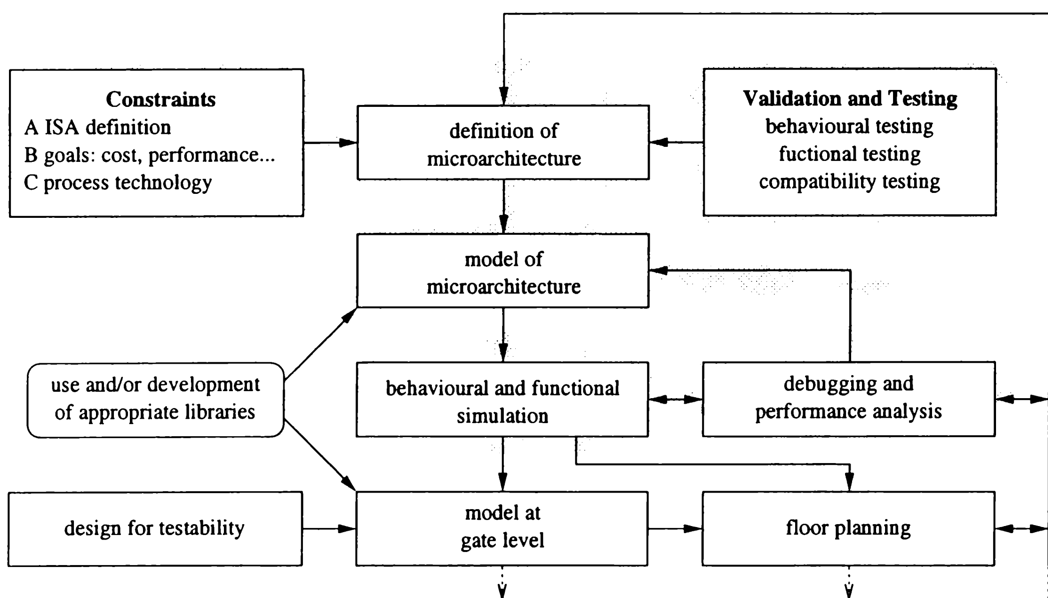


Figure 4.1: Design process for a computer architecture.

To gain an insight into the characteristics of the WarpEngine architecture a faithful description of the known architecture components is required. A simulator modelling the known characteristics of the WarpEngine provides information about performance and costs. This chapter discusses computer architecture simulation techniques, and describes

a novel simulation technique that has been developed to model the speculative architectures.

4.1 Background

In a simulator the level of detail modelled determines the type, amount, and accuracy of the information that can be obtained. The information returned could range from a single number related to system performance through to detailed statistics about individual components. There are a variety of simulation techniques that can be used, including *analytical modelling, trace analysis, instruction set simulation, functional simulation* and *gate level simulation*. A description of these techniques is given by Muller [1993].

Well known simulators include; SPIM [Larus, 1993], a functional level MIPS emulator; SHADE [Cmelik and Keppel, 1994], an instruction set simulator; and SimpleScalar [Burger and Austin, 1997], a configurable MIPS-based architecture simulator. They are used to test the viability of hardware concepts. Trace analysers, such as TETRA [Austin and Sohi, 1993], can be used to determine the performance gains obtained by altering the characteristics of individual components in a system. All these techniques and associated tools have important roles to play in the creation of a new architecture.

Within simulators there exists a clock that is used to determine event ordering. Depending on the model there are two methods of advancing the clock, *fixed increment* and *next event* time advance [Law and Kelton, 1991]. In a fixed increment model the clock is advanced by a constant period, usually one cycle. System components are updated based on their state from the previous cycle and the state of their inputs at the current cycle. With next event time advance, events are queued in time order and the simulation clock is updated to the smallest time in the queue. This can lead to non-uniform clock increments.

4.2 Modelling execution paradigms

This section describes how sequential and out-of-order execution paradigms can be modelled. A hypothetical MIPS-based architecture is used in the examples in this section.

4.2.1 Sequential execution

To examine the process of simulating sequential execution consider the code for the addition of two decimal numbers stored as an array of characters shown in Figure 4.2. MIPS assembly code [Goodman and Miller, 1993] for the loop iteration, lines 8 through 10, is given in Figure 4.3.

```
 1:  unsigned char A[] = {6,5,4,0},
 2:      B[] = {4,3,2,0},
 3:      C[] = {0},
 4:      carry;
 5:
 6:  carry = 0;
 7:  for (int i=0;i<3;i++) {
 8:      C[i] = A[i] + B[i] + carry;
 9:      carry = C[i] / 10;
10:      C[i] %= 10;
11:  }
12:  C[3] = carry;
```

Figure 4.2: C code for the addition of two decimal numbers stored as arrays of characters.

In a sequential architecture execution proceeds by reading an instruction from memory, performing whatever actions are necessary to process the instruction, and incrementing the instruction counter to the next instruction. This process is repeated for each instruction in the program's control path.

The simulation model for this architecture performs operations in the same sequential order. That is, each instruction is loaded and evaluated, updating any outputs, in programmed order. For example, when processing instruction 9 (`addu`) in Figure 4.3 the values in registers `4` and `5` are read, an unsigned addition is performed and the result is placed into register `4`. The simulator's clock is updated to reflect the new program counter value. This sequence is repeated until all instructions have been processed.

```
        ⋮
 1:  la     $3,C
 2:  addu   $3,$6,$3      # loop index in register $6
 3:  la     $4,A
 4:  addu   $4,$6,$4
 5:  la     $5,B
 6:  addu   $5,$6,$5
 7:  lbu    $4,0($4)      # load A[i]
 8:  lbu    $5,0($5)      # load B[i]
 9:  addu   $4,$4,$5
10:  lbu    $5,16($fp)    # load carry
11:  addu   $4,$4,$5
12:  sb     $4,0($3)      # store C[i] = A[i] + B[i] + carry
13:  li     $2,10
14:  divu   $4,$2
15:  mflo   $2
16:  sb     $2,16($fp)    # carry = C[i] / 10
17:  mfhi   $2
18:  sb     $2,0($3)      # store C[i] = C[i] % 10
        ⋮
```

Figure 4.3: MIPS assembly for lines 8 to 10 of the C code in Figure 4.2.

4.2.2 Out-of-order execution

With out-of-order execution, instructions enter the CPU and are stored in an IRB where analysis determines inter-instruction dependencies. Appropriate actions are taken to allow independent instructions to execute in parallel and to ensure dependent instructions are processed in-order. Upon completion of execution instructions are held in the IRB waiting for any prior instructions to be processed so they can be retired in order. Retirement removes processed instructions from the IRB allowing more instructions to enter.

Example 4.1: To show how out-of-order execution is modelled consider an out-of-order machine that has a IRB of five instructions and two general purpose function units with 1-cycle instruction propagation delays. It is assumed that only one memory access can be performed per cycle. The inter-instruction dependencies for instructions 7 to 18 of Figure 4.3 are shown in Figure 4.4 by the solid arrows. The serialised memory access constraint imposes the dependence path denoted by the dotted arrows.

Figure 4.5 shows the contents of the IRB and the instructions executed on each clock cycle. On the first cycle $\texttt{lbu}^7$, $\texttt{lbu}^8$ and $\texttt{addu}^9$ are in the IRB and $\texttt{lbu}^7$ is processed. On the second cycle $\texttt{lbu}^7$ is retired (possibly with other prior instructions) allowing $\texttt{lbu}^{10}$,

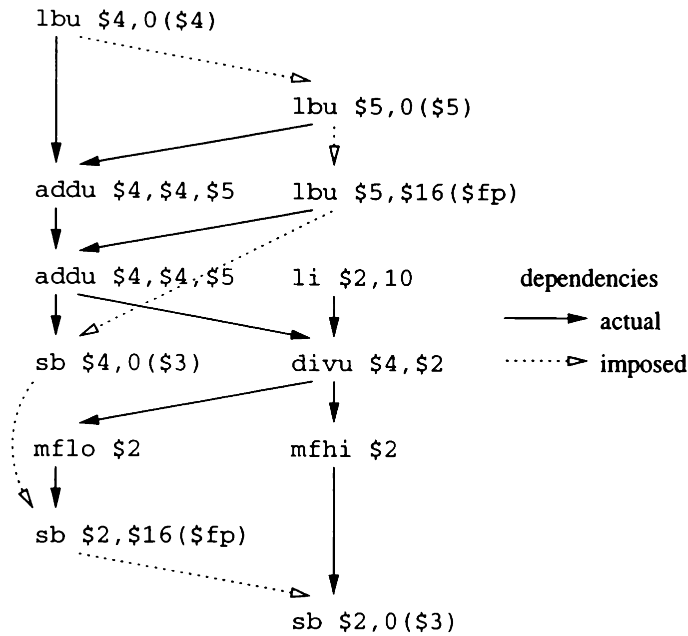


Figure 4.4: Inter-instruction dependencies for instructions 7 to 18 of Figure 4.3.

`addu`^{11} and `sb`^{12} to enter the IRB. At this time only `lbu`^{8} can be processed because `addu`^{9} depends upon it and `lbu`^{10} is blocked by the memory access constraint. On cycle three `addu`^{9} and `lbu`^{10} can be processed in parallel, and `lbu`^{8} is retired allowing `li`^{13} to enter the pending window. The remaining instructions are processed similarly maintaining any dependencies.

clock cycle	*IRB*					*processing*	
1		...	lbu^{7}	lbu^{8}	addu^{9}	lbu^{7}	
2	lbu^{8}	addu^{9}	lbu^{10}	addu^{11}	sb^{12}	lbu^{8}	
3	addu^{9}	lbu^{10}	addu^{11}	sb^{12}	li^{13}	addu^{9}	lbu^{10}
4	addu^{11}	sb^{12}	li^{13}	divu^{14}	mflo^{15}	addu^{11}	**li**^{13}
5	sb^{12}	**li**^{13}	divu^{14}	mflo^{15}	sb^{16}	sb^{12}	divu^{14}
6	mflo^{15}	sb^{16}	**mfhi**^{17}	sb^{18}	...	mflo^{15}	
7	sb^{16}	mfhi^{17}	sb^{18}			sb^{16}	**mfhi**^{17}
8	sb^{18}					sb^{18}	
9	...						

Figure 4.5: Out-of-order execution for instructions 7 to 18 of Figure 4.3.

Note that on cycle four `li`^{13} is processed before `sb`^{12} on cycle five. This out-of-order processing forces the `li`^{13} to wait in the retirement buffer so that on cycle six it can be retired in-order. `mfhi`^{17} could have been executed one cycle earlier because there was a free function unit and no inter-instruction dependencies. However, due to the size limit of the IRB it could not be prepared early enough. If `li`^{13} had been retired one cycle earlier the `mfhi`^{17} could have been processed a cycle earlier.

This is a real-time ordered simulation. Here the simulation clock increments forward at a constant rate, but instructions are processed in an order that differs from that in which they were programmed. This simulation processes instructions in the same order as the architecture it models meaning that the dependence analysis performed by the architecture has to be incorporated into the simulation model.

4.3 Virtual ordered simulation

One way to remove control and data dependence analysis from the simulation model is to process instructions in their programmed order. This order is not necessarily the order in which they would be executed in reality. In-order instruction processing guarantees that instruction operand registers will contain valid values when each instruction is processed, ensuring that control and data dependencies are maintained.

This novel simulation technique is named *virtual ordered simulation*. The name comes from Time Warp where the notion of a program's virtual order is used to generate timestamps. In this simulation methodology instructions are processed in the order in which instructions appear in the program. Real-time timestamps are used to record the times at which instructions would be processed in real architecture.

Virtual ordered simulation differs from other simulation methodologies in that the simulator's clock jumps forward and backward. This occurs because the real-time at which instructions execute is out-of-order with respect to the order in which they are processed in the virtual ordered simulation model.

To model the real-times at which instructions are processed timestamps must be associated with each data and processing resource. That is, timestamps are assigned to registers, memory locations and instruction buffers. An instruction processes data in the same way as a real-time ordered simulation. When data is written to registers or memory the timestamps associated with those storage locations are updated.

Figure 4.6 illustrates the flow of timestamp values during an instruction's processing. Input timestamp values, $t_{in_1}, \ldots, t_{in_n}$, may come from operand registers, instruction buffers, and processing resources. The timestamps of all inputs are compared finding a maximum time, $t_{max} = \max(t_{in_1}, \ldots, t_{in_n})$. This gives the real-time at which all the

instruction's operand registers contain valid values, and there are processing resources available. A time indicating the instruction's processing latency, t_p, is added to t_{max} to give $t_{out} = t_{max} + t_p$. This is the real-time at which output from the instruction is valid. t_{out} is then assigned to all output register or memory location timestamps, and is used to indicate the time at which the instruction's buffer can be freed.

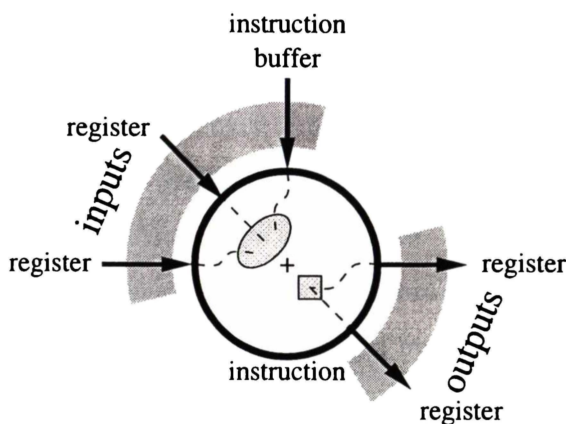


Figure 4.6: The flow of timestamp values during instruction processing.

Example 4.2: To examine the virtual ordered simulation process consider the architecture described in Example 4.1. The state of timestamps in the simulation model are shown in Figure 4.7 for the virtual ordered simulation of instructions 7 to 18 of Figure 4.3.

inst	*invoke*	*IRB*					*procs*		*memory*	*clock cycle*
lbu^{7}	**1**	**1**	1	1	1	1	**1**	1	**1**	1
lbu^{8}	**1**	3	**1**	1	1	1	**2**	1	**2**	2
addu^{9}	**3**	3	4	**1**	1	1	**3**	1	3	3
lbu^{10}	**1**	3	4	5	**1**	1	4	**1**	**3**	3
addu^{11}	**4**	3	4	5	5	**1**	**4**	4	4	4
sb^{12}	**5**	**3**	4	5	5	6	**5**	4	**4**	5
li^{13}	**1**	7	**4**	5	5	6	6	**4**	6	4
divu^{14}	**5**	7	7	**5**	5	6	6	**5**	6	5
mflo^{15}	**6**	7	7	7	**5**	6	**6**	6	6	6
sb^{16}	**7**	7	7	7	8	**6**	**7**	6	**6**	7
mfhi^{17}	**6**	**7**	7	7	8	9	8	**5**	8	7
sb^{18}	**8**	9	**7**	7	8	9	**8**	8	**8**	8
-	-	9	10	**7**	8	9	9	8	9	-

Figure 4.7: Timestamp values during virtual ordered simulation of instructions 7 to 18 of Figure 4.3.

Timestamp values listed in the *invoke* column are determined by the true data and control dependencies between instructions. The *IRB* column lists the timestamp values associated with each of the five instruction buffers. Each value is the time at which the previous

instruction using that buffer was retired. One cycle is added to each buffer timestamp to represent the dependence analysis processing cycle associated with each instruction. The *procs* column maintains one timestamp for each processor that indicates when it is free to process another instruction. These timestamps are updated in such a way that later instructions can be executed as early as possible. The *memory* column keeps track of the next cycle at which the memory system can accept an access. This is required because of the ordered memory operation constraint. The final column, *clock cycle* gives the cycle at which each instruction is processed.

The clock cycle values for each instruction are determined by taking the maximum value of the invoke, the appropriate instruction buffer, where appropriate memory, and the minimum of the processor timestamps. For each instruction the timestamps examined in the clock cycle calculation are highlighted. The timestamps of the resource(s) that determine an instruction's clock cycle have been underlined.

`lbu`^{7} enters the system in the first step and invocation time is set to 1. The maximum of the invoke, the first IRB, and the memory timestamps is 1, which is assigned to the clock cycle. The first instruction buffer's timestamp is set to 3 because the instruction is retired on cycle 2 and one cycle is added for dependence analysis. `lbu`^{8} is then processed. Its invoke time is set to 1 as it has no input dependencies. This time the invoke, the second IRB and memory timestamps are checked. The clock cycle is set to 2 and the second instruction buffer's timestamp is set to 4. `addu`^{9} is the next instruction processed. It is dependent on the outcome of the previous two `lbu` instructions, the latest of which will have a value ready on cycle 3. Because this instruction does not access memory the processing cycle is determined by the invoke and third IRB timestamps.

Processing continues in a similar manner for all instructions. Note that `li`^{13} and `mfhi`^{17} are constrained by the IRB size. `li`^{13} is processed on cycle 4 which puts it out-of-order with respect to `sb`^{12}. As a consequence the corresponding IRB timestamp is set to 7, the value of the previous buffer's timestamp. This represents the extra time taken waiting for `sb`^{12} to be processed and retired.

Comparing the clock cycles to the results in Figure 4.5 it can be seen that each instruction is processed on the same cycle using both simulation techniques.

4.3.1 Trade-offs

When modelling an out-of-order or speculative architecture, virtual ordered simulation has the advantage that it only has to process each instruction once. Using a single thread of control in the simulation model means that context switching overheads are reduced. The trade-off is the extra space required to store the timestamps associated with every storage location.

The amount of control and data coherence detail that can be abstracted away means that a virtual ordered simulator has a quick development life cycle. Virtual ordered simulation provides a good mechanism for testing high level ideas by examining the effects of applying them to the architecture. Ideas that work can be examined in more detail and those that don't can be eliminated quickly.

Virtual ordered simulation requires oracular knowledge of events. This knowledge is used to prioritise the use of resources with instructions earlier in the program given preference. In a real architecture this level of prioritisation may not be possible, meaning that the statistics obtained through virtual ordered simulation under estimate the overheads of processing. It does, however, provide an estimate of performance when using the best scheduling possible techniques.

Since each instruction is processed in program order, the timestamps associated with each storage location can only detect if speculative instructions would have to be rolled back in reality, not how often they were rolled back. To obtain this information timestamps can be extended to record each change in state. Timestamp extensions are discussed in Section 4.3.3.

4.3.2 Limiting resources

In Figure 4.7 there were limits placed on the number of IRB entries, processors, and concurrent accesses to the memory system. To limit concurrent memory accesses to one a single timestamp records the times at which each access finishes its transaction. Each access to memory has its invoke timestamp compared to the global memory access timestamp and the access processed at the greater of the two times. In this way each memory access is processed in order with a maximum of one access at a time.

In situations where there is more than one resource, such as multiple processing units and IRB entries, more complex timestamp structures are required. These resources can be classified by whether they are used and retired in-order or out-of-order in the simulation model. For example an instruction is assigned to an IRB in programmed order, whereas instructions can use processing units at any time.

Processing resources

To model a fixed number of processing resources in a virtual ordered simulation environment a count of the number in use at any time is required. This can be achieved using a sequence of time-range buckets that each contain a count of the resources in use over the time period they cover.

When an instruction is processed the time-range buckets are queried to find the first bucket whose end time is greater than the instruction's timestamp and whose count is less than the resource limit. The start time of this bucket is returned as the earliest time at which a resource is available to process that instruction. The greater of the instruction's timestamp and the returned start time is used as the instruction's processing time in the simulation model. The bucket's count is then incremented to indicate the resource has been used at that time. Each bucket's count will never exceed the resource limit.

This process is shown in Figure 4.8 where there is a processing resource limit of 4. An instruction is invoked at 103 and the time-range buckets are searched. The bucket that starts and ends at 107 is found as the first bucket whose count is less than the limit and start time is greater than or equal to the instruction invoke time. 107 is used as the processing time for the instruction. The count of that bucket is then incremented indicating that only one more processing resource is available at that time.

These buckets can be efficiently implemented using calendar queues and optimisations such as merging adjacent full buckets.

State resources

To model state saving resources, such as IRB entries, a more efficient mechanism can be used. State saving resources are retired in their programmed order. This means that

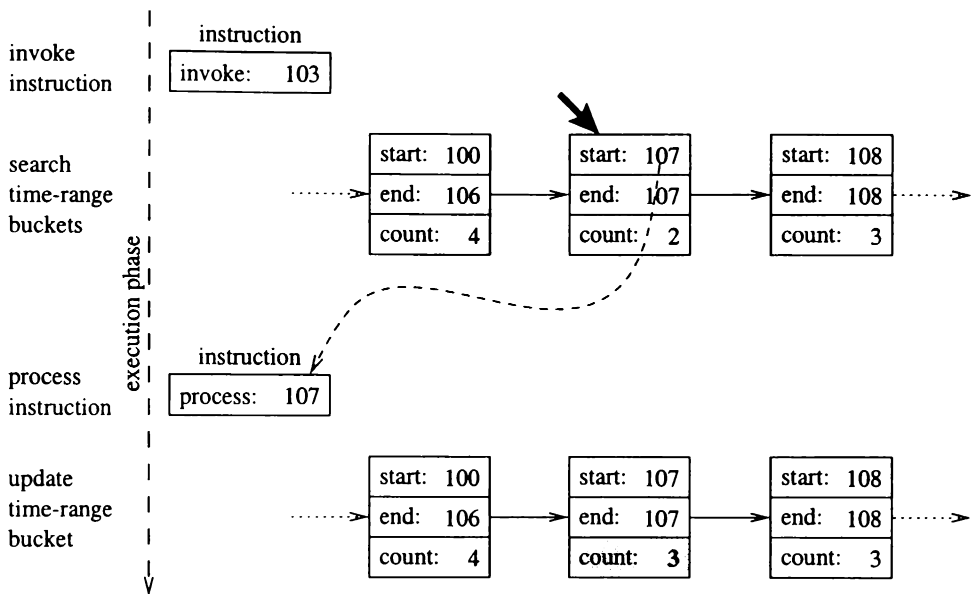


Figure 4.8: Using time-range buckets to model limited processing resources.

successive instructions will release their IRB entries at times greater than or equal to previous instructions.

This can be modelled by an array of timestamps and a current buffer pointer. The time stored in each array element indicates the time at which an associated buffer was freed. The timestamp array size is set to the buffer resource limit and its contents are initially set to 0.

When an instruction is assigned to a buffer its *start time* is calculated from the greater of the instructions invoke timestamp and the timestamp of the current buffer. After the instruction is processed the current buffer's timestamp is updated to a value which is the greater of the executed instruction's end time and the previous buffer's retirement time. The current buffer pointer is then incremented, modulus the resource limit, to point to the next timestamp in the array.

An example of this process is given in Figure 4.9. The executing instruction is invoked at time 96. The limited frames mechanism is consulted and an earliest start time of 97 is returned. The frame is then executed with this new start time and completes at time 98. This time is sent to the limited frames mechanism which updates the freed buffer time and increments the current index. A value of 100 is stored, as opposed to 98, because even though the instruction finished at 98 its resources could not be freed until previous

instructions are retired at time 100.

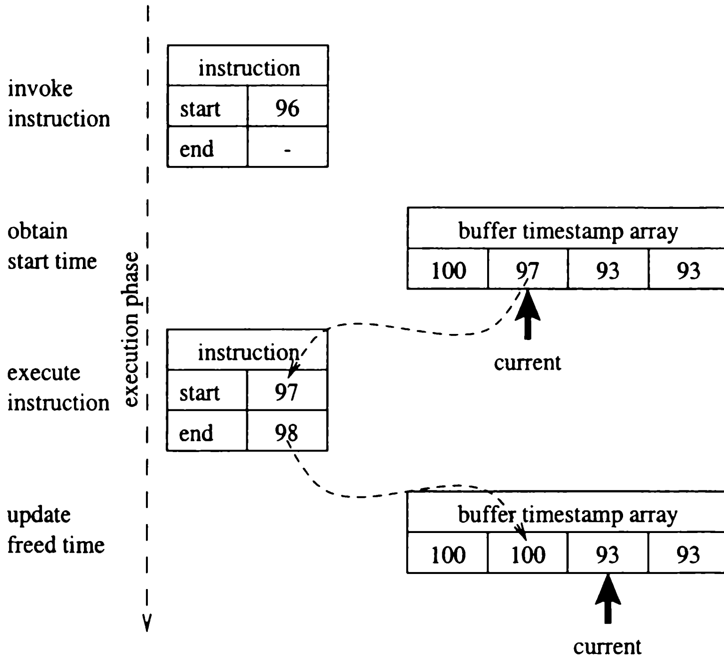


Figure 4.9: Communication between a frame and the limited frames mechanism.

This mechanism is used when limiting the amount of state saving resources in the performance measurements of Chapters 7 and 8.

4.3.3 Extensions

More accurate information can be obtained from the virtual ordered simulation model by replacing timestamps with other forms of timing information. Critical path or state history lists can be used to obtain instruction count and transient state information, respectively. These additions allow the virtual ordered simulation model to emulate some features of real-time ordered simulators and provide information that cannot be obtained easily from real-time ordered simulators.

Critical paths

A critical path through a program is the sequence of instructions that determines the program's overall execution time. These instructions must be processed in order. Information about the mix of instructions on a program's critical path can be used to improve performance.

To compute the critical path in the virtual ordered simulation model each timestamp is replaced by a list of instructions. The list represents the path of instructions of largest latency required to generate the data value stored. When an instruction is processed the critical paths of each input are compared to find the one with the largest time. An entry containing the processed instruction is added to this list and the new list is assigned to each output. If two or more input lists are of the same length the one with the largest number of memory operations is used. Memory operations were chosen because they have a significant impact on performance as is shown in Chapter 6. The creation of an output critical path list is illustrated in Figure 4.10.

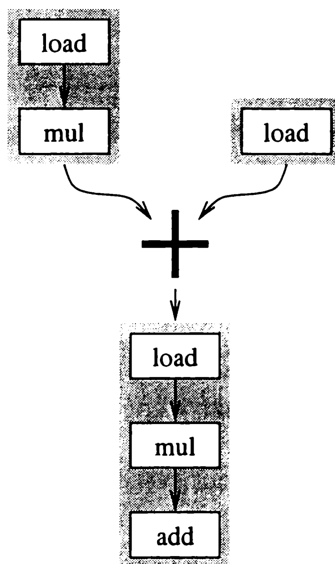


Figure 4.10: Generating an output critical path list from two input critical path lists.

At the completion of simulation the critical path of the whole program is given by the list of instructions that were required to perform the last update of the simulation clock.

When investigating the effects of certain critical path features more efficient mechanisms can be used. In Chapter 6 program instruction mixes on the critical path are examined. To capture this information only instruction counts need to be recorded. As each instruction is processed the instruction counts of its operands are compared to find the largest total count. The instruction distribution of the largest count is passed to each output with processed instruction's count incremented in each.

Also investigated in Chapter 6 is the effect that increasing an instruction's propagation delay has on performance. One way to perform this analysis is to run many simulations

with varying instruction latencies. Another approach is to produce an equation representing the program's critical path length as a function of one instruction type's latency, for example the time to perform a memory fetch. To do this timestamps in the simulation model are replaced by path equation lists.

To represent the execution time of an instruction path two counts that produce a linear equation of the form $mx + c$ are kept. In this equation m is the instruction count for the instruction under examination and c is a constant that is the sum of the time taken to process all the other instructions. Substituting processing latencies for x gives the total program execution time.

When variable processing latencies are in place many paths may be produced that dominate the critical path at different latencies. Domination of the critical path length is shown graphically in Figure 4.11. There are three path equations $x+9$, $2x+7$, and $3x+3$. Initially at $x = 0$ the critical path length is determined by the path with the largest constant part, $x + 9$. This path dominates until the cross over point with the next dominant path is met at $x = 2$, ie. $x + 9 = 2x + 7$. From this point, the $2x + 7$ path dominates until it crosses the $3x + 3$ path at $x = 4$, ie. $2x + 7 = 3x + 3$. From this point on the $3x + 3$ path determines the critical path length. This shows that many linear equations may be required to represent the critical path length.

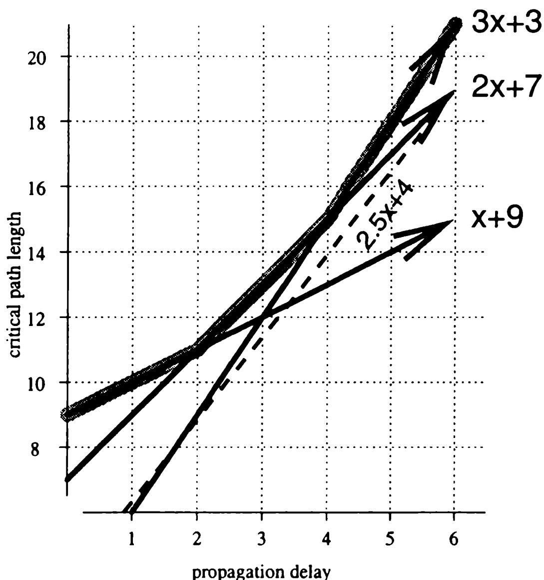


Figure 4.11: Critical path equations with variable instruction processing latencies.

Critical path equations are stored in a list that is ordered by strictly increasing variable

parts, m, and strictly decreasing constant parts, c. In this way any two consecutive equations in the list will produce a cross over point that will dominate the critical path length. Storing the path equations in this order means that for small values of x the equations at the front of the list will dominate critical path length and as the values of x increase the latter equations dominate.

In Figure 4.11 there is an extra, dashed path equation $2.5x + 4$. Although this path dominates each of the individual critical path equations at some point there is always at least one critical path equation that dominates it at any point. Paths which are dominated at all points will be pruned from the set of critical path candidates.

When a new path equation is inserted its place in the list is found by traversing the list comparing the constant parts and then variable parts of equations. Depending on the equations already in the list one of three things occurs:

1. If the new equation's constant part is equal to that of another equation the equation with the larger variable part is kept and the other is removed. If the new equation is inserted all equations after it in the list that have a smaller or equal sized variable part are removed.

2. If there are no equations with equal constant part and the new equation's variable part is equal to that of another equation the equation with the larger constant part is kept and the other is removed.

3. Otherwise, there are no equations with equal variable or constant parts so the new equation is inserted into the list such that the strictly increasing variable and strictly decreasing constant order is maintained.

When the new equation is inserted, cross over points for each pair of path equations are calculated. Determining cross over points allows equations which do not dominate the critical path to be detected because their intersections with adjacent equations will produce a sequence of cross over points that do not increase. Cross over points for two consecutive equations are determined using the equality

$$m_1x + c_1 = m_2x + c_2 \tag{4.1}$$

and solving for x

$$x = \frac{c_1 - c_2}{m_2 - m_1} \tag{4.2}$$

with the knowledge that $m_1 < m_2$ and $c_1 > c_2$.

If any cross over points are not in strictly increasing order the equation causing the problem is removed from the list. To show this, consider the example of inserting the equation $2.5x + 4$ into the equation list comprised of the critical path equations in Figure 4.11. Initially the equation list and its crossover points are

$$\begin{array}{lccccc} \text{equations:} & x+9 & \longrightarrow & 2x+7 & \longrightarrow & 3x+3 \\ \text{cross overs:} & & 2 & & 4 & \end{array}$$

After insertion of $2.5x + 4$ the equation list and cross over points are

$$\begin{array}{lccccccc} \text{equations:} & x+9 & \longrightarrow & 2x+7 & \longrightarrow & 2.5x+4 & \longrightarrow & 3x+3 \\ \text{cross overs:} & & 2 & & 6 & & 2 & \end{array}$$

The cross over points are not strictly increasing. In this case $2.5x + 4$ is dominated by $2x + 7$ for $x < 6$ and by $3x + 3$ for $x > 2$, so $2.5x + 4$ is removed from the equation list.

Pseudo code for the insertion of a path equation into the list of equations is given in Figure 4.12. In this pseudo code the equation list nodes are a structure containing the variable and constant parts of an equation, and reference fields to form a linked list. Descriptions of the procedures called within the code are given in Table 4.1.

procedure	*description*
Constant(E)	Access the constant field of equation node E.
Variable(E)	Access the variable field of equation node E.
$Next(E)$	Obtain the equation in the list after equation node E.
$Previous(E)$	Obtain the equation in the list before equation node E.
$Delete(E)$	Remove equation node E from the list.
$InsertBefore(E, F)$	Insert equation node E in before equation node F in the list.

Table 4.1: Procedure descriptions for the algorithm of Figure 4.12.

Path equation lists are stored everywhere a timestamp needs to be recorded, that is, with each memory location, register and frame in the system indicating the critical path to that point. Each instruction that is executed generates a new set of path equations by merging the path equations of its inputs and adding an appropriate latency to each equation. Merging two lists involves taking each path equation from one list in turn and inserting

{N is new equation to insert}
{H is locative pointing to head of list}

$C \leftarrow H$ {Insertion Phase}
$EquationInserted \leftarrow$ **false**
$ExitLoop \leftarrow$ **false**
while $C \neq \Lambda$ **and not** $ExitLoop$
  if Constant(N) > Constant(C) **then**
    $InsertBefore(N, C)$ {Insert new equation}
    $EquationInserted \leftarrow$ **true**
    $ExitLoop \leftarrow$ **true**
  else if Constant(N) = Constant(C) **then**
    if Variable(N) > Variable(C) **then**
      $InsertBefore(N, C)$ {Insert new equation}
      $EquationInserted \leftarrow$ **true**
      $ExitLoop \leftarrow$ **true**
  else if Variable(N) > Variable(C) **then**
    $C \leftarrow Next(C)$ {Goto next equation in list}
  else
    $ExitLoop \leftarrow$ **true** {New equation is dominated by another equation}

if $EquationInserted$ {Removal Phase}
  $C \leftarrow N$
  while $C \neq \Lambda$
    $D \leftarrow C$
    $C \leftarrow Next(C)$
    if Variable(D) < Variable(N) **then**
      $Delete(D)$ {Dominated by new equation}

  $C \leftarrow H$ {Refinement Phase}
  while $C \neq \Lambda$
    $CrossOver_{next} \leftarrow \frac{\text{Constant}(Next(C)) - \text{Constant}(C)}{\text{Variable}(C) - \text{Variable}(Next(C))}$ {from Equation 4.2}
    $CrossOver_{prev} \leftarrow \frac{\text{Constant}(C) - \text{Constant}(Previous(C))}{\text{Variable}(Previous(C)) - \text{Variable}(C)}$
    $D \leftarrow C$
    $C \leftarrow Next(C)$
    if $CrossOver_{prev} \geq CrossOver_{next}$ **then**
      $Delete(D)$ {Dominated by neighbours}

Figure 4.12: Algorithm for inserting a new path equation into the path equations list.

it into the other using the algorithm above. The instruction processing latency may be added to the variable or constant part of the equations depending on whether it is the variable instruction.

When collecting critical path information a virtual ordered simulator has an advantage a over standard simulator. In a real-time ordered simulation all paths would have to be maintained and at the end of the simulation these paths not affecting execution time could be removed. This is because it is impossible to determine during execution which paths are, and are not, going to form part of the critical path. With speculative execution taking place the correct path to the point of speculation may not be known. Speculative paths would have to be kept until they were discovered to be incorrect. Pruning paths is not always possible as speculative paths may dominate valid paths and that information cannot be discarded until the speculative paths are removed. This may waste considerable computation resources.

State history

In a speculative architecture instructions can be processed with incorrect operands. The basic virtual ordered simulation model only processes instructions with valid inputs meaning that incorrect computation is not observed. To model transient (rolled-back) computation timestamps are replaced with state history lists. For registers and memory locations the state history lists hold all the values and the real-times at which the values were placed in those locations. State history lists associated with IRB and processing resources show the times when those resources are available or busy.

When an instruction is processed the input history lists are merged 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.

A simple example showing the merging of history lists through an addition operation is given in Figure 4.13 a). The first node of the output history list is generated by merging the input history lists in real-time order. When all input history lists are valid at the same point in real-time a node representing that time is generated for the output history list. In this example at time 6 both inputs are valid so an addition is performed resulting in a new

output state at time 7. From this point the input history lists are traversed by stepping through the list with the smallest timestamp. A new node is added to the slot state list whenever the state changes in any of the input lists.

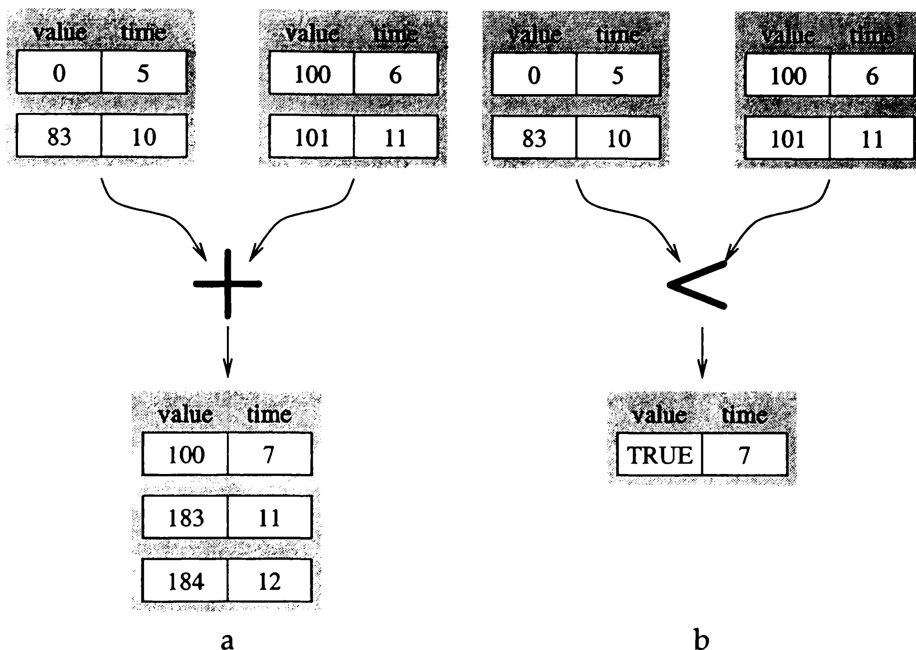


Figure 4.13: Merging of input history lists to form an output history list.

If the result of a computation does not change the state of output then the new entry does not get added to the list. For example, this could occur in a comparison operation where multiple inputs produce the same output. This is shown in Figure 4.13 b) where at time 7 the value TRUE is obtained and subsequent changes of input values produce the same answer. The non-inclusion of a state in the output list indicates the speculative execution on wrong data has produced the right answer.

The process of merging state history lists can be extended to three or more input lists to take into account the availability of processing and state saving resources. For example, the processing of an add instruction may be dependent on the validity of two operands, the availability of an IRB, and the use of a function unit. State history lists for the availability and validity of each component are merged to form the state list for the output's of the instruction.

4.4 WarpEngine simulator

This section describes the WarpEngine simulator which is based on the virtual ordered simulation methodology. The virtual ordered simulation model is applied at two levels, first at the frame level, and then at the instruction level. Frames are processed one at a time in an order defined by their mapping onto the control tree. Within a frame instructions are scheduled in a data flow order but are processed using virtual ordered techniques.

4.4.1 Components

The major components of the WarpEngine virtual order simulator are a *frame stack*, an *executing frame*, a *function unit*, the *memory system*, and a *global clock*. These components and their connections are shown in Figure 4.14.

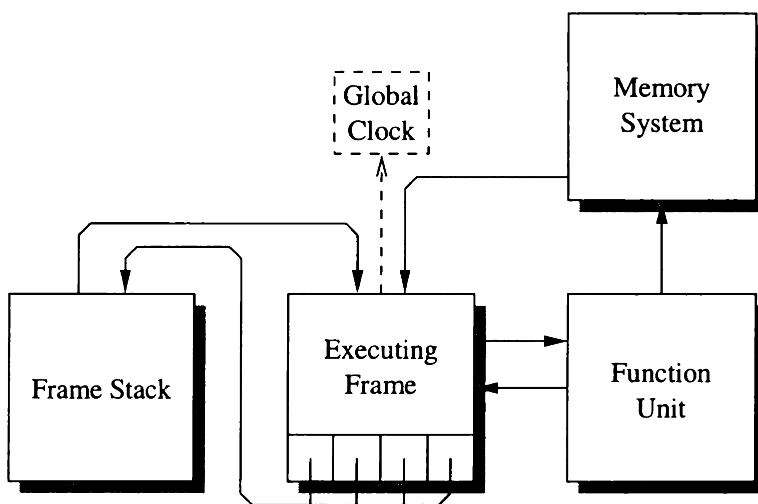


Figure 4.14: Components of the WarpEngine virtual order simulator.

The frame stack provides a mechanism for processing frames in their virtual order. When a frame is processed its dynamically generated children, nodes in the control tree, are pushed on to the frame stack in reverse order. By popping frames off the stack a preorder left-to-right traversal of the control tree is performed. This means that frames are processed in their programmed order.

At any point during simulation there is only one frame being processed and it is held in the executing frame component. The executing frame manages the interactions between

the instructions, communications to and from the function unit, and passes data to child frames.

The function unit is responsible for processing all instructions and forwarding their results to registers in the executing frame, registers in frames on the frame stack, or locations in the memory system. Each operation has a known latency. This, along with a real-time timestamp that is passed in with each instruction is used to determine the times at which results are stored.

The memory system abstracts away the intricacies of a memory controller and the time-space cache. Each location in the memory system has a timestamp associated with it so that store times can be retained. Memory latencies are handled by the function unit.

A global clock records the greatest end time of any frame that has been processed. As each frame is retired the global clock is updated such that it contains the larger of its current value and the frame's end time. Each advance of the global clock represents a time at which frames are retired. When all frames have been processed the value of the clock gives the program's execution time.

4.4.2 Operation

The control mechanism within the simulator operates at two levels. The first is at the frame level where frames are processed in the virtual sequence imposed by the program. Within each active frame in the executing frame component a second level of control takes place allowing individual instructions to be processed in a data flow order.

The frame stack is used for queuing frames in the simulation process. Initially a *starting* frame that is defined within the program executable is placed on the stack. Simulation proceeds by popping a frame off the stack and sending it to the executing frame component where it is processed. Upon completion any children that are generated are pushed onto the stack in reverse order. This sequence is illustrated in Figure 4.15. The processed frame is then removed from the system. Part of the removal process is to update the simulator's global clock. The next frame is then popped off the stack and processed. This cycle continues until the stack is empty, indicating program completion.

Within the executing frame instructions are processed based on the dependencies that

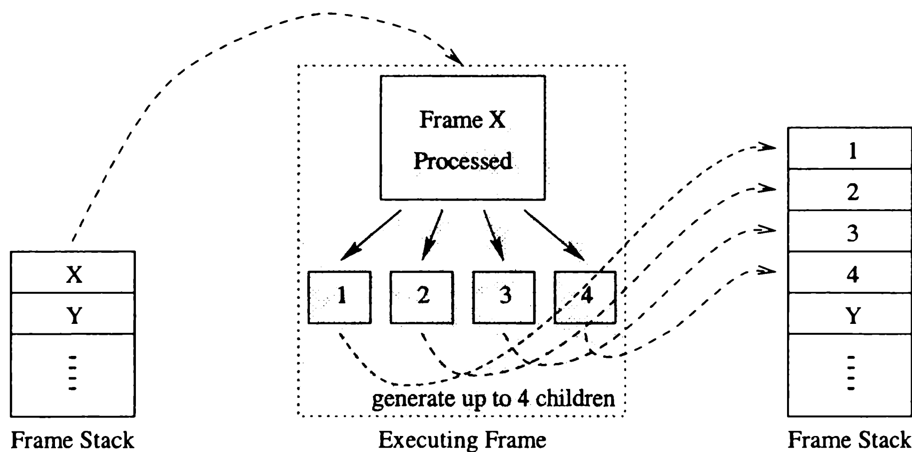


Figure 4.15: Frame stack operations during the processing of a single frame.

hold between them. Neefs et al. [1996] describe a simulation model for data flow scheduling processors. This type of model could be applied to process instructions within a frame. However, since the arrangement of instructions and registers in a frame is static a slight modification of the virtual ordered simulation technique can be used. For each slot in the frame the registers are checked to see if they contain valid values. If they are all valid then the associated instruction is processed. This slot checking process is repeated until all instructions have been processed.

All instructions except memory operations are processed within the function unit. Memory operations and their timestamp values are forwarded to the memory system. When the memory system receives a write request the value at that location and its associated timestamp are updated. For a read the memory system returns the value from the requested address along with a timestamp. This timestamp is the greater of the read request's timestamp and the time at which the value at the given address is valid. When more complex transient state timestamps are used a series of timestamp/value tuples are generated reflecting the many values that may get stored or retrived.

To model limited resources, such as frame state saving space, the timestamp array described in Section 4.3.2 is used. When a frame is processed its timestamp is updated to reflect the time at which resources are available. As each instruction is processed the frame's timestamp is consulted to ensure instructions are processed at a later time. Similar timestamps are kept, and consulted, for other processing resources that are limited. These will be discussed in more detail where they are used in the following chapters.

4.4.3 Performance

The virtual ordered WarpEngine simulator is written in C++ and is compiled with g++ version 2.8.1 at an optimisation level of 3. Table 4.2 lists counts of instructions processed and the time to execute 50×50 matrix multiplication when run on the WarpEngine simualtor, SPIM [Larus, 1993], and SimpleScalar (sim-outorder) [Burger and Austin, 1997]. Also calculated is the number of instructions processed per second (IPS). These numbers are taken from simulation runs on a Digital Celebris GL with a 200MHz Intel Pentium Pro processor with a 256 KB data cache, 60ns RAM, running version 2.0.35 of the Linux operating system.

simulator	*instructions*	*run-time*	*IPS*
SPIM	5325672	3.85	1383291
WarpEngine	3434267	12.8	267884
SimpleScalar	6214288	88.5	70217

Table 4.2: Simulator run times for 50×50 matrix multiplication.

The WarpEngine simulator processes approximately $270,000$ instructions per second whereas SPIM processes around 1.4 million and SimpleScalar $70,000$. SPIM is optimised to process instructions quickly while the WarpEngine simulator is designed to be very flexible allowing many configurations to be tested and it produces more statistical information. Unlike the out-of-order SimpleScalar simulator SPIM does not perform out-of-order execution so no dependence analysis is performed. SimpleScalar models components in more detail than the WarpEngine simulator which explains the slower run times. In this test the WarpEngine simulator was set to produce a similar level of statistical information as SimpleScalar.

These results show that the performance of the WarpEngine simulator has compares well to other real-time order simulators.

4.5 Summary

Virtual ordered simulation, a novel simulation methodology for modelling out-of-order and speculative architectures, has been introduced. It is a computationally efficient simulation model because instructions are inspected and processed once in their programmed

order. This eliminates the need to model control and data dependence analysis because all loads and stores to registers and memory locations occur in the correct order.

To model the times at which events would occur in a real system real-time timestamps are associated with each data storage location. These timestamps can be altered to provide counts of the instructions on the critical path and critical path length equations when examining the effects of varying a single instruction's processing latency. This information is not easily obtained using conventional simulation techniques. Timestamps can also be extended to accurately model the transient state of data and instructions.

The virtual ordered simulation model allows state saving and processing resource contention to be modelled. The latencies incurred through queuing, instruction retirement, and cancelback strategies are modelled by fixed, zero cycle, processing latencies. To model these types of components accurately requires the interactions between real-time ordered events to be observed, which best performed in a real-time ordered simulator.

The WarpEngine virtual ordered simulator falls somewhere between instruction set and functional simulators, and runs at compariable speed to contemporary functional simulators. It provides more information than can be gained from an instruction set simulator but the modelling of component latencies is not as accurate as that of a functional simulator. In the Chapters 7 and 8 the WarpEngine virtual ordered simulator's ability to examine components in isolation is used to provide unique insight into their effect on performance.

Chapter 5
Test programs and compilation

This chapter describes the test programs that were used when performing the experiments described in the following chapters. Each program's control and data structures, algorithm, and performance characteristics are given. The compilation procedure and code optimisations used are also described.

5.1 Test programs

The programs tested on the WarpEngine simulator span the types of operations that are performed in many programs, including matrix and array manipulation, sorting, dynamic structure operations and recursion. Table 5.1 lists the abbreviations used in this document for each algorithm in the test suite. These small loop oriented programs were chosen because no compiler exists for the WarpEngine instruction set and as such they were hand coded in WarpEngine assembly.

These programs do not represent "real world" applications, which contain complex data and control interactions, meaning that the performance results shown here are optimistic. The advantage of using these small loop based benchmarks is that it is possible to determine their algorithmic complexity and the parallelism available in them. This thesis is a preliminary study of the WarpEngines performance and the calculated theoretical limits in programs give a baseline to compare experimental results to.

Although the programs are small the relative amounts of control and data dependence they contain varies. This gives rise to differing amounts of parallelism that can be extracted at compile- and run-time. This is shown in the *work* and speculative *parallelism* measures listed in Table 5.1.

algorithm/implementation	*abbr.*	*work*	*parallelism*
matrix multiplication		$\mathcal{O}(N^3)$	
/tree backboned loops	mat		$\mathcal{O}(N^2)$
/linear backboned loops	matl		$\mathcal{O}(N^2)$
/inner loop linear	matkl		$\mathcal{O}(N^2)$
transitive closure	trans	$\mathcal{O}(N^3)$	$\mathcal{O}(N^2)$
Gauss-Jordan elimination	gj	$\mathcal{O}(N^3)$	$\mathcal{O}(\frac{N^2}{\log N})$
heapsort	heap	$\mathcal{O}(N \log N)$	$\mathcal{O}(\log N)$
quicksort		$\mathcal{O}(N \log N)$	
/left-to-right search	qs1		$\mathcal{O}(1)$
/ends-to-middle search	qs2		$\mathcal{O}(1)$
binary tree insertion		$\mathcal{O}(N \log N)$	
/naive	bin		$\mathcal{O}(N)$
/AVL	avl		?
Fibonacci numbers		$\mathcal{O}(2^N)$	
/standard	fib		$\mathcal{O}(\frac{2^N}{N})$
/data flow	fibf		$\mathcal{O}(\frac{2^N}{N})$

Table 5.1: Algorithm abbreviations and complexity.

Motivation is given for the inclusion of each of the test programs in the test suite. Also given are the performance characteristics of each algorithm. In these evaluations the parameter N is a measure of the problem size. Each program's execution time and parallelism is calculated as a function of the problem size.

These numbers are given as complexity measures using $\mathcal{O}$ notation and only consider complexity of data dependencies in the test programs. Loops are assumed to parallelise perfectly and operate in $\mathcal{O}(1)$ time, meaning that their control mechanism has no impact on performance. The complexity measures report the expected sequential execution time, parallel execution time, and the parallelism that is available when correct data values are processed. That is, the potential performance improvements gained through data value speculation are not considered. This complexity analysis corresponds to the reported execution time and IPC given by the simulator using timestamps.

5.1.1 Matrix manipulation

Three matrix manipulation algorithms, matrix multiplication, transitive closure, and Gauss-Jordan elimination have been included in the test suite. Matrices provide well structured data, and perform operations on multiple instances of data in an orderly man-

ner. In general, this allows parallelism to be easily detected at compile-time. 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 amongst the three algorithms.

Matrix multiplication is coded to multiply 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 $\mathcal{O}(N^3)$. The break down of sequential and parallel components is shown in Figure 5.1. Sequential parts are represented vertically while parallel parts are represented horizontally.

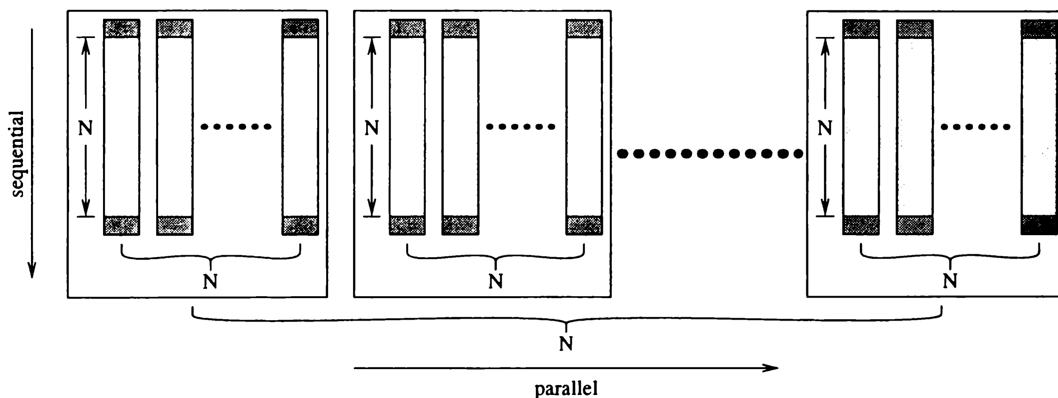


Figure 5.1: Sequential and parallel components within matrix multiplication.

The inner-most loop contains a sequential data dependence chain of length $\mathcal{O}(N)$ through the addition of values to the temporary accumulator variable. This computation could be parallelised through the use of a tree-based reduction mechanism ($\mathcal{O}(\log N)$) but this has not been implemented. The accumulator variable forms a false dependency at the next looping level that can be broken through register and memory location renaming, allowing the inner loops to be scheduled in parallel. There are no dependencies between the iterations of the outer loop, meaning that the next level of loops can also be scheduled in parallel. Therefore, the $\mathcal{O}(N)$ inner loop length constrains parallel execution time, giving $\mathcal{O}(N^2)$ parallelism. A description of *matl* and *matkl* is given in Section 7.1.4.

Transitive closure [Corman et al., 1990] is similar to matrix multiplication in that the parallelism is easy to detect. Again the sequential execution time is $\mathcal{O}(N^3)$ because there

are three levels of nested loops. The sequential and parallel components are shown in Figure 5.2.

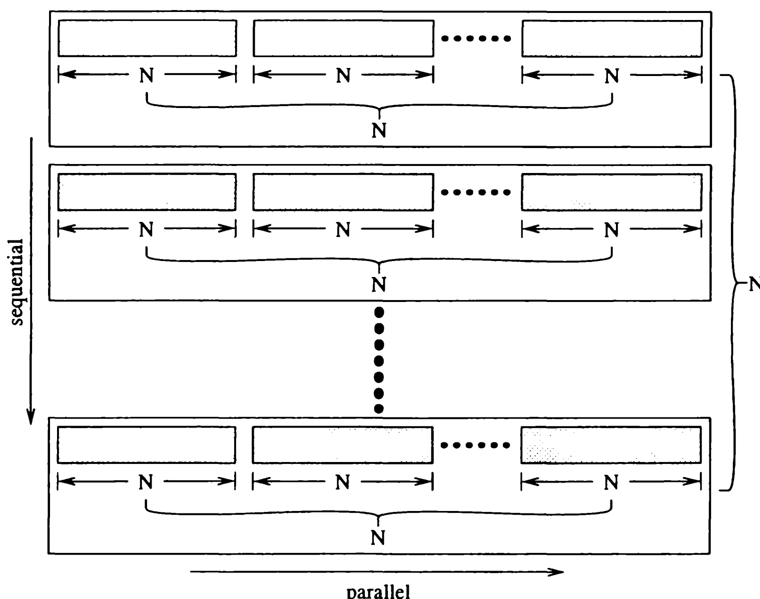


Figure 5.2: Sequential and parallel components within transitive closure.

This time there are no dependencies within the inner two loop iterations allowing them to be scheduled in parallel. There are, however, data dependencies between the iterations of the outermost loop forcing them to be scheduled in sequence. This gives $\mathcal{O}(N)$ parallel execution time and overall algorithmic parallelism is again $\mathcal{O}(N^2)$.

Gauss-Jordan elimination [Press, 1992] is a version of matrix inversion for a $N \times N$ matrix. Unlike matrix multiplication and transitive closure there are control decisions coupled to the value of data that is being manipulated. This complicates the process of determining where parallelism can be extracted.

There are three levels of nested loops giving $\mathcal{O}(N^3)$ sequential execution time. Each outer loop iteration is broken into four phases that occur in sequence. First, a search for the pivot is performed. In a sequential system this would take $\mathcal{O}(N)$ time because all values are checked in order. In a speculative system where all comparisons are performed in parallel only those comparisons that update the *max* variable affect critical path length. For a random matrix the expected number of updates to the *max* variable is

$$\mathcal{H}_n = \sum_{k=1}^{n} \frac{1}{k} = \ln(n) + 0.57721 \tag{5.1}$$

which is $\mathcal{O}(\log N)$ [Corman et al., 1990].

Next, two rows of the matrix are swapped, with each of N elements swapped in parallel. Similarly, in the next phase N elements are inverted in parallel. Finally, there are $N - 1$ rows subtracted from the pivot row. All these rows and their elements can be subtracted in parallel.

The outer loop has to be processed in sequence due to subtractions that occur giving parallel execution time of $\mathcal{O}(N \log N)$. This gives parallelism of $\mathcal{O}(\frac{N^2}{\log N})$.

5.1.2 Sorting

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

Heapsort [Press, 1992] maps a binary tree structure onto an array. Although not lending itself to parallelism the advantage of heapsort is that the worst case and average sequential execution time are both $\mathcal{O}(N \log N)$ [Lewis and Denenberg, 1991]. This is independent of the data being manipulated.

Two passes are performed, the first building the heap, and the second pulling values out in order. Each pass calls the `heapify` function N times, which in turn calls itself recursively $\log_2 N$ times. During each of these passes a root element within the array is modified N times. This forces all top level calls to `heapify` to be serialised. In a speculative system these top level calls could be scheduled in parallel but are essentially pipelined through roll-backs and re-execution caused by the changing value of the root node. This pipelining produces an $\mathcal{O}(N)$ critical path, giving $\mathcal{O}(\log N)$ parallelism.

Quicksort [Quinn, 1987] has worst case sequential execution time $\mathcal{O}(N^2)$, but the expected sequential execution time is $\mathcal{O}(N \log N)$ [Lewis and Denenberg, 1991]. This algorithm lends itself to parallel scheduling. Singh et al. [1991] describe two parallel implementations of quicksort. Although intended for mesh-based multiprocessors they could be adapted to run on the WarpEngine. The types of parallel optimisations used by Singh

et al. have not been considered here.

In quicksort there are two phases of execution. In the first a pivot element is found that splits the array. The second phase involves two recursive calls to the *sort* routine which work on the two pieces of the split array. These pieces are independent allowing the two calls to be scheduled in parallel.

The sequential and parallel components of quicksort execution are shown in Figure 5.3. On the first recursive call there is a single $\mathcal{O}(N)$ sequential path to find the pivot. This path depends on the number of updates to the pivot pointer variable, which occurs $N/2$ times on average. In the expected case this splits the next level in two, which can be scheduled in parallel. These paths contain approximately half as much work as at the previous level. This splitting and approximate halving continues for approximately $\log N$ levels, giving parallel execution time $\mathcal{O}(N)$. In the worst case the pivot will be at one or other ends of the array and the split will create an array of size $N - 1$ and one of size 0, giving $\mathcal{O}(N^2)$ parallel execution time. Therefore, the expected parallelism is $\mathcal{O}(\log N)$, and worst case parallelism is $\mathcal{O}(1)$.

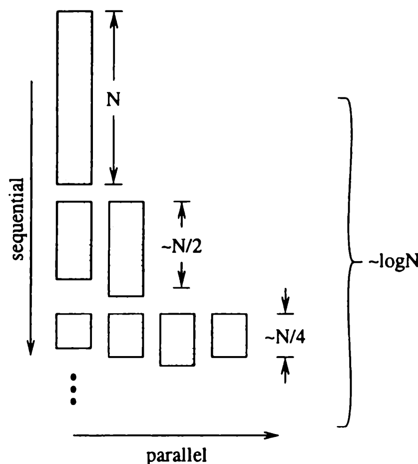


Figure 5.3: Sequential and parallel components within quicksort.

Two versions of quicksort have been tested which differ in the way they select the pivot. The first selects a pivot value and moves from left-to-right through the array swapping elements until all elements less than the pivot value are to the left of it. The second version selects a pivot value and then moves in towards the centre swapping from either end of the array until the pivot is in the correct position. These pivot selection mechanisms have no impact on parallelism estimates. They do, however, have a significant impact

on performance in a speculative architecture as shown in Chapter 7. In a speculative environment the recursive sort calls can be "pipelined" if 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. In the ends-to-middle pivot search both ends of the sub-arrays need to be known before the searches can begin. The effects that the pivot selection mechanism has on performance will be examined further in Section 7.1.4.

5.1.3 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 the sorting routines the data determines the program's control flow, though 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 is unpredictable and control is highly coupled to data. These routines have been included to show that a speculative architecture can extract parallelism from programs with complex and unpredictable control flow and dynamic data structures.

Naive binary tree insertion creates a tree with no explicit attempt made to balance it. Therefore, the tree is not guaranteed to be balanced giving worst case sequential execution time $\mathcal{O}(N^2)$, the equivalent of inserting to the tail of a linked list. However, because the values inserted are random the expected sequential execution time is $\mathcal{O}(N \log N)$ [Lewis and Denenberg, 1991].

The insertion of a key is performed in two stages. First, a search is performed to find the key's location in the tree. Next, the key is added at this location if it is not already in the tree. To gain parallelism the search phases for multiple keys can be performed in parallel. Searching only requires reads from the data structure so no data dependencies between search phases exist.

However, possible data dependencies arise between searches when a new node is added to the tree. Figure 5.4 shows the parallel insertion of two nodes, X and Y, into a binary tree. In a) the search paths diverge and the insertions of the new nodes, indicated by the shaded arcs, can be performed in parallel because there are no data dependencies. In b) because the insertions are ordered Y's search path depends on the insertion of X. This

means that Y's search and insertion has to wait for X's insertion to complete.

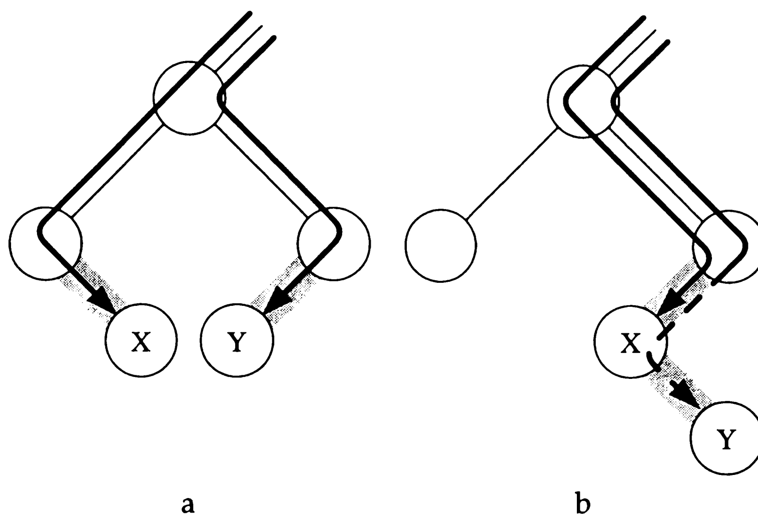


Figure 5.4: Multiple keys inserted into a binary tree a) with no dependencies, and b) with dependencies.

To allow multiple insertions to occur in parallel, a lock on tree node addresses would normally have to be used. As searches progress they acquire the lock for a node, perform a comparison and then obtain the lock for the next node. When the second lock is acquired the first can be freed allowing other insertions to use it. This procedure continues until the insertion is complete.

In the WarpEngine where insertions may occur out-of-order the timestamped memory system emulates the locking operations. It removes the need for explicit locks by detecting when speculative reads have been given incorrect values. These reads are re-satisfied and the effected search process is rolled back and re-executed. This allows the search phases to occur in parallel with insertion phases pipelined on the true data dependencies that exist, giving an expected $\mathcal{O}(\log N)$ parallel execution time. In the worst case the insertion pipeline would be N stages giving $\mathcal{O}(N)$ parallel execution time. With expected sequential execution time $\mathcal{O}(N \log N)$ and worst case $\mathcal{O}(N^2)$, the expected parallelism is $\mathcal{O}(N)$ in either case.

When manipulating dynamic structures a *free list* is kept to control memory allocation. Access to this list is sequentialised through a datum which contains a tail pointer. If the time between updates to the tail pointer is short then the processing to update the pointer can have a big impact performance, as shown in Section 7.1.4. To overcome this problem the binary tree insertion routines have been implemented with parallel free lists.

AVL binary tree insertion [Lewis and Denenberg, 1991] maintains a balanced tree. This gives amortised insertion time $\mathcal{O}(\log N)$, and sequential execution time $\mathcal{O}(N \log N)$. 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 internal nodes get rotated to maintain a balanced tree.

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 rotates have been processed. This locking of the root node would serialise computation removing most opportunities for parallelism.

Many of the rotations that occur will not affect the root node, 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 hard.

5.1.4 Recursion

Initially the recursive Fibonacci number generation algorithm was included in the test suite to show that recursive routines can be programmed on the WarpEngine. It is not a computationally efficient routine but it maps nicely onto the WarpEngine's tree-structured control mechanism.

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. This produces 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 additions of the values returned from the recursive calls introduces a sequential dependency path of length N. This gives a parallel execution time of $\mathcal{O}(N)$ and parallelism of $\mathcal{O}(\frac{2^N}{N})$.

Each call of `fib` produces a result which is stored on a stack. The second invocation of `fib` at each level uses stack locations that have been used by the first invocation. This means that locations get written to in the second invocation erasing the old values stored

there. To execute `fib` calls in parallel would require that this return stack be duplicated for each call.

This is one area where the features of a timestamped memory system show out. Timestamping stack locations renames the location in a temporal manner. This has the effect of producing separate stacks for each `fib` call. The ordering mechanisms in the memory system simulate duplication of memory locations through temporal memory location renaming.

A purely data flow version has also been coded exploiting features of the WarpEngine to minimise or eliminate memory usage. Data can be passed directly between code blocks without using a memory based stack that is normally required for recursion.

5.2 Control tree optimisations

The preceding complexity analysis assumed perfect loop control allowing all loop interactions to be scheduled in parallel at the same time. In code the problem is that a straight mapping of sequential control onto the WarpEngine's control tree produces a linear backbone. In cases where the work in an iteration is small a linear control backbone may not schedule instructions quickly enough, impeding performance.

At run-time the bounds of `for` loops are known when the loop is entered. Iteration count knowledge can be used to generate a flatter, wider control tree with the loop counter incorporated into the control mechanism. This is shown in Figure 5.5 for a `for` loop with 10 iterations. Each control node in the tree splits the number of remaining loop iterations in half. If there are only one or two iterations left then they are invoked directly. All the loop iterations are invoked in less steps down the control tree than with a linear backbone.

This works well for loops with independent iterations. A problem arises if there is a data dependency, excluding the loop counter variable, across the iterations. In these situations early loop iterations need to be started as early as possible. Figure 5.6 shows a more effective `for` loop control tree for 10 iterations. At control node a one iteration is invoked along with another control node b. At b the remaining loop iterations are split in half, 4 to the left and 5 to the right in this case. Control nodes c and f are invoked in parallel

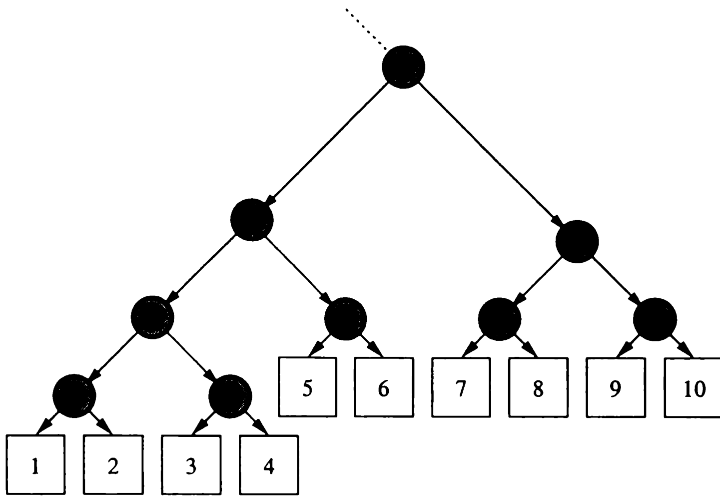


Figure 5.5: Control trees for a `for` loop with a tree-based control backbone.

performing the same actions as *a*. This splitting and firing of loop iterations continues to control node *e*. After invoking iteration 4 only one iteration remains in that sub-tree so it gets invoked as well. A similar sub-tree is produced for the remaining half of the loop iterations starting at control node *f*.

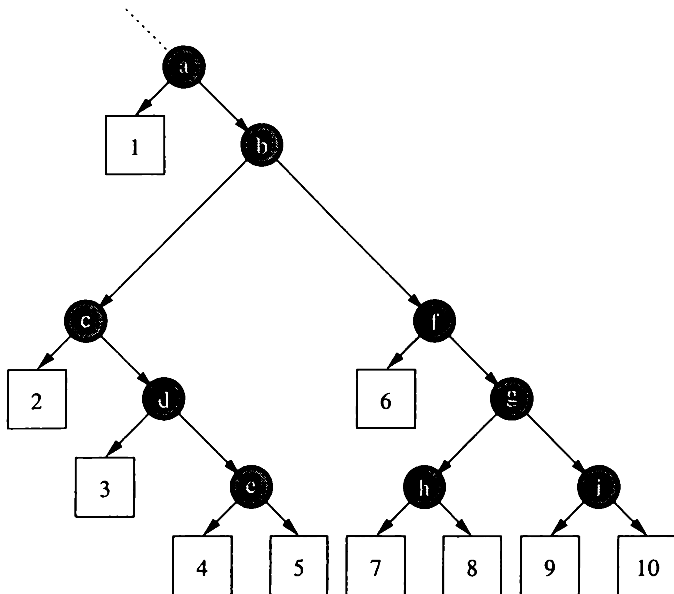


Figure 5.6: Early start tree-based control backbone for a `for` loop.

Tree-based backbones can also be used in loops where the bounds are not known upon entry. Figure 5.7 shows the control tree generated for a `while` loop. Here the right hand edge acts as a linear backbone, but the number of loop iterations fired by this backbone doubles at each level down the tree. As with the `for` loop tree early loop iterations are

scheduled before later ones.

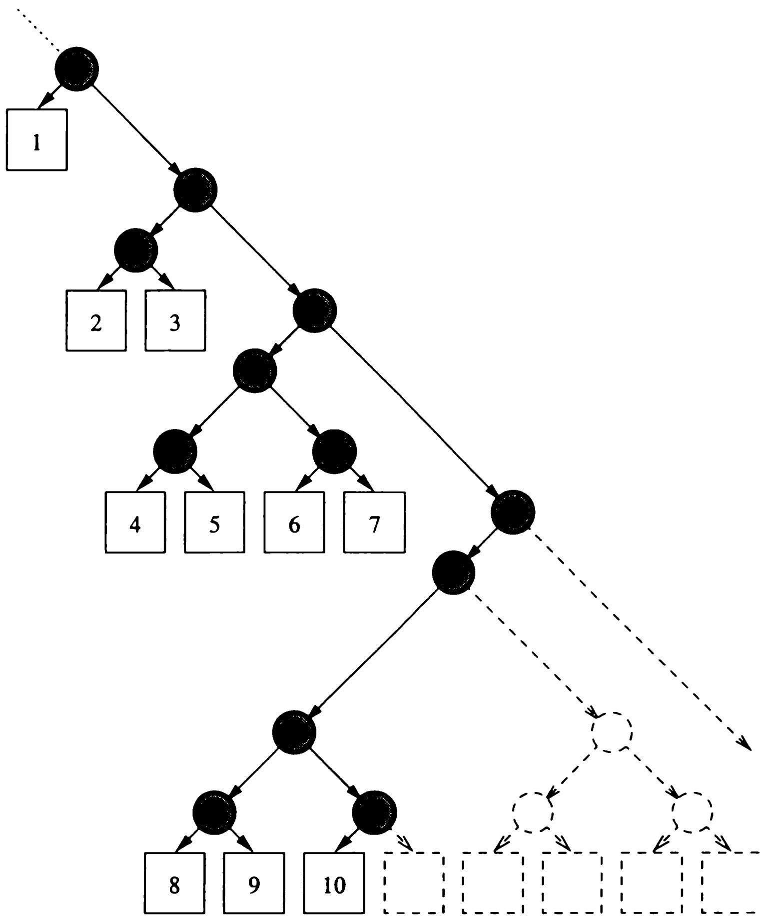


Figure 5.7: Tree-based control backbone for a `while` loop.

The intention of the tree-based loop construct is to try to get code performing real work, rather than loop control, in parallel and as quickly as possible. The advantages of tree-based loop backbones are that iterations are invoked earlier and the control mechanism is distributed. This allows any extra loop control computation to be hidden in parallelism.

These looping structures generate code to be executed quickly but they also consume processing and state saving resources at the same rate. One drawback of this is that these resources may be wasted if loop iterations are rolled back or squashed. This can occur in situations, such as while loops, where the loop control code is fast relative to the code in the loop iterations and it runs ahead invoking iterations that should never be executed. Care must be taken to ensure that speculation on control flow does not waste too many resources. This is a trade-off between the aggressiveness of speculation and the resources available.

5.3 Compilation

To date no compiler exists for the WarpEngine instruction set and producing one was beyond the scope of this thesis. Due to the WarpEngine's unique block based instruction set and tree based control mechanism it would require significant effort to modify an existing compiler for a contemporary architecture to produce efficient machine executables for the WarpEngine. The well established techniques used in VLIW compilers to group instructions [Biglari-Abhari et al., 1998] could be modified to generate WarpEngine code blocks. The process of finding independent instructions to form a VLIW instruction would need to be replaced by a mechanism to find tightly coupled instructions to form WarpEngine code blocks.

External to the fixed sized instruction grouping process techniques would need to be developed to generate an efficient control structure. To handle the tree based control mechanism special code libraries that generate tree based loop and branch control structures could be employed. Knowing when to use structures of different shapes to obtain best performance in a certain processing environment is an area of interesting research. This is tightly coupled to load balancing and resource scheduling and again investigation into this area is beyond the scope of this thesis.

The effort of building a compiler is also compounded by the infancy of the instruction set. The instruction set was in the early stages of development and had not been fully tested, meaning that it was open to change.

With all these considerations in mind it was deemed impractical at this early stage of the WarpEngine's development to build a compiler to test its features. Instead the suite of test programs were compiled manually.

Each program is first written in C [Kernighan and Ritchie, 1988] and compiled to a known architecture. This code is then tested for algorithmic correctness. The C code provides a template when coding the algorithms in WarpEngine assembly code. In the process of converting from C to WarpEngine assembly every attempt was made to emulate the functionality of a compiler. The assembly code is written such that it can be closely mapped back to the original C code. Assembly code correctness is determined by comparing its output, when run on the simulator, to the output of the C equivalent.

The C code for each of the test programs is given in Appendix B. Corresponding WarpEngine assembly code can be found in Littin [1999b]. The assembly code is converted into a machine executable using the WarpEngine Assembler [Littin, 1998], which produces an ELF-like [ELF, 1993] machine code executable. This machine executable is then run on the simulator.

5.3.1 Techniques used

The algorithms have been coded assuming a single thread of control and a simple shared memory model. This means that all operations are ordered, including those to memory. (No locking on memory locations or other synchronising constructs are used.) Ordering memory operations ensures that the memory contents will be consistent with a sequential implementation of the algorithm.

Like any speculative architecture the WarpEngine can use branch prediction to improve performance by predicting branch outcomes based on previous execution of those instructions. In the investigations performed here no dynamic hardware branch prediction mechanisms have been incorporated into the simulation model and no attempt to predict branch outcomes through code annotations has been performed at compile-time. Control independence information is used to determine points of convergence within code so that blocks can be speculatively scheduled for execution. The only blocks that are statically scheduled to execute in parallel are those that are guaranteed to unconditionally execute. All other blocks wait until their enabling condition becomes true.

When compiling code the philosophy used is to fan out blocks as much as possible using the control tree, and to maximise the number of instructions per block. No loop unrolling is performed, only consecutive instructions are placed in a single block. To parallelise loops the tree-based control backbones of Section 5.2 are implemented.

In all the test programs `for` loops use a tree-backbone similar to the one illustrated in Figure 5.6. The WarpEngine allows up to four children to be created by a code block. A control tree is produced that can schedule up to three loop iterations at a time along with split node that schedules the remaining iterations in two equal sized subtrees.

Matrix multiplication has been coded with different combinations of `for` loop control

backbones. `mat` has all tree backbones, `matl` has all linear backbones, and `matkl` has a linear back-boned inner loop with the outer two tree backbones. This is to show effect that these combinations of loop control structures have on performance and is examined in Section 7.1.4.

After experimentation it was discovered that no additional performance was gained using tree-back-boned `while` loops in `qs2` and `avl`. The data dependencies between loop iterations neutralised the parallel scheduling of the tree-back-boned loop control. In these programs `while` loop control is implemented with a simple linear backbone.

Although compiled by hand the generation of fixed-sized instruction blocks can be performed by a compiler. Fixed sized blocks are necessary for VLIW machines. Compiler techniques for generating fixed size groups of instructions have been proposed and implemented [Biglari-Abhari et al., 1998], and could be adapted to meet the requirements of the WarpEngine.

5.4 Instruction set evaluation

The initial impetus behind building the virtual order simulator was to test the adequacy of an instruction set developed for the WarpEngine paradigm. The main objective was to determine if the proposed instruction set was sufficient to allow sequential programs to be mapped onto the tree-based control structure, and whether the control structure provided an adequate mechanism for scheduling 100s of instructions in parallel.

The WarpEngine's instruction set characteristics had been defined as a block based architecture mapped onto a tree control structure with data flow control within blocks. Arithmetic, logic and floating point operations are based on instructions from conventional architectures. Special instructions are in place to control the dynamic creation of nodes in the control mechanism, and to pass data between blocks. The functional capabilities of these extra instructions had been defined but their implementation had not.

A simulator was built to model the WarpEngine instruction set [Cleary, 1995a]. Many of the test programs were coded for this instruction set, and executed on the simulator to test the ability of WarpEngine's instruction set to handle programming concepts such as conditional control flow, loops, recursion, arrays and pointers. The coding of these

programs provides evidence that conventional sequential programs can be written using the WarpEngine instruction set and that a sequential control path can easily be mapped onto a tree-based control mechanism.

Coding these programs also exposed some of the limitations of this version of the instruction set. As a result a second version of the instruction set was developed [Cleary, 1995b]. The second version of the instruction set is used throughout this thesis.

5.5 Summary

A suite of test programs has been coded for the WarpEngine instruction set. These programs are used throughout the experimental sections of this thesis. Complexity analysis has been performed to determine the parallelism available in terms of problem size when these programs are run on a speculative architecture. These complexity measures are used in the following chapters to show that the WarpEngine is capable of extracting the parallelism that is available.

The process of coding these programs has shown that loops and branching structures in sequential code can be mapped onto a tree-based control mechanism. The coding process has also tested the capabilities of a proposed instruction set and highlighted weaknesses within it. Information produced from simulation runs and the development of the virtual ordered simulator itself has aided in the development of an improved version of the instruction set.

Chapter 6
Critical instructions

The creation of a simulator and a suite of test programs has shown that the WarpEngine's instruction set can be used to code programs written in high level sequential languages. As a first investigation into the performance of the WarpEngine it is interesting to see what instructions are used during execution. In a parallel system it is also interesting to see how the mix of instructions that determines execution time (those on the program's critical path) compare to those of the program as a whole. The instructions that occur frequently on the critical path contribute the most to execution time meaning that their processing latencies should be as small as possible.

A *path* within a program is the sequence of instructions from the start to end of a task. The time taken to execute that task is given by the count of instructions and their processing latencies on a path. A *critical path* is one whose length is greater than that of any other path [Ralston and Reilly, 1993]. The length of the critical path determines the program's execution time, with maximum performance gained by minimising the length of this path.

In a sequential architecture all instructions are on the critical path because they are processed in order. Minimising the processing times of instructions is one way to shorten the critical path length. Another is to execute instructions in parallel, which produces many paths and reduces the number of instructions on the critical path. Instruction processing optimisations can be performed to further shorten the critical path length.

This chapter compares instruction mixes of programs as a whole to that of their critical path when processed on a system with unbounded resources. Also investigated is the affect of altering an individual instruction's processing latency.

6.1 Instruction usage

One way to improve performance is by optimising individual instructions in an attempt to reduce their processing times. Performance will benefit most by improving those instructions that occur most often on the program's critical path. Knowing the instruction mix ratio on a program's critical path is essential in this process. However, if the proportional mix of instructions on the critical path is similar to that of the program as a whole then this computation would not be required. This section investigates whether the mix of instructions on a program's critical path matches that of the program as a whole.

6.1.1 Method

The distribution of instructions for the program as a whole is calculated by keeping counts of the number of times each instruction is executed. Collecting instruction counts on the critical path is an involved process. Instruction counts are kept for each path in the program using the mechanism described in Section 4.3.3. At the end of execution only the critical path remains, with instruction counts obtained from it.

As the critical path generally contains fewer instructions than the program as a whole a direct comparison of the instruction counts is not meaningful. From a set of instruction counts a normalised distribution is generated by dividing each count by the total number of instructions.

For ease of visualisation instructions have been grouped as shown in Table 6.1. Logic, floating point and integer arithmetic have been grouped into three categories Logic, Float, and Integer, respectively. The remaining categories represent individual instructions constituting the fundamental control and data flow operations performed in the WarpEngine.

6.1.2 Results

Figures 6.1 and 6.2 graph the dynamic instruction mix proportions of the program as a whole and of the critical path for each of the test programs. From initial observations it is clear that for each problem the instruction mixes on the critical path vary from the

key	*description*
Float	floating point operations.
Logic	bitwise logical operations.
Integer	arithmetic operations.
Compare	the *cmp* instruction which includes all comparison tests.
Move	inter-frame data movement operations, the *mv* instruction.
Load	memory read operations, the *ma* instruction.
Store	memory write operations, the *st* instruction.
Child	frame creation operations, the *child* instruction.

Table 6.1: Description of keys in instruction mix graphs

program as a whole, as well as across problems. Although not shown, the instruction mix ratios hold for varying problem sizes and data sets for each problem.

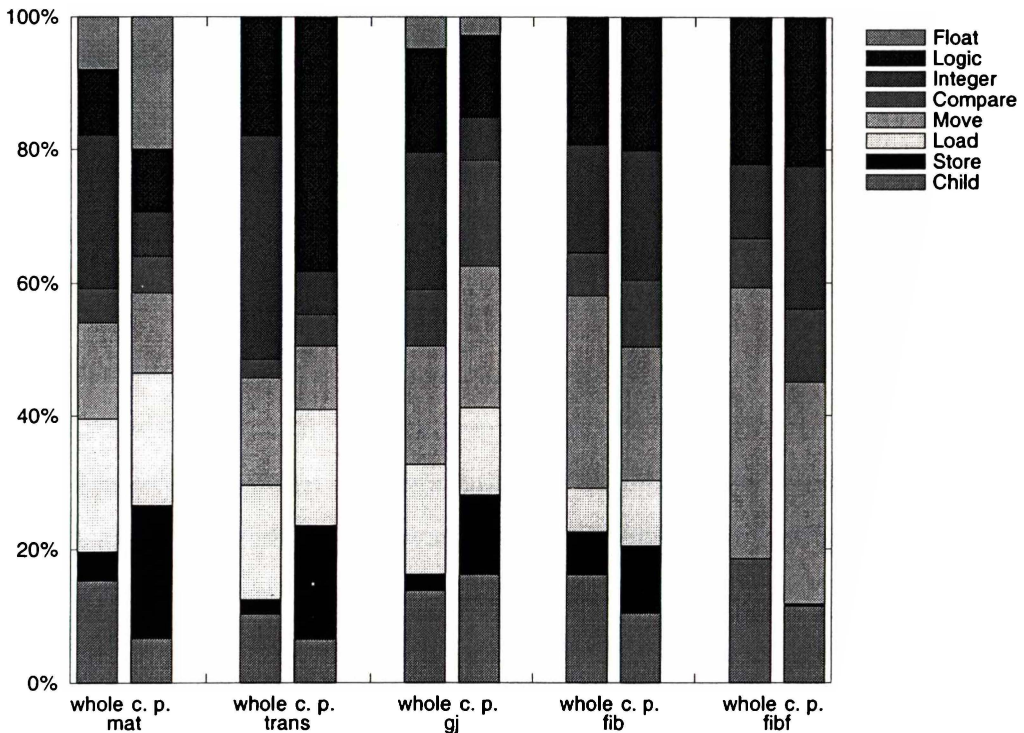


Figure 6.1: Instruction mixes for *mat*(50), *trans*(50), *gj*(30), *fib*(30), and *fibf*(30).

In general the instruction mix ratios on the critical path differs from that of the program as a whole. This indicates that decisions about instruction optimisations should be based on critical path ratios rather than overall program instruction mix. In most cases the ratio of memory operations on the critical path is greater than for the program as a whole, indicating an increased dependence on the memory system. This is expected as program critical paths are highly dependent on the data flow within the program when there are

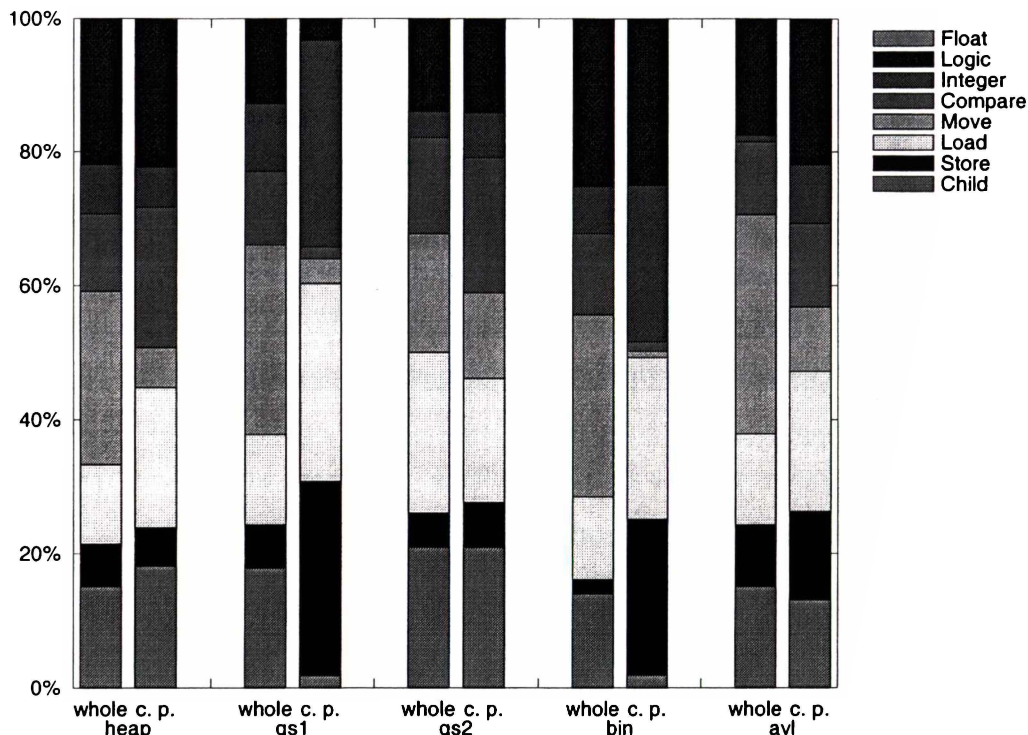


Figure 6.2: Instruction mixes for *heap*(2000), *qs1*(2000), *qs2*(2000), *bin*(2000), and *avl*(2000).

arbitrary processing resources available.

In all but *qs2* there is an increase in the proportion of loads and stores on the critical path. From the graphs, ignoring *fibf* because it is designed not to access memory, the average proportion of memory operations of the programs as a whole is 20.0% ± 4.9%, with a minimum of 12.9% and maximum of 29.1%. This is comparable to the 20% seen in other architectures [Hennessy and Patterson, 1996a]. The ratio of memory operations on the critical path is significantly higher in most cases, rising to an average of 34.6% ± 12.4%, with a minimum of 19.8% and maximum of 58.5%. Some of the test programs show a near doubling in the proportion of memory operations on the critical path, *mat* from 24.2% to 39.8%, *trans* from 19.3% to 34.5%, *qs1* from 20.0% to 58.5%, and *bin* from 14.5% to 47.5%.

Along with the increase in memory operations comes a general decrease in the proportion of child operations. The exceptions to this are *gj* and *heap*. There is also a decrease in the proportion of inter-block move operations, with the exception of *gj*. This shows that there is a trade-off between memory operations, and block creation and data movement operations.

The instruction mix ratios of other instructions tend to stay more constant or produce no discernible patterns in the comparisons. The instruction ratios that do vary seem to be dependent on the characteristics of the program. This provides more evidence for the case that critical path instruction mix ratios should be used when optimising instruction processing latencies.

6.2 Critical paths

Instruction mixes on critical paths show a high proportion of memory operations. This means that program execution time is very sensitive to memory access latencies. In the WarpEngine data movement instructions are used to communicate data between parent and child blocks directly whereas memory operations communicate data over an arbitrary number of instructions. The memory system of the WarpEngine is likely to be distributed meaning that memory access latencies could be high compared to other instruction latencies. This section examines how the variation in average memory access latency affects program execution time.

6.2.1 Method

Within the simulation model instruction processing times are modelled with fixed latencies. One method of investigating the effect that the processing latency of a single instruction has on program execution time is to run many simulations with varying instruction latencies. Another approach is to produce a mathematical equation representing program execution time as a function of the variable instruction's latency. For sequential execution the equation will be a linear function of that instruction's latency as there is only one instruction path. In a parallel architecture it is possible that there may be multiple critical paths of equal length comprised of different instruction mixes. Varying a single instruction's latency can produce a non linear change in execution time. To capture this information a group of piece-wise linear equations is used.

The technique described in Section 4.3.3 is used to find critical path equations for each of the test programs. The varying processing latency instruction examined here is the memory read operation. This is the operation that is most likely to be affected when

scaling up the number of processing resources in the WarpEngine. With larger and more complex memory hierarchies the time taken for values to filter through to RAM and other distributed caches is likely to increase. The time taken for the value of a store to filter through the memory system is felt by dependent reads, not the write that performed it.

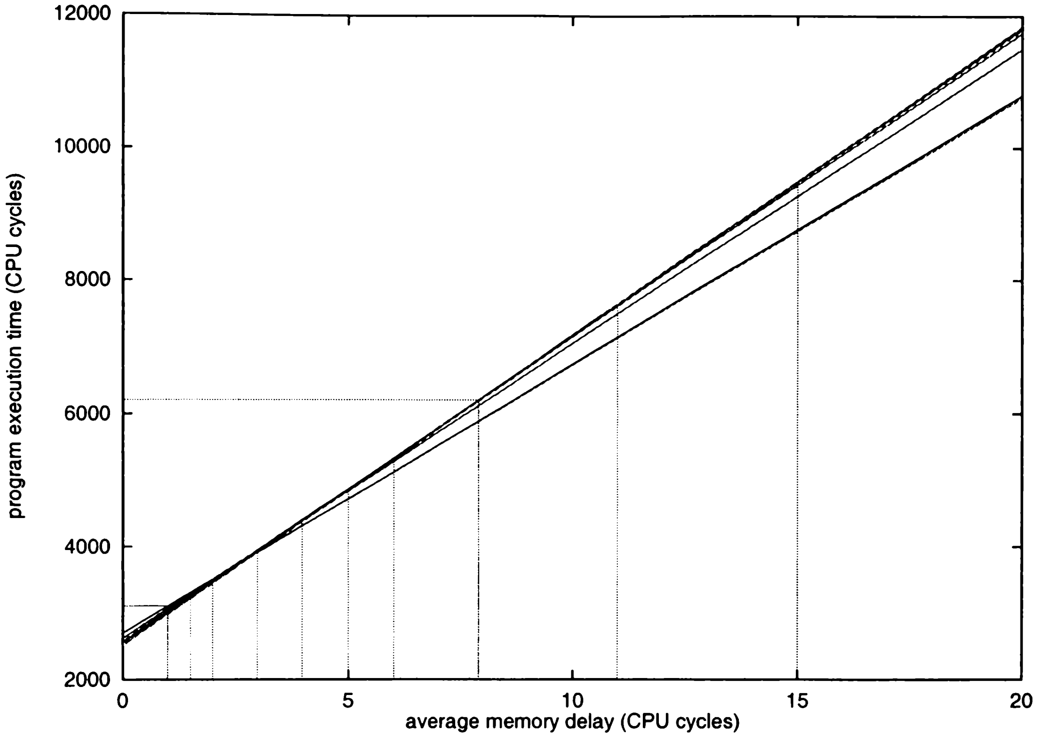
6.2.2 Results

Figures 6.3 to 6.5, and C.1 to C.6, show the critical path equations when average memory read latency is varied for each of the test programs at a given problem size. All other instructions have been modelled with a single cycle processing latency. *fibf* has not been plotted because varying read latency has no affect on program execution time.

Each inclined line in the plots represents one of the critical path equations with the overall critical path length at any read latency given by the maximum of any of the equations at that point. The range over which each equation dominates the critical path is also given in the associated tables. Vertical lines on the graphs indicate the points at which successive path equations intersect. There are two horizontal lines, the lower of which indicates the program's execution time when the average memory read latency is one cycle. The upper line represents the point at which the program execution time is double that of the lower. Its intersection with the dominant critical path equation, indicated with another vertical line, gives the read processing latency that will double execution time.

In all but *fib* the program execution doubling time occurs before the read latency reaches 8 cycles. From the instruction mix graphs the average number of reads on the critical path is $19.4\% \pm 5.8\%$. This produces a normalised critical path equation of $0.194x + 0.806$ and execution time of 1 at $x = 1$. Doubling the execution time and solving for x, $0.194x + 0.806 = 2$ gives $x = 6.15$. This value for average read latency that doubles execution time collaborates with the results shown in the graphs.

Another observation is that the most activity, in terms of critical path cross over points, occurs in the range of 1 to 20 cycles. A small increase in read time within this range can have a greater than linear impact on execution time. Initial estimates of the average read processing latency were near the middle of this range meaning that varying this latency slightly will have a significant impact on performance.

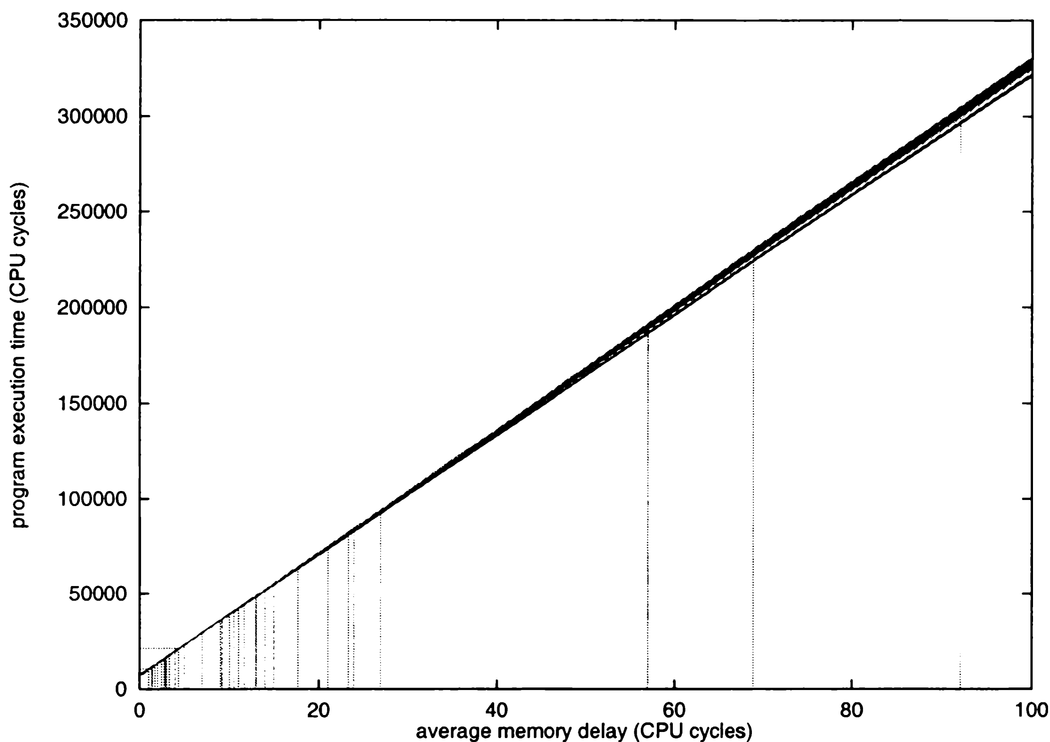


equation	*range*
$403x + 2701$	$[0, 1.5)$
$405x + 2698$	$[1.5, 2)$
$444x + 2620$	$[2, 3)$
$458x + 2578$	$[3, 4)$
$461x + 2566$	$[4, 5)$
$462x + 2561$	$[5, 6)$
$463x + 2555$	$[6, 11)$
$464x + 2544$	$[11, 15)$
$466x + 2514$	$[15, \infty)$

Figure 6.3: Critical path equations for *gj* (30).

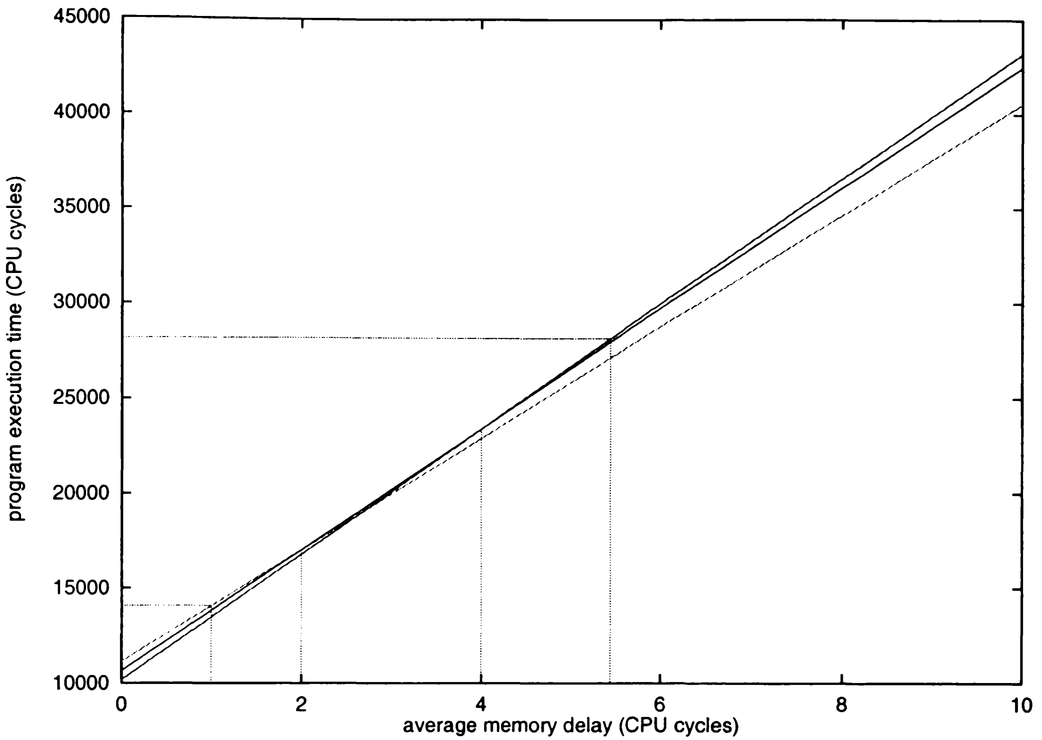
In many cases the effect of memory latencies on parallel program execution time is not linear. There are multiple critical path equations for five of the nine test programs. In these programs, as the read processing latency increases paths with more memory operations on them become more prominent. In these graphs it appears that the critical path equations are fairly similar with only a small difference in the gradients of the path equations. The ratios of the highest to lowest gradients in critical path equations for each program are shown in Table 6.2. The largest variations in gradients are 15.6% for *gj*, and 12.3% for *avl*. All others have gradient variations of less than 4%.

It is notable that the programs with the greatest variation in critical path equation gra-



equation	*range*	*equation*	*range*
$3134x + 7520$	$[0, 1.4)$	$3220x + 7130$	$[10.5, 11)$
$3139x + 7513$	$[1.4, 1.7)$	$3223x + 7097$	$[11, 11.6667)$
$3149x + 7496$	$[1.7, 2)$	$3226x + 7062$	$[11.6667, 13)$
$3151x + 7492$	$[2, 2.5)$	$3230x + 7010$	$[13, 14)$
$3173x + 7437$	$[2.5, 2.81818)$	$3231x + 6996$	$[14, 15)$
$3184x + 7406$	$[2.81818, 3)$	$3232x + 6981$	$[15, 17.6667)$
$3185x + 7403$	$[3, 3.33333)$	$3235x + 6928$	$[17.6667, 21)$
$3188x + 7393$	$[3.33333, 4)$	$3236x + 6907$	$[21, 23.3333)$
$3191x + 7381$	$[4, 5)$	$3239x + 6837$	$[23.3333, 24)$
$3193x + 7371$	$[5, 7)$	$3240x + 6813$	$[24, 27)$
$3201x + 7315$	$[7, 9)$	$3242x + 6759$	$[27, 57)$
$3203x + 7297$	$[9, 9.2)$	$3243x + 6702$	$[57, 69)$
$3208x + 7251$	$[9.2, 10)$	$3244x + 6633$	$[69, 92)$
$3218x + 7151$	$[10, 10.5)$	$3245x + 6541$	$[92, \infty)$

Figure 6.4: Critical path equations for *qs1* (2000).



equation	*range*
$2941x + 11141$	$[0, 2)$
$3185x + 10653$	$[2, 4)$
$3304x + 10177$	$[4, \infty)$

Figure 6.5: Critical path equations for *avl* (2000).

dients are those with the more complex control structures. The *gj* and *avl* routines are significantly more complex than the others. This indicates that more complex real-world applications are likely to produce many critical path equations.

6.3 Summary

The instruction mix on the critical path differs from that of the program as a whole. In general, there is an increase in the proportion of memory operations on the critical path. With the increase in memory operations there is a decrease in the number of inter-block register move operations. This is a trade-off in the proportion of short-distance and long-distance data value communication operations.

Memory operations make up an average of 35% of operations on the critical path. With

problem	*size*	*gradient* *low*	*high*	*ratio*
mat	50	51	52	1.020
trans	50	51	51	1.000
gj	30	403	466	1.156
heap	2000	14733	14733	1.000
qs1	2000	3134	3245	1.035
qs2	2000	35927	35927	1.000
bin	2000	2022	2060	1.019
avl	2000	2941	3304	1.123
fib	30	29	29	1.000

Table 6.2: Ratio of the highest to lowest critical path equation gradients.

this high proportion any latencies in processing memory operations can have a significant impact in execution time. Program execution time can be doubled with only a small increase in the average number of cycles to process read operations.

As the average read processing latency increases, paths with higher numbers of reads in them dominate the critical path. This is especially noticeable in programs with complex control and data structures. This produces a greater than linear increase in execution time with respect to increasing read processing latency.

The results from this chapter show that it is crucial to optimise the memory system to avoid delays in storing and returning data.

Chapter 7
Performance

An architecture that speculates on data values and uses control independence information can extract large amounts of parallelism. The WarpEngine has this potential.

Two areas that are critical to parallel execution are accessing the memory system and scheduling instructions to be processed concurrently. The previous chapter has shown that a high proportion of instructions on the critical path are memory accesses. Restricting these operations in any way, in turn increasing their average processing latency, could have a significant impact on performance.

In the WarpEngine frames are used to hold instruction blocks when they are processed. Limiting the availability of these state saving resources will also have an impact on performance.

This chapter shows the amount of parallelism, in terms of instructions processed per clock cycle, that a WarpEngine with unlimited resources can obtain. These results are used as a baseline for comparison in later evaluation where constraints are placed on the order that memory operations occur and limits are placed on the amount of frame resources available.

Performance metric

To evaluate architectural characteristics a performance metric is required. One measure is to take the total execution time in seconds, but this depends on accurate modelling of system components. Since precise modelling of components is not done a more useful measure is the number of CPU cycles required the execute the program. This is independent of the technology that could be used to implement the WarpEngine.

A program's parallel execution time is given by the time at which the last instruction is retired. This instruction will be on the program's critical path meaning that the number of clock cycles to process the critical path will give program execution time.

Program performance is measured by the average number of instruction processed per cycle, or its IPC. This value is calculated by dividing the number of instructions processed by the length of the critical path.

$$\text{IPC} = \frac{\text{instructions executed}}{\text{length of critical path}}$$

7.1 Unlimited resources

This section investigates the amount of parallelism obtained from the test programs when executed on the WarpEngine with unbounded processing and state saving resources available. Memory bandwidth is also assumed to be unlimited, meaning that instruction loading and data storage and retrieval is not hindered in any way. With these fully relaxed resource constraints the intention is to demonstrate that the tree-based control mechanism and timestamped memory system are adequate to support large scale speculative execution.

Each test program is processed on the simulator with various sized data sets to examine the effect that increasing problem size has on parallelism. The sorting and dynamic structure programs base their control decisions on the values of data being manipulated. For these programs a variety of data sets are used at each of the data set sizes. Simulation results are then plotted to see if the curves produced match the complexity measures calculated in Chapter 5.

In parallelism studies [Lam and Wilson, 1992; Wall, 1991; Postiff et al., 1998] all instruction processing latencies are set to one cycle. This gives a measure of the IPC, but it is not representative of the instruction latencies found in modern architectures. Here test programs are run on the simulator with both single cycle latencies and more typical instruction processing latencies. The instruction latencies used in this section are displayed in Table 7.1. These values are estimates of processing delays obtained from informal discussions [Glew, 1997]. Other combinations of instruction latencies have been examined with the results obtained similar to those shown later in this chapter. This investigation

is interested in relative performance changes rather than absolute values meaning that imprecise instruction latencies are tolerable.

instruction	*cycles*
CHILD	15
ST	2
MV MA	8
CMP	2
ADD SUB	2
MUL	5
DIV	50
SPLIT	2
AND OR XOR	2
ADDF SUBF	4
MULF	6
DIVF F2I	50
I2F	10

Table 7.1: Processing latencies assigned to instructions to give a typical distribution.

7.1.1 Results

Figures 7.1 to 7.8, and C.7 to C.8, graph the IPC obtained against problem size when running the test programs on a WarpEngine with unlimited resources. The sorting and binary tree insertion routines make control decisions based on the data being processed. These routines were run with several different data sets. The graphs of their performance with respect to problem size produce an IPC band. In the binary tree insertion routines, larger problem sizes are supersets of smaller problems. Related problems have been graphed with connecting lines. Also plotted on *gj*, *heap*, *qs1* and *qs2* are complexity measure curves that were calculated in Chapter 5.

In each graph the upper line, group of lines, or cluster of points represent the performance gained when single cycle instruction processing latencies are used and the lower ones when typical instruction processing latencies are used. For each program the shapes of the IPC curves are similar for the single cycle and typical cycle instruction processing latencies. With the programs that make dynamic control decisions the same peaks and troughs are present in both curves. This is easily seen in the *gj* plot (Figure 7.2) and when comparing individual data sets in *qs1*, one of which has been plotted with connecting lines (Figure 7.4).

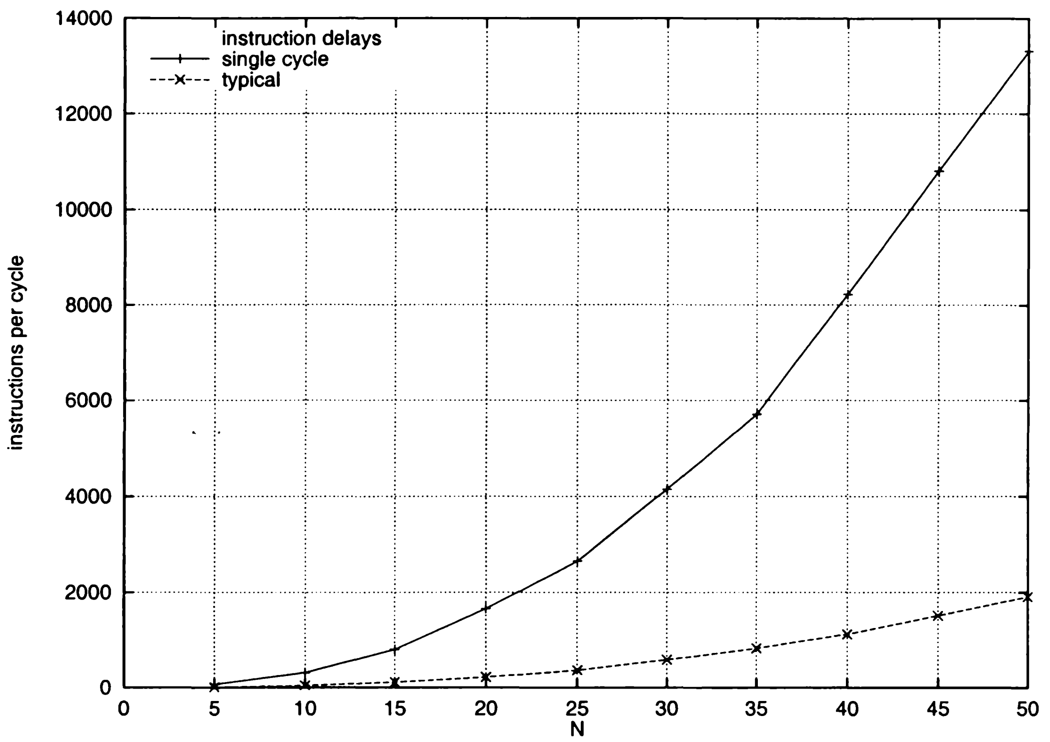


Figure 7.1: IPC vs problem size for *mat*.

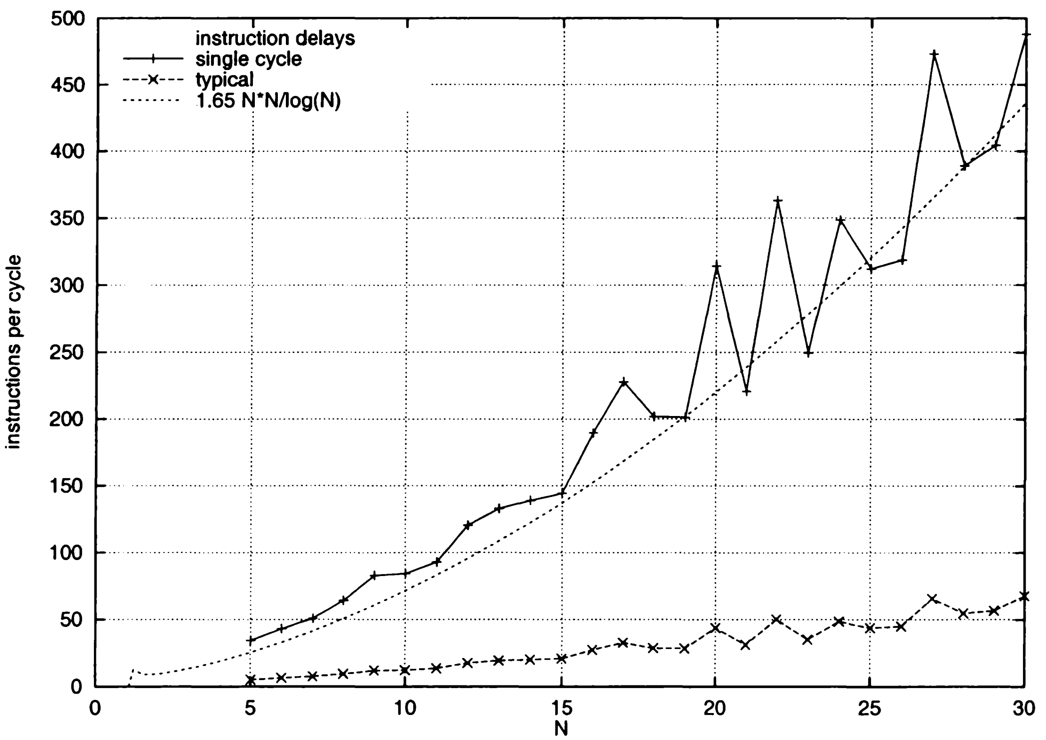


Figure 7.2: IPC vs problem size for *gj*.

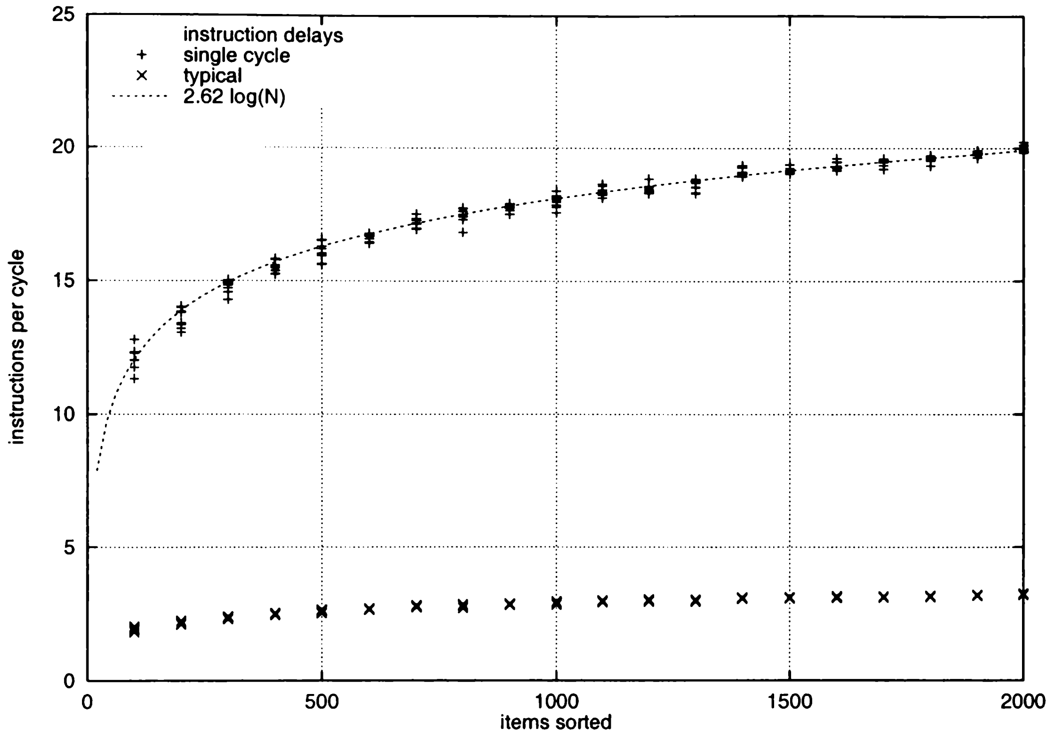


Figure 7.3: IPC vs problem size for *heap*.

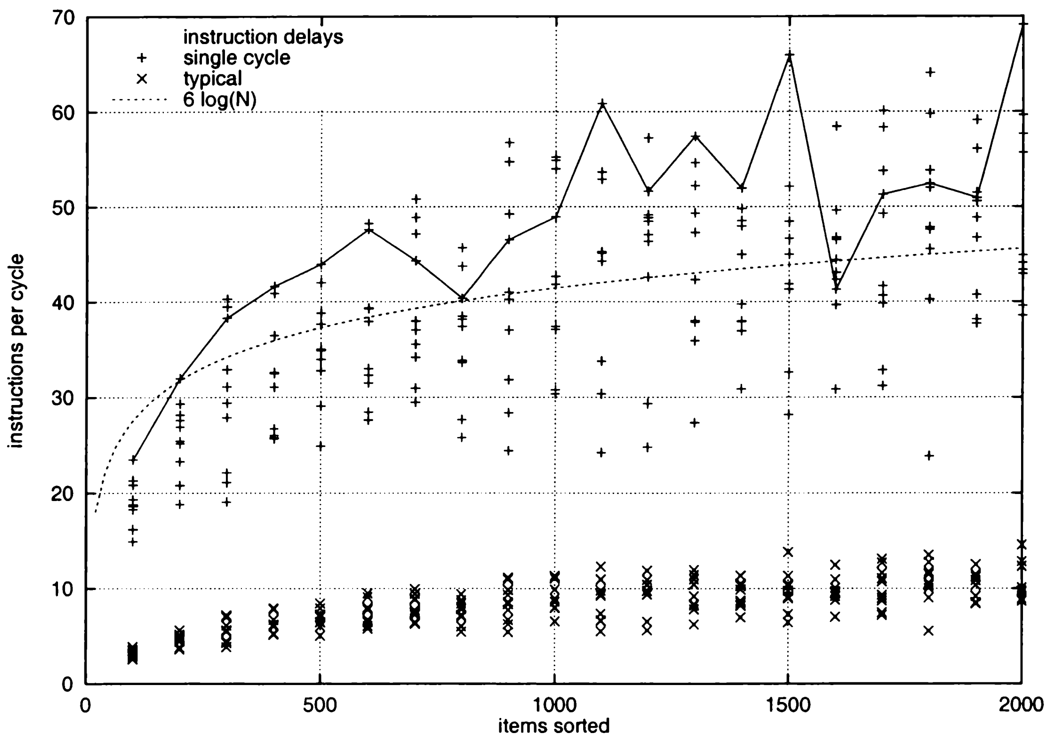


Figure 7.4: IPC vs problem size for *qs1*.

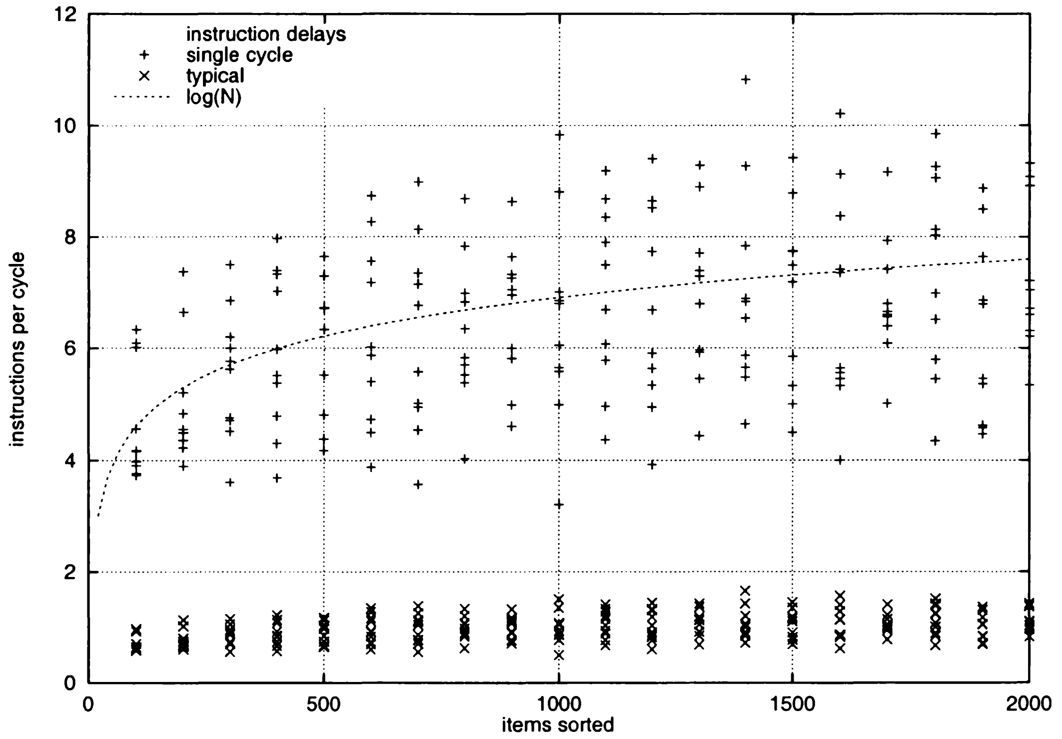


Figure 7.5: IPC vs problem size for *qs2*.

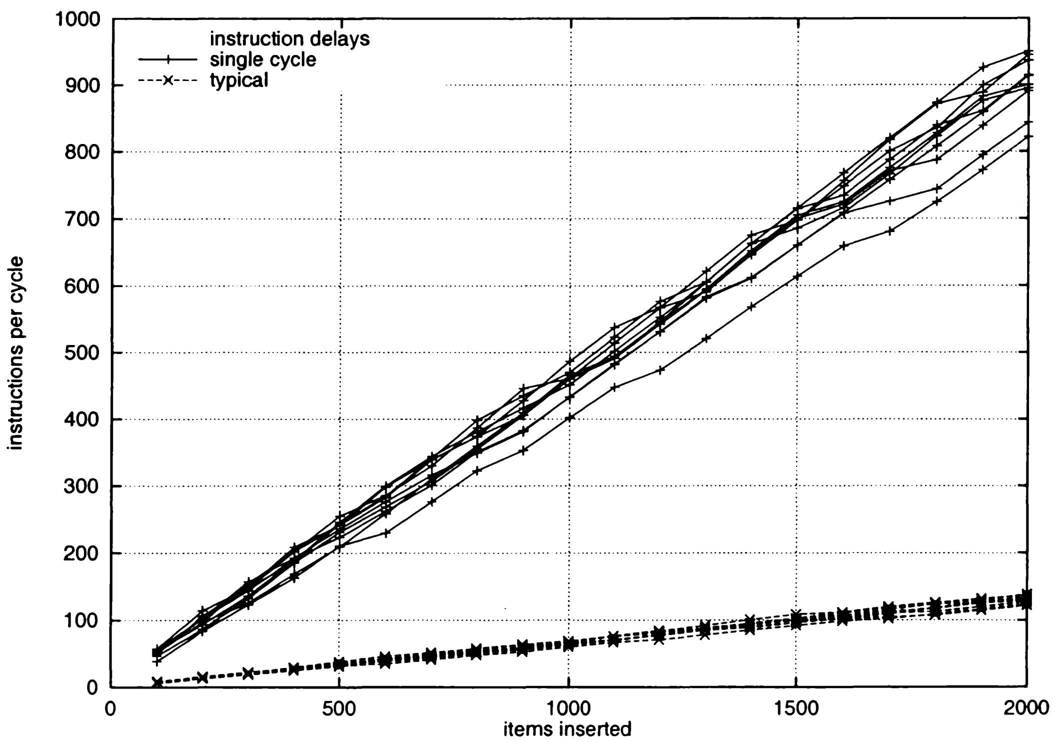


Figure 7.6: IPC vs problem size for *bin*.

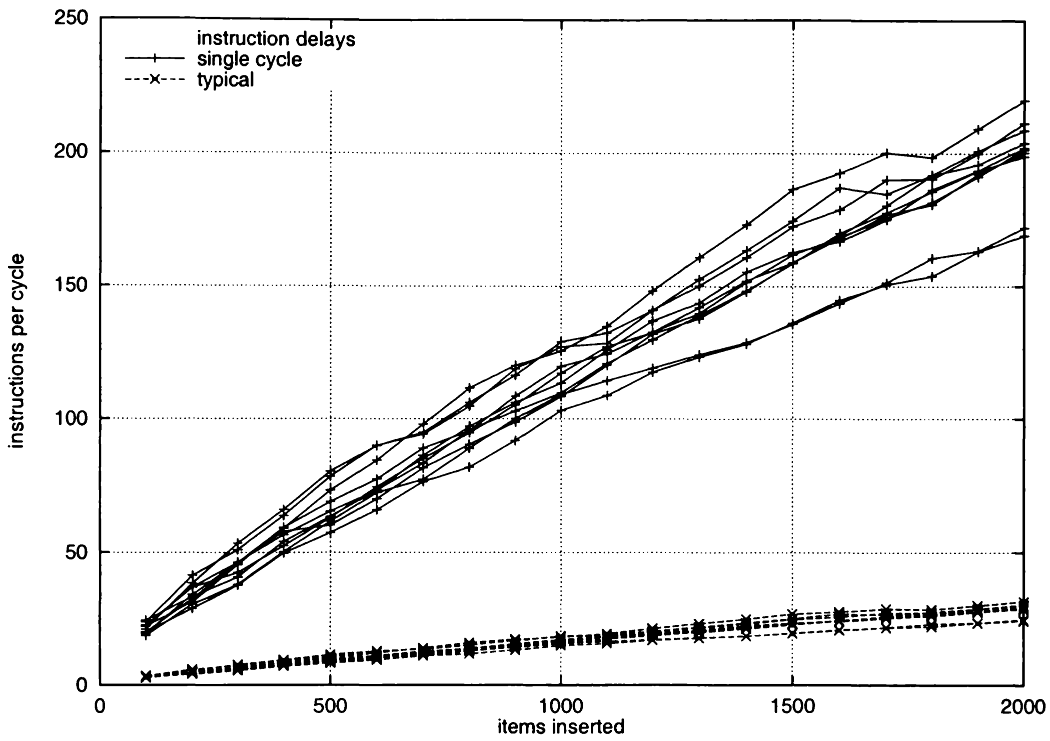


Figure 7.7: IPC vs problem size for *avl*.

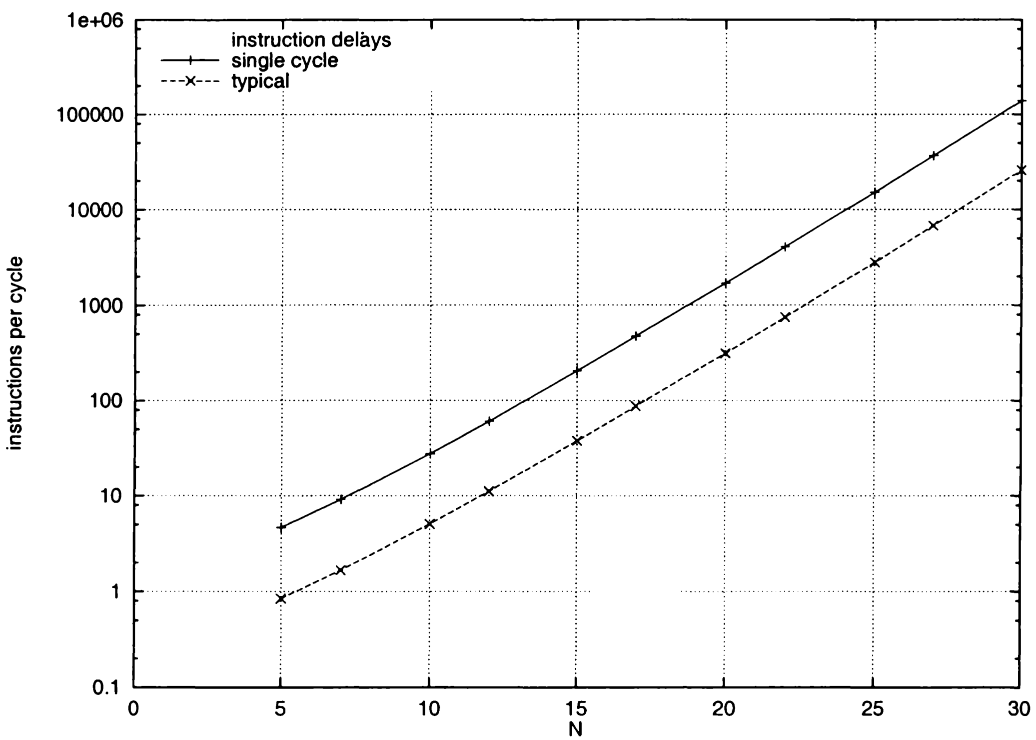


Figure 7.8: IPC vs problem size for *fib*.

Qualitatively, instruction processing latency has no effect on parallelism obtained, but quantitatively the magnitude of the IPC obtained varies. This means that either single or typical instruction latencies can be used when comparing the relative changes in performance. All analysis from this point will only consider the case where single cycle instruction processing latencies are used.

When examining parallelism with typical instruction processing latencies in all but *heap* and *qs2* the IPC obtained exceeds 10 for larger problem sizes. In *mat*, *trans*, *fib*, and *fibf* where parallelism is detectable at compile-time performance exceeds 100 IPC for larger problem sizes. Even in the programs that make dynamic control decisions, where parallelism is hard to detect, a significant amount of parallelism is obtained.

The level of IPC seen here may seem high and it might be thought that this is due to the low level of complexity in the test programs. The results seen here reflect those observed in other ILP investigations. Similar levels of parallelism, up to 4000 IPC, have been shown to exist in the more complex SPEC95 suite of benchmark applications [Postiff et al., 1998].

7.1.2 Complexity analysis

To determine if the WarpEngine can extract the parallelism available in a program the shape of the IPC curve can be compared to its complexity measure calculated in Chapter 5. *mat* (Figure 7.1) and *trans* (Figure C.7) show polynomial increase in IPC with respect to problem size. This matches the $\mathcal{O}(N^2)$ level of parallelism calculated. The levels of IPC shown in these routines would be expected from any parallel architecture with unbounded resource limits.

With *gj* (Figure 7.2) there are rises and falls in IPC between successive problem sizes with a general greater than linear but less that polynomial trend. The equation $\frac{1.65N^2}{\log_{10} N}$ gives an approximation to the plotted data points, indicating that the predicted $\mathcal{O}(\frac{N^2}{\log N})$ parallelism is obtained.

For the three sorting routines (Figures 7.3 to 7.5) an $\mathcal{O}(\log N)$ equation has been plotted to approximately bisect the IPC data points. In *heap* the data points fit closely to the equation showing that the parallelism obtained is as predicted. With *qs1* the equation does not fit well at smaller problem sizes (< 250), but at larger problem sizes it does indicating the

predicted parallelism is obtained.

The IPC band for *bin* (Figure 7.6) can be approximated by the equation $0.45N$ indicating that calculated $\mathcal{O}(N)$ parallelism is obtained. *avl* (Figure 7.7) shows a slightly less than linear parallelism trend. Although its parallel complexity was not calculated this result shows that significant amounts of parallelism can be obtained.

fib (Figure 7.8) and *fibf* (Figure C.8) produce a slightly greater than linear line when plotted on a logarithmic scale indicating an exponential trend. Again this corresponds with the $\mathcal{O}(2^N)$ calculated parallelism complexity.

These graphs show that the WarpEngine has the potential to extract the parallelism that is available, even in programs where complex control decisions hide the parallelism. They also show that the WarpEngine ISA does not limit parallelism.

7.1.3 Data sets

In the sorting and binary tree insertion routines the IPC obtained for a given problem size varies across data sets. The variation in IPC can come from either a variation in the number of instructions processed, the length of the critical path, or a combination of these. Figures 7.9 and 7.10 plot the instruction counts and critical path lengths for *qs1*, respectively. For a given data set size there are variations in both instruction counts and critical path length. The combination of variations produces a greater variation in IPC.

Similar fluctuations in instruction counts and critical path lengths are evident in *qs2*, but not in *heap*. In *heap* there are only slight variations in IPC, instruction counts, and critical path lengths. On each iteration of the sort routine the same number of comparisons are performed independent of the data processed. Any data dependent operations that cause variations are small, for example, the swapping of two elements. This leads to consistent processing times across data sets.

The binary tree routines also show variations in IPC for a given problem size. With *bin* there are variations in both instruction counts and critical path length. This can be seen in Figures 7.11 and 7.12, respectively. The variation in instruction counts can be attributed to the shapes of the trees produced. The shape of the tree affects the length of paths taken when finding insertion points.

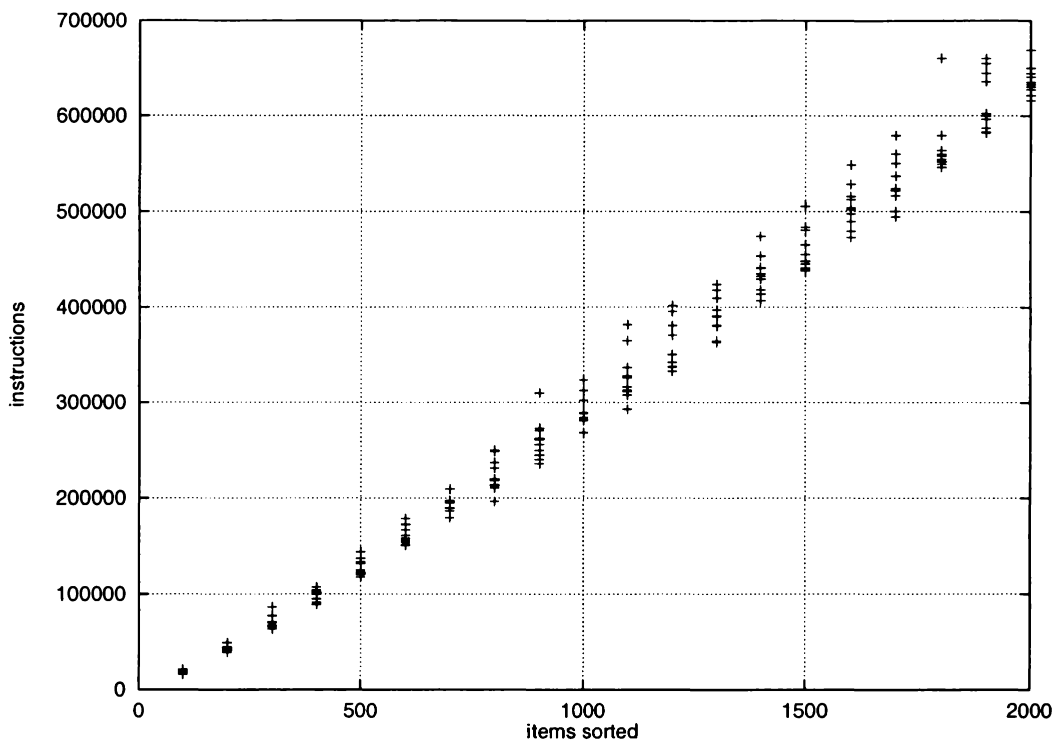


Figure 7.9: Instruction count vs problem size for *qs1*.

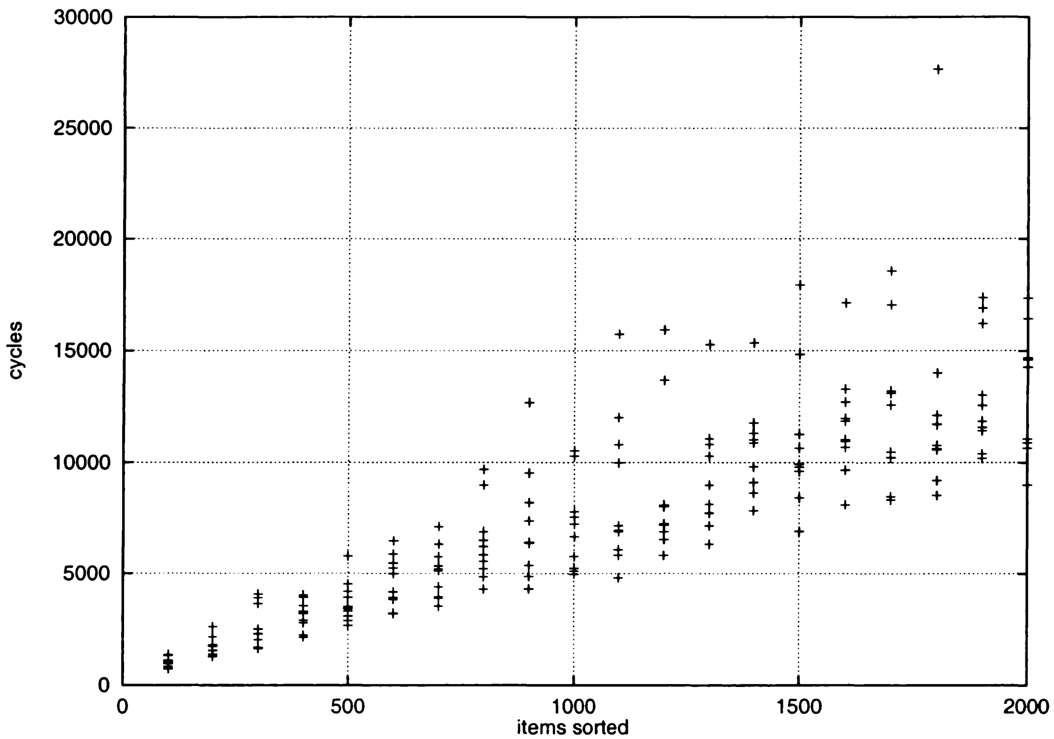


Figure 7.10: Critical path length vs problem size for *qs1*.

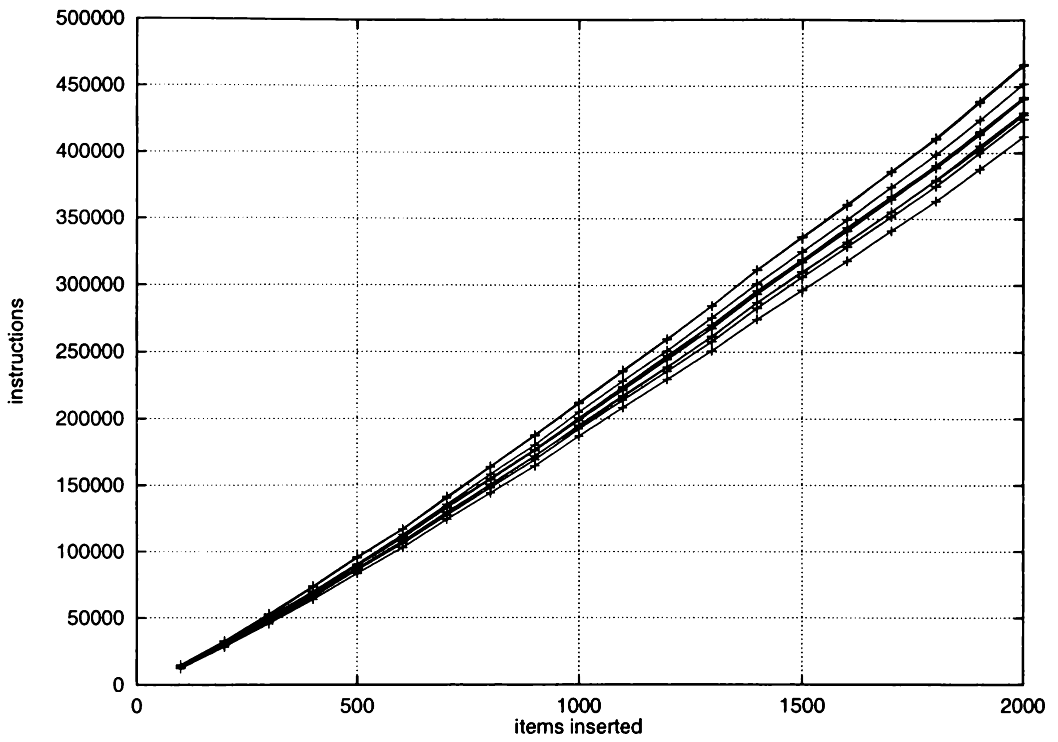


Figure 7.11: Instruction count vs problem size for *bin*.

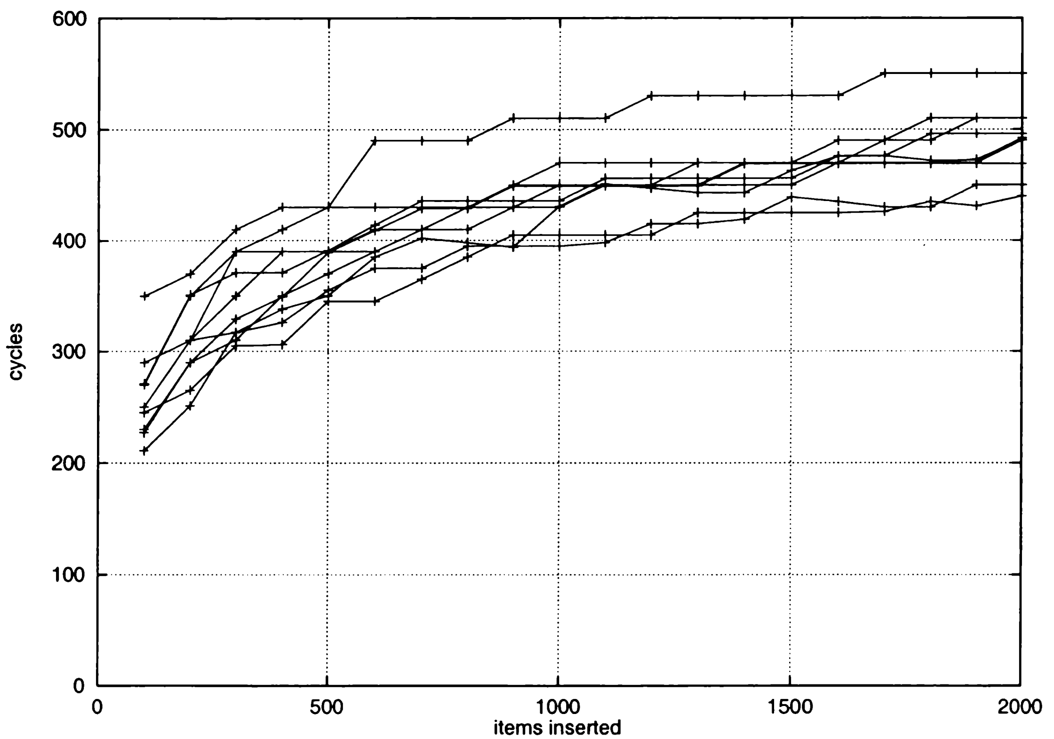


Figure 7.12: Critical path length vs problem size for *bin*.

In *avl* the instruction counts are fairly constant and only the critical path lengths show variation. Figure 7.13 shows *avl's* instruction counts. With *avl* the trees produced are balanced, so insertion of the ith item from any data set requires the traversal of a path which varies by at most one step. The major action affecting the instruction count is the number of rotations incurred to re-balance the tree. On average, across data sets of the same size, the total number of rotations will be the same. This gives rise to near constant instruction counts, but the combination of rotations affects critical path length.

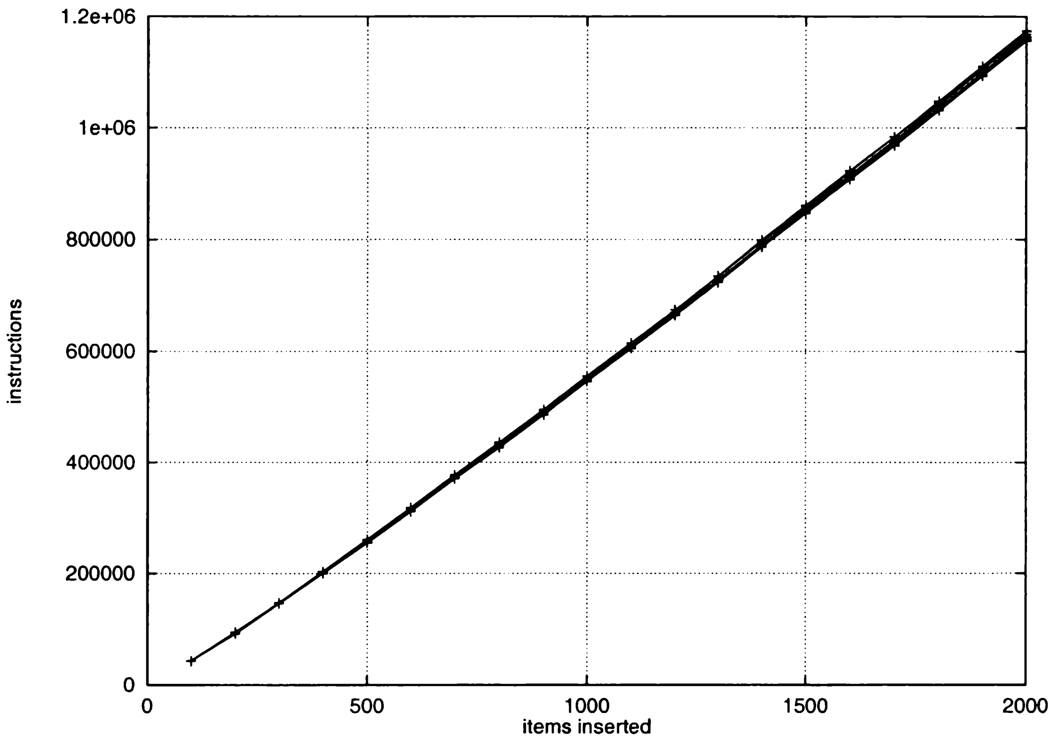


Figure 7.13: Instruction count vs problem size for *avl*.

7.1.4 Algorithmic decisions

For a given problem, such as sorting, the choice of algorithm can have a significant impact on performance. From figures 7.3, 7.4 and 7.5 it can be seen that *qs1* allows more parallelism to be obtained than either *heap* or *qs2*. The same data is sorted with algorithms of the same sequential complexity yet the performance obtained varies.

Figure 7.14 graphs the critical path lengths against problem size for all three of the sorting algorithms tested. For each problem size on average *qs1* has the shortest critical path followed by *heap*. In this case critical path rankings map to IPC rankings, but in general

this may not be true.

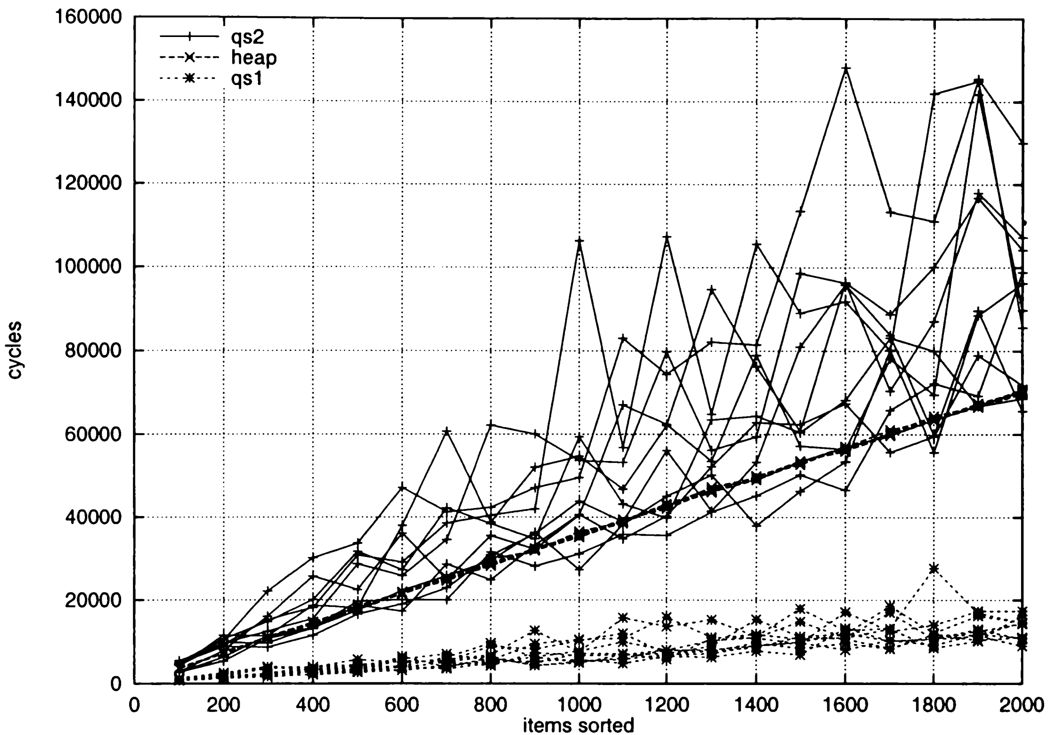


Figure 7.14: Critical path length vs problem size for the three sorting routines.

It is not entirely clear why *qs1* performs significantly better than *qs2*. Although, it is worth noting that the sub-sorts after the partition step can start earlier in *qs1*. The pivot selection process can progress in *qs1* without knowing the upper bound of the array because it steps through the array from the start. In *qs2*, where the pivot is selected by alternating from either end of the array, the array size has to be known before this process can start. Sub-sorts cannot effectively start until the entire partition step is complete.

Another area where algorithmic decisions can impact on performance is in loop control. The run-time control mechanism in the WarpEngine allows both linear and tree loop control backbones to be generated. There is a trade-off in the extra computation required to generate tree loop control over linear loop control with the extra parallelism that can be obtained by scheduling loop iterations earlier using tree-based loop control.

To test the effects that loop control mechanisms have on performance matrix multiplication has been coded with combinations of linear and tree-based loop control structures. The combinations examined here are all three nested loops *tree*-based, all three loops *linear*, and only the *inner linear*. Instruction counts and critical path length, and IPC for these

loop combinations within matrix multiplication are shown in Figures 7.15 to 7.17.

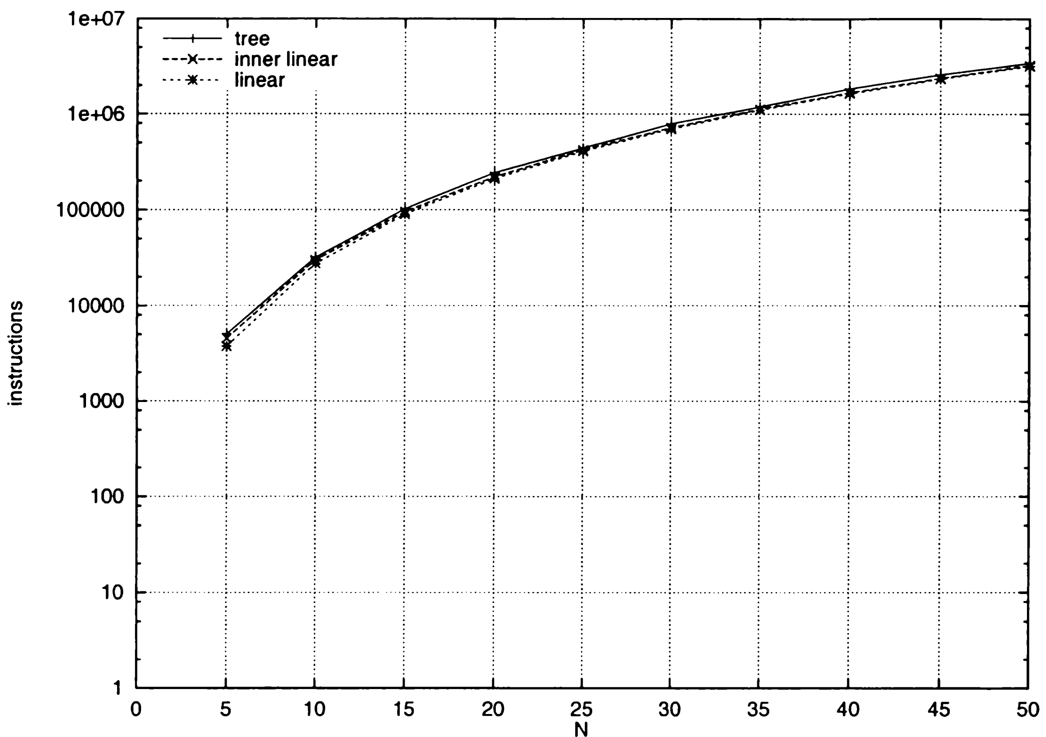


Figure 7.15: Instruction counts for *mat* with different loop constructs.

At any loop size the version of matrix multiplication with all linear loops has the lowest number of instructions processed. This is as expected, because linear loop control requires less computation. However, the version with all tree loops has the shortest critical path lengths for larger problem sizes. For small numbers of iterations a linear loop performs better than a tree loop. This can be seen for a problem size of 5 where the critical path length is shortest for the version that has a combination linear and tree-based of loops. In this case the overhead of the tree loop mechanism on the inner loop degrades performance. For larger loop sizes the critical path for tree loops is limited by the data dependencies between inner loop iterations. At larger sizes linear loops do not allow iterations to be fired early enough to gain the maximum amount of parallelism.

Another place where programming structures can have a significant impact on performance is the maintenance of free lists during dynamic memory allocation. Updating the free list can form a sequential path through the program, which can have an effect on parallelism. This was noticeable with *bin* which was initially programmed with a single free list. The critical path length for updating the free list is long compared to that of

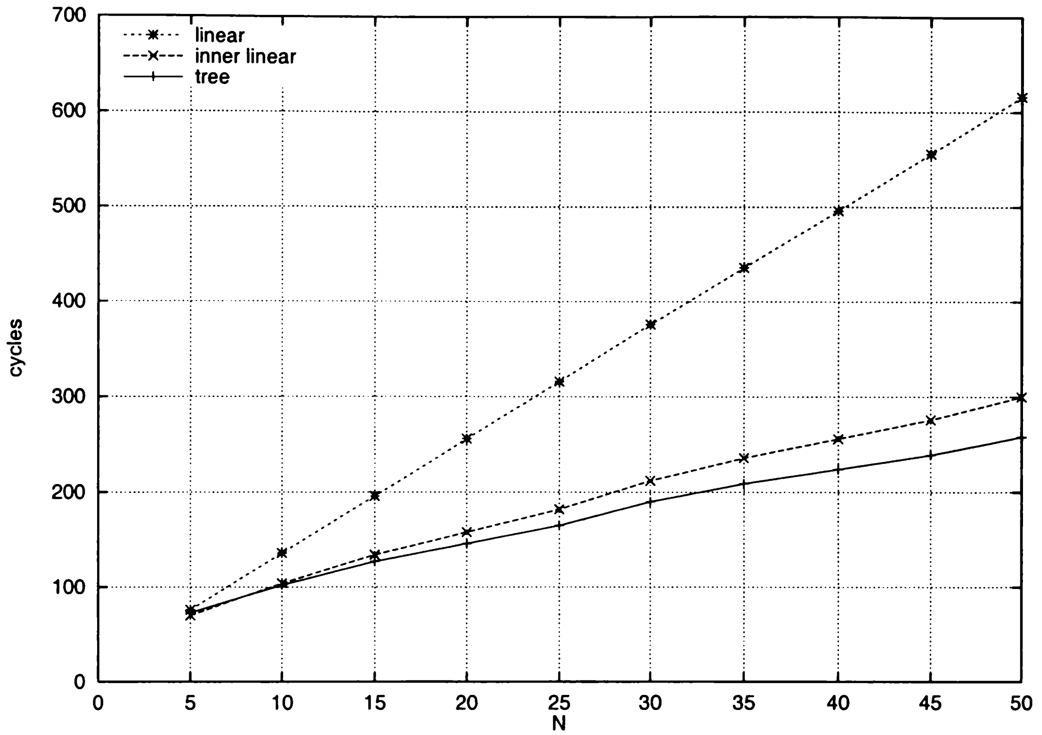


Figure 7.16: Critical path length for *mat* with different loop constructs.

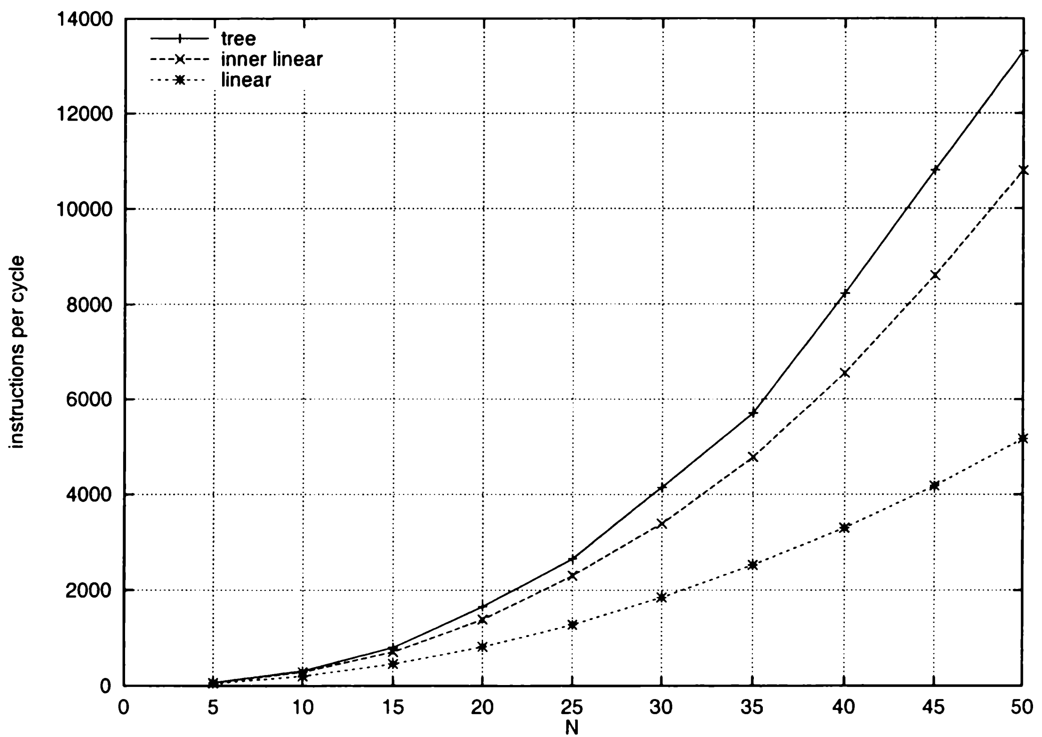


Figure 7.17: IPC for *mat* with different loop constructs.

the insertion process, especially when insertions are performed in parallel. This can be seen in Figure 7.18 where *bin* programmed with a single free list gives $\mathcal{O}(\log N)$ parallelism because it contains an $\mathcal{O}(N)$ critical path. With multiple free lists performance is not hindered allowing maximum parallelism to be obtained.

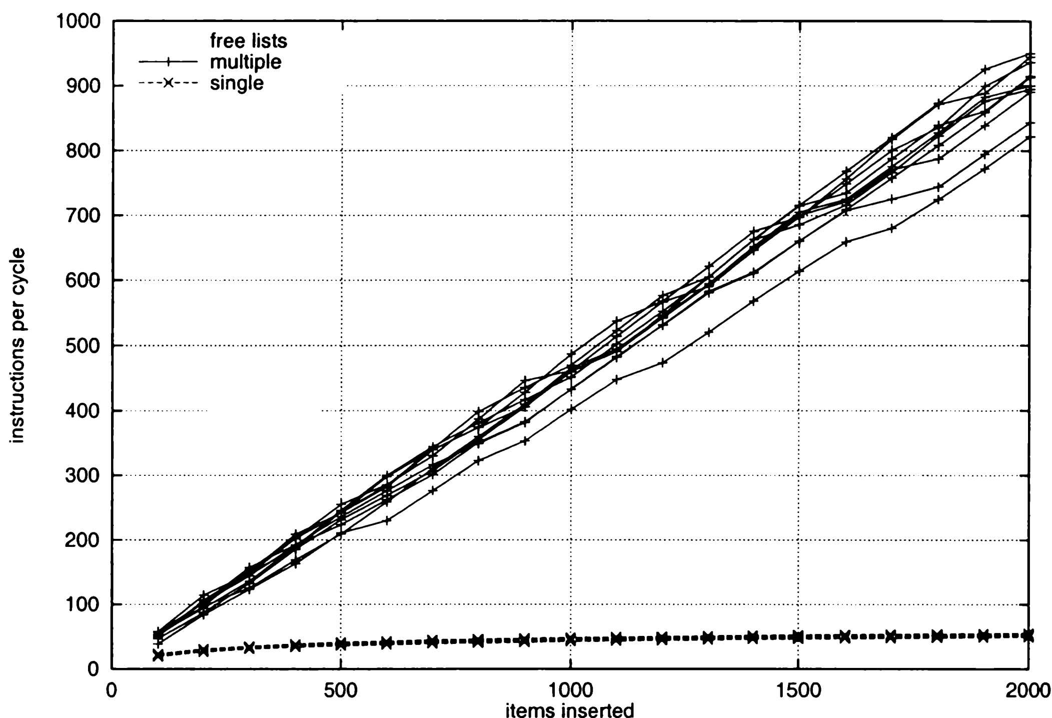


Figure 7.18: IPC for *bin* with a single and multiple free lists.

The code for loop control mechanisms and free list maintenance is generated at compile-time. To obtain best performance these constructs must be structured to promote parallelism. The choice of algorithms used to solve problems also has an impact on the parallelism that can be obtained. Again it is important to use those algorithms that allow parallelism to be extracted.

7.2 Memory access constraints

Memory accesses are a significant proportion of the operations on a program's critical path meaning that the memory system could become a bottleneck to performance. To gain the levels of parallelism seen in the previous section the WarpEngine uses a timestamped memory system to allow memory accesses to occur out-of-order. If certain ac-

cesses are not allowed to pass other accesses, in programmed order, the potential for performance is reduced. It is a question as to whether out-of-order memory accesses are a requirement for obtaining parallelism.

This section investigates the requirements of a timestamped memory to allow memory accesses to occur in any order. It will be shown that data value speculation, achieved by out-of-order memory accesses, is necessary to achieve large scale ILP.

7.2.1 Method

To test memory access order requirements several *machines* with various ordering restrictions are modelled with the WarpEngine simulator. These machines can be classified by the set of memory access re-orderings that they allow. For example, if reads can be re-ordered then this is said to be a Reads can pass Reads (RR) machine. Similarly if a write can be interchanged with an earlier read, in program order, then the machine is said to be a Writes can pass Reads (WR) machine. Writes can pass Writes (WW) and Reads can pass Writes (RW) round out the possible re-orderings.

A range of possible restrictions on the re-ordering of memory operations is considered from the purely sequential case where no re-ordering is permitted to the completely permissive one where memory operations may occur in any order so that the parallelism is restricted only by true data dependencies. To examine the performance properties of each of the reordering types a set of abstract machine models are defined and the potential parallelism for each machine is analysed. The machines are formed by taking combinations of the memory ordering relaxation types and are summarised in Table 7.2. They each assume unlimited processing resources and memory bandwidth.

At the extremes there are two machines, one with all ordering constraints in place and the other with all ordering constraints relaxed. The totally constrained machine is included to give a baseline. All other machine combinations incorporate the RR model, with the fully relaxed machine only constrained by the true data dependencies of the program. A RAR hazard violation has no effect on the outcome of a program so no address disambiguation between read accesses is necessary, meaning that the functionality of the RR machine is easy to implement in hardware.

machine	*description*
NONE	All memory accesses happen in their virtual order.
RR	Reads can pass reads.
RR-WW	Reads can pass reads and writes can pass writes.
RR-WR	Reads and writes can pass reads.
RR-WR-WW	Reads and writes can pass reads and writes can also pass writes.
RR-RW	Reads can pass reads and writes.
RR-RW-WW	Reads can pass reads and writes, and writes can pass writes.
RR-RW-WR	Reads can pass reads and writes, and writes can pass reads.
ALL	Memory accesses can happen in any order.

Table 7.2: Abstract machine models used in memory order constraint tests.

If memory operations are sent to different addresses then the instructions are not constrained by any hazards. When an access to memory is made, one of the parameters given is the address of a memory location. To allow the accesses to happen out-of-order the associated address of a memory access must be compared against all other pending accesses. This comparison can be performed either when the access enters the memory system, or when the address of the access is known. The difference between the two schemes is that in the first the memory operation has to wait for all its parameters, whereas in the second it only has to wait for the address parameter.

In the second case *early address knowledge* is used to potentially improve performance because in many cases addresses are static or computed from loop indices and may be known much earlier than the actual value. The hope is that by doing the check early more re-ordering will be possible resulting in more parallelism. The trade-off is that extra hardware or software is required to determine when the address parameter is valid. The effect of having early address knowledge when performing memory address disambiguation is examined for each of the machine models.

To model these memory ordering machines using the virtual ordered simulator two extra timestamps are used. These record the time that the last read and last write occurred. When a memory operation is issued its timestamp is compared to the read and write timestamps. The memory operation's timestamp is updated to indicate that it has been prevented from passing other memory operations based on the characteristics of the machine modelled. When using early address knowledge the timestamp of the address operand is considered in calculations rather than the timestamp of instruction as a whole.

7.2.2 Results

Figures 7.19 to 7.22, and C.9 to C.12, show the IPC obtained for each of the test programs with varying data set sizes running on each of the machines. *fibf* has not been included because it performs only one memory access. In these plots memory address disambiguation is performed only when both the address and the value to be written are available.

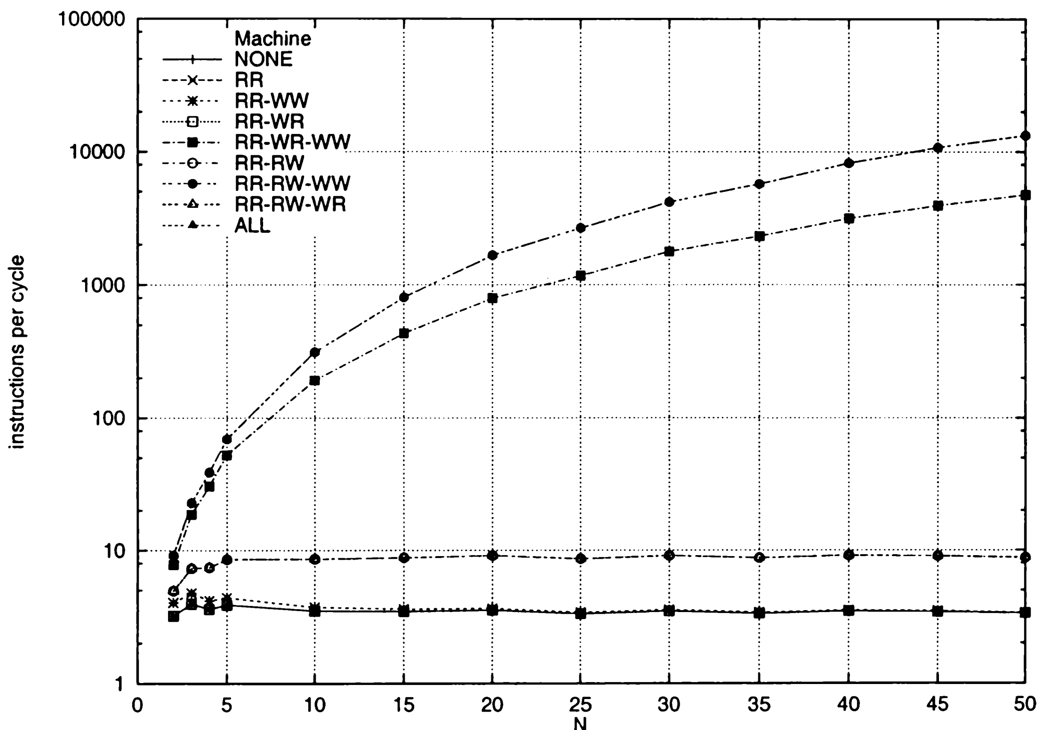


Figure 7.19: IPC vs problem size with memory ordering constraints in place for *mat*.

In general the NONE and RR machines greatly restrict performance when compared to the ALL machine and when more relaxations are in place performance improves. In *mat* (Figure 7.19), *qs2* (Figure C.11), and *fib* (Figure 7.22), the RR-RW-WW machine extracts as much parallelism as the ALL machine and performs better than all the others for all but *bin* (Figure C.12), and *avl* (Figure C.13). The RR-WR-WW machine performs well for *qs1* (Figure 7.21), *bin* (Figure C.12), *avl* (Figure C.13), and *fib* (Figure 7.22). The other machines do not fare so well.

Problem size has an effect on the performance achieved in all the machines. In the more relaxed machines performance improves with problem size. In these machines the fluctuations between sampled problem sizes seen when unlimited resources are in place are

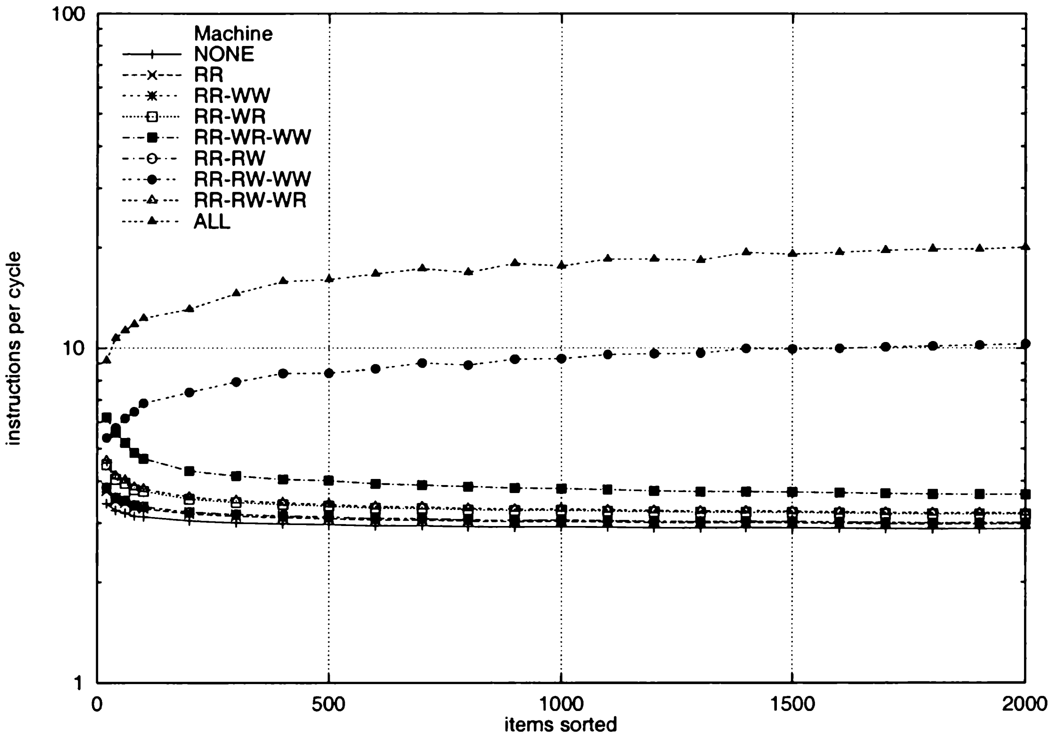


Figure 7.20: IPC vs problem size with memory ordering constraints in place for *heap*.

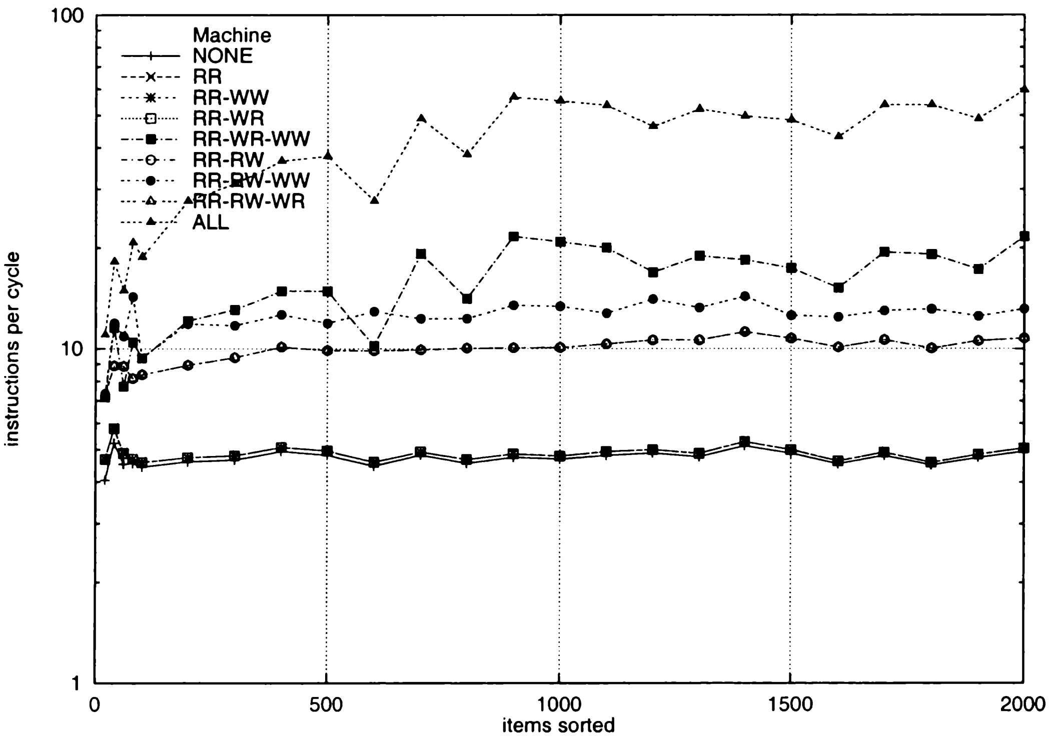


Figure 7.21: IPC vs problem size with memory ordering constraints in place for *qs1*.

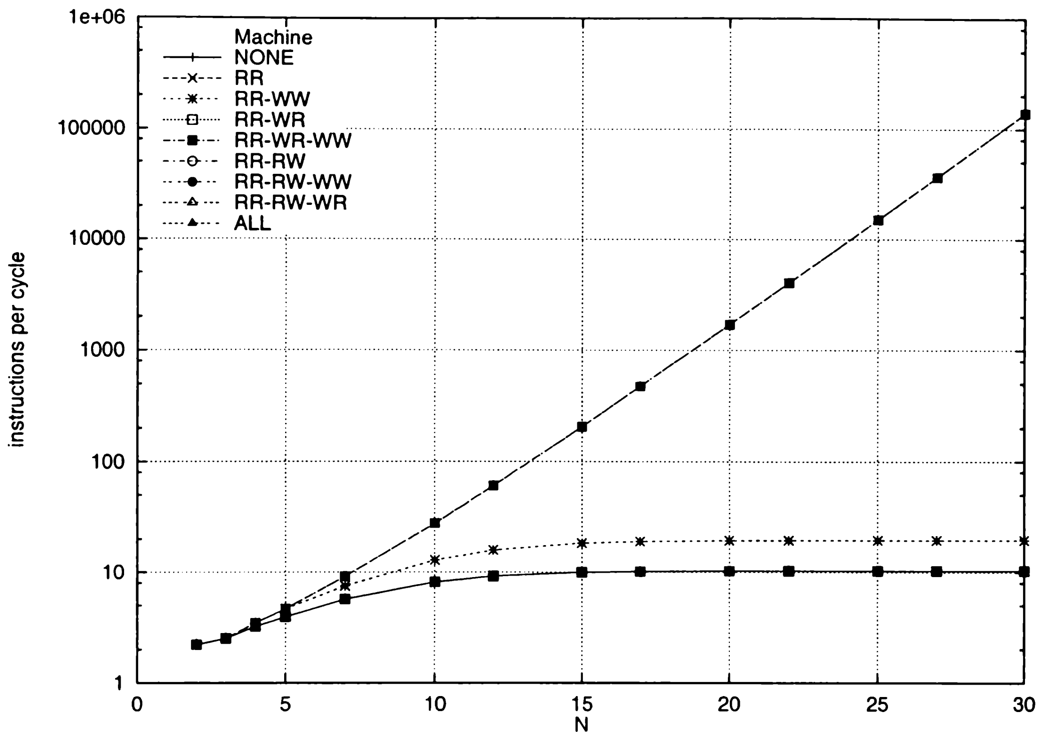


Figure 7.22: IPC vs problem size with memory ordering constraints in place for *fib*.

also prominent. The more restrictive machines tend to smooth the performance curves eliminating the fluctuations. In some cases performance deteriorates with problem size. This can be seen in *heap* (Figure 7.20), *bin* (Figure C.12), and *avl* (Figure C.13). This phenomenon will be discussed in Section 7.3. In almost all cases the relative performance of the machines is maintained across a wide range of problem sizes. This indicates that any problem size can be chosen for further analysis without loss of generality.

Table 7.3 shows the parallelism obtained for each algorithm and at a given problem size when run on each of the abstract machine models when early address knowledge is not used. Machines that extract the maximum available IPC are highlighted. In general, the more relaxed the memory constraints the better the performance obtained. This table also shows that each algorithm reacts differently to the ordering constraints. In most cases the maximum potential parallelism in not realised until the ALL machine is used.

Table 7.4 shows the parallelism obtained when early address knowledge is used with each of the machine models. Recall that in this case address disambiguation for writes is done as soon as the address is available rather than waiting for the data value as well. None of the results for the ALL machine are affected because it does not wait for disam-

problem	*size*	*NONE*	*RR*	*RR WW*	*RR WR*	*RR WR WW*	*RR RW*	*RR RW WW*	*RR RW WR*	*ALL*
mat	50	3.34	3.34	3.38	3.34	4730	8.80	**13311**	8.80	**13311**
trans	50	3.39	4.62	4.72	4.62	444.9	280.5	21261	280.5	**21550**
gj	30	3.69	4.66	4.67	4.90	9.27	135.1	483.5	135.1	**488.2**
heap	2000	2.88	2.98	3.00	3.19	3.65	3.00	10.30	3.22	**20.01**
qs1	2000	4.95	5.06	5.06	5.06	21.59	10.74	13.13	10.74	**59.65**
qs2	2000	1.81	1.99	2.00	1.99	2.02	2.04	**5.34**	2.04	**5.34**
bin	2000	2.04	2.04	2.17	2.05	946.5	32.84	535.27	32.84	**950.4**
avl	2000	2.47	2.47	2.52	2.49	124.0	2.85	46.84	2.87	**198.9**
fib	30	10.33	10.33	19.57	10.33	**139580**	10.33	**139580**	10.33	**139580**

Table 7.3: IPC for each problem at a given data set sizes for each of the memory order constraint machines.

biguation before allowing passing. In a similar vein it would be expected that the greatest effect might be on the more restrictive machines. All machines show improved performance except in the cases where maximum performance had previously been achieved.

problem	*size*	*NONE*	*RR*	*RR WW*	*RR WR*	*RR WR WW*	*RR RW*	*RR RW WW*	*RR RW WR*	*ALL*
mat	50	6.57	6.57	6.70	151.44	8005	5180	**13311**	5180	**13311**
trans	50	3.63	7.26	7.41	7.26	10490	501.7	21333	501.7	**21550**
gj	30	4.73	6.45	6.45	6.91	14.79	165.2	483.7	165.2	**488.2**
heap	2000	2.97	3.04	3.04	3.26	3.69	3.04	10.45	3.26	**20.01**
qs1	2000	5.49	5.64	5.81	5.64	24.29	10.81	13.19	10.81	**59.65**
qs2	2000	1.88	2.03	2.03	2.03	2.05	2.07	**5.34**	2.07	**5.34**
bin	2000	2.06	2.06	2.19	2.06	**950.4**	38.05	547.9	38.05	**950.4**
avl	2000	2.61	2.61	2.67	2.64	131.1	2.87	46.84	2.90	**198.9**
fib	30	**139580**	**139580**	**139580**	**139580**	**139580**	**139580**	**139580**	**139580**	**139580**

Table 7.4: IPC for each problem at a given data set sizes for each of the memory order constraint machines using early address knowledge.

Clearly early address disambiguation can be important for regular computations. Consider some interesting cases: on the RR-RW machine *mat* improved by a factor of more than 500 (from 8.8 to 5180), *fib* improved by a factor of 13000, *gj* improved by 22% and *trans* by 78%; on RR-RW-WW no program improved by more than 2%; and of course on ALL there was no change.

Programs with regular addressing patterns and addresses that are determined on the basis of index values in loops show the best improvement. The results bear this out with the matrix manipulation programs showing significant improvements on the more restrictive machines. The dynamic problems show little improvement for any of the machines. Any improvements in these problems are more noticeable in the restrictive machines.

The *fib* IPC numbers are equal to the maximum for all machines. This is because the addresses for the results of each recursive call are determined on the way down the recursive calls and are then used on the way back up. So even for the NONE machine the address disambiguation can be performed early allowing maximum performance to be obtained. This shows that early address disambiguation can improve performance in systems with restricted memory access orders, but only if addressing patterns are regular.

7.2.3 Machine rankings

Contemporary ILP processors allow some memory accesses to occur out-of-order w.r.t. their programmed order on a limited scale. The HP PA-8000 achieves out-of-order memory accesses on a small scale using a 28 entry Address Re-order Buffer [Gwennap, 1996]. There is a trade-off between the complexity of re-ordering hardware and the performance attainable. A study by INTEL [1995] concludes that for the Pentium Pro architecture:

- Writes can be constrained from passing other writes, for only a small impact on performance.
- Writes can be constrained from passing reads, for an inconsequential performance loss.
- Constraining reads from passing other reads or writes has a significant impact on performance.

In summary they propose that a machine that allows reads to pass other reads and writes is a good compromise between complexity and the performance obtained. It is a question of whether this is the correct evolutionary path to take.

To show the paths of best improvement Figures 7.23 to 7.26, and C.14 to C.18 graph the IPC measurements of Tables 7.3 and 7.4 in a lattice form. The dashed lines drawn between machines link those that differ in only one re-ordering attribute. In each figure, part a) is not using, and b) is using early address knowledge.

The arrows between machines indicate those on the path of best evolutionary improvement for each program. From a given machine an upward arrow indicates the machine immediately above it with the highest performance. For example, in Figure 7.23 the RR

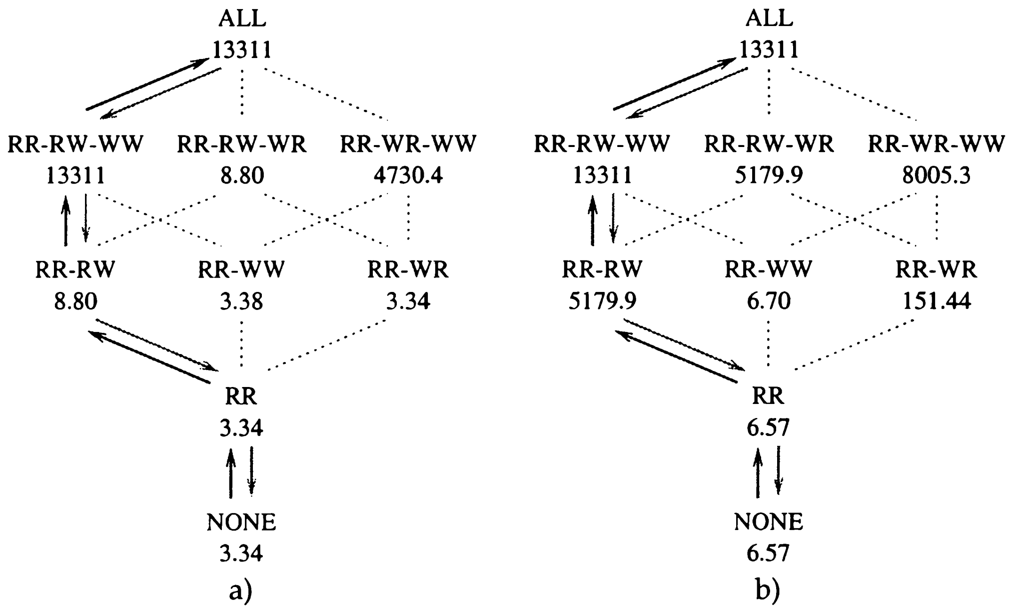


Figure 7.23: Memory constraint ordering mesh for *mat* (50).

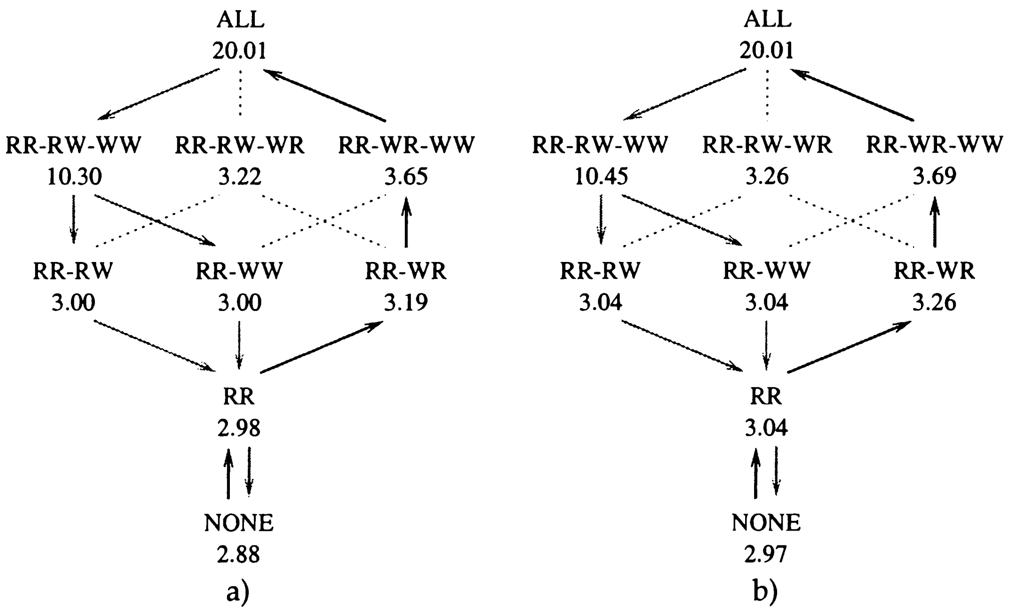


Figure 7.24: Memory constraint ordering mesh for *heap* (2000).

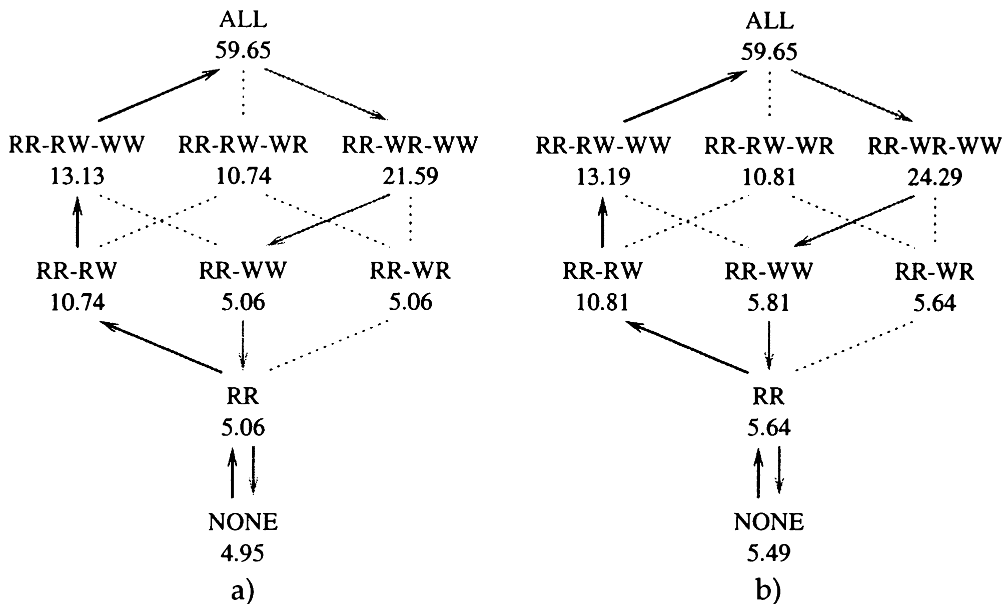


Figure 7.25: Memory constraint ordering mesh for *qs1* (2000).

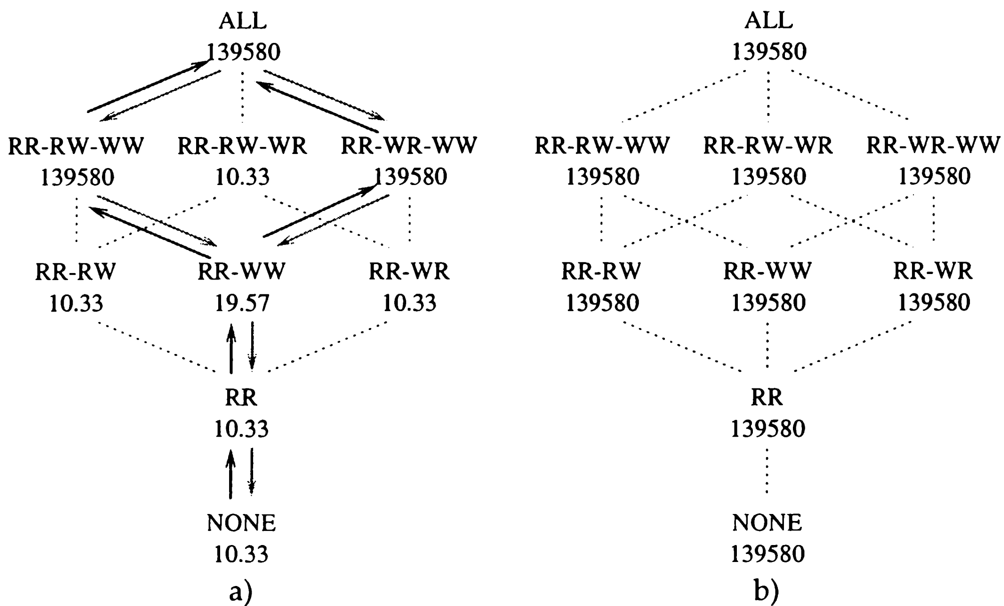


Figure 7.26: Memory constraint ordering mesh for *fib* (30).

machine has three machines immediately above it (RR-RW, RR-WW, RR-WR), and an arrow link is connected to RR-RW which has the best parallelism of the three. Similar downward arrows are also added indicating the best machine below a given machine. Following the upward arrows forms a path of *greatest improvement* when moving from the most to least restrictive machines.

From the meshes there are four sets of paths. They are grouped as: *mat*, *trans*, *gj*, and *qs2*; *qs1*, *avl*, and *bin*; *heap*; and *fib*. The most favoured upward path is, or contains, RR → RR-RW → RR-RW-WW → ALL. The most favoured downward path is, or contains, ALL → RR-RW-WW → RR-RW → RR. Some lattices have different paths in the upward and downward directions. For example, *heap* (Figure 7.24) and *qs1* (Figure 7.25). *heap* (Figure 7.24) and *fib* (Figure 7.26) have multiple paths in one or both directions and do not include the most favoured path in the upward direction. *qs1* and *fib* do not contain the most favoured path in the downward direction. Also, there is no change in preferred path when early address knowledge is used, except for *fib* where all paths become equally favourable.

The RR-RW machine sits on the most favoured upward path, concurring with the earlier statement that an RR-RW machine was a good compromise between complexity and available parallelism. In the lattices there is a large step from this machine to the best machine one level above it, which is usually the RR-RW-WW machine, and this machine tends to get near maximum parallelism for most algorithms.

These results support the statement that a RR-RW machine does allow significant amounts of parallelism to be extracted in some cases. The RR-RW machine gives IPC of 50 or more for three of the nine algorithms, but the other six languish between 2 and 11. To consistently move into the region of more than 15 IPC clearly requires less restrictive machines. Adding WR capability to give the RR-RW-WR machine achieves no noticeable improvement. Adding WW to give RR-RW-WW does significantly improve the performance of all the problems. To move from the RR-RW-WW to the ALL machine would require little or no extra hardware support. This means that to extract large levels of IPC an architecture with the capabilities of the ALL machine is required.

7.3 Limiting state saving resources

The previous section has shown that allowing out-of-order memory accesses is critical to obtaining good performance on our benchmarks. In the WarpEngine the time-space cache provides a mechanism to allow out-of-order memory accesses to occur. State saving is used to capture speculative memory accesses to allow data dependence violations to be detected. Similarly, instructions executed speculatively need to have their operand's state saved. This is achieved through the use of frames.

The evaluations in previous sections have measured performance on the WarpEngine where unlimited processing and state saving resources are available. This section investigates the effect on performance of limiting the number of frames available for instruction state saving. The usage of frames during execution and their relationship to memory state saving resources is investigated in Chapter 8.

7.3.1 Method

Placing an upper limit on the number of frames in the system is comparable to setting the size of the IRB in contemporary ILP architectures. In the simulator this is achieved using the model extension described in Section 4.3.2. Simulations were then performed on each of the test programs and data sets with various frame limits in place.

In this section an upper frame limit of 1024 has been chosen which is the equivalent of an IRB of 16384 instructions. Contemporary production ILP processors have the space to implement instruction buffers of the order of 32 to 64 instructions (2 to 4 frames). These architectures cannot make use of larger IRBs because non-perfect branch prediction prevents them from being filled with valid speculative instructions. The WarpEngine, on the other hand, could replace some space dedicated to caching with frame buffers which are easier to fill.

7.3.2 Results

Figure 7.27 plots IPC against a range of frame limits for each of the test programs at a fixed problem size. In general, as expected, the amount of IPC obtained increases as

more frames are made available. There is never a decrease in performance. Each plot increases at sub-linear rate, with greater performances gains between frame limits made at the lower end of the scale.

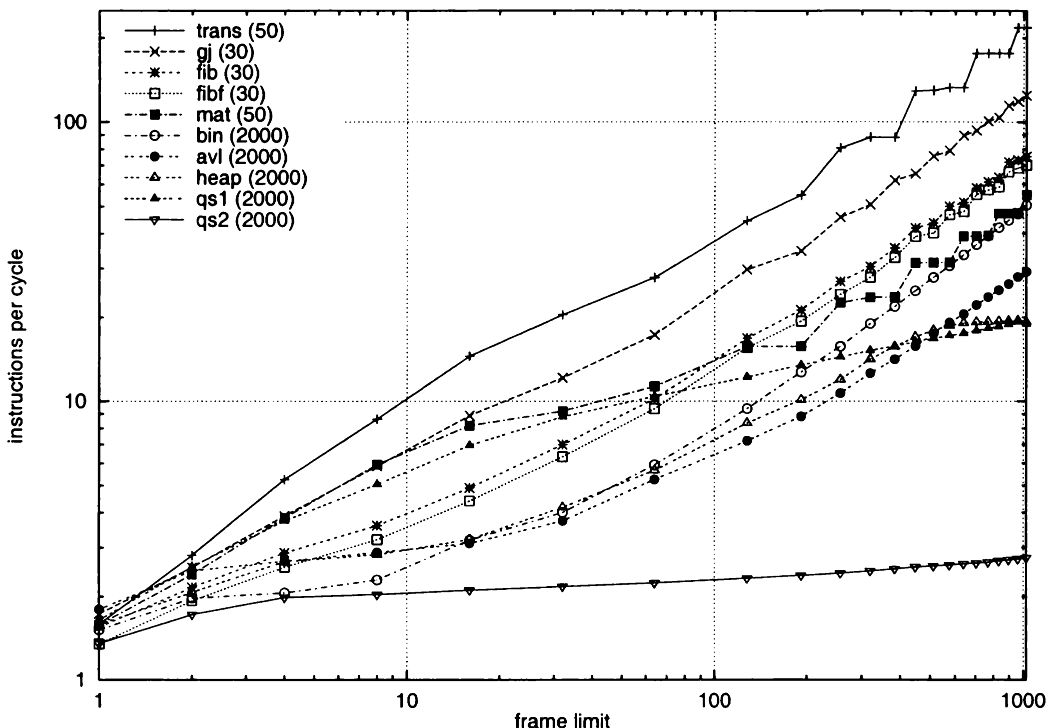


Figure 7.27: IPC vs number of frames for each problem.

In the plot for *trans* (2000) there is a striking effect with the steps in performance between sampled frame limits in the 256 to 1024 range. Although not as prominent, similar steps can also be seen in the *mat*. An explanation of the cause of these plateaus and jumps in performance is given in Section 8.2.

In most cases the parallelism gained at a 1024 frame limit is well below the maximum attainable. Table 7.5 lists IPC values for each problem in Figure 7.27 when restricted to 1024, and 10000 frames, and the maximum IPC available. At the 1024 frame limit *heap* reaches 96% of its maximum IPC. The next closest is *qs2* at 52% of its maximum. All other programs still have the majority of the parallelism to be extracted. At the 10000 frame limit more parallelism is gained. At this point *gj*, *qs1*, and *avl* obtain more than 60% of the parallelism that is available.

Figure 7.28 graphs IPC against frame limit for each *heap* at a variety of problem sizes. Frame limits have been sampled in powers of 2 ranging from 1 to 1024.

problem	size	frames 1024	frames 10000	maximum IPC
mat	50	55.0	414.4	13311
trans	50	217.7	1407	21550
gj	30	124.4	477.0	488.2
heap	2000	19.28	19.94	20.01
qs1	2000	18.93	36.18	59.65
qs2	2000	2.751	3.427	5.337
bin	2000	50.49	324.5	950.4
avl	2000	29.19	126.4	198.9
fib	30	75.35	505.1	139580
fibf	30	70.11	472.6	133637

Table 7.5: IPC for each program with frame restrictions.

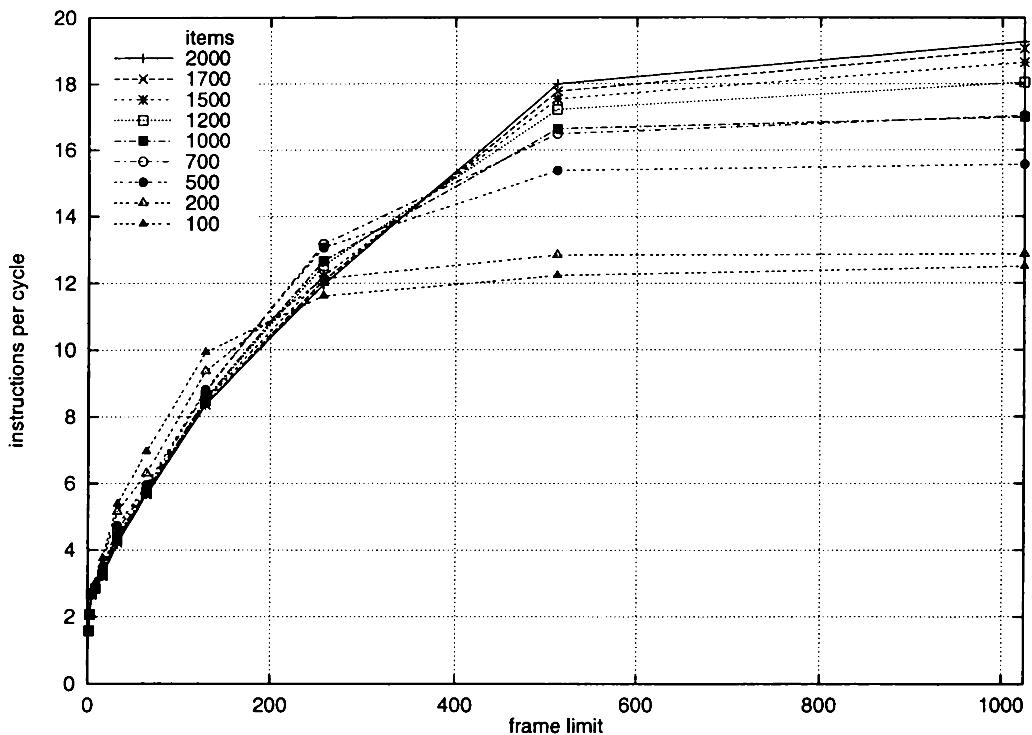


Figure 7.28: IPC vs frame limit for *heap*.

This graph shows that the parallelism obtained is similar across problem sizes for small frame limits, but varies more at larger frame limits. Similar fan-out in IPC obtained occurs with the other test programs. The parallelism obtained increases with frame limit, with more parallelism extracted from larger problem sizes. For small frame limits performance is limited by a lack of state saving resources, and for larger frame limits performance is limited by the parallelism available in the problem.

As the IPC obtained approaches the IPC limit of the program there are minimal gains made by increasing the frame limit. For example, the maximum IPC available for *heap* (100) is 12.29. At the 512 frame limit 11.93 IPC is obtained and at the 1024 frame limit 12.17 IPC is obtained. There is only a 2% gain in performance with a doubling of the frame limit. This means that only the larger problem sizes, that typically contain more parallelism, benefit from larger frame limits.

For frame limits up to 64, a higher level of IPC is obtained for the smaller problem of *heap* (100) than for any of the others. In fact the IPC obtained is inversely related to problem size at these smaller frame limits. This phenomenon occurs in many of the other test programs.

Plotting IPC against problem size for fixed frame limits allows this decreasing performance to be seen more easily. Figures 7.29 to 7.33, and C.19 to C.22, graph IPC for a range of frame limits (1 to 1024) against a range of problem sizes for each of the test programs.

In *gj* (Figure 7.30) there are fluctuations in IPC. These are due to the data being processed because the same peaks and troughs are seen in the unlimited resources case. Fluctuations between sampled problem sizes are also present in the sorting and dynamic structure routines. These fluctuations are smoothed out at smaller frame limits. The effects on performance of frame limits are similar to those present when restrictive memory order constraints are in place.

In many of the program's a drop-off in IPC is observed across a range of frame limits. This is most noticeable in the larger frame limits graphed for *mat* (Figure 7.29) and *bin* (Figure 7.32) and is also noticeable in the plots where frame limits are less than 64, *heap* (Figure 7.31) and *avl* (Figure C.22). The complexity analysis of Chapter 5 shows that larger problem sizes contain more parallelism in all the test programs. This visible decline

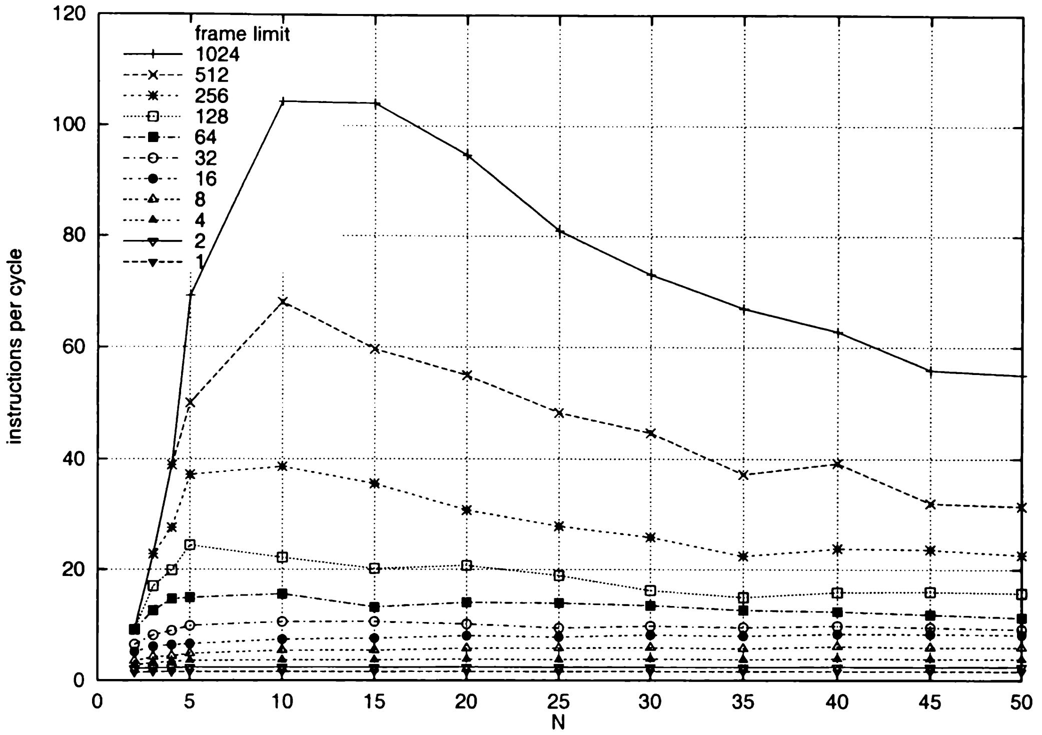


Figure 7.29: IPC vs problem size with limited frames for *mat*.

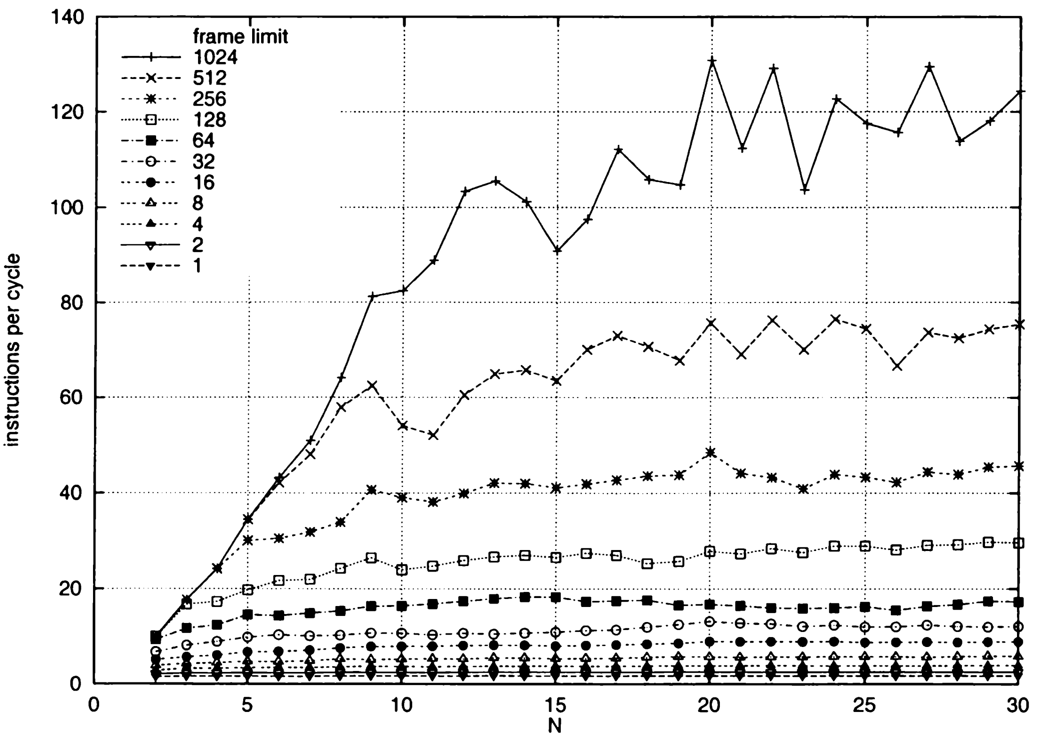


Figure 7.30: IPC vs problem size with limited frames for *gj*.

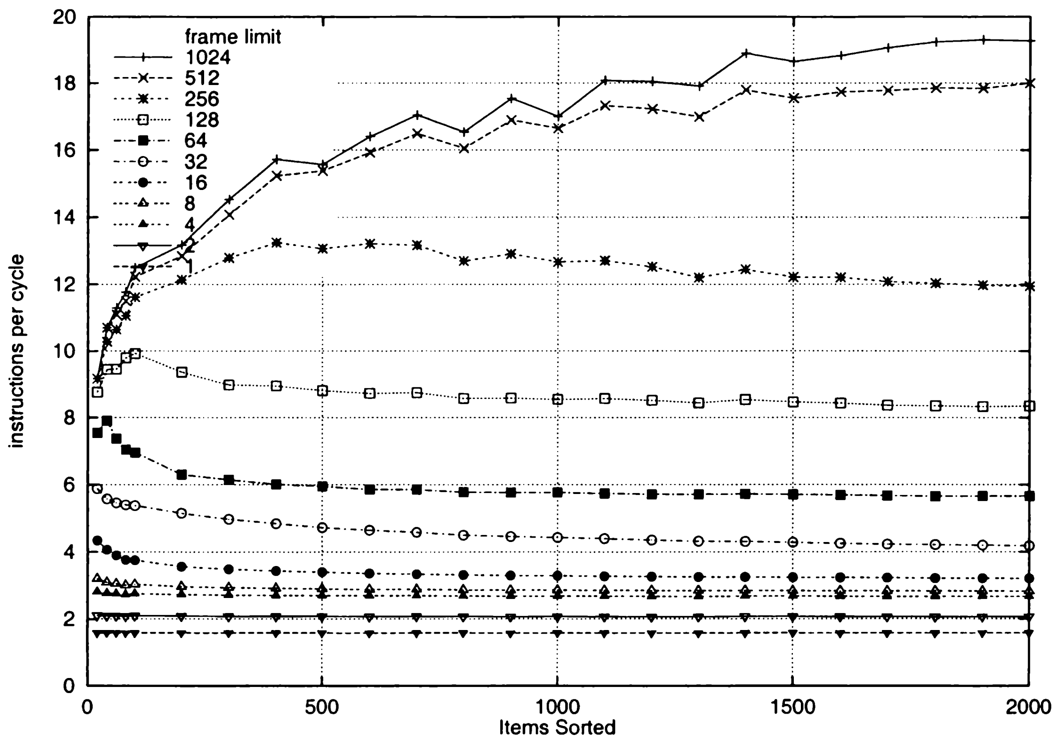


Figure 7.31: IPC vs problem size with limited frames for *heap*.

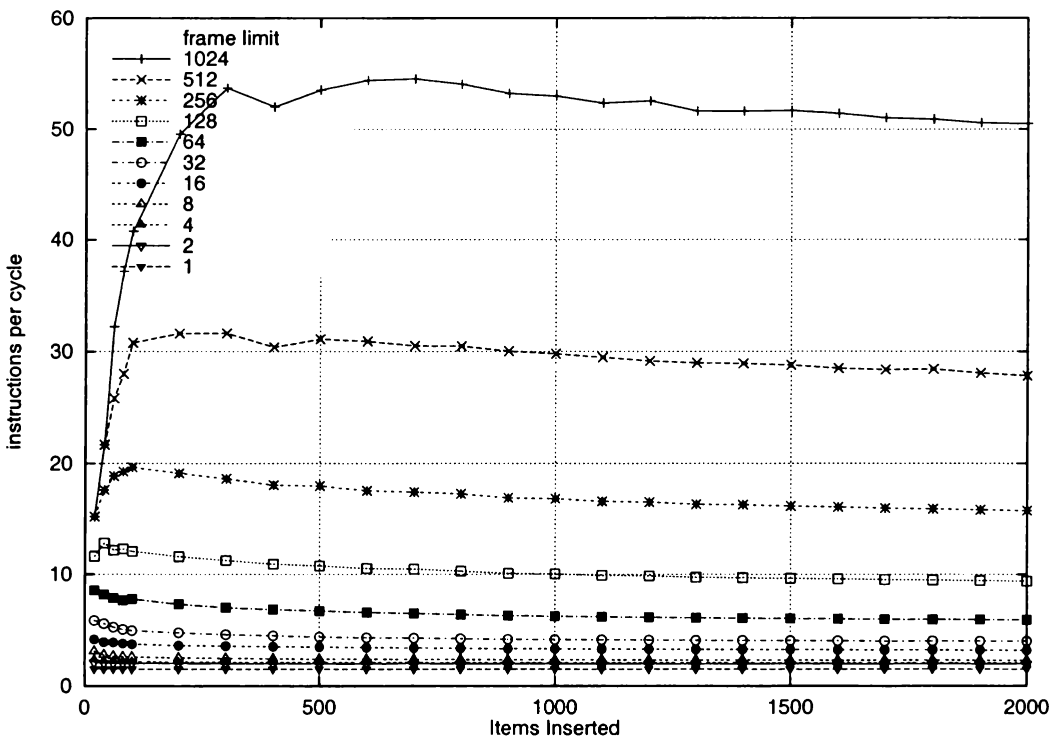


Figure 7.32: IPC vs problem size with limited frames for *bin*.

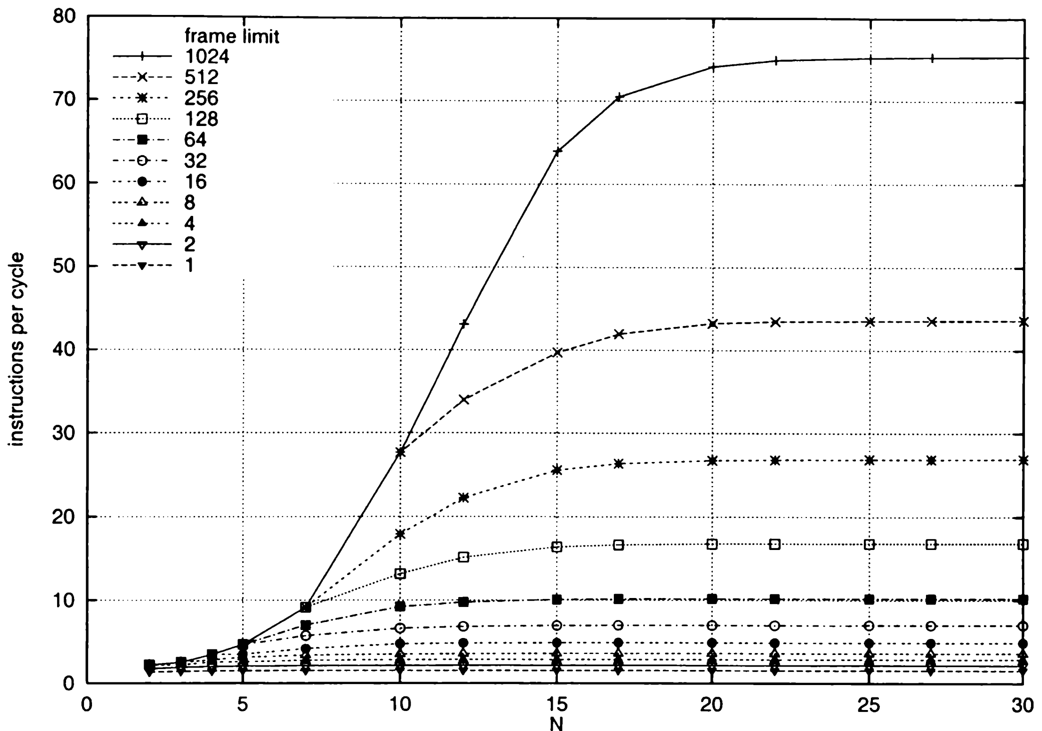


Figure 7.33: IPC vs problem size with limited frames for *fib*.

in performance with respect to problem size seen here produces the fundamental paradox that *larger problem sizes provided less parallelism*. The reason for this decline is discussed in Section 7.3.3.

At the other end of the scale is *fib* (Figure 7.33) in which IPC is initially restricted by small problem size, then rises to a frame limit dependent peak and remains there. In this case increased problem size only improves the potential for extracting parallelism.

7.3.3 Declining performance

In the programs that show declining performance with increasing problem size there are sequential paths that increase in length with respect to problem size. In *mat* this path is the summation in the inner loop which increases in proportion to problem size. With *bin* the sequential path is determined by the number of steps in the search phase. This increases as a logarithmic proportion of problem size. In *heap, avl* and *fib* the sequential paths exist but they are interlaced with other processing that can be performed in parallel, so their impact on performance is not so apparent.

The sequential paths of *mat* and *fib* can be processed in parallel. However, long sequential paths delay the state saving resources of independent speculative events from being freed because instructions are retired in their programmed order. This delay in turn hinders performance because other independent instructions cannot be processed.

To show this phenomenon consider the simple example of the speculative execution of nested loops illustrated in Figure 7.34 with the following parameters:

$$
\begin{aligned}
f &= \text{frame limit} \\
m &= \text{number of inner loop iterations} \\
n &= \text{number of outer loop iterations} \\
c &= \text{count of instructions in inner loop iteration} \\
p &= \text{fractional overlap between inner loop iterations}
\end{aligned}
$$

```
for (i=0;i<m;i++) {
   for (j=0;j<n;j++) {
      some
      instructions
   }
}
```

Figure 7.34: Example of nested loops.

Using these parameters equations defining the program's execution time can be formulated, assuming there are no overheads in parallelising the loops. The program's instruction count is

$$mnc \tag{7.1}$$

and the program's critical path length with unlimited frame resources is

$$mc(1-p)+cp \tag{7.2}$$

This is the time required to execute one of the outer loop iterations taking into account the inner loop iteration overlap. Dividing equation 7.1 by 7.2 gives that parallelism available in the program. In this case it is $\mathcal{O}(n)$.

Taking into account frame limits a series of functions for execution time can be derived over certain ranges of m. These ranges, which are dependent on the relative values of m

and f, are $m < f-1$, $f-1 \leq m < 2f-2$, and $m \geq 2f-2$. For the range $m < f-1$ the mathematics involved is complicated by the frames having to wait to be retired in order. The area of interest in this example is where m and n are large compared to f. Therefore the equations in the range of $m < f-1$ need not be considered.

In the range $f-1 \leq m < 2f-2$ the critical path length is given by the recursive function

$$length = e(n) \tag{7.3}$$

$$e(i) = \begin{cases} 0 & \text{if } i = 0 \\ s(i) + b(m) & \text{otherwise} \end{cases} \tag{7.4}$$

$$s(i) = \begin{cases} 0 & \text{if } i \leq 1 \\ \max(e(i-2), s(i-1) + b(m-f+1)) & \text{otherwise} \end{cases} \tag{7.5}$$

$$b(i) = ic(1-p) + cp \quad \forall i > 0 \tag{7.6}$$

Here $b(i)$ is the time to execute an inner loop of length i, $s(i)$ is the start time of the ith outer loop iteration, and $e(i)$ is the end time of the ith outer loop iteration.

For the range $m \geq 2f-2$ execution time is given by the equation

$$time = n((m-f)c(1-p)+c) + (f-1)c(1-p) \tag{7.7}$$

If the value of n is fixed then the parallelism gained for large m is given by dividing equation 7.1 by equation 7.7.

$$\lim_{m\to\infty} \frac{mnc}{n((m-f)c(1-p)+c) + (f-1)c(1-p)} = \frac{1}{1-p} \tag{7.8}$$

Therefore in the limit the performance is given by the overlap between inner loop iterations. Performance is independent of the number of outer loop iterations.

In problems like matrix multiplication the inner loop size is proportional to the outer loop size. This means altering n affects m at the same rate. Setting m to n gives performance, in the limit, of

$$\lim_{n\to\infty} \frac{n^2c}{n((n-f)c(1-p)+cp) + (f-1)c(1-p)} = \frac{1}{1-p} \tag{7.9}$$

The same result as equation 7.8. The case where $m = n$ is graphed in Figure 7.35 which plots the parallelism gained for $f = 16$, $c = 3$, and $p = 1/3$. The vertical lines indicate

the boundaries of regions for each of the execution time equations. This curve is not too dissimilar to the IPC plots of *mat* in Figure 7.29.

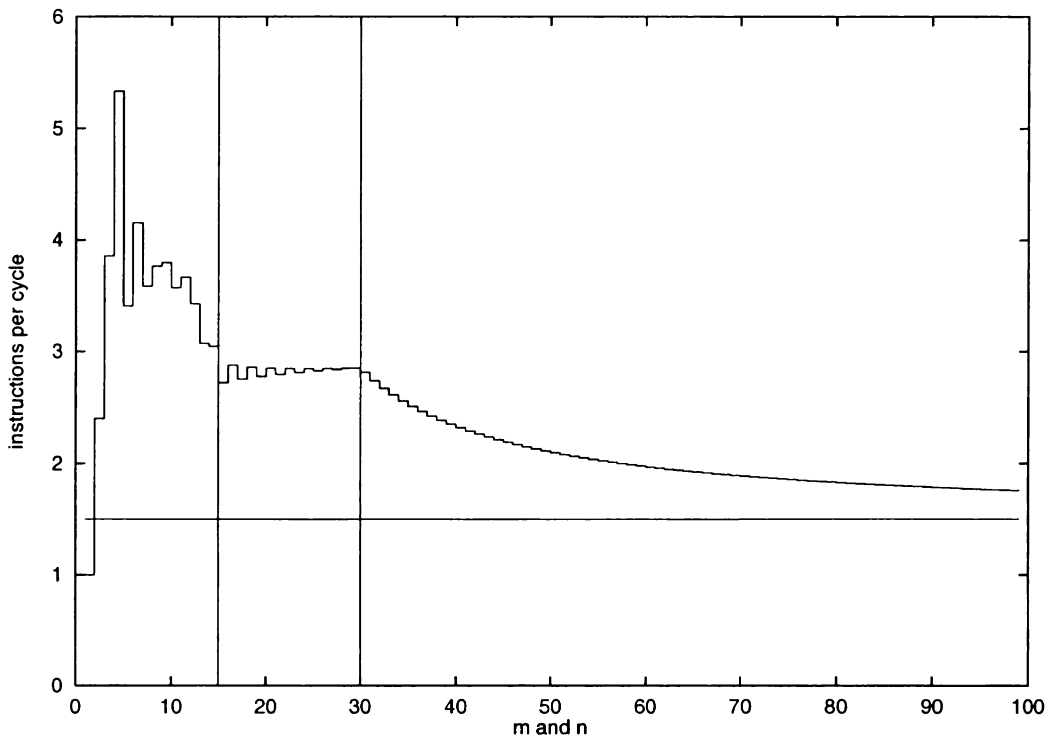


Figure 7.35: IPC vs number of loop iterations for nested loops example.

In summary, for large values of m relative to f the parallelism obtained is independent of the frame limit, but is dependent on the overlaps within the inner loop iterations. This shows that any algorithm that has sequential paths inside parallel threads that are tied to problem size will suffer from decreased performance with increased problem size when frame resources are limited.

7.4 Summary

In this chapter it has been shown that the ideas behind the WarpEngine are adequate to extract parallelism available in programs. The WarpEngine is capable of finding and utilising parallelism even in situations where parallelism is typically hard to detect and extract. Although modelling unlimited resources is not realistic from an absolute performance point of view it shows that there are no fundamental problems when scaling up the architectural ideas of the WarpEngine. The tree-based program control mechanism

is capable of scheduling enough instructions to obtain parallelism and can effectively hide loop control by distributing it. Speculating on data values through the use of the timestamped memory system allows large scale IPC to be obtained.

The performance gained is affected by data sets and algorithmic decisions. In programs that make control decisions based on the data manipulated, variations in IPC for a given problem size shows that dynamic parallelism available can be extracted by the WarpEngine. The choice of loop control structures and dynamic memory allocation mechanisms also affect performance. The implementation of these components of a program would be performed at compile-time meaning that these types of optimisations can be used to assist in the extraction of parallelism.

Another area that affects performance is constraining memory accesses to occur in order. On some of our benchmarks simply allowing reads to happen out-of-order can give a noticeable increase in performance. The addition of early address detection mechanisms can further improve performance. Some regular programs are not affected by constraints on memory accesses, but more complex programs clearly need the greater freedom of full re-ordering. To reliably achieve IPC greater than 20 on our benchmarks it is shown that the memory system must allows all accesses to occur out-of-order. The WarpEngine provides this capability through its timestamped memory system and in particular the time-space cache. While some of the simpler programs achieved good performance on more restrictive machines the more complex programs, such as *heap*, *qs1*, and *avl*, improve significantly with each step in capability. This phenomenon is expected to be more marked for more realistic programs with less regular data and control dependencies.

When state saving resources are limited performance is reduced, as expected. In general an increase in frame limit produces a less that linear improvement in performance. The level of improvement depends on the algorithm and the size of the problem being processed. In programs, such as *mat* and *bin*, there are long sequences of instructions that cause a noticeable decrease in performance as problem size is increased. This produces the fundamental paradox *that increased problem size gives less parallelism.* This phenomenon is tied to the in-order retirement of instructions. Potential solutions to this problem are given in Chapter 8 where the usage patterns of frames are examined. It is important to minimise the effects of these long sequential paths because problem sizes increase at a similar rate to processor performance.

The effects that memory access constraints and state saving space limits have on performance shown here apply to any architecture that uses speculation to obtain large levels of parallelism. The WarpEngine's time-space cache provides a solution to out-of-order memory accesses, but the usage of state saving resources must be examined to determine how they should be implemented.

Chapter 8
Resource usage

In any architecture that uses speculation space must be set aside so that state saving can be performed. In the WarpEngine state saving resources are required to maintain memory consistency and to retain traces of speculatively executed instructions. Memory consistency is maintained by the time-space cache, whereas instruction traces are retained in frames.

Performing large scale speculation implies that large amounts of state saving is required. Information about the amount of state saving space required and how it is used during execution can be used when making hardware design decisions.

This chapter investigates how state saving space is use during the execution of programs. Three areas are investigated when considering the use of state saving resources: the slots within a frame; the frames in the system; and the entries in the time-space cache.

8.1 Slot usage

Slot usage, or frame utilisation, is defined as the number of slots within each frame that contain instructions that will be processed. There are conditional instructions which may not get processed but some state saving space still has to be set aside for them. Slot usage determines how efficiently individual frame state saving resources are being used. This frame utilisation could have an effect on frame design decisions.

The count of slots used per frame can be obtained from two places: statically from the program executable, or dynamically from instruction counts when executing programs. Static counts are easily obtained at compile-time and are constant across problem sizes.

Dynamic counts are more accurate but require extra slot usage counts to be maintained during execution.

The mean and standard deviation of slots used in the static program executables and during the execution of the test programs are listed in Table 8.1. In both cases there is an overall frame utilisation of approximately 50% with a standard deviation of approximately 25%. This means that most frames contain between 4 and 12 instructions. Given that static and dynamic counts give similar values the static counts from program executables, which are easier to calculate, will be used in further analysis.

		static				*dynamic*			
		mean		*std dev*		*mean*		*std dev*	
problem	*size*	*slots*	*%*	*slots*	*%*	*slots*	*%*	*slots*	*%*
mat	20	10.2	63.5	4.01	25.1	7.80	48.7	4.52	28.2
trans	20	10.2	63.6	4.02	25.1	11.0	68.6	3.12	19.5
gj	15	9.28	58.0	4.39	27.5	8.48	53.0	4.77	29.8
heap	200	8.03	50.2	5.08	31.8	8.80	55.01	5.09	31.8
qs1	500	8.22	51.4	4.43	27.7	7.11	44.4	3.83	23.9
qs2	200	7.44	46.5	3.87	24.2	5.85	36.6	2.60	16.3
bin	500	7.53	47.1	3.02	18.9	9.26	57.8	2.26	14.1
avl	200	7.30	45.6	4.43	27.7	7.96	49.8	4.19	26.2
fib	15	6.00	37.5	5.79	36.2	6.60	41.2	4.96	31.0
overall		8.24	51.5	4.34	27.1	8.10	50.6	3.82	23.9

Table 8.1: Average number of slots used per frame.

Fifty percent slot usage over all frames means that at any time half the frame state saving resources that are available will not be in use. Whether this low usage is a major concern depends on the cost of implementing slot state saving resources in hardware. This result suggests that state saving resources need to be cheap in terms of the hardware they require. Such a cost analysis is beyond the scope of this thesis.

8.2 Frame usage

When a block is processed the frame it is assigned to will go through two states. It will start in an active state performing work to process the instructions in the block and then progress to a waiting state until the block is retired, freeing the frame resource. This section will investigate for what period of time frames are in use, and when in use how long are they active or waiting.

This section examines the state of frame resources during execution. First their usage, when no limits are in place, is examined to show the frame limits required to obtain maximum performance. Then the variation of frame usage in a limited resource situation is examined.

8.2.1 Method

Each test program is run on the simulator with information regarding the number of frames and time-space cache entries in use at any given point during execution recorded. To collect this information the time-range buckets described in Section 4.3.2 are modified to kept counts for the different states that a frame can be in.

With unlimited state saving resources at any time during execution a frame can be in one of three states, *active*, *waiting*, or *unused*. A frame is deemed *active* when one or more of its slots are being processed, or waiting to be processed. This represents the time from when the frame is invoked until the last instruction has finished executing. After the last instruction has completed the frame goes into a *waiting* state until it is retired. If the transition from *active* to *waiting* occurs within the range of a bucket only the active count of that bucket is incremented.

When there is a frame limit an extra state is possible. A speculative block may get assigned a frame and at later time the frame is needed for an earlier block in the virtual sequence. In this situation the speculative block is cancelled, wasting any processing performed, and the earlier block is assigned the frame. In this situation a *wasted* count is incremented.

Similar information is kept for the number of read and write entries that must be stored in the time-space cache. As with frames the space for time-space cache entries must be retained until they can be retired. The counts in appropriate buckets are incremented to reflect this.

8.2.2 Results

For most programs similar shaped usage curves are produced across the range of problem sizes tested. This can be seen for *gj* with unlimited resources in Figures C.23 to C.26.

The program problem sizes examined here have been selected so phenomena associated with each program is clearly visible, and so the number of data points plotted is not too excessive. In the limited resource results a frame limit of 1024 has been chosen for the same reasons.

8.2.3 Unlimited resources

The results on frame resource usage are reported in terms of the number of frames that are used and the state that each frame is in. Figures 8.1 to 8.10 plot frame usage over the duration of each program's execution for a given problem size. In these tests no restrictions are placed on the total number of frames available. In each of the frame usage plots,

- the darker region at the bottom represents the number of frames that are *active*ly processing instructions, and
- the lighter region above represents the number of frames that have finished execution and are *waiting* to be retired.

The *waiting* counts have been plotted cumulatively with the *active* counts so that the waiting curve gives the total number of frames in use at any point during execution. The number of waiting frames is the difference between the curves.

Most noticeable across all of the graphs is that the total area of the waiting region is larger than the active region. This indicates that much of the work that is done speculatively is correct, and the speculative execution does not get rolled back. The frames that are waiting are not contributing to improving performance because they have performed their useful work and are only taking up state saving resources.

In all but *qs2*, early in computation there is a rapid build up in the number of frames that are in use. This is most visible in *mat*, *trans*, and *fib* where execution time is short and large amounts of parallelism are gained. There is an initial exponential increase in the number of active frames. This maps to the tree-based looping constructs, and the tree-based control mechanism, which has the ability to quadruple the number of blocks activated at each level in the control tree. The exponential increase in blocks activated shows that the

tree-based control mechanism can effectively schedule instructions to promote parallel execution.

In *mat* (Figure 8.1) and *trans* (Figure 8.2) the number of waiting frames increases to a higher peak than the active frames. This occurs because later independent loop iterations are performing speculative execution on valid data, and the instructions within the frames do not get rolled back. They are, however, prevented from retiring by earlier loop iterations that are still active.

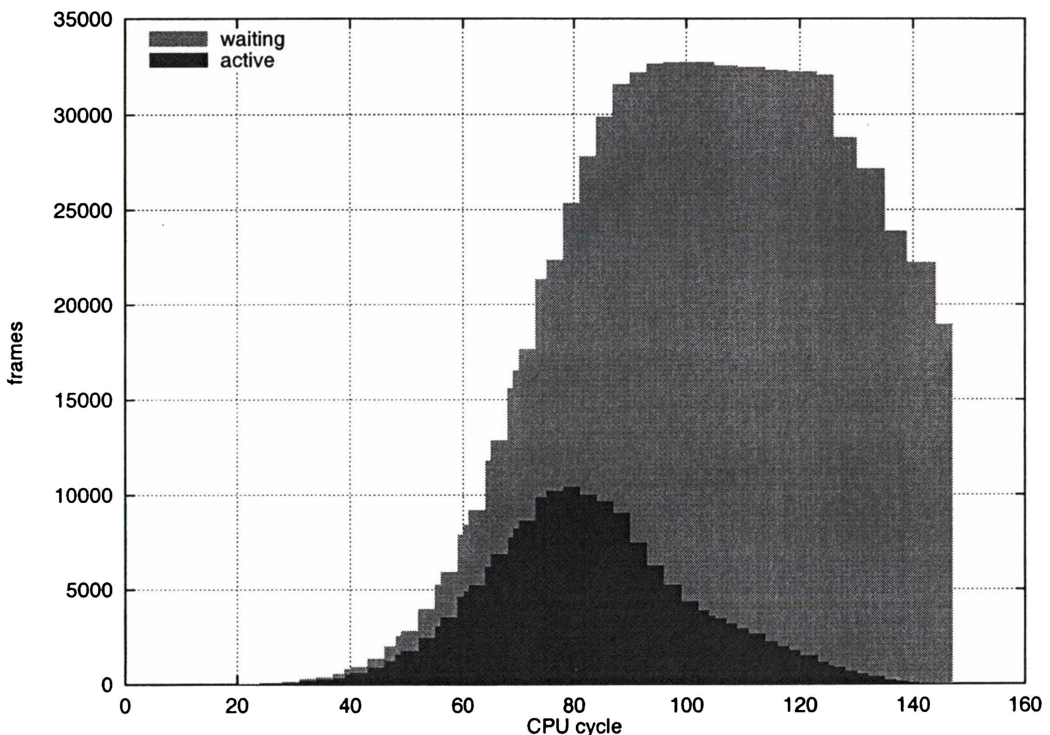


Figure 8.1: Frame usage over time for *mat* (20).

With *mat* once the peak is met at 90 cycles the total number of frames used declines slowly. At this time the number of active frames is dropping quickly but the number of waiting frames is increasing at a slightly slower rate. This is where the computation of the sequential chain of additions in the inner loop is occurring in parallel across all the iterations of the outer loops. Only the frames of the first inner loop iteration are freed, the rest must wait for the entire loop to complete. This gives the slow decline in the total number of frames used. Beyond 125 cycles, multiple iterations of inner loops are retired in parallel. The steps occur because the tree loop control construct start iterations in groups.

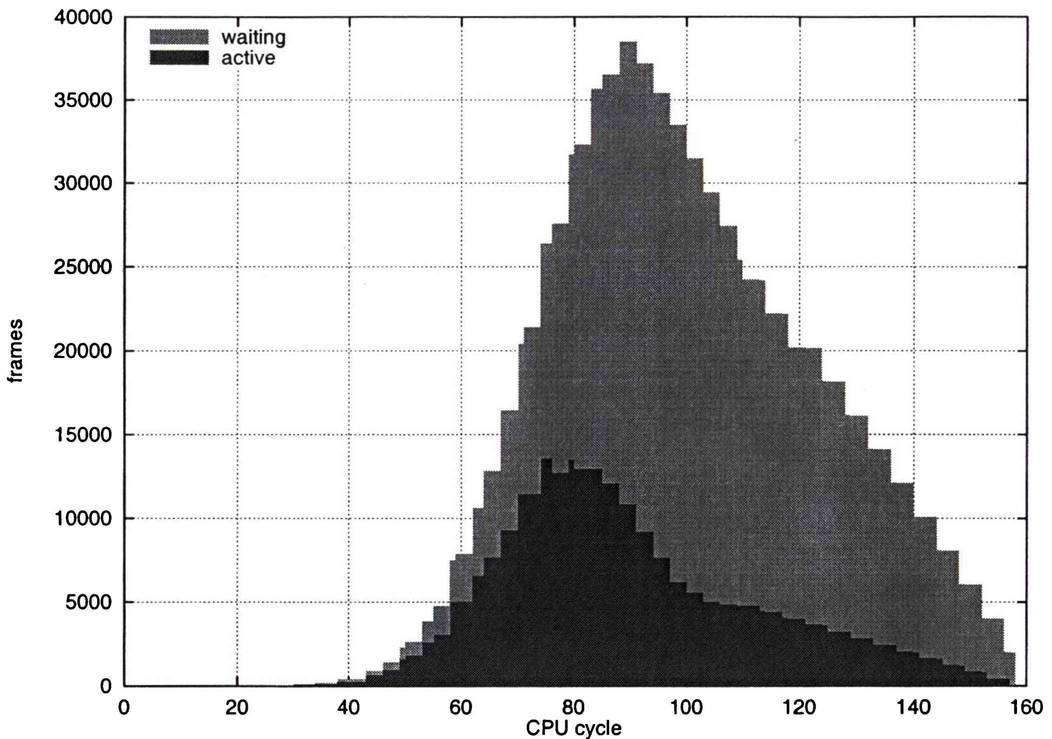


Figure 8.2: Frame usage over time for *trans* (20).

From the peak *trans* shows a steady decline in active and waiting instructions. This is because the inner loops contain many instructions that can run in parallel and the outer loop applies a sequential order. The instructions in the inner loops are retired frequently.

In both *mat* and *trans* the sequential part of the computation can be seen in the decline of active instructions in the latter half of their computations. At this point instructions are executed and rolled-back, or are waiting for valid data to manipulate. The constant decline shows the sequential computation occurring as fewer instructions are waiting for valid data. With correct data they perform correct computation and go to the waiting state.

gj (Figure 8.3) shows many small spikes on the declining slopes of both the active and waiting frames. There is one spike for each row in the matrix that is being inverted, which is 15 rows and corresponding spikes in this case. This shows that instruction processing is occurring in bursts.

With *heap* (Figure 8.4), both the active and waiting curves have an initial, relatively large, spike followed by steadily declining slopes. The constant decline in the active region in-

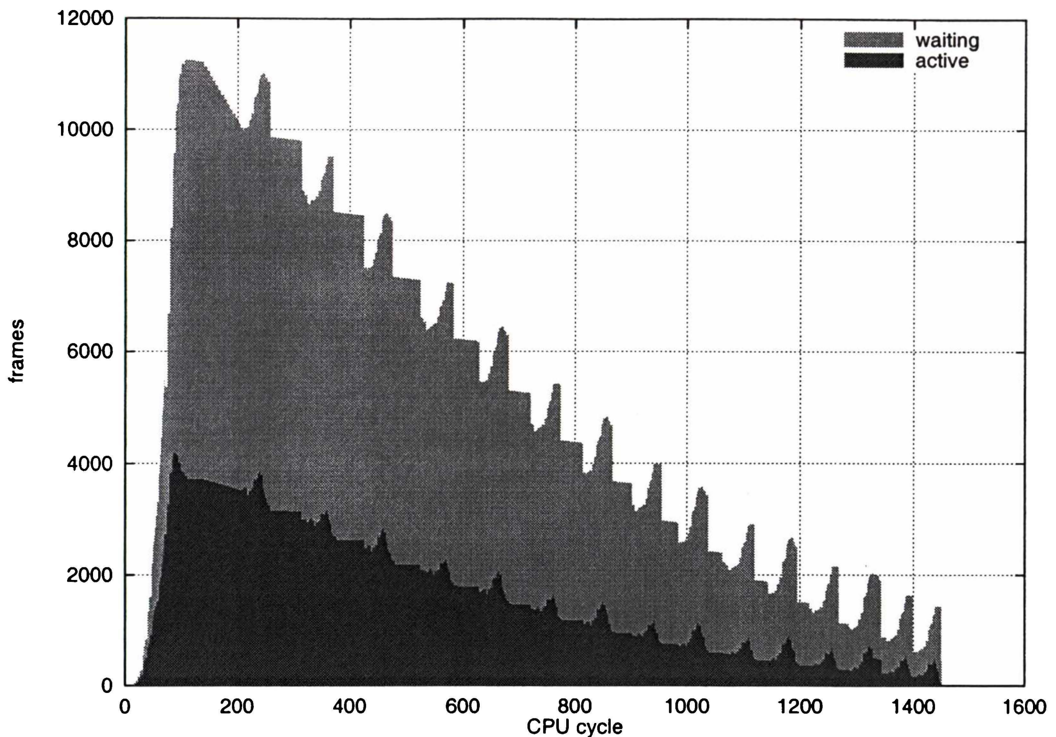


Figure 8.3: Frame usage over time for *gj* (15).

dicates that initially many roll-backs, but are less frequent as time goes on. Dependencies between loop iterations through the root item mean that speculative iterations will get rolled back. The size of the sorting array reduces at a constant rate, one item per iteration. As time progresses the likelihood of an event being rolled back is also reduced at the same rate. Thus there is a reduction in the number of active frames.

In the curves for *qs1* (Figure 8.5) and *qs2* (Figure 8.6) there are points at which the number of active and waiting frames approach zero after having been large. This is especially noticeable in *qs1* where the first sequential pivot finding phase finishes at about 1300 cycles. From that point to the end of execution the number of waiting frames increases significantly while the number of active frames produce little peaks and troughs. This happens because the recursive calls split the problem into smaller independent parts which execute in parallel, but all the speculative recursive calls have to wait to be retired.

qs2 shows the same increase in waiting frames toward the end of execution. There is, however, less parallel computation performed in the first half of execution where the pivot is found at the first level. With the ends-to-middle pivot searching requiring knowledge of the array bounds, speculative computation for the recursive calls cannot proceed

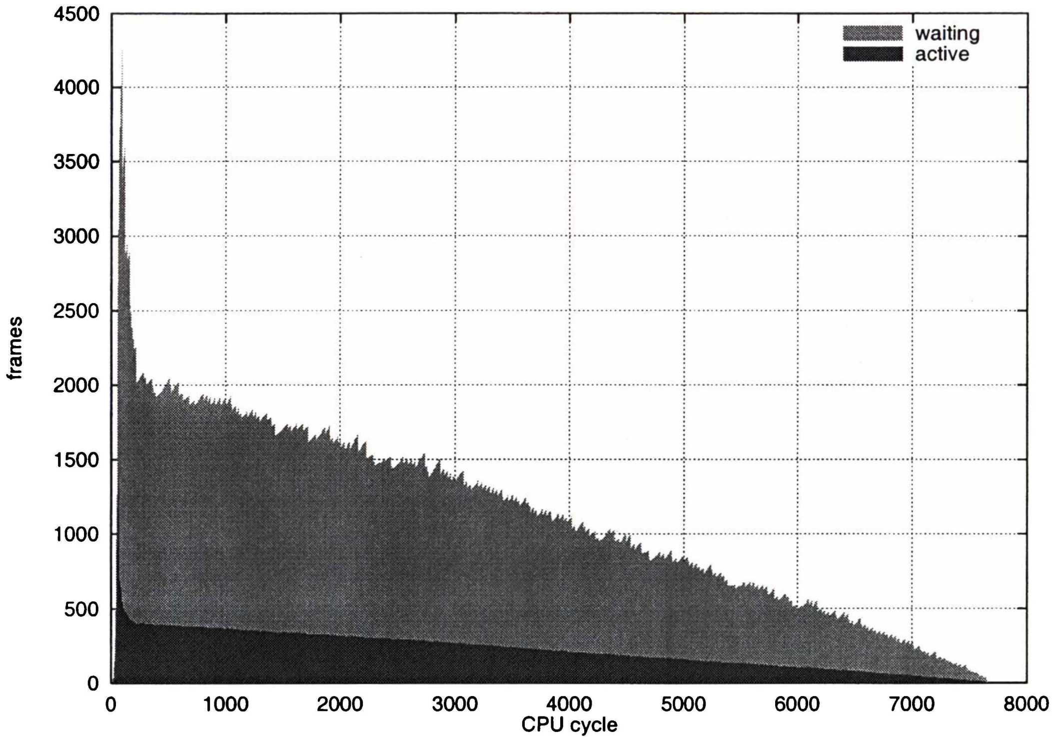


Figure 8.4: Frame usage over time for *heap* (200).

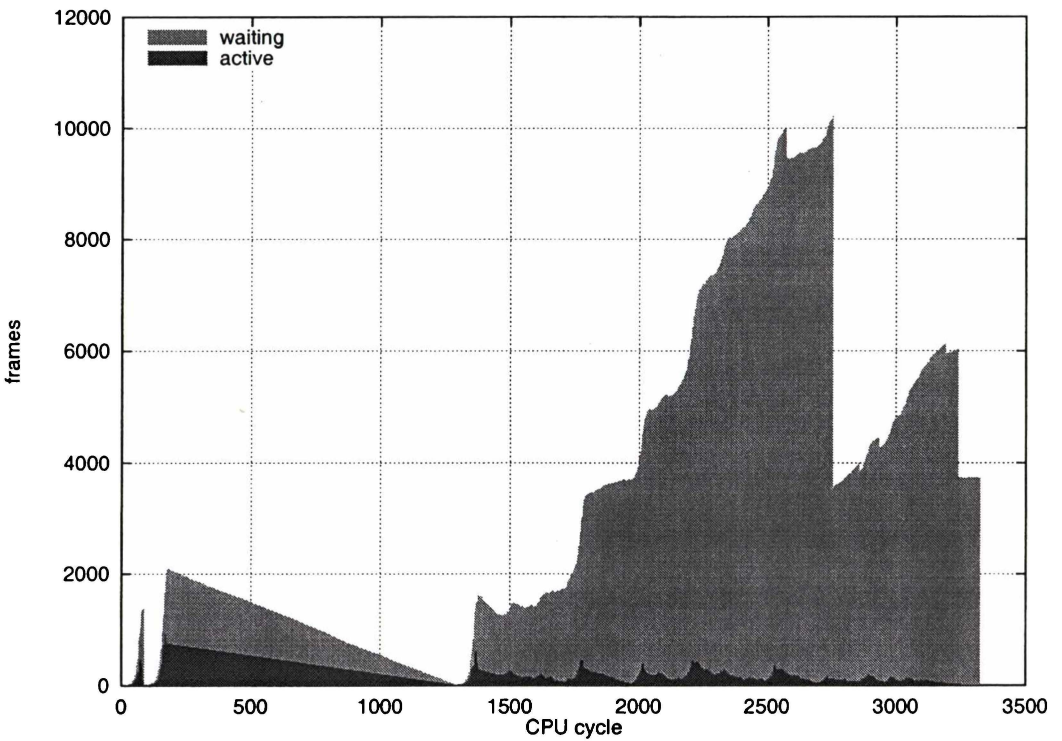


Figure 8.5: Frame usage over time for *qs1* (500).

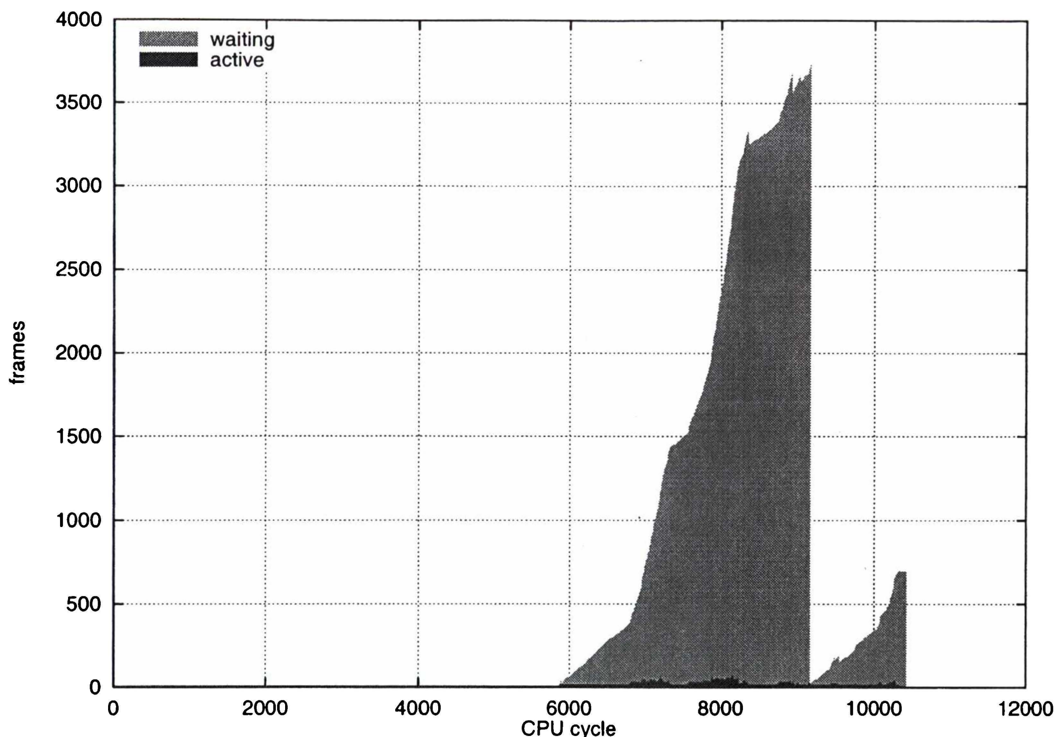


Figure 8.6: Frame usage over time for *qs2* (200).

until the parent pivot has been found. This sequentialises computation, which be seen by the very small active area in the graph. The large areas of waiting frames map to the recursive calls to the right which speculatively perform correct computation and then wait to be retired.

The active frames in *bin* (Figure 8.7) rise to a peak and then drop off at a slow rate with little peaks. After the initial peak all insertions are running in parallel. As more items are inserted into their correct places in the tree fewer insertions are rolled-back increasing the number of waiting frames. The waiting frames increase at a logarithmic rate to a relatively large peak and then drop off quickly. The large ratio of waiting to active frames can be attributed to the increasing number of independent insertions that can be performed in parallel as the tree gets larger.

avl (Figure 8.8) produces similar shaped curves to *bin*, although the ratio of active to waiting frames is greater. The extra work is due to the complex nature of the rotations taking place to maintain the balanced tree. The balancing process causes more frames to be rolled-back.

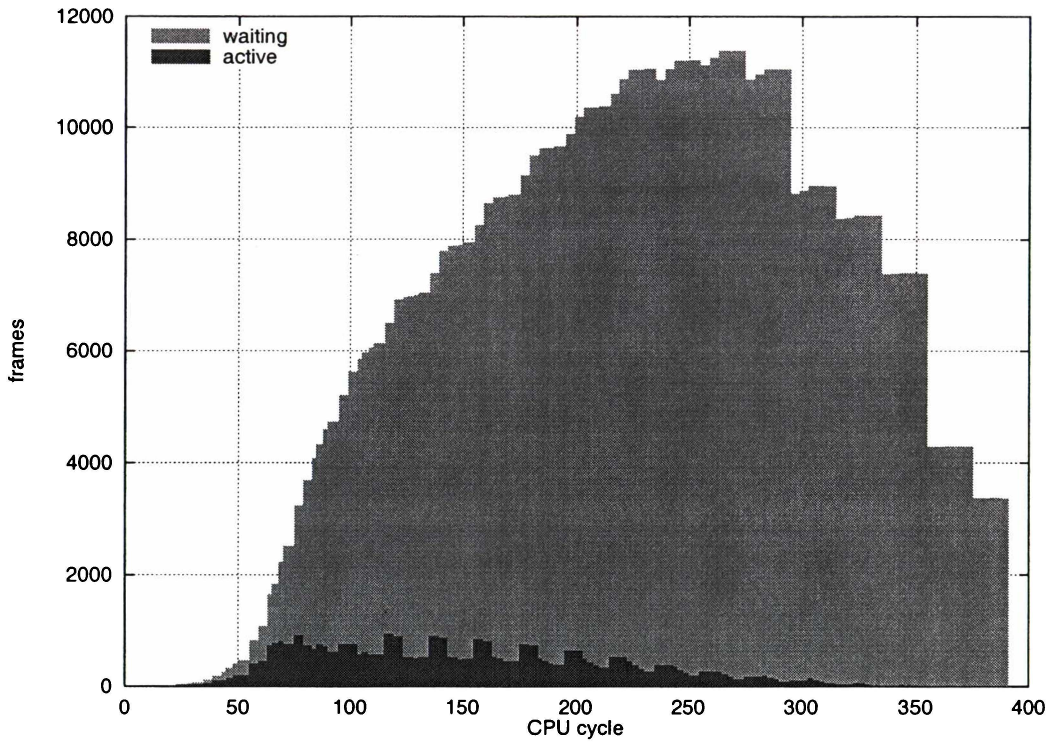


Figure 8.7: Frame usage over time for naive *bin* (500).

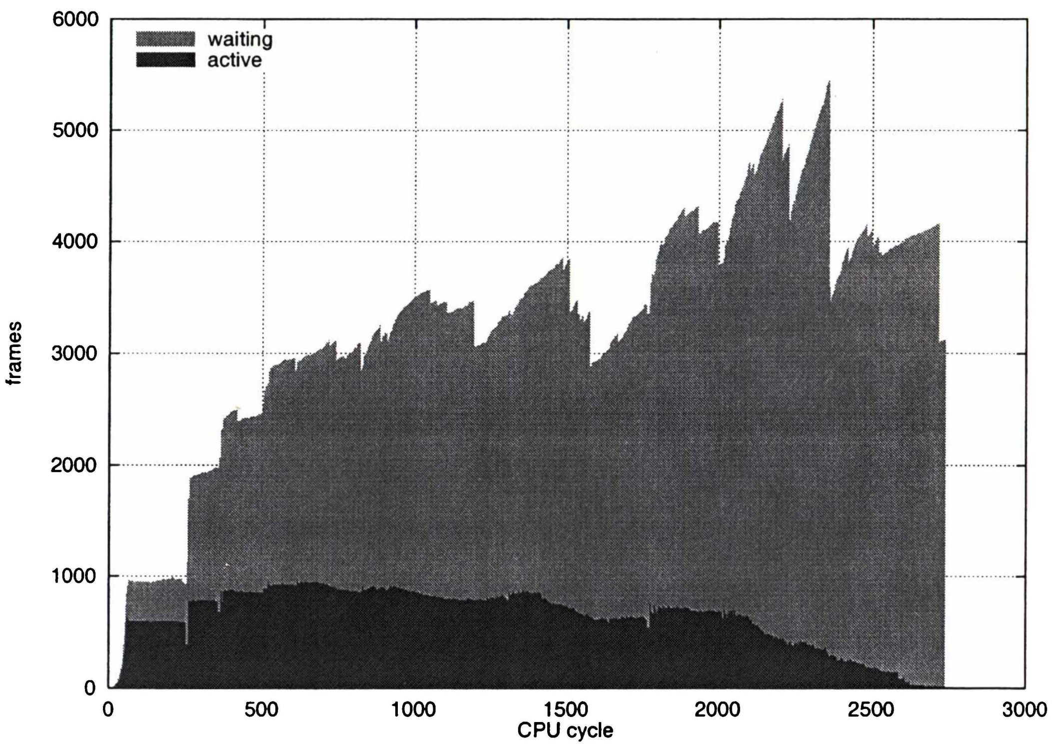


Figure 8.8: Frame usage over time for *avl* (200).

The active frame curves of *fib* (Figure 8.9) and *fibf* (Figure 8.10) look similar indicating that the same computation is performed. The overall shape of the waiting frame curves is also similar with the only difference being that *fibf*'s is 15 cycles shorter. This difference is caused by the 15 read/write dependencies of the recursive call stack which fall on the critical path. *fibf* avoids these extra cycles by passing the values produced directly to their destination registers, thus saving one instruction per result, a total of 15 in all.

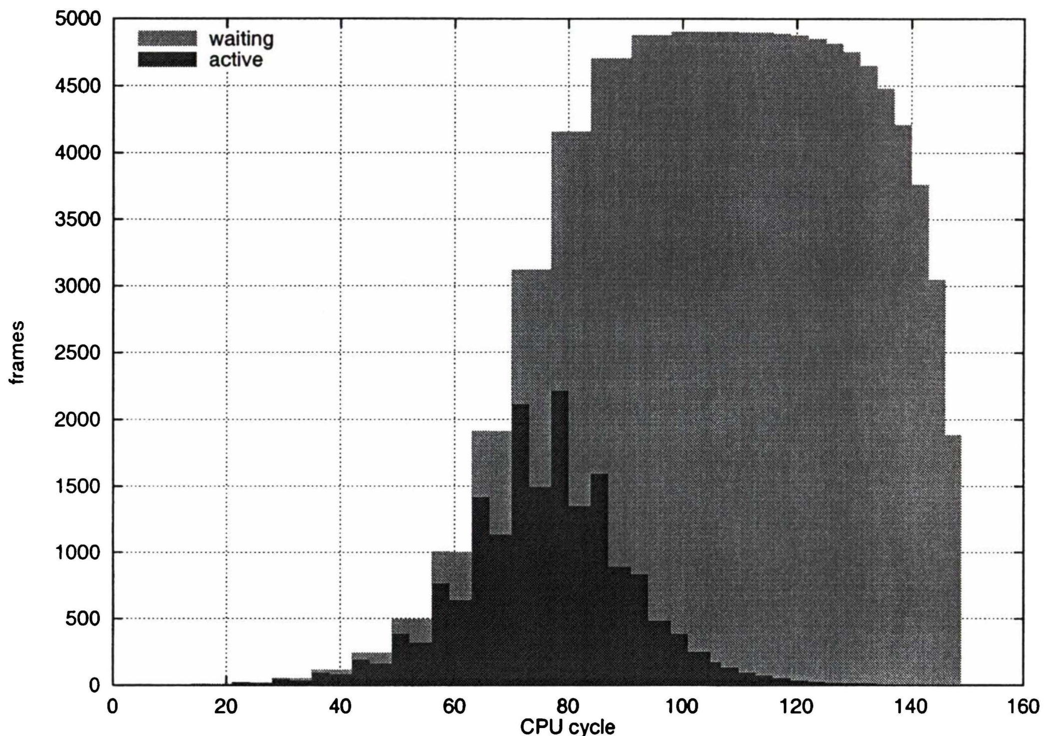


Figure 8.9: Frame usage over time for *fib* (15).

In summary the shapes of the frame usage curves is highly coupled to the algorithm and data being manipulated, with less of a dependency on the data set size. Only the frame count and number of CPU cycles alters with a change in problem size. In general the number of active frames is only a small proportion of frames that are in use at any time.

8.2.4 Limited resources

It has been shown how frames are used during execution when resource contention is not an issue. This section investigates how frame usage varies when there is a constraint on the maximum number of frames available.

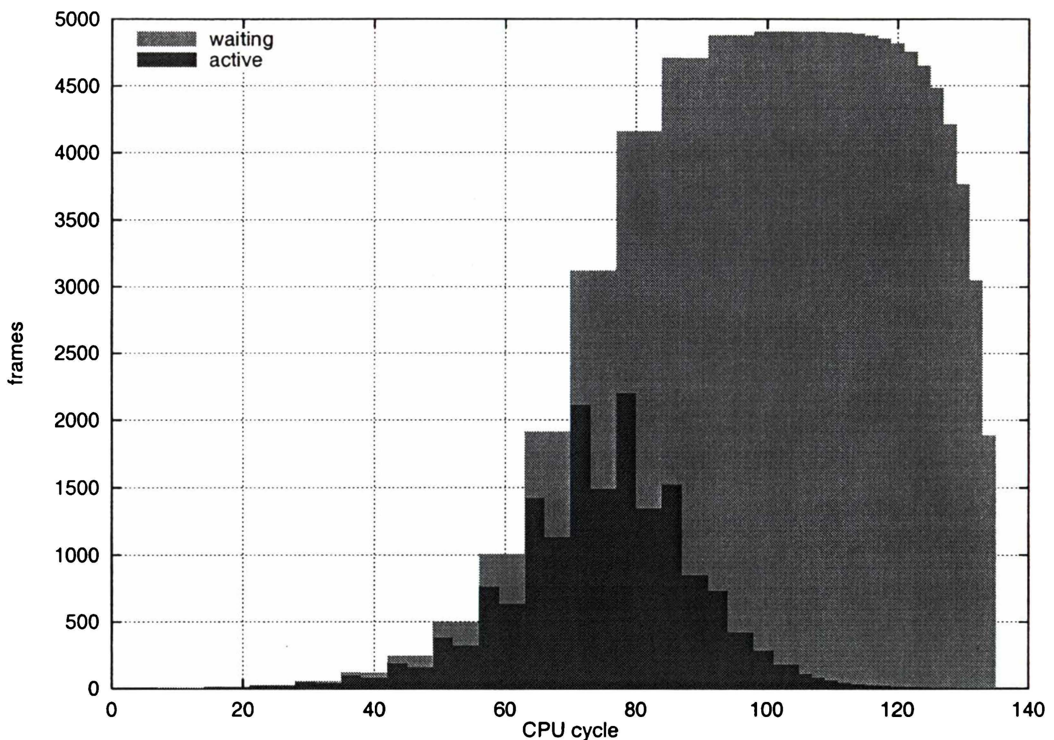


Figure 8.10: Frame usage over time for *fibf* (15).

When there are frame limits speculative blocks may get assigned a frame which is then claimed by another earlier block. The speculative block is cancelled and given another frame when one becomes available. To show this, in the following graphs the *active* region has been split into *valid* and *wasted* regions. A frame resource is *wasted* if the block is processed but for some reason is terminated rather than retired. A block of this sort does not contribute to the execution of the program because all its actions are lost.

Figures 8.11 to 8.16, and C.27 to C.30, show frame usage during the execution of the test programs when a frame limit of 1024 is set. The uncoloured regions in each graph represent the number of frames that are *unused*.

As with unlimited frames the number of active frames is generally smaller than the number of waiting frames in all of the graphs. A typical pattern that emerges is the number of frame resources that are used, either active (valid or wasted) or waiting, peaks early and then drops rapidly. Subsequently the number of used frames only approaches or reaches the limit in short bursts.

mat (Figure 8.11), *trans* (Figure C.27), and *gj* (Figure C.28) have lots of spikes in the number of frames that are active. The spikes in wasted frames indicate that control speculation is taking place but there are not enough frame resources available so some of speculative frames have to be cancelled. If more frames were available they could be put to use. The large oscillations in the total number of frames used can be attributed to the regular computation that is performed. That is, the same computation is performed multiple times in parallel with the frames of speculative blocks being freed in groups.

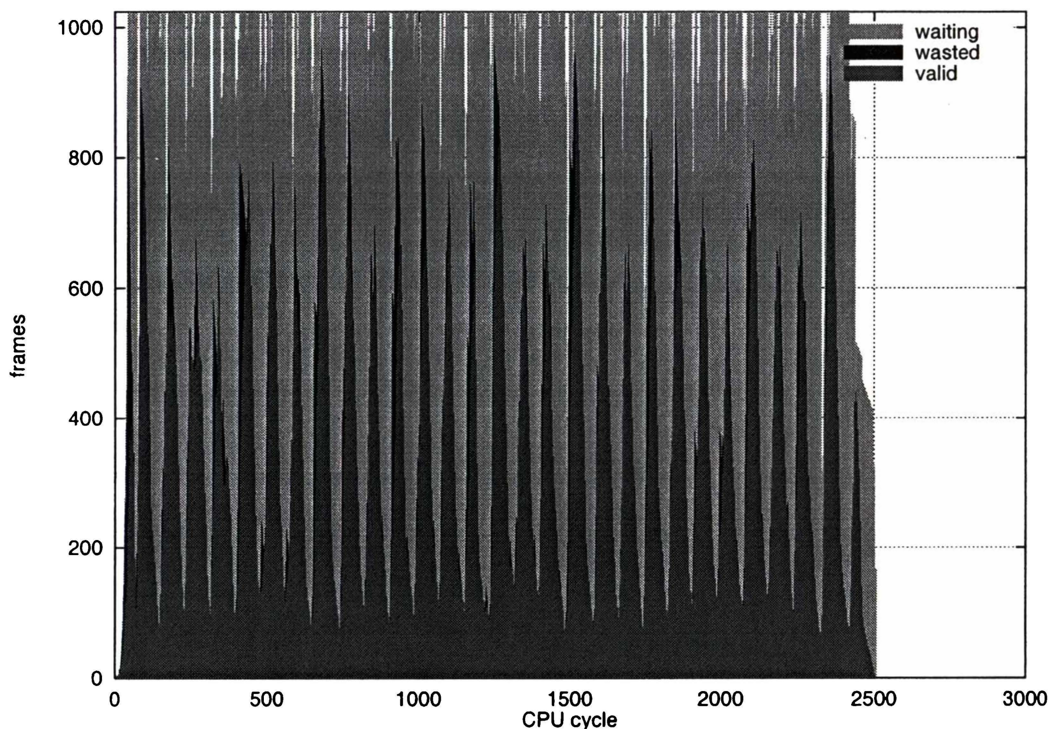


Figure 8.11: Frame usage over time for *mat* (20) when limited to 1024 frames.

The curves for *heap* (Figure 8.12) look remarkably similar to their unlimited counterparts. There is an initial peak which is wasted when resources are limited, and then a large majority of frames sit in the waiting state. Up to 4000 cycles the number of valid active frames remains relatively constant at 200 and then tapers off to zero like the unlimited case. There is only a minor increase in execution time.

In *qs1* (Figure 8.13) there are large areas of wasted frames indicating that lots of processing is not assisting computation. Computation for the right-most half of the problem is performed speculatively. The frames used for this are claimed by earlier less speculative computation resulting in the computation being wasted. In contrast there are relatively

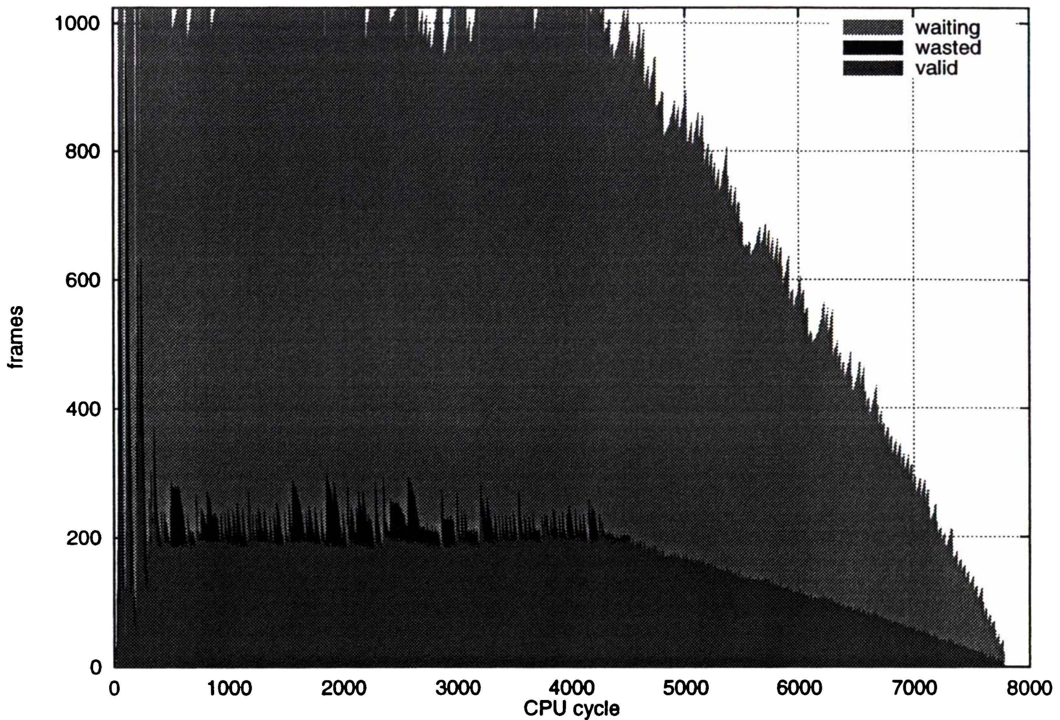


Figure 8.12: Frame usage over time for *heap* (200) when limited to 1024 frames.

few wasted frames in *qs2* (Figure 8.14). As with the unlimited resource case speculative computation cannot proceed due to the pivot selection phase. The relatively small number of wasted frames in the 6000 to 9000 cycle range indicates that not much speculative computation is lost due to the limited frames. The result is that *qs2*'s curves look very similar to the unlimited frames case.

Like the matrix manipulation programs *bin* (Figure 8.15) and *avl* (Figure C.29) contain spikes of wasted frames. In both cases the initial wasted frames spike corresponds to all the insertions starting in parallel, but then many of them are cancelled because frame resources become saturated. In *bin* the total number of frames used sits near the limit more than for many of the other programs. The oscillations apparent in *trans* and *gj* are not seen because the speculative work performed is not as regular. This means that speculative blocks are freed in a less regular manner and frames can be reassigned quickly enough to maintain maximum utilisation.

fib (Figure 8.16) and *fibf* (Figure C.30) have a similar initial shape to their unlimited frames counterparts and then have a series of similar shaped oscillations. These oscillations are present across a range of frame limits. This can be seen in Figure 8.17 which shows frame

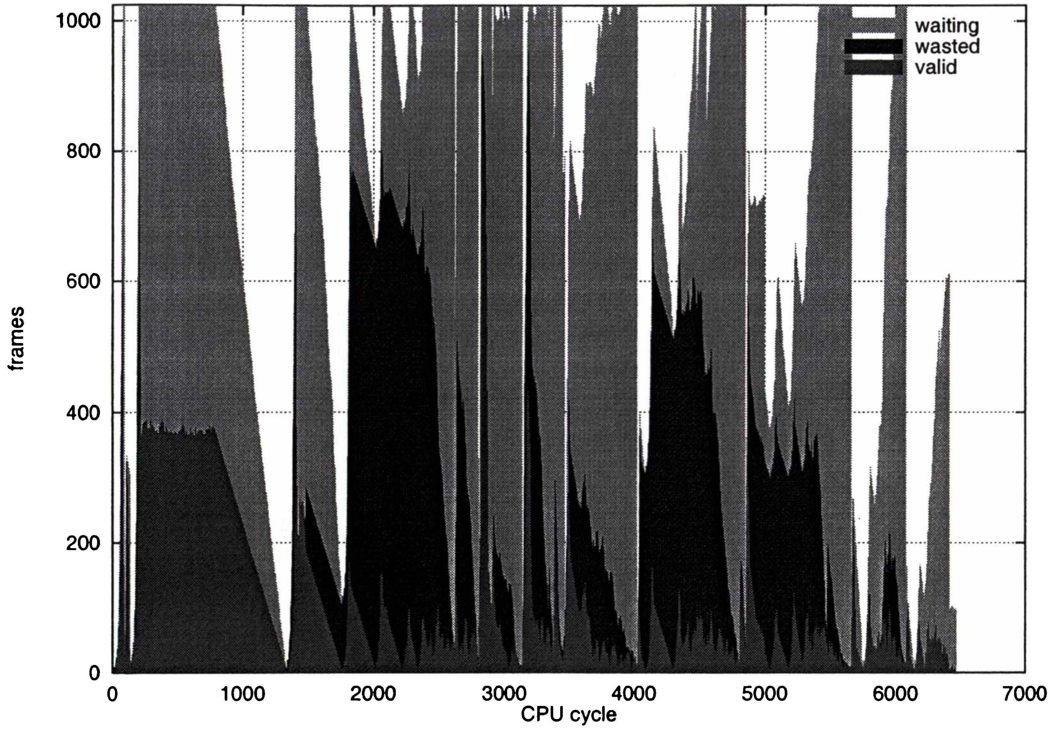


Figure 8.13: Frame usage over time for *qs1* (500) when limited to 1024 frames.

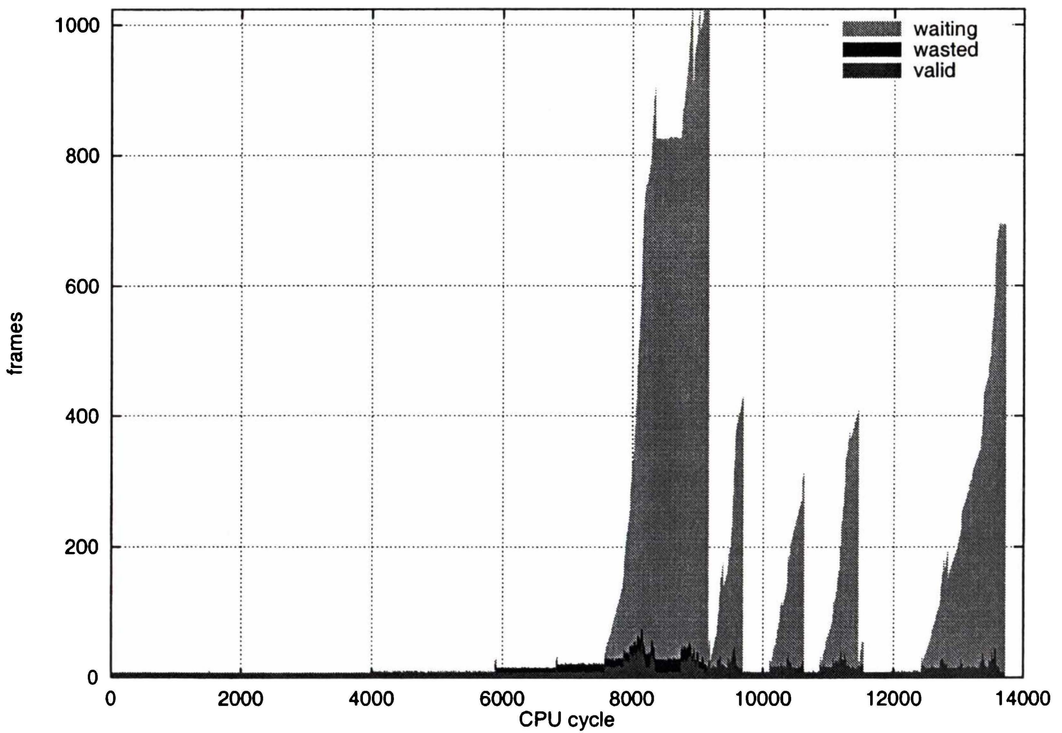


Figure 8.14: Frame usage over time for *qs2* (200) when limited to 1024 frames.

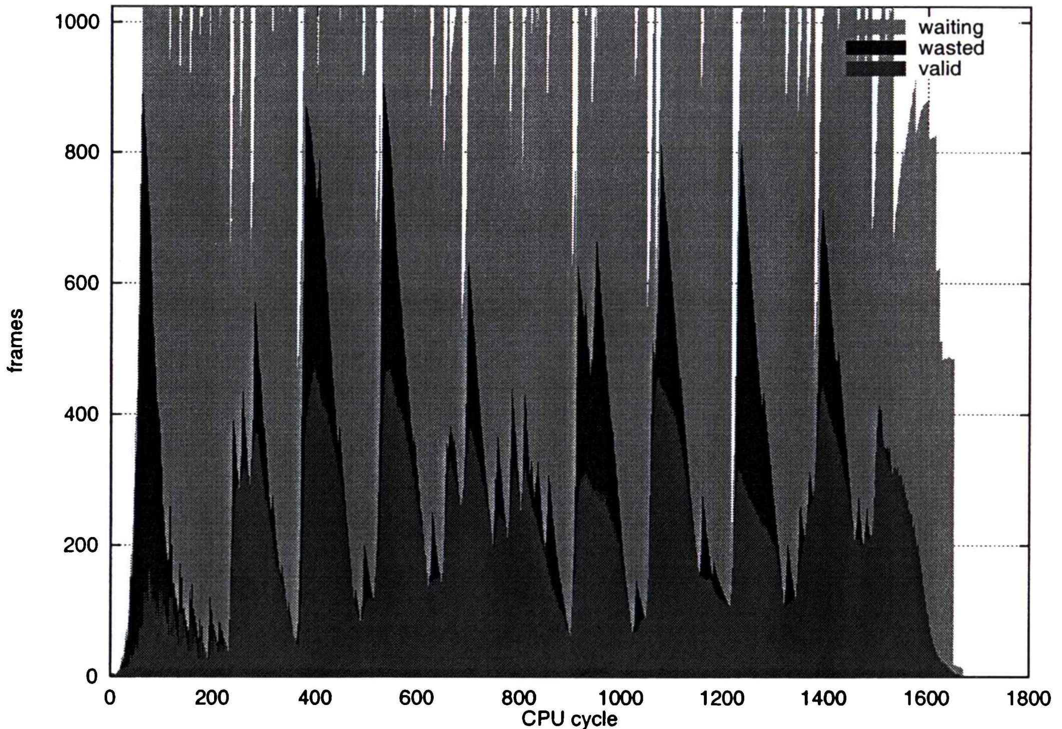


Figure 8.15: Frame usage over time for *bin* (500) when limited to 1024 frames.

usage for *fib* (17) for frame limits of 1024, 2048, and 4096. The curves at each of the three frame limits produce three initial peaks and then a series of peaks. More oscillations are present when fewer frames are available. With an increase in frame limit a greater proportion of the frames are in the waiting state. The graph approaches the unlimited frames shape as more resources are made available.

The oscillations are also present as problem size increases within a fixed frame limit. Figure 8.18 shows frame usage for *fib* with a frame limit of 1024 for problem sizes of 12, 15, 17, and 20. The frame usage oscillations present in the smaller problem sizes migrate through to the larger problem sizes. The initial spike for the problem size of 12 is present in the other plots. The same can be seen for the entire curve of problem size 15 which is visible at the beginning of the size 17 and 20 plots. This is not surprising with *fib* as larger problem sizes incorporate the computation of the smaller problems.

Similar oscillations have been observed in the matrix manipulation algorithms. However, the sorting and dynamic structure algorithms do not exhibit the same patterns with increasing problem size. This may be due to the more unpredictable nature of the computation taking place in these dynamic routines. The structured matrix routines extenuate

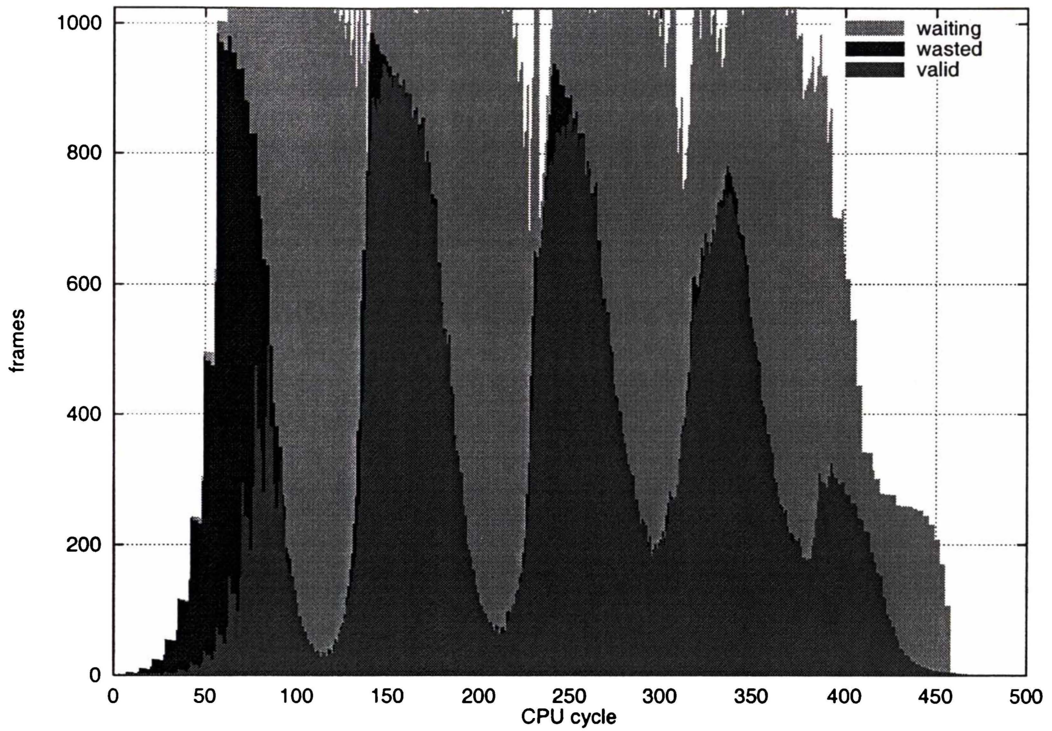


Figure 8.16: Frame usage over time for *fib* (15) when limited to 1024 frames.

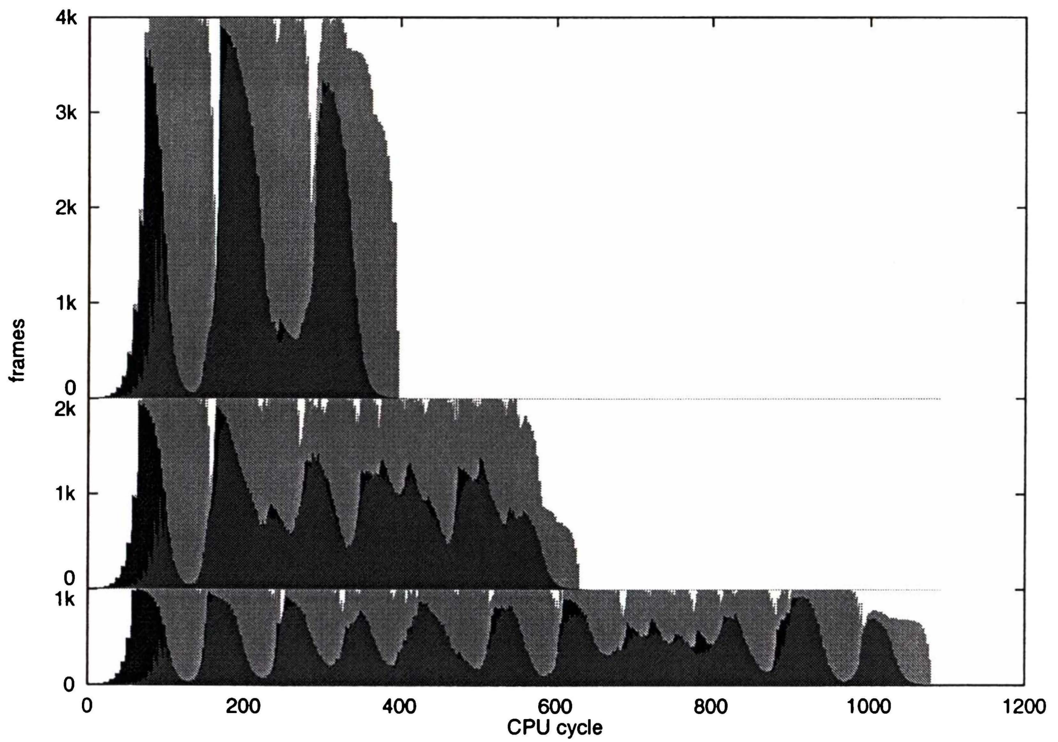


Figure 8.17: Frame resource usage over time for *fib* (17) with various frame limits.

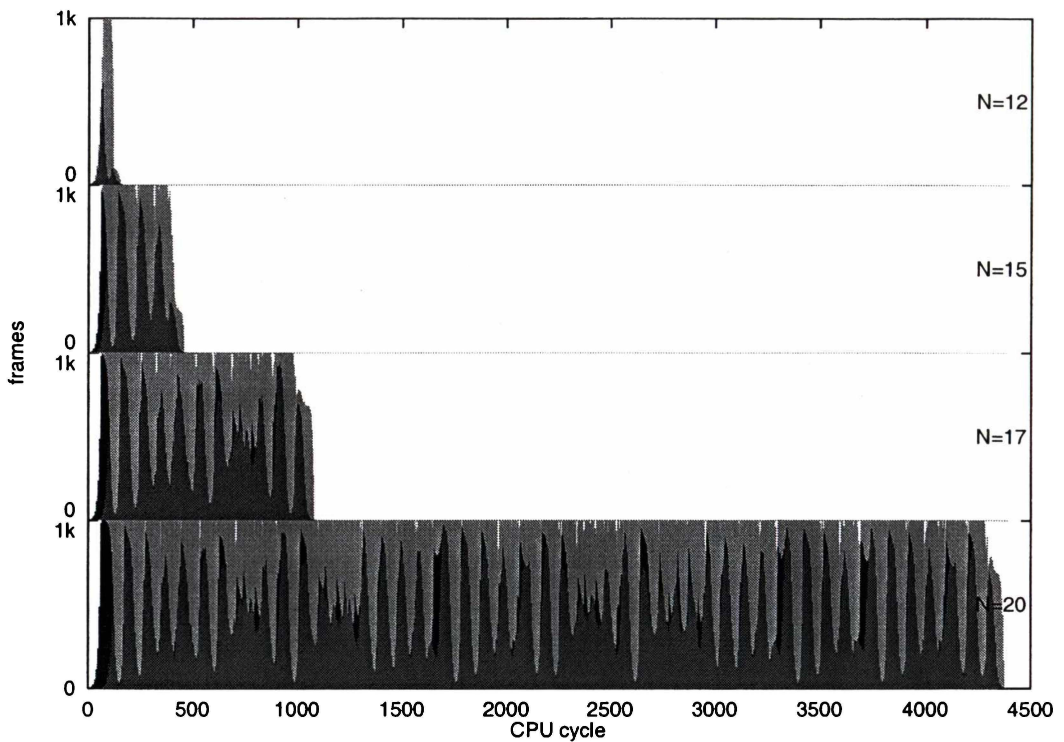


Figure 8.18: Frame resource usage over time for Fibonacci at various problem sizes when limited to 1024 frames.

computation trends by performing the same computation many times.

The efficiency with which frames are used declines as the number of frames increases. This can be seen in Table 8.2 which gives the average number of active and in-use frames for *mat* (20) over a range of frame limits. With a limit of one or two almost 100% of frames are used throughout the program's execution. As the frame limit increases the ratios drop, with only 16.4% of frames active and 70.1% of frames used at a frame limit of 1024.

Also given in Table 8.2 is the ratio of waiting to active frames for each frame limit. This ratio also increases with the frame limit, indicating that as more state saving resources are made available a greater fraction of them are taken up by frames that have finished doing useful work and are waiting in the system to be fossil collected.

In summary, frames are used less efficiently as more are made available. A greater proportion of them sit in the waiting state, preventing other blocks from using them and in turn preventing more useful work being performed. This contributes to the sub-linear performance improvements seen in Section 7.3.

frame limit	*ave. active* #	%	*ave. in use* #	%	*waiting/ active*
1	1.00	100	1.00	100	0.00
2	2.00	100	2.00	100	0.00
4	3.32	82.9	3.87	96.8	0.17
8	5.43	67.9	7.66	95.7	0.41
16	8.60	53.7	14.2	88.5	0.65
32	13.7	42.8	27.9	87.3	1.04
64	20.5	32.0	50.3	78.6	1.46
128	32.3	25.2	100	78.3	2.10
256	52.0	20.3	193	75.4	2.71
512	97.2	19.0	386	75.3	2.97
1024	168	16.4	718	70.1	3.28
2048	303	14.8	1347	65.8	3.45
4096	514	12.5	2475	60.4	3.82
8192	909	11.1	4549	55.5	4.00
16384	1658	10.1	8601	52.5	4.19

Table 8.2: Average frame utilisation for *mat* (20) over a range of frame limits.

In many cases there are large oscillations in the numbers of frames that are in use. This is caused by frames being freed in groups, and blocks not being activated quickly enough to use the freed frames instantly. This phenomena may be less prominent in a real machine as retirement of frames will not be instantaneous as has been assumed in the simulation model.

8.2.5 Code scheduling

A compiler can reorder some code, particularly nested loops, to use frames more efficiently. The independence of the inner loop to the other loops in matrix multiplication provides a straightforward example. Consider the matrix multiplication code in Figure 8.19.

```
for (i=0;i<N;i++)
    for (j=0;j<N;j++)
        for (k=0;k<N;k++)
            Z[i][j] = Z[i][j] + A[i][k] * B[k][j];
```

Figure 8.19: Code for matrix multiplication.

The inner loop contains a potentially long sequential data dependency chain. The outer two loops allow many of these long sequential data dependency chains to be executed

independently in parallel. If the inner loop (`k`) control is moved up a level (between `i` and `j`) there will be an inner loop of independent tasks. The sequential dependence chain is still there but it encompasses a group of independent tasks. These can run in parallel and not take up as many frame resources while waiting to be freed. If the `k` loop control is moved all the way out then more independent tasks can be run in parallel in the inner loops.

Figure 8.20 shows how frame resource usage varies over time for three different orderings of the loops in *mat* (20) when the frame limit is set to 1024. As the sequential data dependency chain is moved from inner to outer loops the execution time reduces. There is only a four cycle execution time improvement when moving from i-k-j to k-i-j but the is a noticeable reduction in the number of waiting frames. The freed frames allow blocks later in the virtual sequence to be assigned frames and to use processing resources earlier. At higher frame limits there is a greater improvement going from i-k-j to k-i-j indicating that for a 1024 frame limit i-k-j provides enough instruction independence to provide near maximal performance.

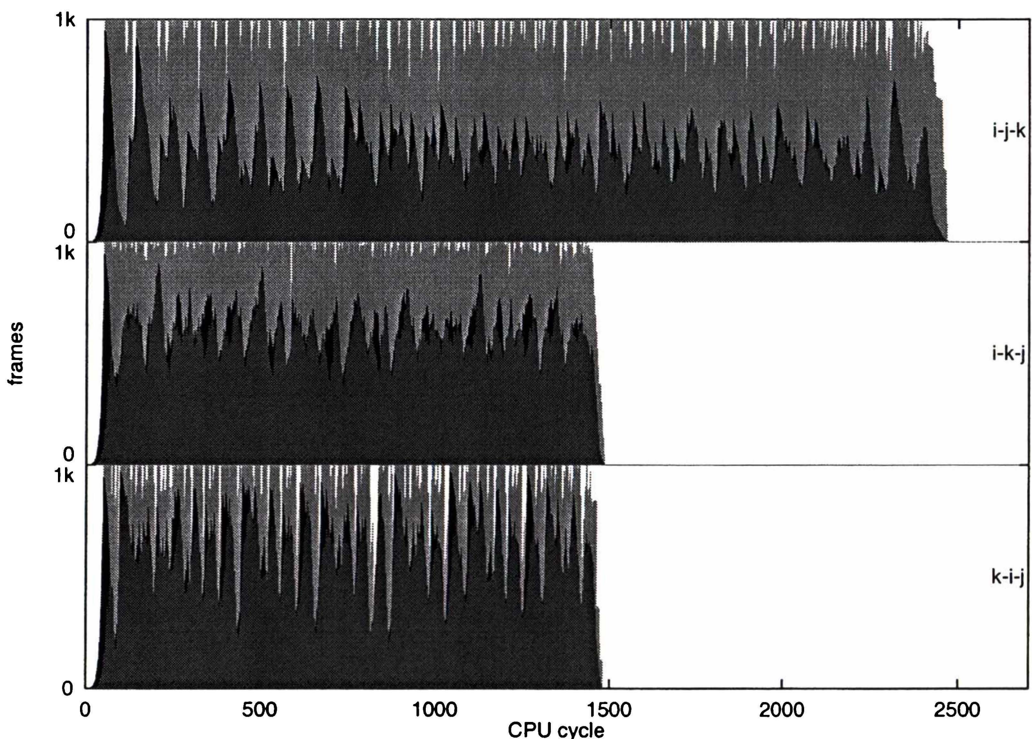


Figure 8.20: The effect of nested loop re-orderings on frame usage for *mat* (20).

This does not solve all of the wasted frames problem, but it can significantly improve

performance. Re-arranging the code so that independent events are scheduled close together in parallel allows better performance to be obtained. Long sequential paths create false retirement dependencies between state saving resources. Increasing the parallelism between steps in a sequential path by interleaving many sequential paths reduces the number of false dependencies for a given instruction window size. This is important because it allows state saving resources to be better used when they are limited.

Code could be rearranged by a compiler to maximise the independence of close instructions. Vector compiler technology is capable of detecting the situations where many independent data values have the same operations performed on them. Vector instructions are then generated to parallelise computation. These ideas can be extended and applied to the WarpEngine.

8.3 Time-space cache usage

Frames are used to store the state of speculative instructions and their operands. Similarly space is needed to store speculative memory accesses. In the WarpEngine speculative memory accesses are stored as entries in the time-space cache. The size of the time-space cache required to hold these entries will have an impact on hardware design decisions. This section examines the number of time-space cache entries during execution of the test programs.

In this section time-space cache resource usage is reported in terms of the number of entries. In the time-space cache a write entry contains an address, a timestamp, and a value, while a read entry contains an address, a timestamp, and a destination register. Slightly different information is stored for each type of entry but for the purposes of this investigation it is assumed that the same amount of space is required to store each. This means that the space required for time-space cache entries will be proportional to the total number of time-space cache entries.

8.3.1 Results

Figure 8.21 plots the number of time-space cache entries required for *mat* (20) when there are unlimited resources. This graph is representative of all programs where the number

of read entries is greater than the number of write entries and the sum of the time-space cache entries produces a curve similar in shape to the frames used curve. *fibf* is an obvious exception because there are no reads and only a single write is performed.

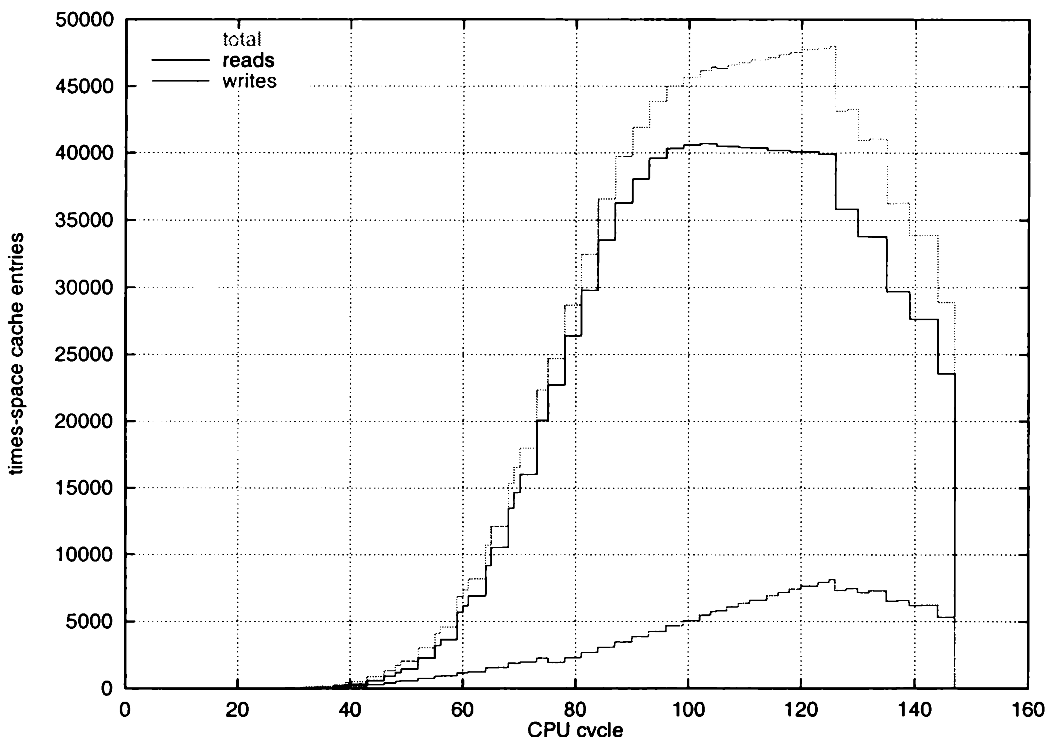


Figure 8.21: Time-space cache entries over time for *mat* (20).

The magnitude of the number of time-space cache entries differs from the number of frame resources used but the shapes of the curves are similar. This indicates that the number of time-space cache entries is a constant multiple of the number of frames in use. Figure 8.22 graphs the relative number of time-space cache entries against the number of frames in use during execution of *mat*. This gives a running ratio of time-space cache entries per frame over the duration of the program.

Although not displayed, in all of the test programs the total number of time-space cache entries is less than double the number of frames in use over the entire duration of execution. Table 8.3 lists the numbers of frames, reads, and writes for each program and problem size examined here. Also given are the average number of reads, writes and total memory accesses per frame. It can be seen for *mat* (20) the average number of reads and writes per frame closely matches the final numbers of corresponding time-space cache entries in Figure 8.22. The other programs show similar results.

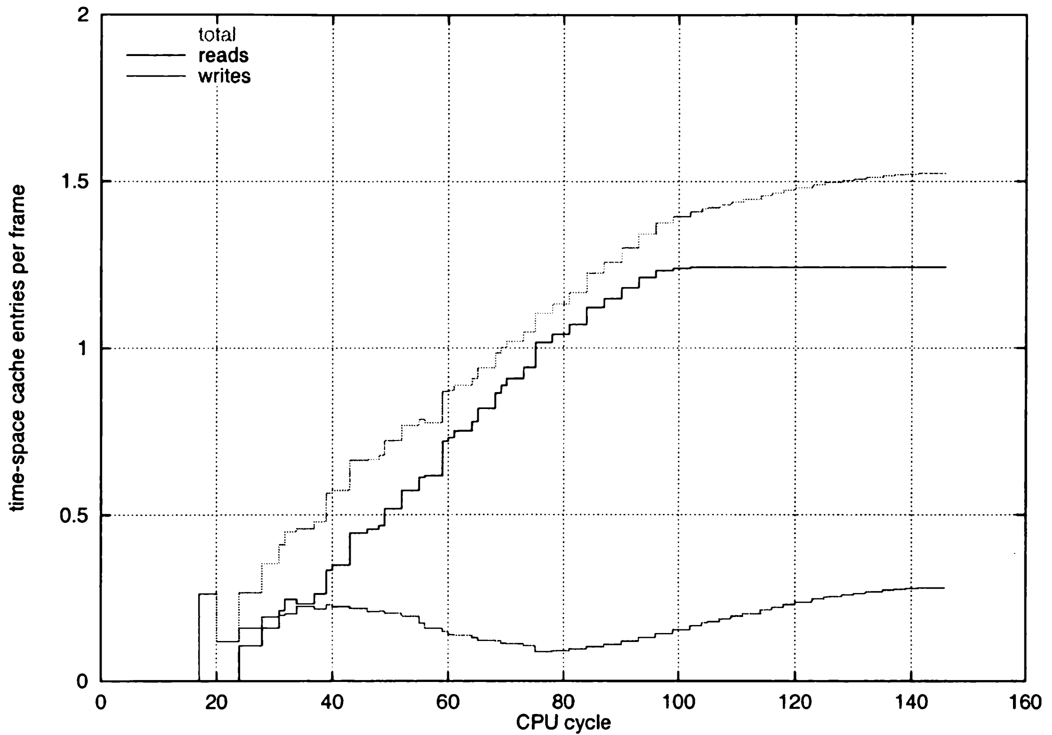


Figure 8.22: Time-space cache entries per frame for *mat* (20).

		total			*average per frame*		
problem	*size*	*frames*	*reads*	*writes*	*reads*	*writes*	*total*
mat	20	34040	41200	10040	1.21	0.29	1.51
trans	20	40420	64000	8420	1.58	0.21	1.79
gj	15	27622	32186	6013	1.17	0.22	1.38
heap	200	14300	11034	5800	0.77	0.41	1.18
qs1	500	20890	15597	7627	0.75	0.37	1.11
qs2	200	8654	14360	3130	1.66	0.36	2.02
bin	500	13235	11311	2492	0.85	0.19	1.04
avl	200	14062	13215	9894	0.94	0.70	1.64
fib	15	4933	1972	1973	0.40	0.40	0.80
				overall average	1.04	0.35	1.39

Table 8.3: Number of frames, reads, and writes used in the test programs.

It is more interesting to know how time-space cache entries vary when frames are limited. When limiting frame resources the question arises as to whether the same time-space cache ratios hold as with unlimited frames. Figures 8.23 to 8.26, and C.31 to C.35, graph the number of time-space cache entries per frame in use over each program's execution with the frame limit set to 1024.

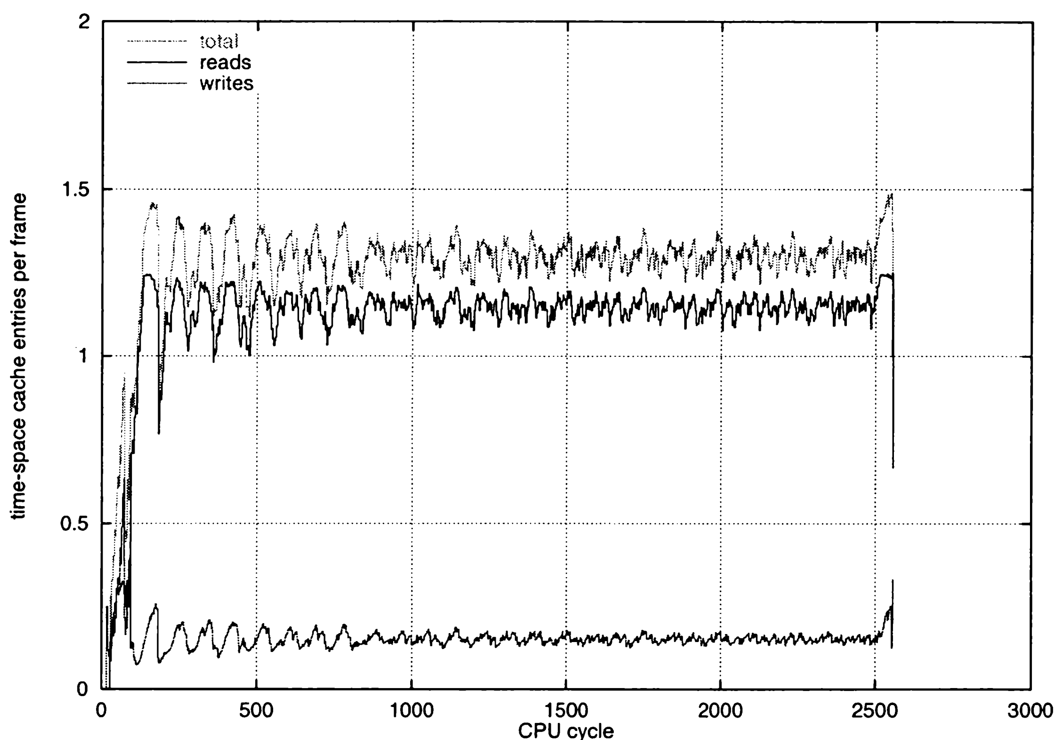


Figure 8.23: Time-space cache entries per frame for *mat* (20).

With larger frame limits the plots are smoother and their shape approaches the corresponding unbounded curves. At smaller frame limits the plots more closely resemble the distribution of memory accesses in the sequential instruction trace. That is, peaks and troughs in the ratio of time-space cache entries tend to fluctuate more and they map closely to the number of memory accesses that are in individual code blocks.

It is very rare for the ratio of time-space cache entries to exceed twice the number of frames in use. The ratios for read, write, and total time-space cache entries still sit close to their corresponding memory instruction to frame ratios of the programs as a whole. There are a few spikes which exceed 2, near the middle of *qs2* (Figure 8.24), at the beginning and end of *fib* (Figure 8.26), and *avl* (Figure 8.25) reaches 2 near the end. At these points there are several writes occurring and there are very few frames in use.

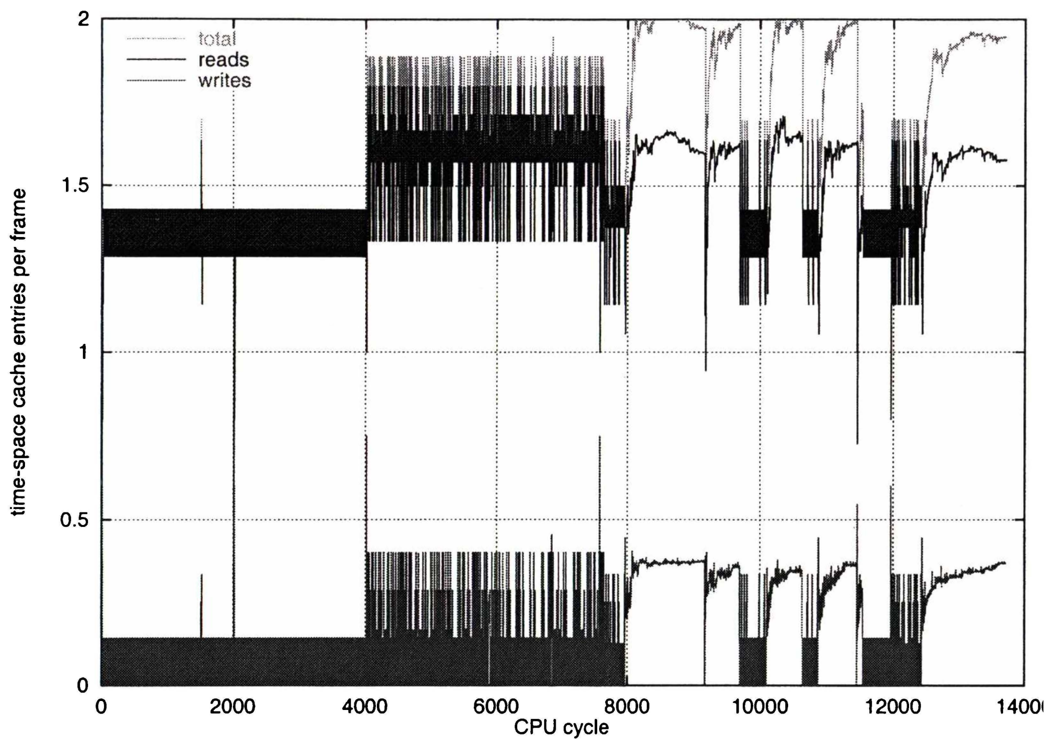


Figure 8.24: Time-space cache entries per frame for *qs2* (200).

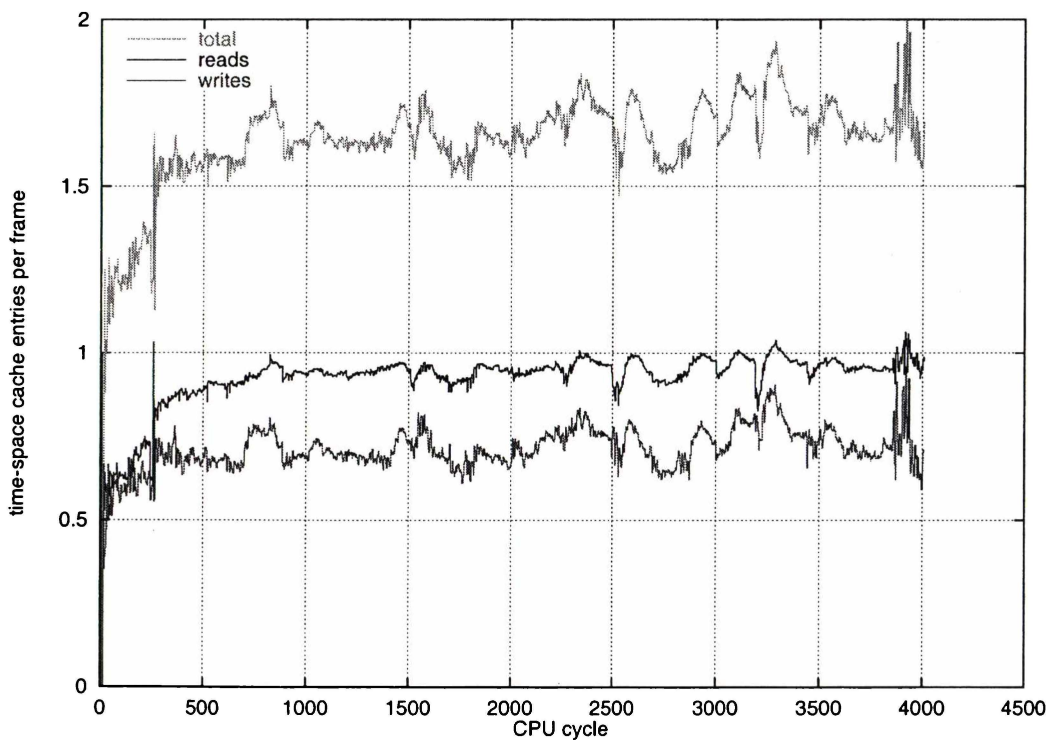


Figure 8.25: Time-space cache entries per frame for *avl* (200).

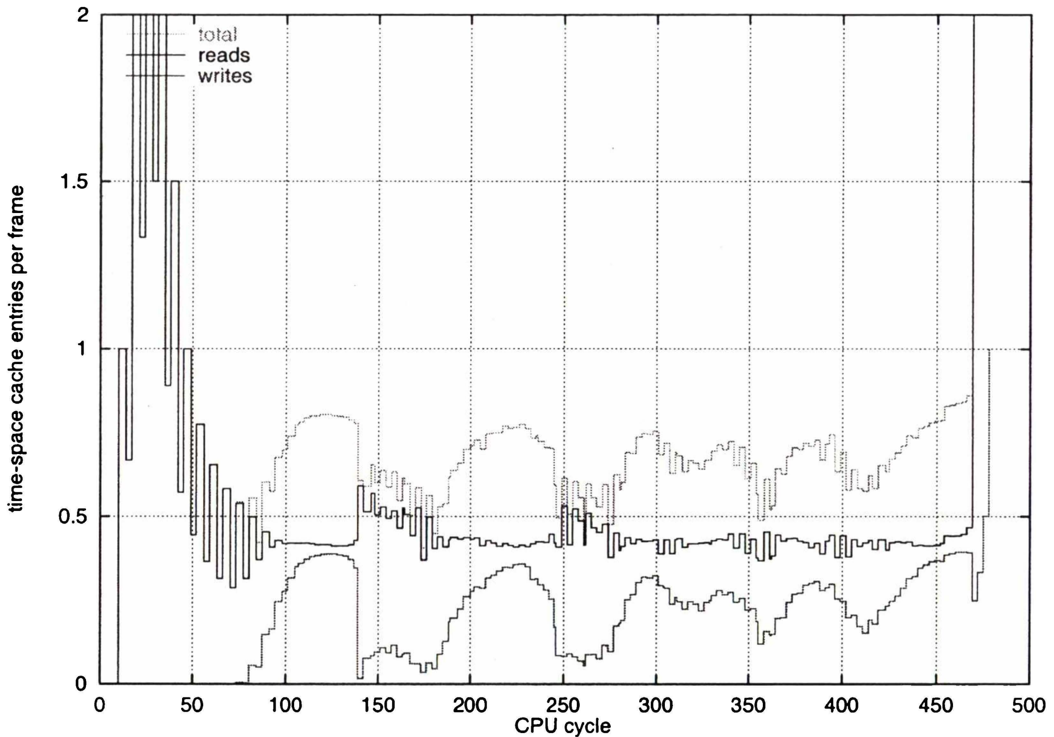


Figure 8.26: Time-space cache entries per frame for *fib* (15).

These results show that at any point during execution the space needed for the time-space cache is proportional to the number of frames that are in use. In only a few cases did the number of read and write entries exceed twice the number of frames in use. The number of read and write entries are consistent across a range of applications. With these two properties the requirements for the size of the time-space cache can be easily predicted given a maximum limit for the number of frames.

8.4 Possible solutions

In Section 7.3 a problem with long sequential paths when state saving resources are limited was discussed. With the knowledge gained in this chapter about how frames are used during execution this problem can be examined further and potential solutions to the problem can be discussed.

Sequential paths that are executed in parallel cause a build up of waiting blocks on those paths that are not holding back the retirement process. These waiting blocks prevent frames being allocated to other blocks where useful work could be performed. This

severely restricts the performance of a machine with a limited amount of state saving resources because processing resources are not used to their potential. To improve performance frames must be freed as early as possible.

This section looks at some ways to reduce the number of waiting frames so that performance can be improved in a limited frame resource environment. Possible hardware solutions include storing less state and out-of-order retirement.

8.4.1 Storing less state

One way to reduce the resource usage is to reduce the amount of state stored. In the results above it was assumed that the instructions as well as intermediate results were stored together with active logic to detect dynamic changes in the state of the frame. Instead, just the intermediate results might be stored.

The cost of this solution is that if an instruction needs to be re-executed then that instruction would need to be re-fetched and loaded into a frame. A strategy with some promise is to initially store a frame in its fully expanded form and if it has been inactive for some period of time discard part of the information and move it to less active storage. It might also be possible at this point to take advantage of the fact that about 50% of the instructions in a block are unused and record only those that are used. This is difficult because it is faster to process a frame if the hardware is kept uniform and simple. The effectiveness of this strategy would depend on correctly identifying frames that can be compacted, and minimising the number that would need to be recalled and expanded.

A further step in reducing storage usage is to discard all or some of the intermediate results in the frame. In the WarpEngine it is necessary at a minimum to record values transmitted directly from other frames. This is achieved through instructions that allow closely related frames to directly transmit data to each other without using the memory subsystem. A typical frame has between one and three such inputs so the total space for a frame might be reduced to about two words of input data, an address for the frame, and a few bits of control information. The disadvantage of this approach is that if any instruction in a frame needs to be re-executed the entire frame will need to be re-loaded and re-executed.

As well as the information in the frame itself there is space taken up in the time-space cache. The amount of information stored could be reduced by recording a single entry for all memory loads within a range of addresses, such as a record. Any write within the range of addresses causes all loads in the range to be re-satisfied. This reduces the number time-space cache entries but creates false dependencies which could cause some unnecessary re-execution of frames.

Another means of decreasing the cost, if not the size, of frames is to use cheaper slower memory, such as an off-chip cache or on-chip DRAM. When a speculative block is thought to have completed its execution the frame contents could be swapped out to memory. This incurs extra costs if the frame has to be loaded back to re-execute instructions, but the advantage is the frame resource can be assigned to another block.

8.4.2 Out-of-order retirement

Another solution is to allow frames to be retired out-of-order. This is not a trivial task, as discussed in Chapter 2. A sophisticated mechanism for detecting when a frame can no longer be squashed is required.

For a frame to be retired two conditions must be satisfied.

1. The frame should not share any memory addresses with earlier frames.
2. All data values inherited by the frame must be safe. That is, they are not subject to possible squashing and re-execution.

In some code, such as matrix multiplication, where all addresses are computed from loop indices and no code is conditionally executed it is conceivable that a mechanism capable of determining frame "safety" could be developed. However, in other problems actions such as a write via a pointer can produce unpredictable data flow preventing all subsequent frames from being retired.

Detecting safety for out-of-order retirement is essentially the same problem as detecting safety for execution in non-speculative parallel machines. For example, if all the iterations in a loop are known to be independent of one another then they can be retired independently. The advantage of this approach is that much work has been done on the

static detection of code independence, and much is known about how compilers can detect it and how programmers can assert it with pragmas. The disadvantage is that the transparency of ILP is lost, and the programmer becomes involved in the extraction of parallelism.

8.5 Summary

The management of state saving resources is critical to the performance of the WarpEngine. As more frames are made available a greater proportion get used to hold the state for speculative blocks that have been processed. On average blocks are processed earlier but have to wait longer to be retired. This explains why performance obtained is not proportional to the frame resources that are available.

The large numbers of waiting frames hinder performance because no computation can be performed by those frames. To achieve maximum performance the numbers of waiting and unused frames must be minimised. If the waiting frames can be freed from the system early greater performance can be gained. This has been shown with the reduction in waiting frames when the loops in matrix multiplication are reordered.

Code rescheduling at compile-time is one means of improving performance. Other ways of improving performance in hardware are to store less state information, to migrate waiting frames to cheaper hardware, and to retire frames out of programmed order. The design of hardware to store frames needs to consider these alternatives.

The poor utilisation of state saving resources used to perform useful work is made worse with only 50% of slots within frames used. There is a trade-off in wasted state saving space of fixed sized code blocks against the performance that can be obtained. The static arrangement of instructions in fixed sized frames is easier to implement than complex re-order buffers and allows data flow techniques to be employed providing state saving at the instruction level.

Another insight gained from this section is that the number of time-space cache entries required is proportional to the number of frame resources used. In general no more than 2 time-space cache entries are required for each frame in the system. The ratio of time-space cache entries to frame resources is independent of the number of frames available.

If implemented in CMOS a single frame requires around 55000 transistors and each time-space cache entry requires around 2500 transistors [Calvert, 1997]. The space required for time-space cache entries is less than 10% of that required for frames, meaning that frames are the limiting factor when allocating chip resources to state saving.

Chapter 9
Summary & conclusions

This thesis performed an initial evaluation of capabilities of a new computer architecture, the WarpEngine. It examined the features and limitations of a block based instruction set that employs a tree based control mechanism. With only a preliminary design of components and an untested instruction set complex components were modelled with simple propagation delays. The results obtained give a feel for performance is possible, and which components limit performance, rather than hard performance statistics.

9.1 Implications for CPU design

Simulation of the WarpEngine has shown that a speculative architecture can extract high levels of parallelism. It has been shown that parallelism is available in a range of programs including those that traverse deep and complex data structures that are not easy to explicitly parallelise. To obtain high levels of parallelism false control and data dependencies must be eliminated. This can be achieved by speculating on the outcomes of control decisions and the values of data in memory.

The virtual ordered simulation model has provided insight that is not easily obtained using traditional simulation techniques. Using this simulation model the limitations of components in contemporary architectures have been examined. This work has shown that the areas critical to obtaining good performance are: scheduling large numbers of instructions; accessing the memory system; and the use of state saving resources.

To obtain large amounts of parallelism many instructions must be processed in parallel. A distributed control mechanism that schedules control independent instructions in parallel is required. A linear control mechanism, such as a program counter, is not sufficient

to schedule enough instructions to extract large amounts of parallelism. A distributed control mechanism scales well because scheduling of code at control independent points is not dependent on the accuracy of branch prediction mechanisms.

Memory operations, on average, form 35% of the instructions on a program's critical path, almost double the proportion of memory operations of the program as a whole. Small variations in the average processing latency of memory reads have a significant impact on program execution time and performance. When attempting to obtain larger levels of parallelism reducing memory access latencies becomes more important. It has been shown that memory accesses must be allowed to occur out-of-order. This helps to reduce average memory latencies, breaks false data dependencies, and allows greater numbers of instructions to be speculatively executed.

The use and management of state saving resources is also critical to performance. The amount of space needed for saving the state of speculative instructions grows more quickly than the parallelism that can be extracted. Long control and data dependencies between instructions force later instructions that have successfully executed to wait for earlier ones to retire before they can retire themselves. In some cases this phenomenon leads to the paradoxical result that increasing the size of a problem can lead to worse performance.

To gain maximum performance using a fixed set of state saving resources the time instructions spend waiting to be retired must be minimised. That is, state saving resources must be freed as early as possible when instruction have completed their execution. Possible hardware and software solutions include: storing less state information which allows more instructions to be stored in a given amount of space; out-of-order retirement which frees state saving resources early; and code rearrangement to schedule independent instructions close to each other. This work has shown that it is clearly important to investigate these and other alternatives.

9.2 The WarpEngine

The components of the WarpEngine provide potential solutions to problems associated with speculative execution. In the WarpEngine frames and the time-space cache provide

an alternative to the re-order buffers of contemporary architectures. Speculation is used to eliminate false dependencies between instructions and true dependencies are maintained by rolling back and re-executing instructions when violations are detected.

Dynamic re-order buffers and dependence analysis hardware are replaced by statically arranged instruction buffers and hardware which has the ability to detect changes in operand values. This scales well because data flow processing is limited to a statically arranged fixed size group of instructions within frames and the frames themselves are processed independent of one another. Instruction control flow is distributed using the tree-based control mechanism with control independent instructions scheduled in parallel. It has been shown that this mechanism is sufficient to provide the numbers of instructions needed to obtain large levels of parallelism.

The time-space cache allows memory operations to occur in any order promoting data speculation. It has been shown that the size of the time-space cache is a fixed and small proportion of frame size meaning that the number of frames is the limiting factor when allocating chips resources to state saving.

Using the WarpEngine architecture it is possible to build re-order buffers with thousands of instructions and to use them to extract significant amounts of ILP. But to do this efficiently and effectively requires careful attention to the issue of compact storage of saved state.

9.3 Conclusions

The results of this thesis show a gain performance an order of magnitude greater than that obtained by executing instructions sequentially. To achieve this level of performance is has been shown that:

1. Speculating on the outcome of branch decisions and value of data in memory can be used. These forms of speculation help to eliminate complex control and data dependence analysis in the instruction re-order buffer allowing a greater number of instructions to be scheduled in parallel.

2. A distributed control flow mechanism provides a solution to reducing false con-

trol dependencies. The tree-based control mechanism of the WarpEngine generates many independent control paths which can be processed in parallel.

3. The memory system should allow accesses to occur out-of-order to eliminate false data dependencies. To complement out-of-order accesses a timestamped memory hierarchy can be employed to ensure true data dependencies are maintained.

It has been shown that the use and management of state saving resources is critical to efficiently performing large scale speculative execution. In particular, the in-order retirement of instructions and committing data to memory can severely restrict performance in code where long sequential paths are prevalent. The issues associated with state saving resource retirement need to be investigated further.

This work has also shown that the theory behind the WarpEngine provides the mechanisms to extract large amounts of instruction level parallelism. The WarpEngine architecture provides solutions to many of the issues faced with any form of speculative processing, although, it is a question as to whether some components can be implemented efficiently.

Many of the results obtained here were gathered using the fresh insights gained using a virtual ordered simulation model. This simulation model proved valuable in that it aided in the processes of determining and refining the WarpEngine instruction set, an important part of architecture design. Fast simulation runs are obtained because simple constants are used to model complex instructions propagation delays and issues such as bandwidth and resource contention are not modelled accurately. The trade-off is that the statistics produced only give an indication of the performance that can be obtained.

9.4 Future work

To enhance the results gathered here more accurate modelling of resource overheads, contention and communications is needed. An interesting question that arises is how much extra memory traffic is generated with speculative loads and stores taking place? There could be a reduction or increase in the number of writes and reads. This cannot be determined with the current simulation model because the memory system model is not detailed enough, and the time-space cache is not considered a separate entity to the rest of

the memory system. Also, the effects of bandwidth restrictions on memory accesses must be examined. Different queuing policies could have varying impact on performance.

Detailed design and modelling of: the time-space cache and memory hierarchy; state saving management, cancelback and retirement mechanisms; and inter-frame communications is required. Different strategies need to be examined to determine their impact on performance. These issues are best investigated with a real-time ordered simulation model as the effects of concurrent processing need to be observed.

Timestamps are an integral part of the WarpEngine meaning that efficient timestamp schemes need to be developed and tested. In hardware a fixed amount of space will be assigned to each timestamp meaning that timestamp rescaling will have to take place. The effects that rescaling will have on performance must also be examined. This area is being examined in the PhD thesis of David McWha at the University of Waikato.

The test programs used may be considered "toy" benchmarks because they are small in size. More complex "real world" problems and standard benchmark test suites must be examined to verify the results obtained here. A compiler is required because these programs are typically large.

Appendix A
WarpEngine Instruction Set

This appendix contains the WarpEngine instruction set. This is version 2 (as of November 2, 1995 [Cleary, 1995b]) 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 mi ds`	Execute child c at address (a1), using a2 as migration hint (0 is a null that supplies no hint). mi specifies how migration is to be treated: 00 - don't migrate; 01 - migrate only if no free frames on local machine; 10 - migrate optionally ; 11 - always migrate to different machine. 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`	`(sop d)x3`	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 the C source code for the test programs used throughout this thesis. The C code and corresponding WarpEngine assembler code can be found at `www.cs.waikato.ac.nz/timewarp/wengine/testcode/` [Littin, 1999b].

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;
        }
}
```

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]));
}
```

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);
}
```

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("singular matrix\n");
            exit(1);
          }
    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;
      }
}
```

Heapsort [Press, 1992]

```
int table[TABLE_SIZE];

void Swap(int *a, int *b) {
    int temp;
    temp = *a; *a = *b; *b = temp;
}

void Heapify(int i, int j) {
    if ((RC(j) >= i) && (table[RC(j)] <= table[LC(j)])
        && (table[RC(j)] < table[j])) {
        Swap(&(table[j]),&(table[RC(j)]));
        Heapify(i, RC(j));
    }
    else if ((LC(j) >= i) && (table[LC(j)] < table[j])) {
        Swap(&(table[j]),&(table[LC(j)]));
        Heapify(i, LC(j));
    }
}

void InitialiseHeap() {
    int i;
    for (i=0;i<TABLE_SIZE;i++) {
        Heapify(0,i);
    }
}

void HeapSort() {
    int i;
    InitialiseHeap();
    for (i=0;i<TABLE_SIZE-1;i++) {
        Swap(&(table[i]),&(table[TABLE_SIZE-1]));
        Heapify(i+1,TABLE_SIZE-1);
    }
}
```

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);
    }
}
```

Naive binary tree insertion

```
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]

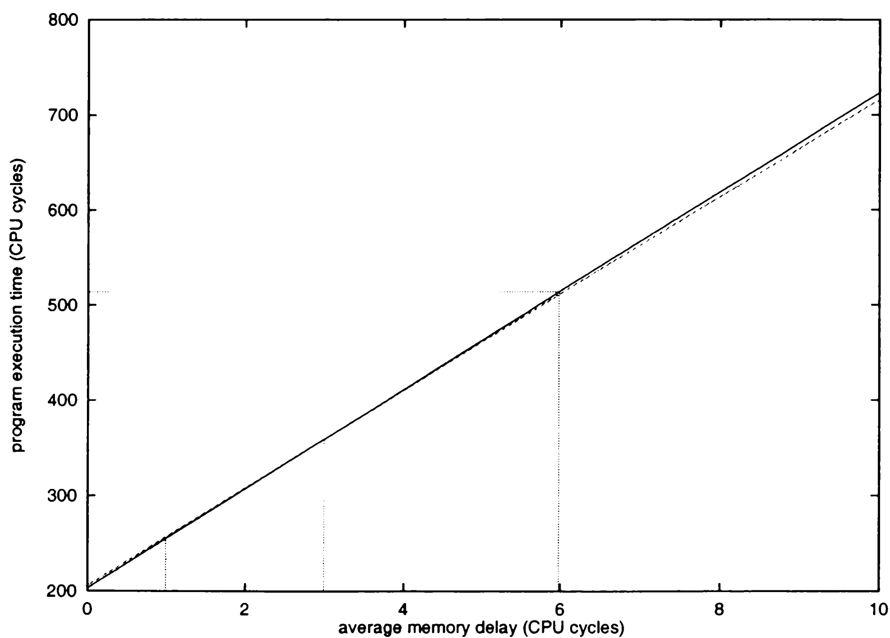
```
struct node {
    int key;
    int bal;
    node *left, *right;
};

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);
    }
  }
}
```

Appendix C
Extra graphs

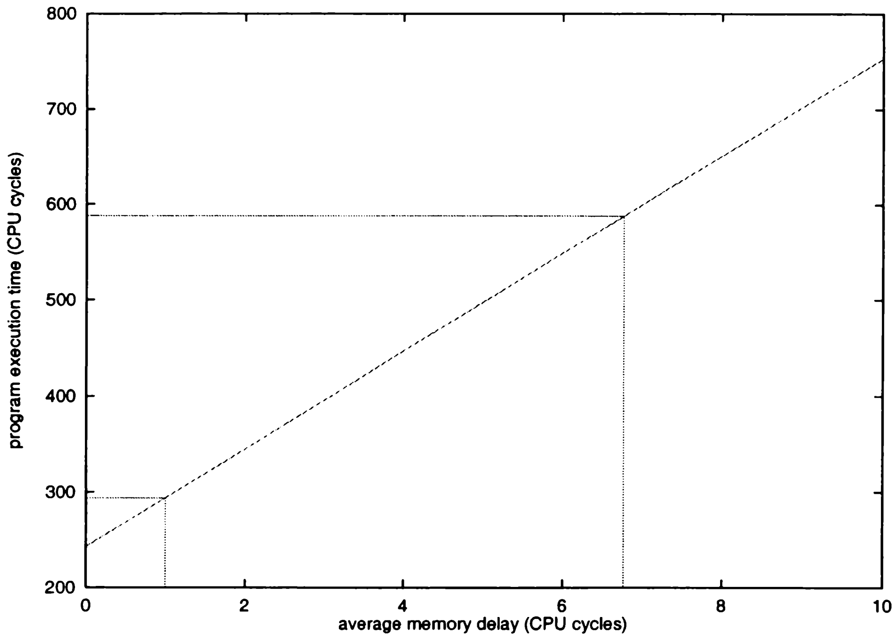
This appendix contains graphs for the test programs that were not displayed in Chapters 6 to 8.

C.1 Critical path equations



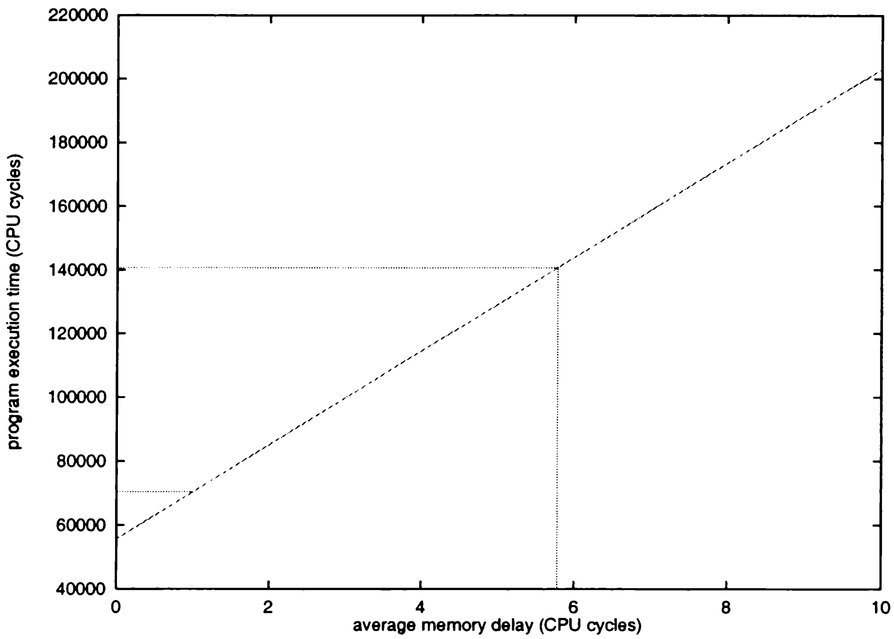
equation	*range*
$51x + 206$	$[0, 3)$
$52x + 203$	$[3, \infty)$

Figure C.1: Critical path equations for *mat* (50).



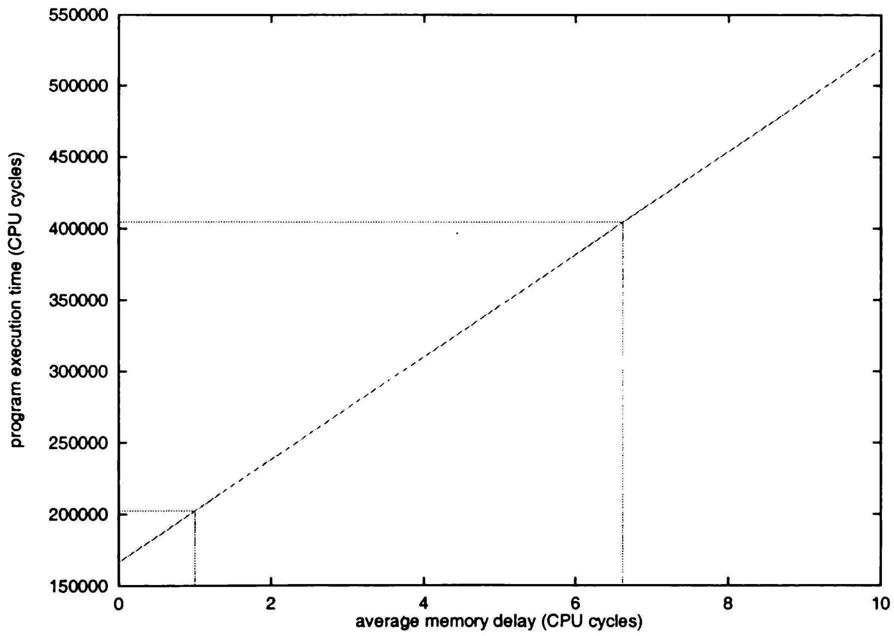
equation	*range*
$51x + 243$	$[0, \infty)$

Figure C.2: Critical path equations for *trans* (50).



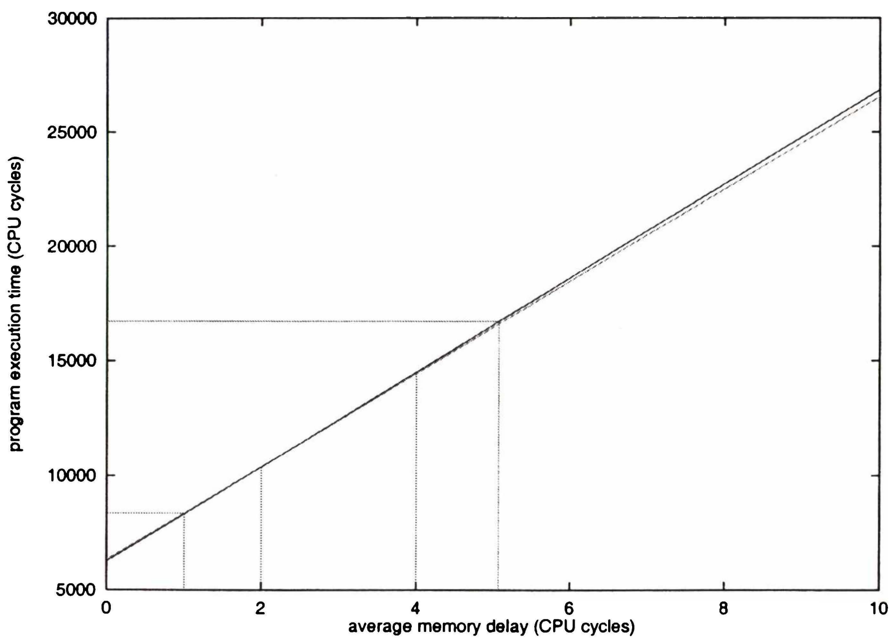
equation	*range*
$14733x + 55672$	$[0, \infty)$

Figure C.3: Critical path equations for *heap* (2000).



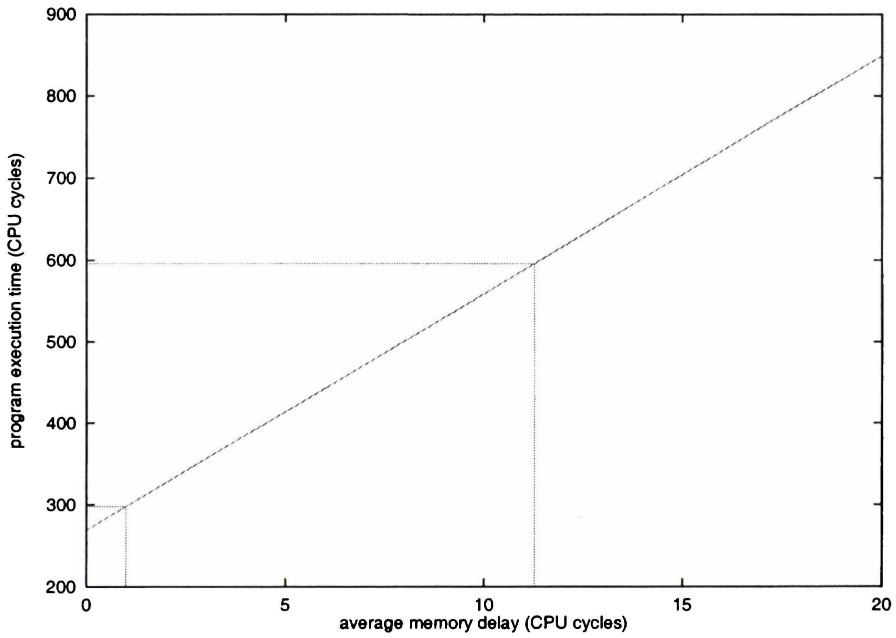
equation	*range*
$35927x + 166482$	$[0, \infty)$

Figure C.4: Critical path equations for *qs2* (2000).



equation	*range*
$2022x + 6337$	$[0, 2)$
$2059x + 6263$	$[2, 4)$
$2060x + 6259$	$[4, \infty)$

Figure C.5: Critical path equations for *bin* (2000).



equation	*range*
$29x + 269$	$[0, \infty)$

Figure C.6: Critical path equations for *fib* (30).

C.2 IPC with unlimited resources

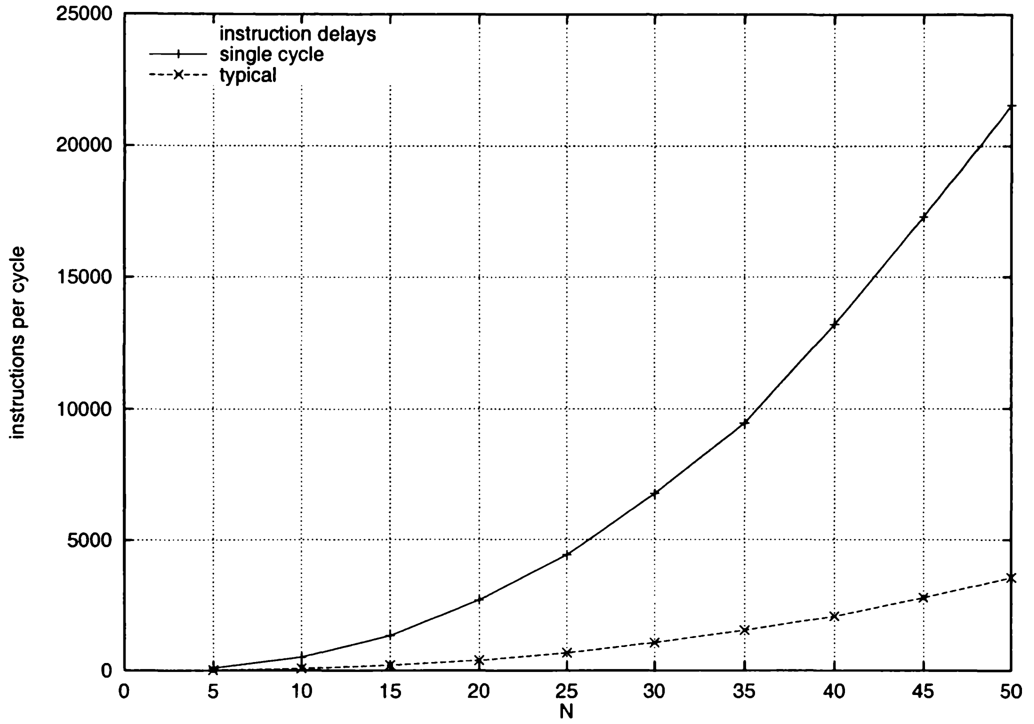


Figure C.7: IPC vs problem size for *trans*.

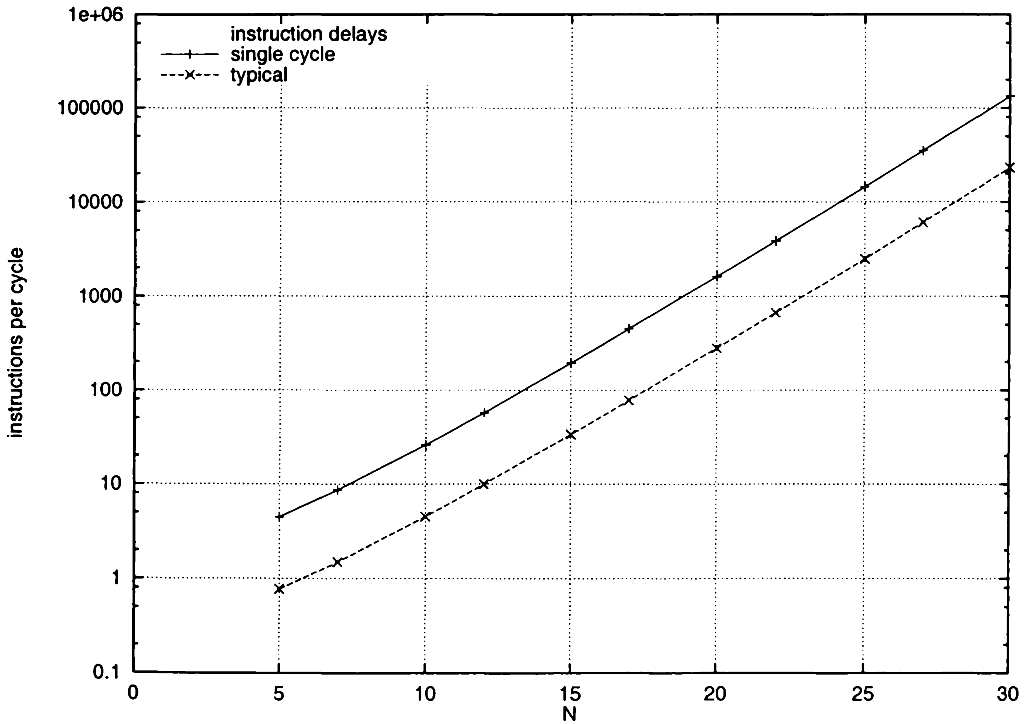


Figure C.8: IPC vs problem size for *fibf*.

C.3 IPC with memory access constraints

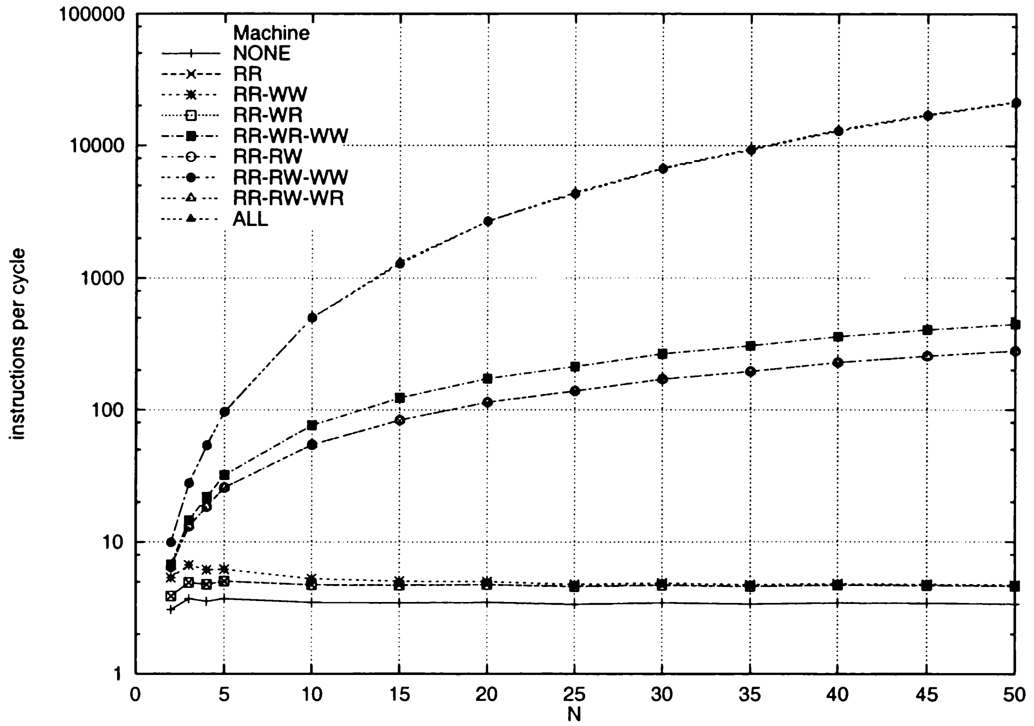


Figure C.9: IPC vs problem size with memory ordering constraints in place for *trans*.

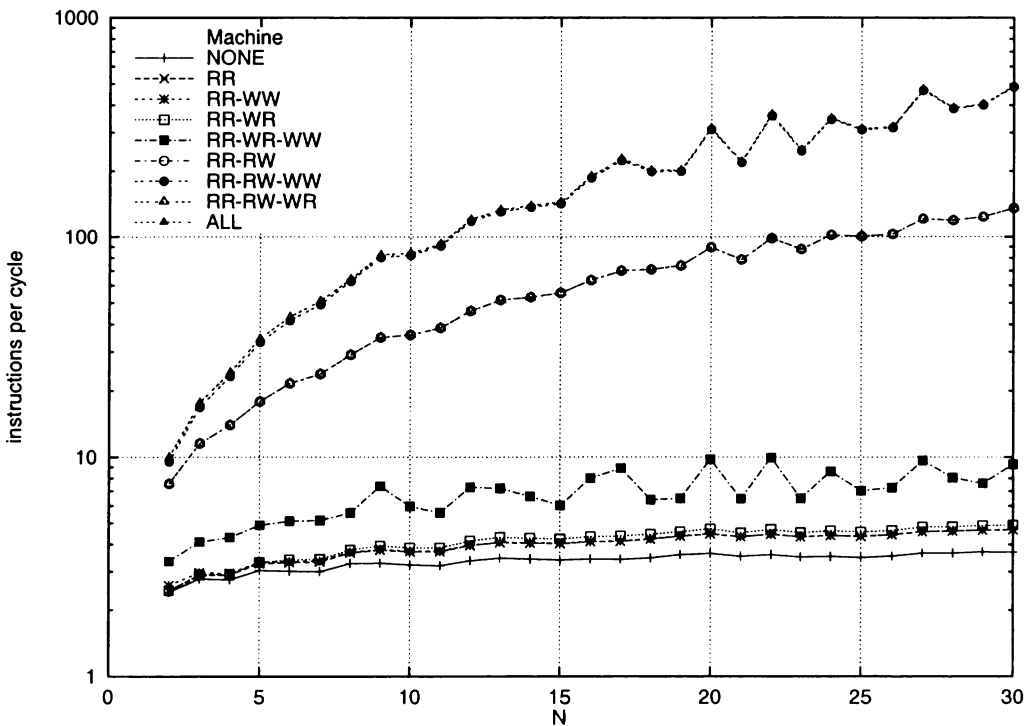


Figure C.10: IPC vs problem size with memory ordering constraints in place for *gj*.

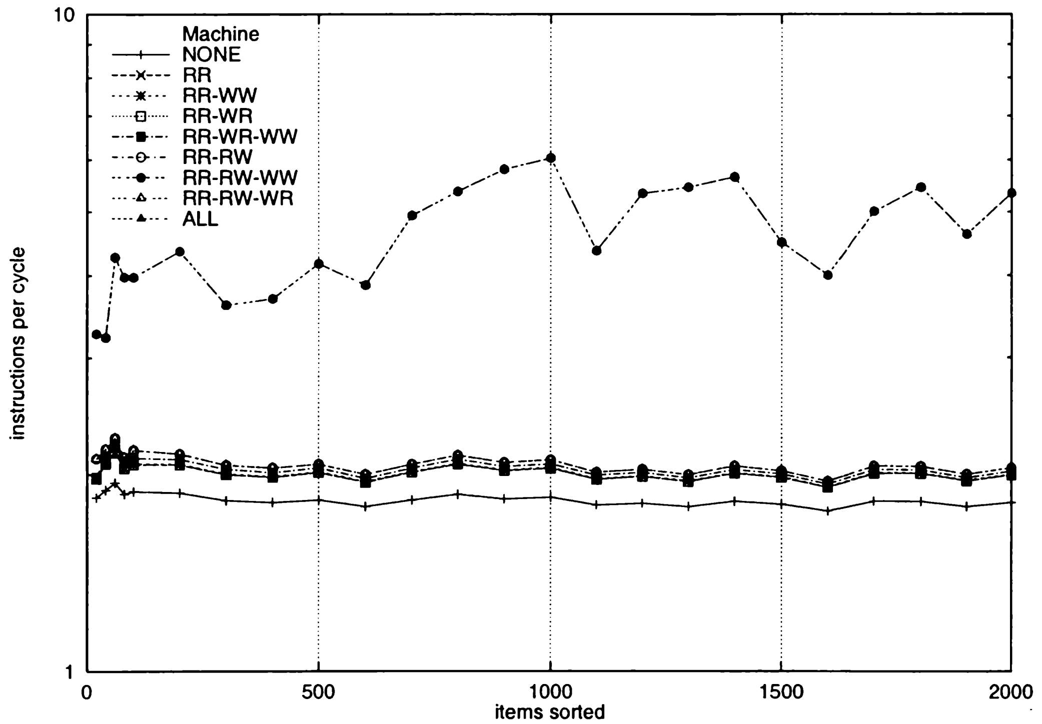


Figure C.11: IPC vs problem size with memory ordering constraints in place for *qs2*.

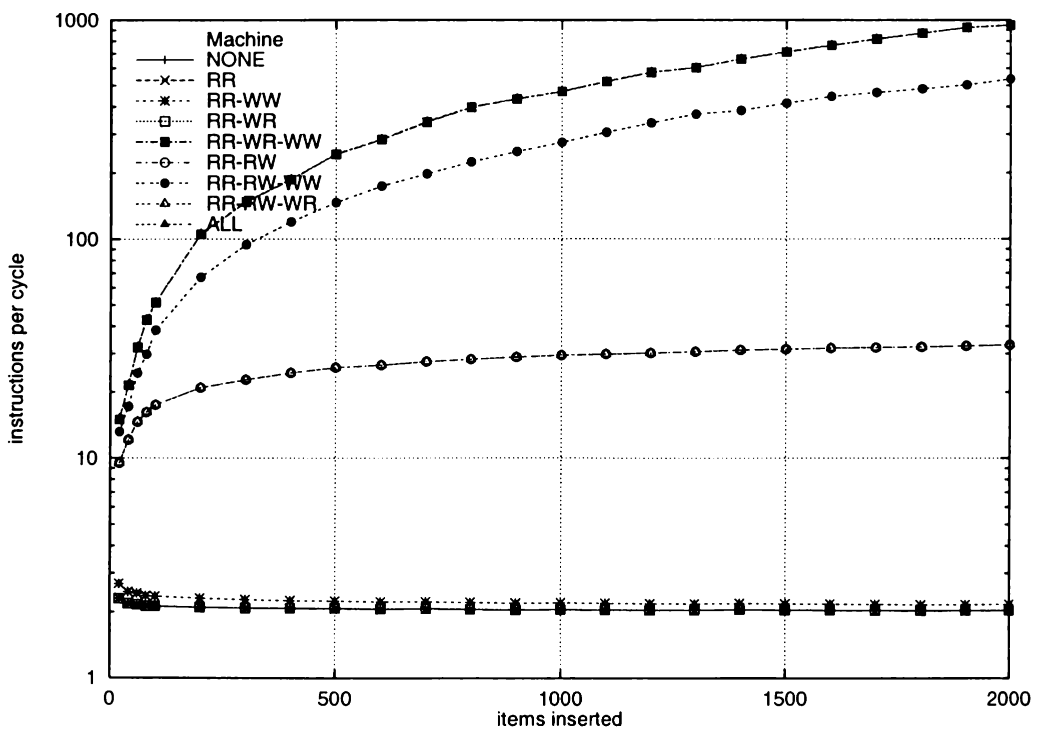


Figure C.12: IPC vs problem size with memory ordering constraints in place for *bin*.

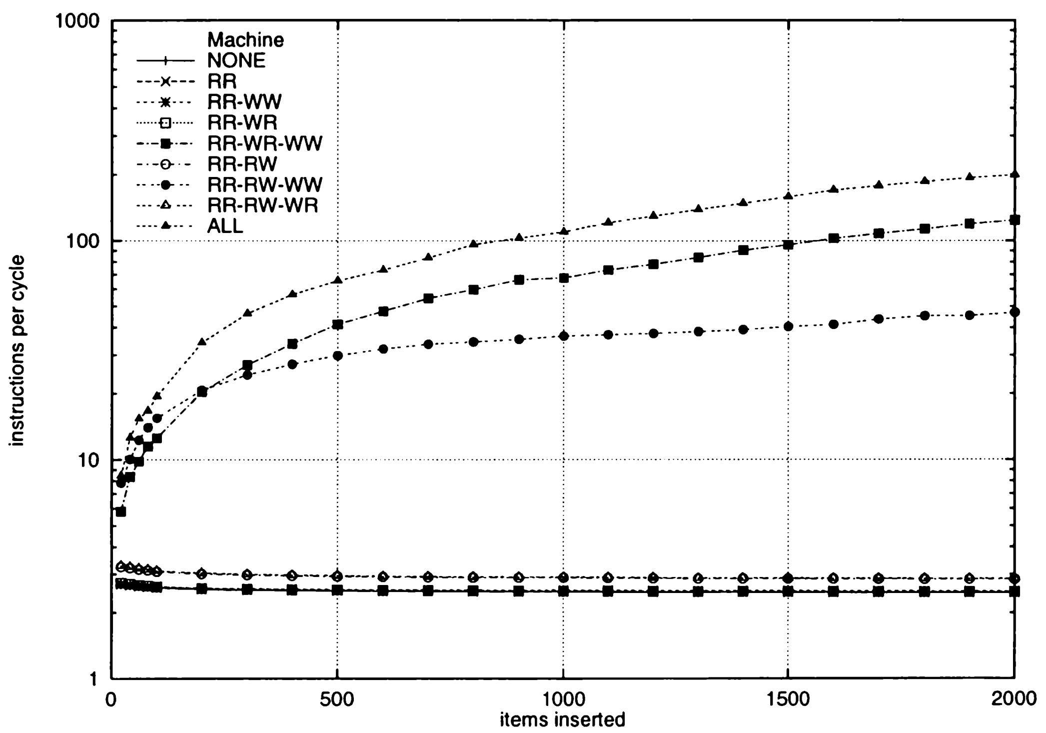


Figure C.13: IPC vs problem size with memory ordering constraints in place for *avl*.

C.4 Memory constraint meshes

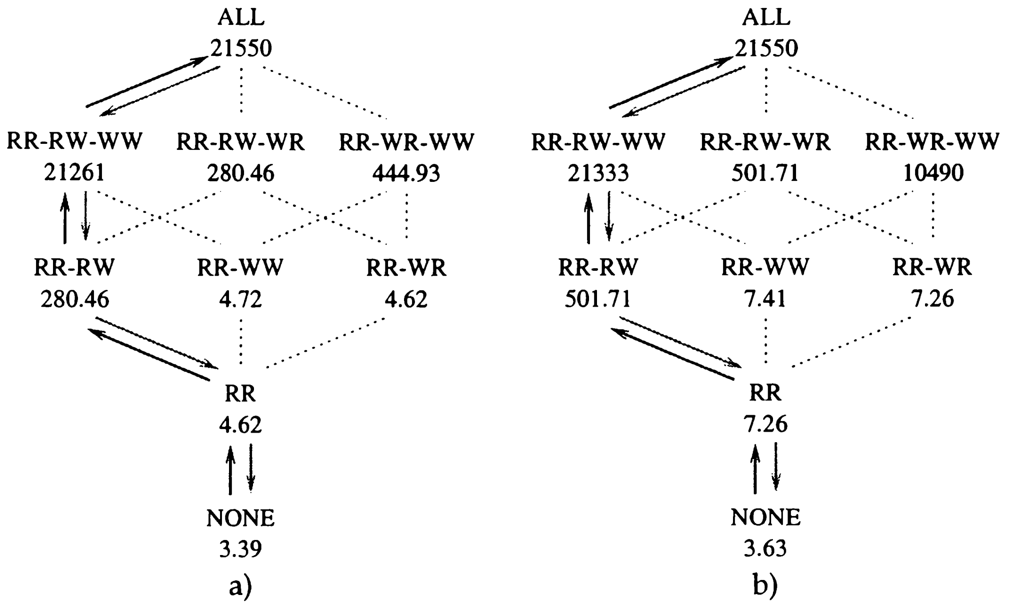


Figure C.14: Memory constraint ordering mesh for *trans* (50).

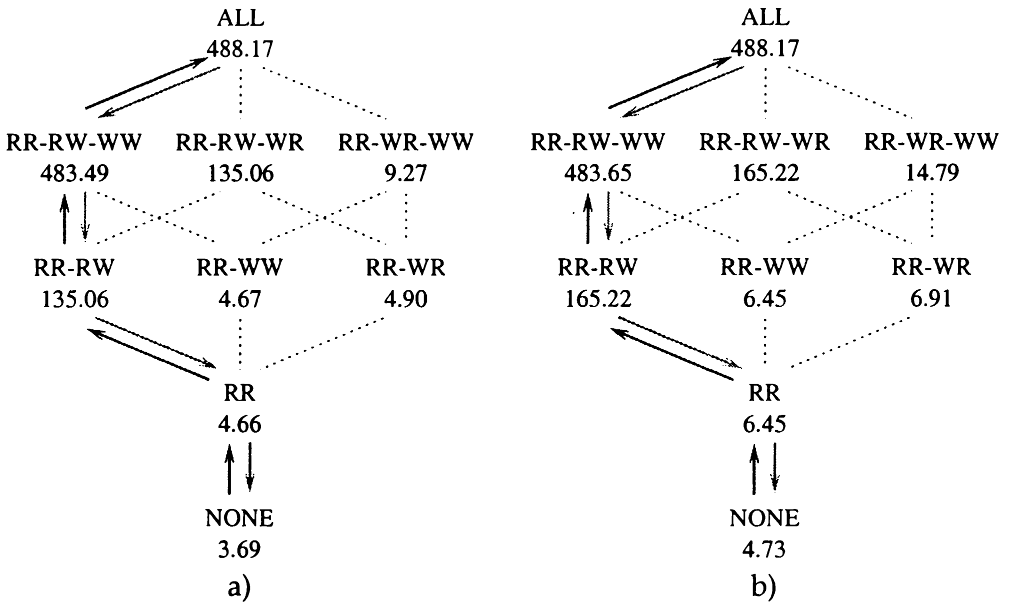


Figure C.15: Memory constraint ordering mesh for *gj* (30).

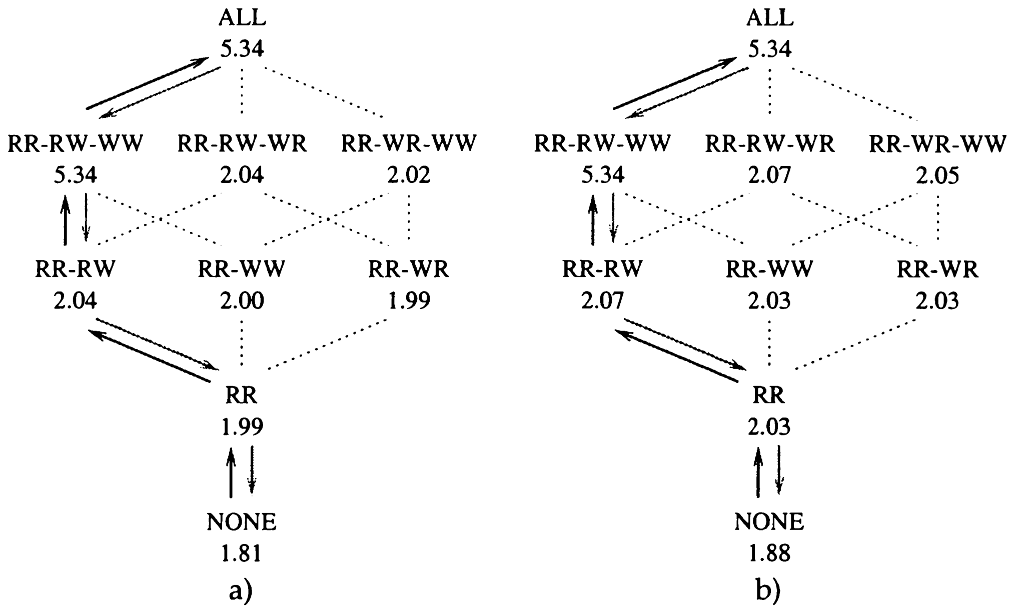


Figure C.16: Memory constraint ordering mesh for *qs2* (2000).

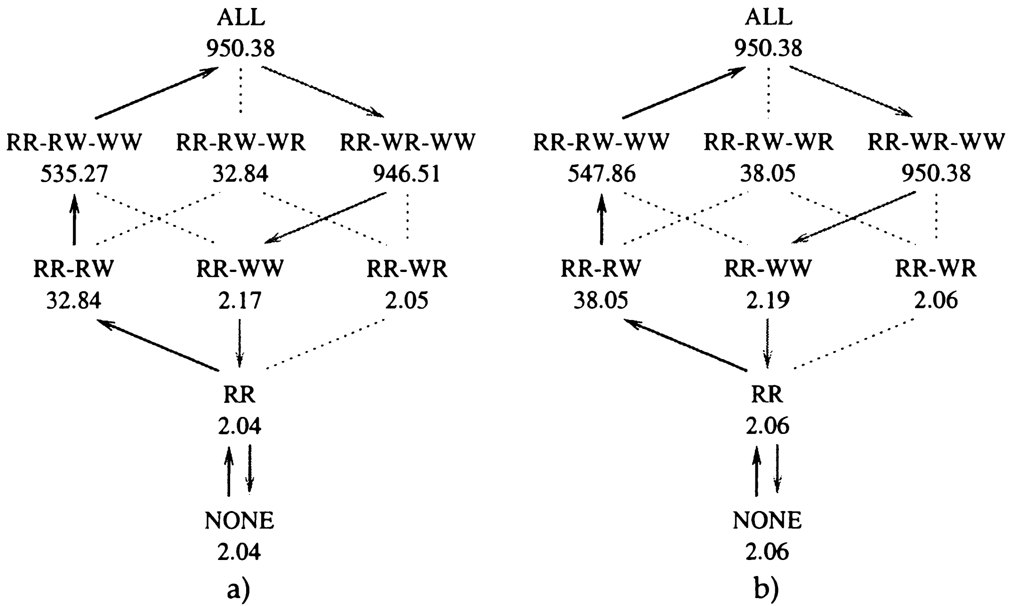


Figure C.17: Memory constraint ordering mesh for *bin* (2000).

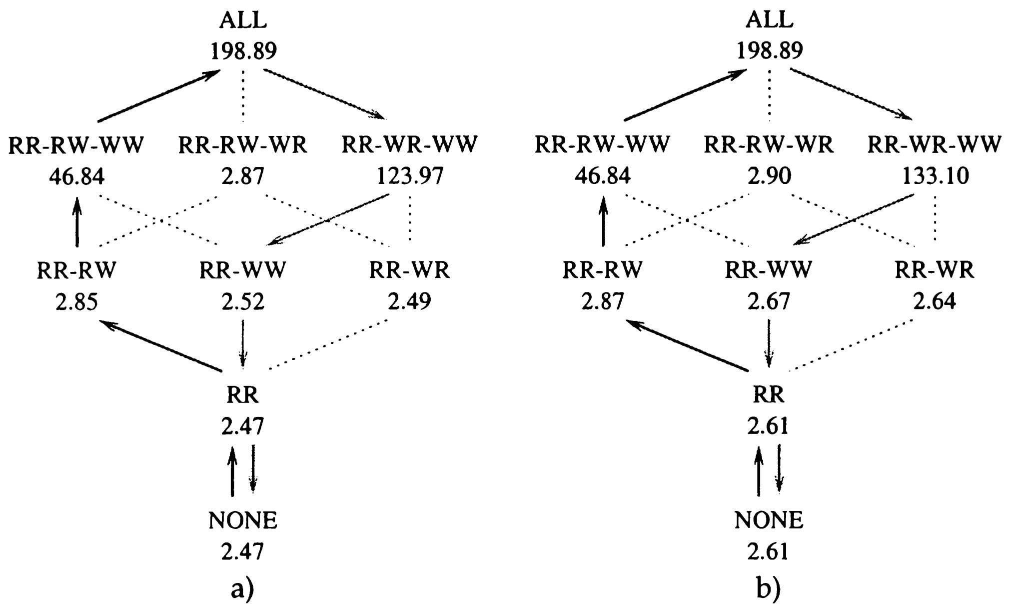


Figure C.18: Memory constraint ordering mesh for *avl* (2000).

C.5 IPC with frame limits

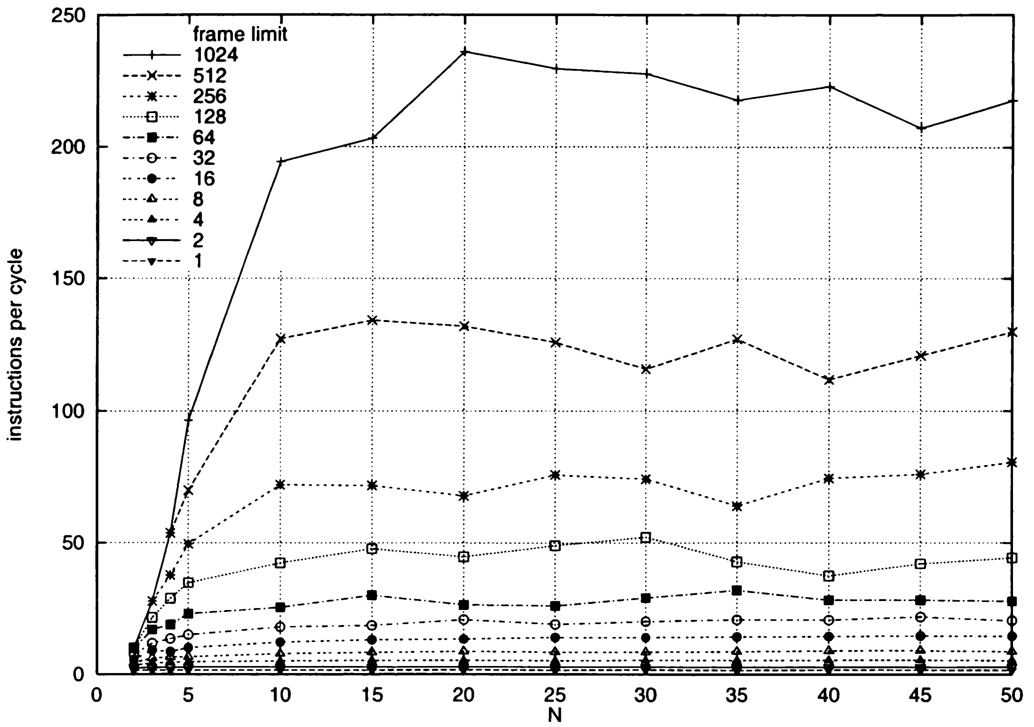


Figure C.19: IPC vs problem size with limited frames for *trans*.

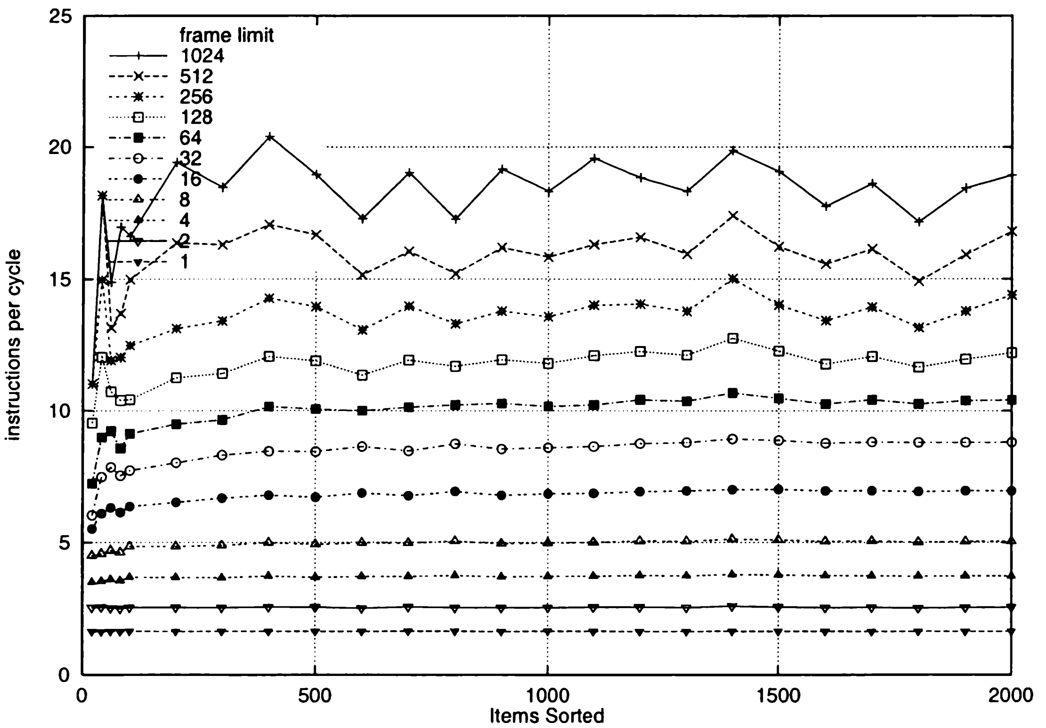


Figure C.20: IPC vs problem size with limited frames for *qs1*.

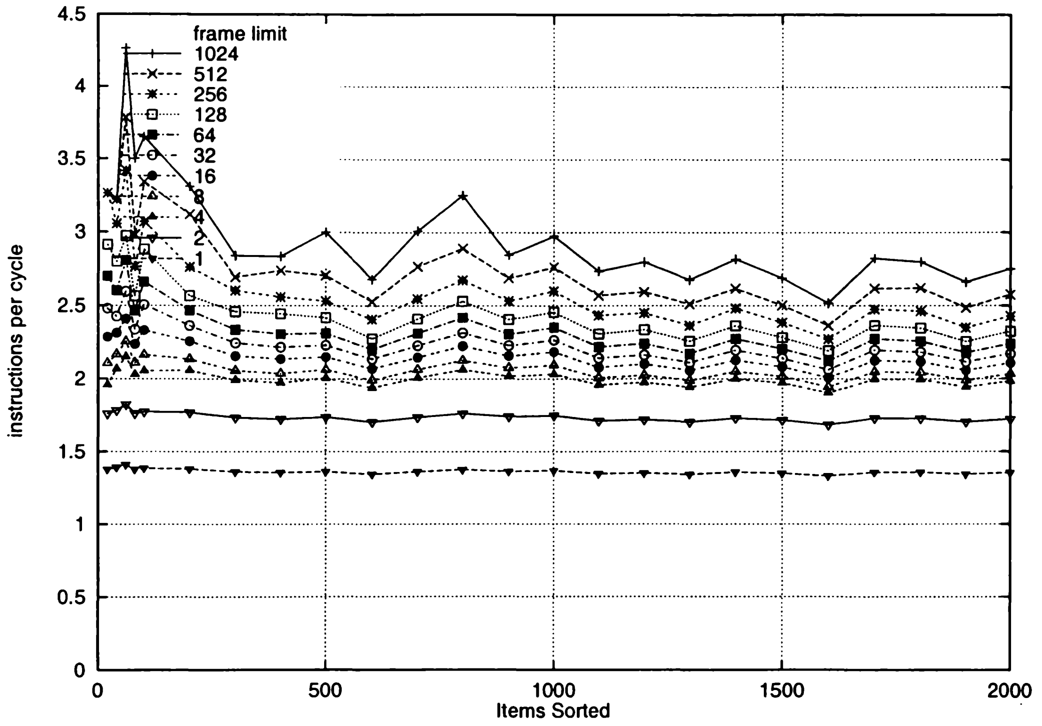


Figure C.21: IPC vs problem size with limited frames for *qs2*.

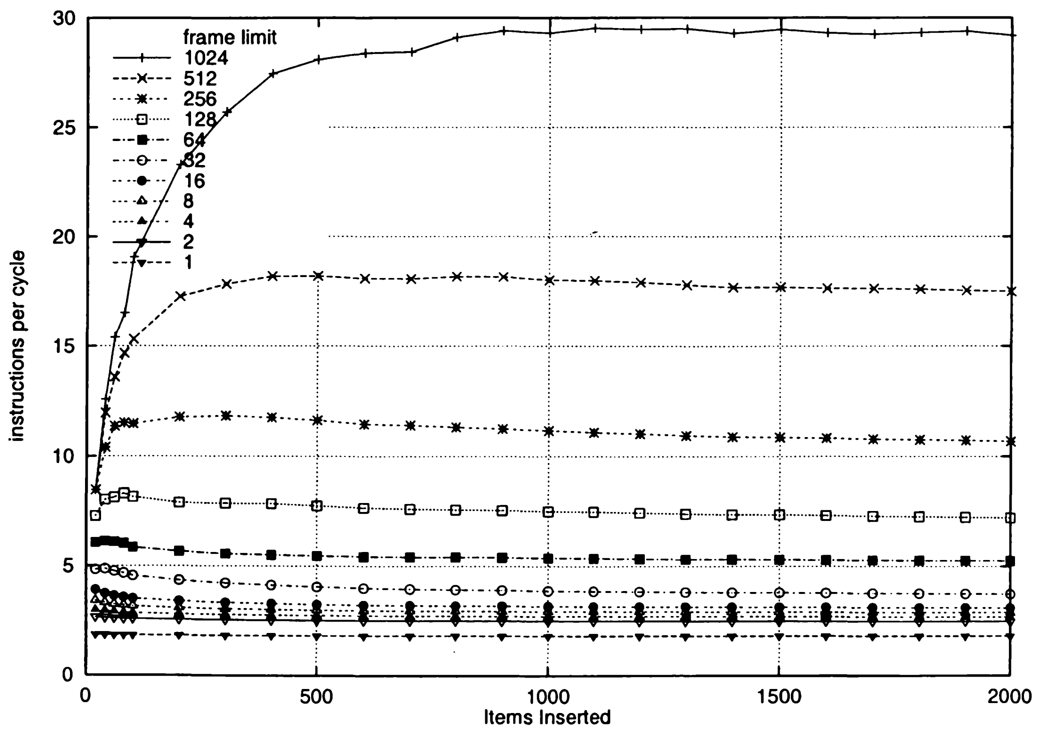


Figure C.22: IPC vs problem size with limited frames for *avl*.

C.6 Unlimited resource frame usage

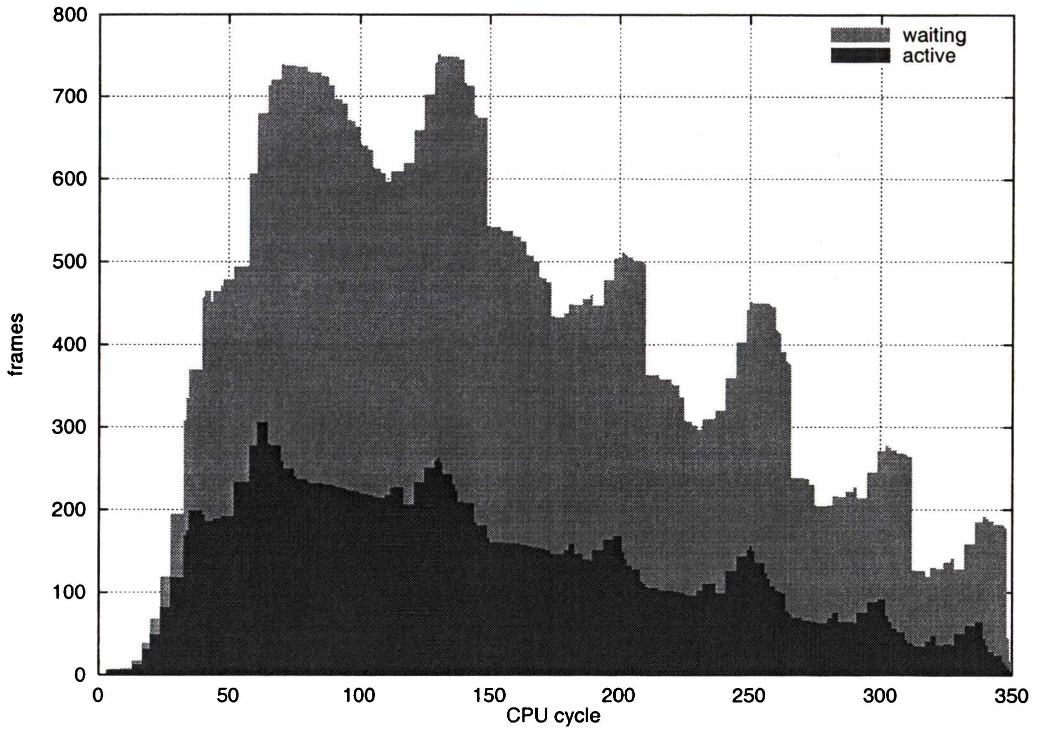


Figure C.23: Frame usage over time for *gj* (5).

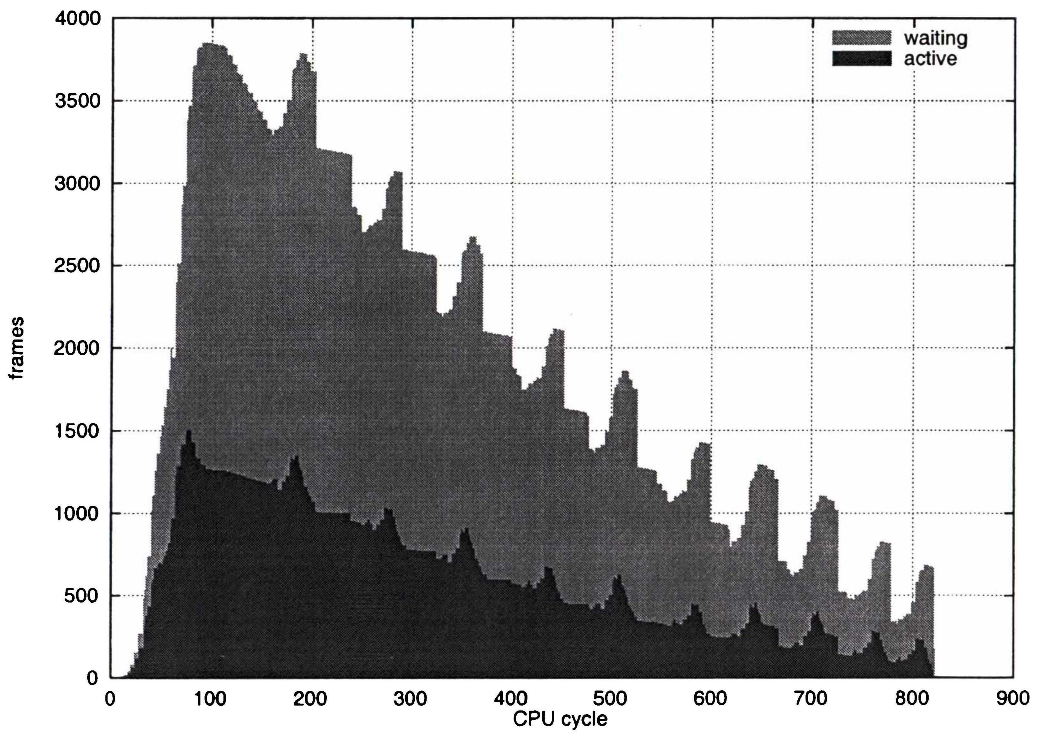


Figure C.24: Frame usage over time for *gj* (10).

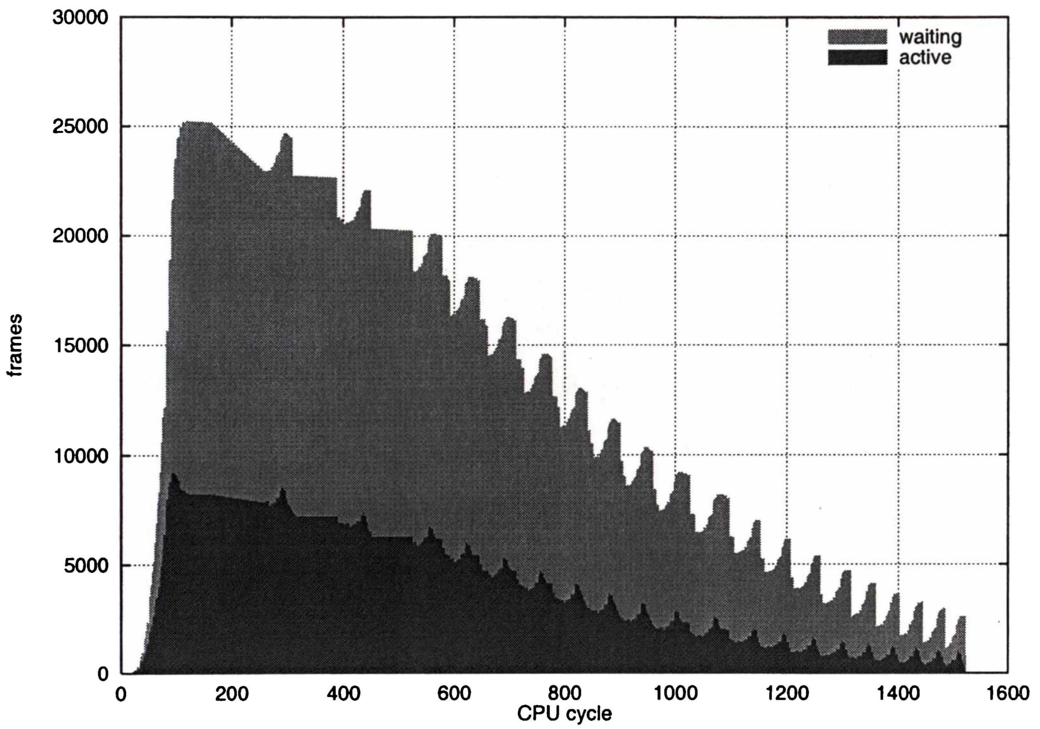


Figure C.25: Frame usage over time for *gj* (20).

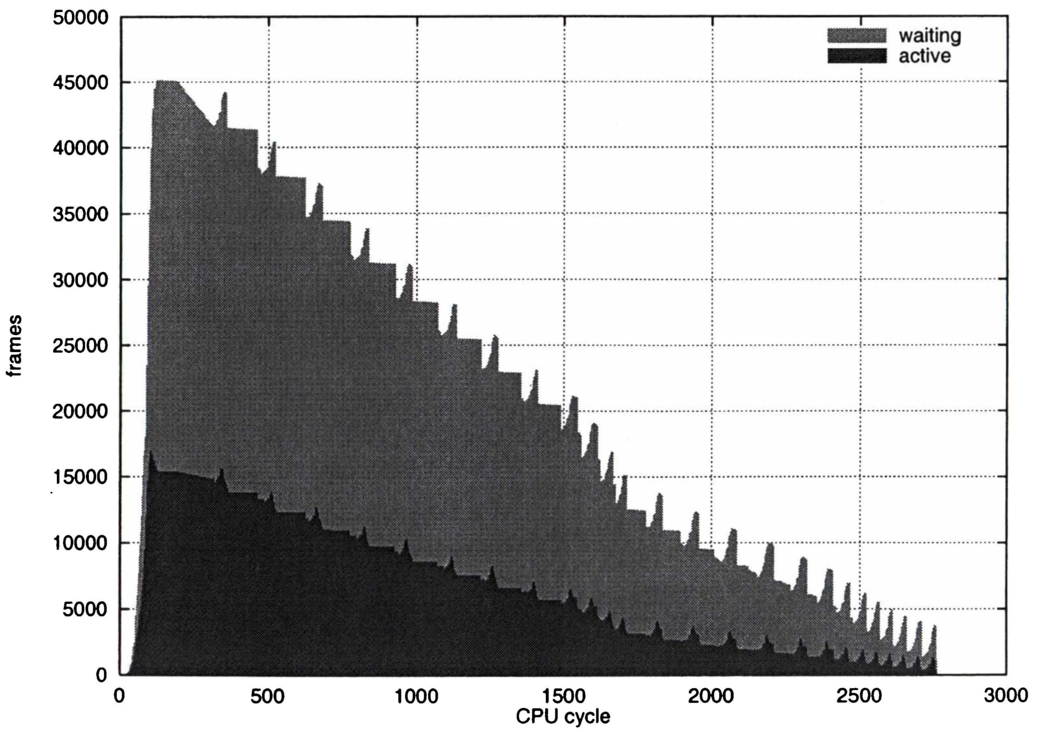


Figure C.26: Frame usage over time for *gj* (25).

C.7 Limited resource frame usage

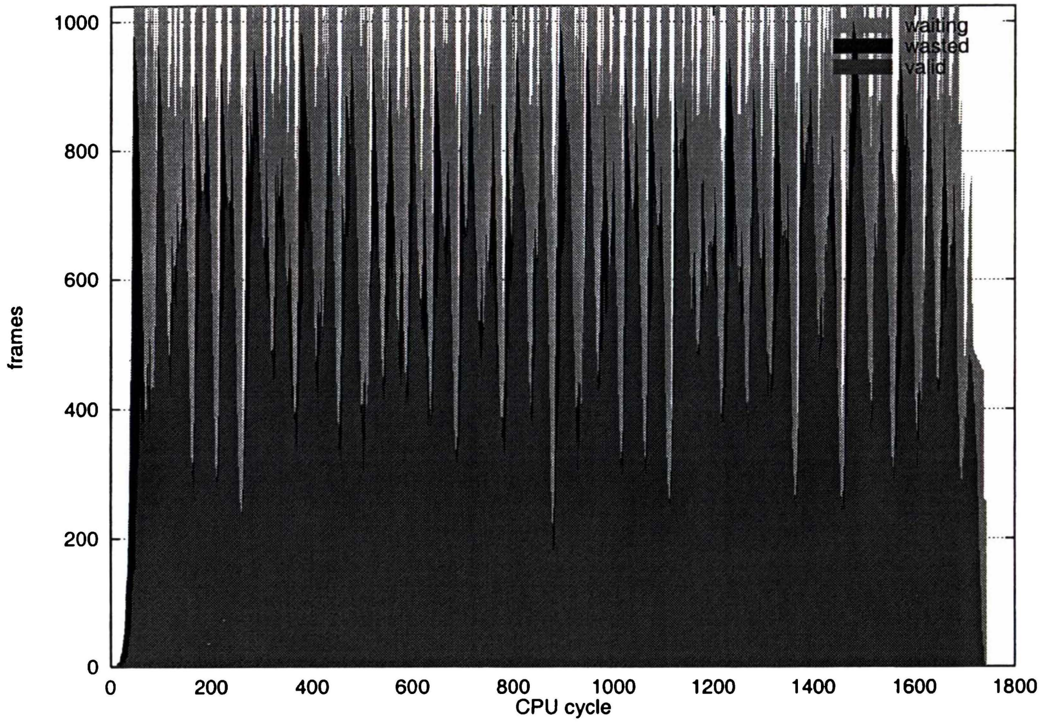


Figure C.27: Frame usage over time for *trans* (20) when limited to 1024 frames.

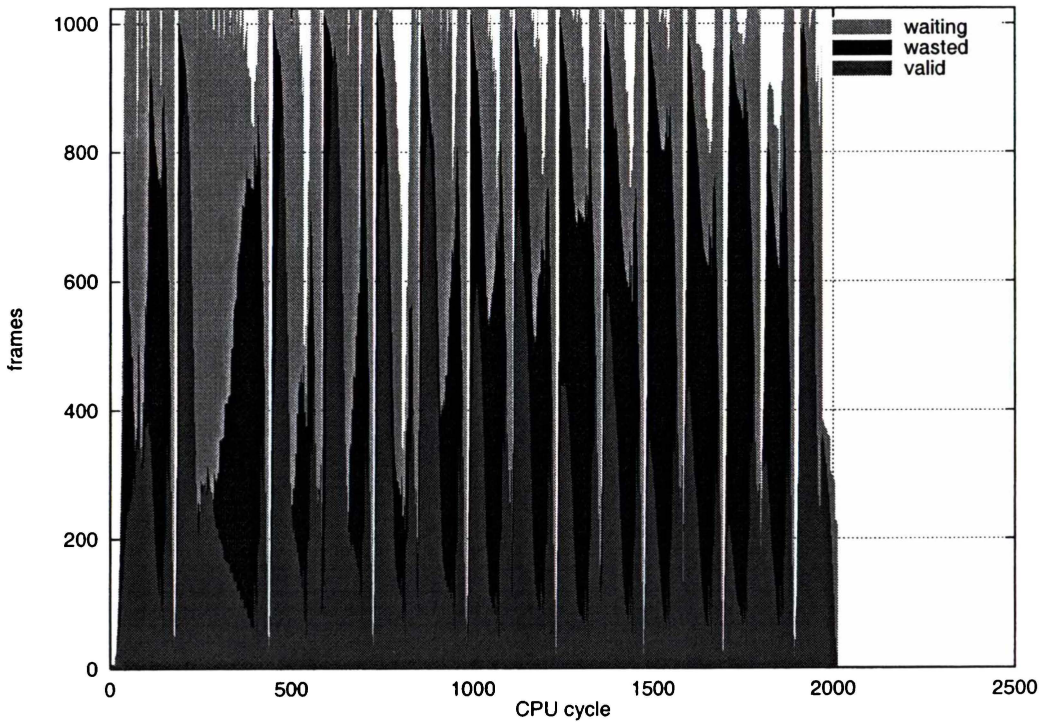


Figure C.28: Frame usage over time for *gj* (15) when limited to 1024 frames.

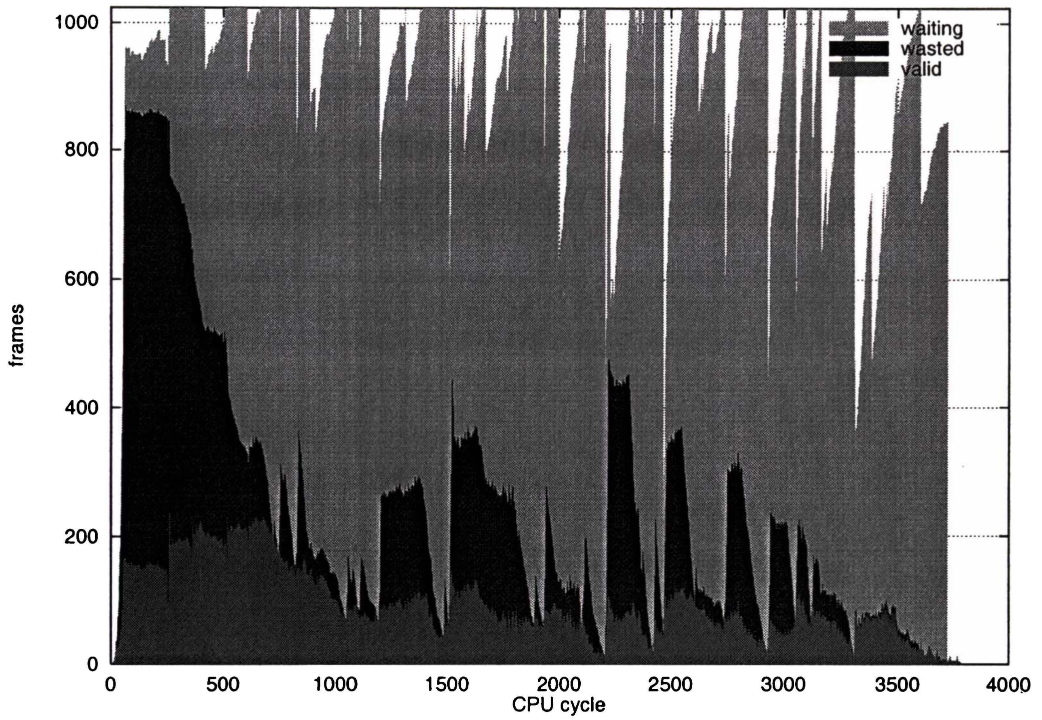


Figure C.29: Frame usage over time for *avl* (200) when limited to 1024 frames.

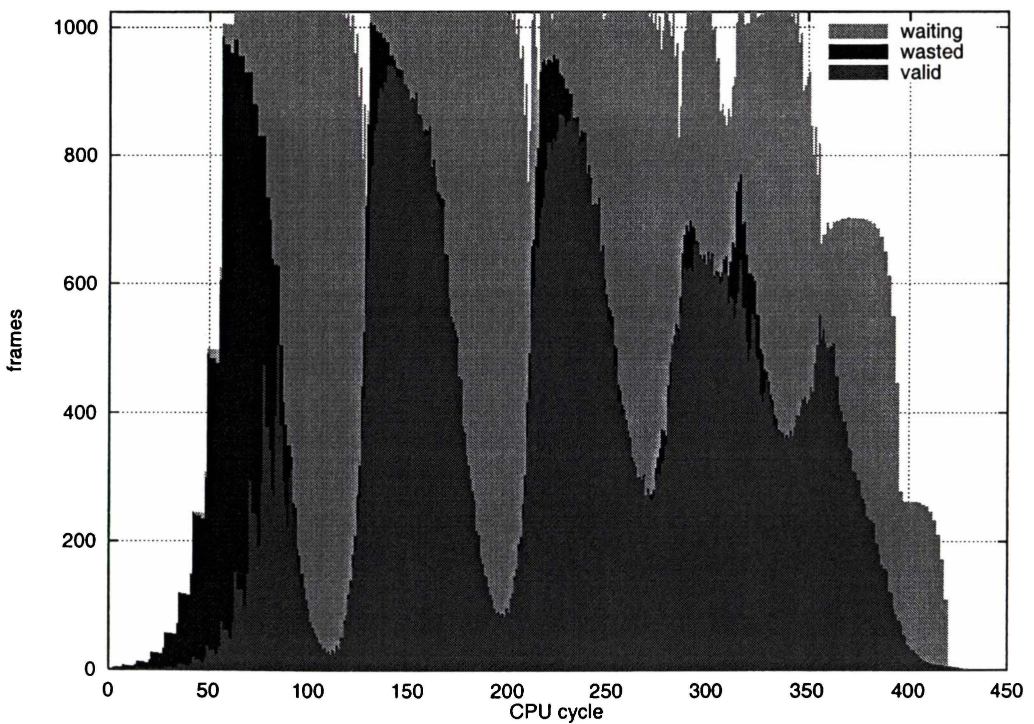


Figure C.30: Frame usage over time for *fibf* (15) when limited to 1024 frames.

C.8 Time-space cache entries per frame

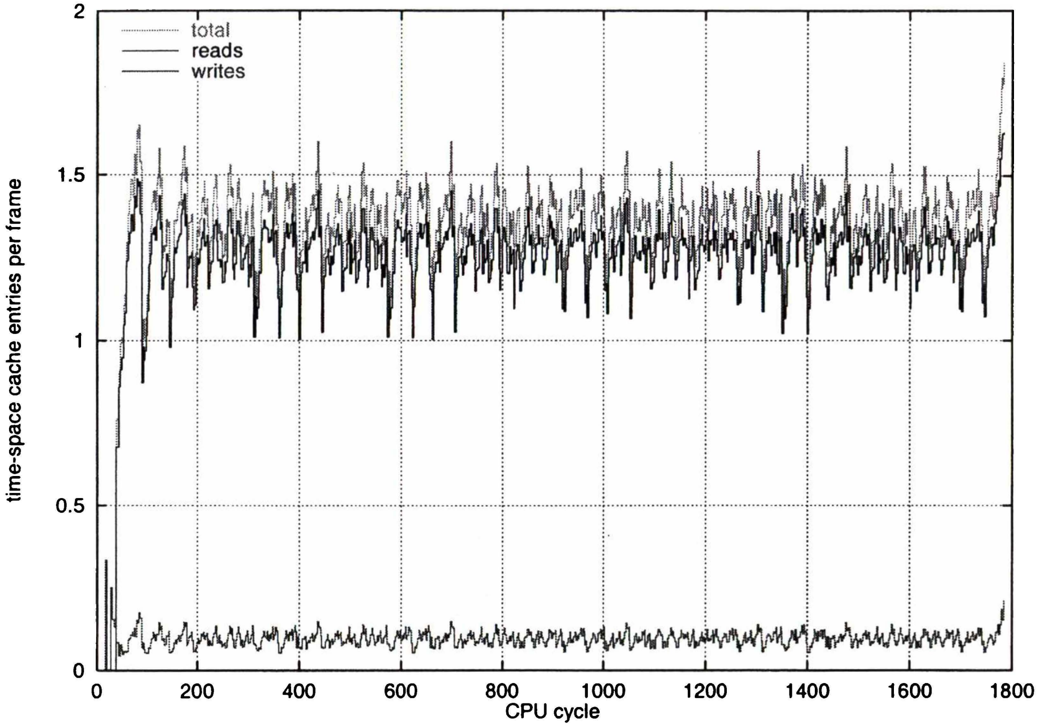


Figure C.31: Time-space cache entries per frame for *trans* (20).

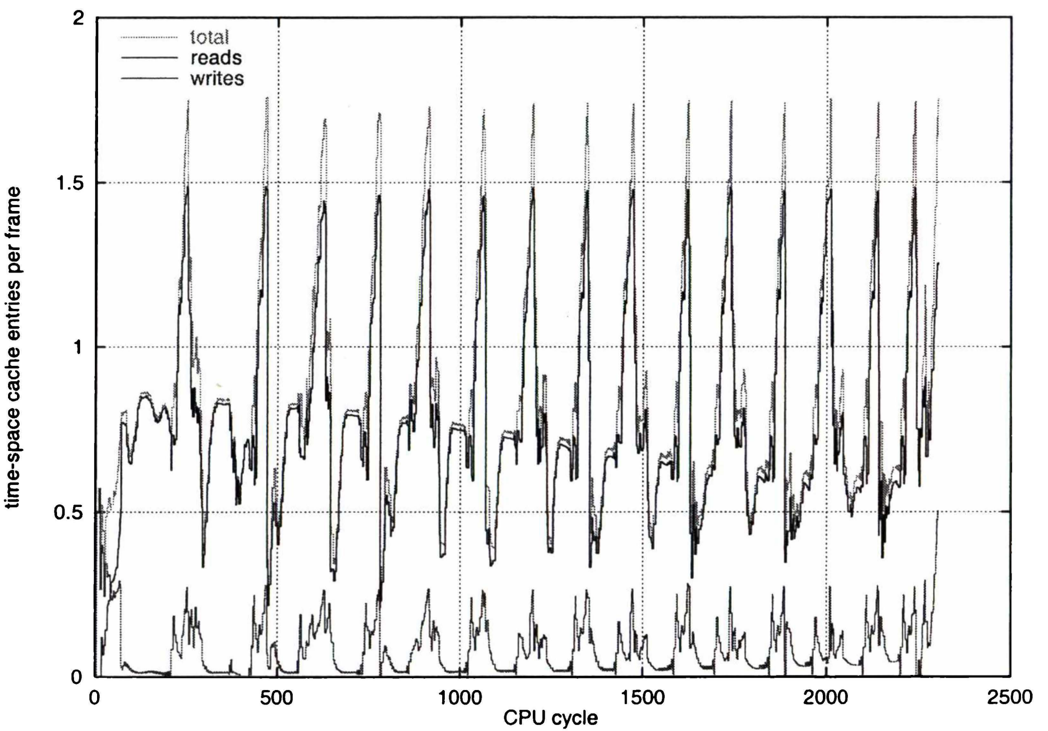


Figure C.32: Time-space cache entries per frame for *gj* (15).

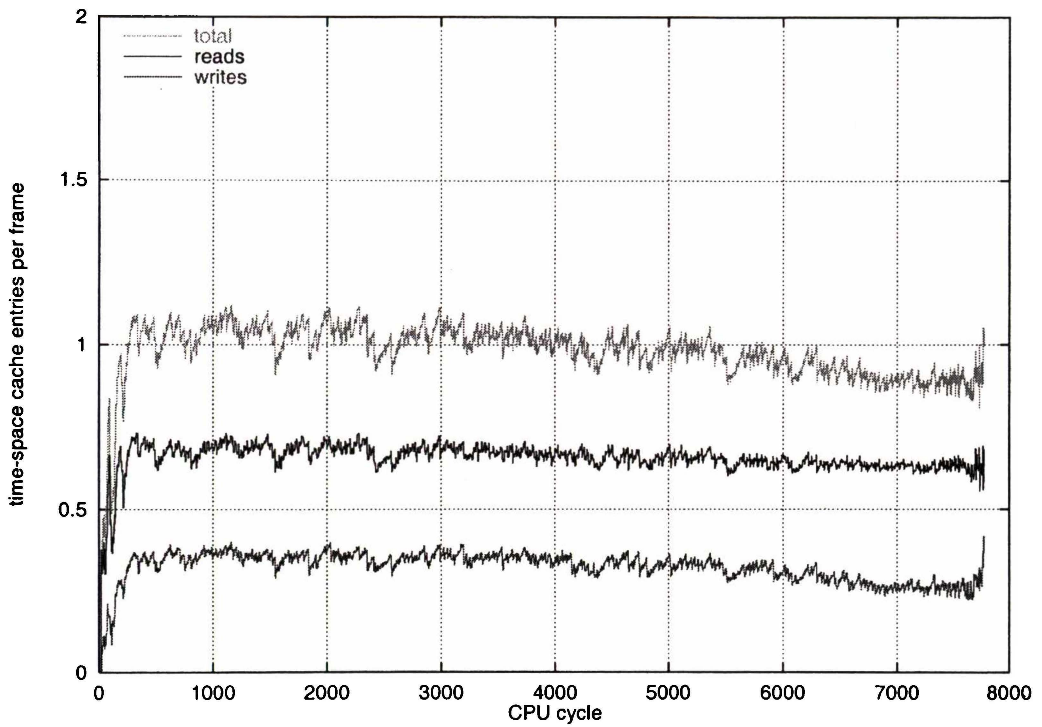


Figure C.33: Time-space cache entries per frame for *heap* (200).

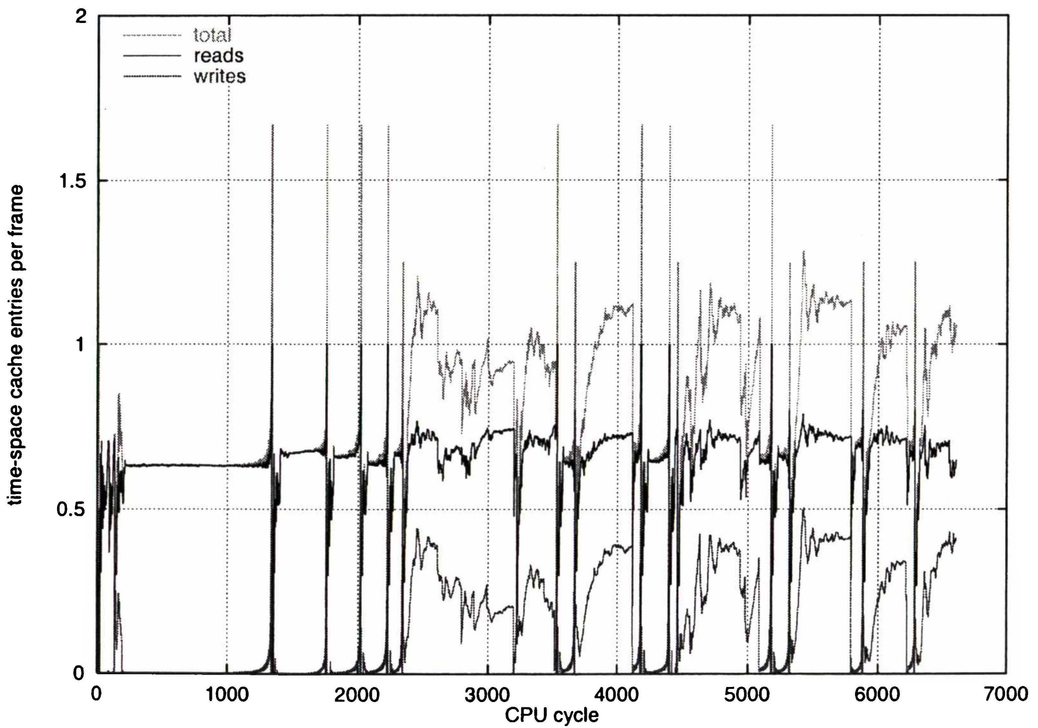


Figure C.34: Time-space cache entries per frame for *qs1* (500).

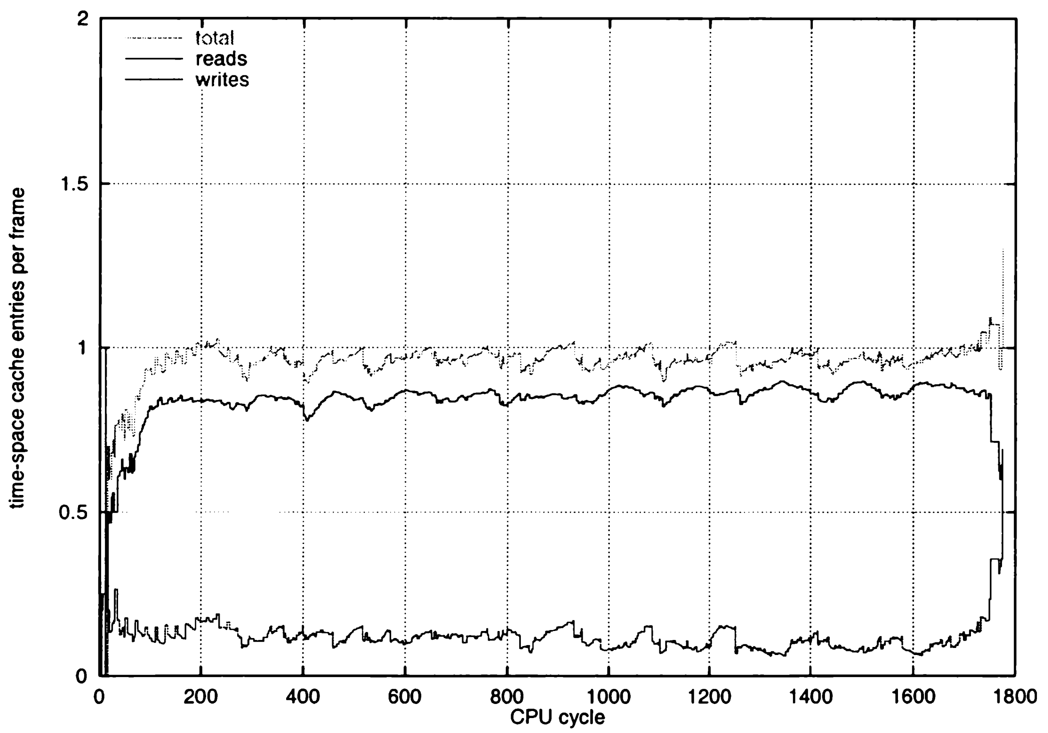


Figure C.35: Time-space cache entries per frame for *bin* (500).

Bibliography

Adve, S. V. and Hill, M. D. [1989]. Weak ordering - a new definition and some implications. Technical Report 902, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin 53706.

Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K. L., Kranz, D., Kubiatowicz, J., Lim, B.-H., Mackenzie, K. and Yeung, D. [1995]. The MIT Alewife machine: Architecture and performance. In *22nd Annual International Symposium On Computer Architecture (ISCA-22)*. New York: ACM Press.

Agarwal, A., Chaiken, D., Johnson, K., Kranz, D., Kubiatowicz, J. and Kurihara, K. [1991]. The MIT Alewife machine: A large-scale distributed-memory multiprocessor. In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers.

Aho, A. V., Sethi, R. and Ullman, J. D. [1986]. *Compilers, Principles, Techniques, and Tools*, chapter 10.5, (pp. 611–612). Addison-Wesley Publishing Company.

Ang, B. S., Arvind and Chiou, D. [1995]. StarT the next generation: Integrating global caches and dataflow architecture. Technical Report CSG354, Massachusetts Institute of Technology.

Austin, T. M. and Sohi, G. S. [1993]. Evaluation of serial program performance on fine-grain paralle processors. Technical Report 1162, University of Wisconsin-Madison.

Back, A. and Turner, S. [1994]. Time-stamp generation of the parallel execution of program control structures. Technical report, University of Exeter, Exeter EX4 4PT England.

Bashe, C. J., Johnson, L. R., Palmer, J. H. and Pugh, E. W. [1986]. *IBM's Early Computers*, chapter 11, (pp. 416–458). MIT Press.

Bellenot, S. [1990]. Global virtual time algorithms. In Nicol, D. (Ed.), *1990 SCS multiconference on Distributed Simulation*, Volume 22 (pp. 122–130). San Diego, California.

Bennett, B. S. [1995]. *Simulation Fundamentals*, chapter 2, (pp. 16–17). Prentice Hall International.

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 (ACAC'98)*, Volume 20 (pp. 37–46). Perth, Australia: Springer-Verlag.

Burger, D. and Austin, T. M. [1997]. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin-Madison.

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.

Calvert, J. [1997]. Design of the execution control structure for the WarpEngine optmistic cpu. Master's thesis, University of Waikato.

Charlesworth, A. E. [1981]. An approach to scientific array processing: The architectural design of the AP-120B/FPS-164 family. *IEEE Computer, 14(9)*, 18–27.

Cleary, J. G. [1995a]. WarpEngine instruction set. Internet Web Page. URL www.cs.waikato.ac.nz/timewarp/wengine/instset/we_inst.may51995.html.

Cleary, J. G. [1995b]. WarpEngine instruction set: Version 2. Internet Web Page. URL www.cs.waikato.ac.nz/timewarp/wengine/instset/we_inst.nov21995.html.

Cleary, J. G., McWha, J. A. D. and Pearson, M. W. [1997]. Timestamp representations for virtual sequences. In *11th Workshop on Parallel and Distributed Simulation (PADS'97)*. Lockenhaus, Austria.

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.

Cmelik, B. and Keppel, D. [1994]. Shade: A fast instruction-set simulator for execution profiling. In *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*.

Corman, T. H., Leiserson, C. E. and Rivest, R. L. [1990]. *Introduction to Algorithms*. New York: McGraw-Hill Book Company.

Culler, D. E. [1993]. Two fundamental limits on dataflow multiprocessing. In *The IFIP Working Group 10.3 (Concurrent Systems) Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*. Elsevier Science Publishers. Also Report No. UCB/CSD 92/716 University of California, Berkeley.

Dennis, J. B. [1980]. Data flow supercomputers. *IEEE Computer, 13(11)*, 48–56.

Dutta, S. and Franklin, M. [1995]. Control flow prediction with tree-like subgraphs for superscalar processors. In *Proceedings of 28th Annual International Symposium on Microarchitecture (MICRO-28)*.

Dwyer, H. and Torng, H. C. [1992]. An out-of-order superscalar processor with speculative execution and fast, precise interrupts. *Proceedings of IEEE.*

ELF [1993]. *Executable and Linkable Format (ELF) V1.1.* Tools Interface Standards (TIS).

Fillo, M., Stephen W, K., Dally, W. J., Carter, N. P., Chang, A., Gurevich, Y. and Lee, W. S. [1995]. The M-Machine multicomputer. Technical Report AI1532, AI Lab, Massachusetts Institute of Technology.

Fisher, J. A. [1981]. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers, 30(7),* 478–490.

Franklin, M. [1993]. *The Multiscalar Architecture.* PhD thesis, University of Wisconsin–Madison.

Franklin, M. and Sohi, G. S. [1992]. The expandable split window paradigm for exploiting fine-grain parallelism. In *19th Annual International Symposium On Computer Architecture (ISCA-19).* New York.

Franklin, M. and Sohi, G. S. [1996]. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions on Computers, 45(5),* 552–571.

Fujimoto, R. M. [1989]. The virtual time machine. In *International Symposium on Parallel Algorithms and Architectures* (pp. 199–208).

Fujimoto, R. M. [1990a]. Parallel discrete event simulation. *Communications of the ACM, 33(10),* 30–53.

Fujimoto, R. M. [1990b]. Time warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation, 6(3),* 211–239.

Glew, A. [1997]. Instruction propagation delays. Email. Personal Correspondence.

Gomes, F. [1992]. Global virtual time for optimistic parallel simulation. Technical report, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada.

Gomes, F., Cleary, J. G. and Unger, B. [1992]. GVT approximation in optimistic parallel discrete event simulation: A surevey. Technical report, Department of Computer Science, University of Calgary, Calgary, Alberta, Canada.

Goodman, J. and Miller, K. [1993]. *A Programmer's View of Computer Architecture with assembly language examples from the MIPS RISC architecture.* Saunders College Publishing.

Gurd, J. R., Kirkman, C. C. and Watson, I. [1985]. The manchester prototype dataflow computer. In *Communications of the ACM,* Volume 28 (pp. 34–52).

Gwennap, L. [1996]. Digital 21264 sets new standard clock speed, complexity, performance surpass records, but still a year away. Internet Web Page. URL www.digital.com/alphaoem/papers/microrep/digital2.htm.

Heinemann Publishers (NZ) Ltd [1990]. *Eton Statistical & Math Tables*. Octopus Publishing Group (NZ) Ltd.

Hennessy, J. L. and Patterson, D. A. [1996a]. *Computer Architecture A Quantitative Approach* (2nd Ed.). San Francisco: Morgan Kaufmann Publishers, Inc.

Hennessy, J. L. and Patterson, D. A. [1996b]. *Computer Architecture A Quantitative Approach* (2nd Ed.), chapter 4 (figure 4.60), (p. 359). San Francisco: Morgan Kaufmann Publishers, Inc.

Hum, H. H. J., Maquelin, O., Theobald, K. B., Tian, X., Tang, X., Gao, G. R., Cupryk, P., Elmasri, N., Hendren, L. J., Jimenez, A., Krishnan, S., Marquez, A., Merali, S., Nemawarkar, S. S., Panangaden, P., Xue, X. and Zhu, Y. [1995]. A design study of the EARTH multiprocessor. In *The International Conference on Parallel Architectures and Compilation Techniques (PACT'95)* (pp. 59–68). Limassol, Cyprus.

Hunt, D. [1997]. Advanced performance features of the 64-bit pa-8000. Internet Web Page. URL www.hp.com/ahp/framed/technology/micropro/pa-8000/docs/advperf.html.

INTEL [1995]. PENTIUM PRO processor at 150 mhz, 166 mhz, 180mhz and 200 mhz. INTEL Corporation Datasheets. Order Number: 242769-003.

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 *3rd International Conference on ASPLOS* (pp. 272–282). New York.

Kernighan, B. W. and Ritchie, D. M. [1988]. *The C Programming Language* (2nd Ed.). Prentice Hall.

Lam, M. S. and Wilson, R. P. [1992]. Limits of control flow on parallelism. In *19th Annual International Symposium On Computer Architecture (ISCA-19)* (pp. 46–57). New York.

Lamport, L. [1979]. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers, C-28(9)*, 690–691.

Larus, J. R. [1993]. *SPIM S20: A MIPS R2000 Simulator.* Computer Sciences Department, University of Wisconsin-Madison.

Law, A. M. and Kelton, W. D. [1991]. *Simulation Modeling & Analysis*, chapter 1, (pp. 8–9). McGraw-Hill.

Lee, J. K. F. and Smith, A. J. [1984]. Branch prediction stategies and branch target buffer design. *IEEE Computer, 17(1)*, 6–22.

Lethin, R. [1994]. The Jellybean machine. Internet Web Page. URL cva.stanford.edu/j-machine/cva_j_machine.html.

Lewis, H. R. and Denenberg, L. [1991]. *Data Structures & Their Algorithms*. Harper Collins.

Lipasti, M. H. and Shen, J. P. [1996]. Extending the dataflow limit via value prediction. In *Proceedings of the 29th Annual ACM/IEEE Symposium on Microarchitecture* (pp. 226–237). Paris, France.

Littin, R. H. [1998]. Assembler for the WarpEngine simulator. Internet Web Page. URL www.cs.waikato.ac.nz/timewarp/wengine/instset/assembler.html.

Littin, R. H. [1999a]. Data and control speculative execution. In Yeates, S. (Ed.), *The 3rd New Zealand Computer Science Research Students' Conference (NZCSRSC'99)* (pp. 14–21). Hamilton, New Zealand.

Littin, R. H. [1999b]. WarpEngine test programs. Internet Web Page. URL www.cs.waikato.ac.nz/timewarp/wengine/testcode/.

Littin, R. H., McWha, J. A. D., Pearson, M. W. and Cleary, J. G. [1998]. Block based execution and task level parallelism. In Morris, J. (Ed.), *The 3rd Australasian Computer Architecture Conference (ACAC'98)*, Volume 20 of *Australian Computer Science Communications* (pp. 57–66). Perth, Australia: Springer-Verlag.

McWha, J. A. D. [1999]. Using timestamps to maintain causal dependencies. In Yeates, S. (Ed.), *The 3rd New Zealand Computer Science Research Students' Conference (NZCSRSC'99)* (pp. 38–45). Hamilton, New Zealand.

MIPS Technologies, Inc [1994]. MIPS RISC technology R10000 microprocessor technical brief. Technical report, Silicon Graphics, Inc.

MIT [1995]. Project monsoon. Internet Web Page. URL www.csg.lcs.mit.edu/monsoon/.

Motorola [1997]. PowerPC 750 RISC microprocessor technical summary. Technical report, IBM.

Muller, H. L. [1993]. *Simulating Computer Architectures*. PhD thesis, University of Amsterdam, Amsterdam.

Neefs, H. [1996]. A preliminary study of a fixed-length block-structured instruction set architecture. Technical Report 96-07, Electronics and Information Systems Department, University of Gent, Belgium.

Neefs, H., Bosschere, K. D. and Campenhout, J. V. [1996]. A C++ simulator modelling a modern data-flow scheduling microprocessor. In *Seminar on Parallel Computing*. Noordwijk aan Zee (NL).

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*.

Oehring, H., Sigmund, U. and Ungerer, T. [1999]. Simultaneous multithreading and multimedia. In *Workshop on Multi-Threaded Execution, Architecture and Compilation (MTEAC 99) in conjunction with Fifth International Symposium on High Performance Computer Architecture (HPCA-5)*. Orlando.

Oplinger, J., Heine, D., Liao, S.-W., Nayfeh, B. A., Lam, M. S. and Olukotun, K. [1997]. Software and hardware for exploiting speculative parallelism with a multiprocessor. Technical Report CSL-TR-97-715, Computer Systems Laboratory, Stanford University, Stanford, CA 94305-4070.

Patterson, D. A. [1980]. The case for the reduced instruction set computer. *Computer Architecture News, 8(6)*, 25–33.

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 '97* (pp. 290–295). Bangalore, India.

Perleberg, C. H. and Smith, A. J. [1993]. Branch target buffer design and optimization. *IEEE Transactions on Computers, 42(4)*, 396–412.

Pollock, F. [1999]. New microarchitecture challenges in the coming generations of cmos process technologies. In *32nd Annual International Symposium on Microarchitecture emphasizing Instruction-Level Parallelism (ILP) processing*. Haifa, Israel. Keynote Presentation.

Postiff, M. A., Greene, D. A., Tyson, G. S. and Mudge, T. N. [1998]. The limits of instruction level parallelism in SPEC95 applications. In *ASPLOS VIII INTERACT3 Workshop*.

Press, W. H. [1992]. *Numerical Recipes in C*. Cambridge University Press.

Quinn, M. J. [1987]. *Designing Efficient Algorithms for Parallel Computers*. McGraw–Hill.

Ralston, A. and Reilly, E. D. (Eds.). [1993]. *Encyclopedia of Computer Science* (3rd Ed.) (p. 1061). IEEE Press.

Rau, B. R., Yen, D. W. L., Yen, W. and Towle, R. A. [1989]. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *IEEE Computer, 22(1)*, 12–34.

Ross, P. E. [1996]. Moore's second law. *Forbes, 157(6)*, 116–117.

Rotenberg, E., Jacobson, Q., Sazeides, Y. and Smith, J. [1997]. Trace processors. In *Micro-30*. Research Triangle Park, North Carolina.

Rotenberg, E., Jacobson, Q. and Smith, J. [1998]. A study of control independence in superscalar processors. Technical report, University of Wisconsin - Madison.

Russell, R. M. [1978]. The CRAY-1 computer system. *Communications of the ACM, 21(1),* 63–72.

Scott, A., Burkhart, K. P., Kumar, A., Blumberg, R. M. and Ranson, G. L. [1997]. Four-way superscalar PA-RISC processors. *Hewlett-Packard Journal.* Article 1.

Shen, J. P. and Nagle, D. F. [1998]. Computer architecture: A.d. 2000 and beyond. Internet Web Page. URL www.ece.cmu.edu/pubs/vision/arch.html.

Shriver, B. and Smith, B. [1998]. *The Anatomy of a High-Performance Processor, A Systems Perspective,* chapter 1, (p.5). IEEE Computer Society Press.

Siemers, C. and Möller, D. P. F. [1998]. The >S<puter: A novel microarchitecture model for execution inside superscalar and vliw processors using reconfigurable hardware. In Morris, J. (Ed.), *3rd Australasian Computer Architecture Conference (ACAC'98),* Volume 20 (pp. 169–178). Perth, Australia: Springer-Verlag.

Singh, V., Kumar, V., Agha, G. and Tomlinson, C. [1991]. Efficient algorithms for parallel sorting on mesh multicomputers. *International Journal of Parallel Programming, 20(2),* 95–131.

Smith, A. J. [1982]. Cache memories. *Computing Surveys, 14(3),* 473–530.

Smith, J. E. [1981]. A study of branch prediction strategies. In *8th International Symposium on Computer Architecture (ISCA-8).*

Smith, J. E. and Pleszkun, A. R. [1985]. Implementation of precise interrupts in pipelined processors. In *12th Annual International Symposium On Computer Architecture (ISCA-12).*

Smith, M. D., Horowitz, M. and Lam, M. S. [1992]. Efficient superscalar performance through boosting. In *Fifth Conf. on Architectural Support for Programming Languages and Operating Systems* (pp. 248–259). Boston.

Smith, M. D., Johnson, M. and Horowitz, M. A. [1989]. Limits on multiple instruction issue. In *3rd International Conference on ASPLOS* (pp. 290–302). New York.

Snelling, D. F. [1993]. *The Design and Analysis of a Stateless Data-Flow Architecture.* PhD thesis, University of Manchester, Manchester M13 9PL, UK.

Snelling, D. F. and Egan, G. K. [1994]. A comparitive study of data-flow architectures. Technical Report UMCS-94-4-3, University of Manchester, Manchester M13 9PL, UK.

Sohi, G. S., Breach, S. and Vijaykumar, T. N. [1995]. Multiscalar processors. In *22nd International Symposium on Computer Architecture (ISCA-22)* (pp. 414–425). New York: ACM Press.

Stallard, P. W. A., Muller, H. L. and Warren, D. H. D. [1993]. Performance evaluation of parallel programs on the data diffusion machine. In *Performance Evaluation of Parallel Systems PEPS'93* (pp. 94–101). Universtiy of Warwick, UK.

Sun Microsystems [1999]. The UltraSPARC processor technology white paper. Internet Web Page. URL www.sun.com/microelectronics/whitepapers/.

Theobald, K. B., Gao, G. R. and Hendren, L. J. [1993]. Speculative execution and branch prediction on parallel machines. In *The 7th ACM International Conference on Supercomputing* (pp. 77–86). Tokyo, Japan.

Tomasulo, R. M. [1967]. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal, 11*, 25–33.

Wall, D. W. [1991]. Limits of instruction-level parallelism. In *4th International Conference on ASPLOS* (pp. 176–188). New York.

Yasugi, M., Matsuoka, S. and Yonezawa, A. [1992]. ABCL/onEM-4: a new software/hardware architecture for object-oriented concurrent computing on an extended dataflow supercomputer. In *6th ACM International Conference on Supercomputing* (pp. 93–103). Washington D.C.

Yeager, K. [1996]. The MIPS R10000 superscalar microprocessor. *IEEE Micro, 16(2)*.