Instruction Scheduling in Micronet-based Asynchronous ILP Processors

Salvador Sotelo Salazar

Doctor of Philosophy
University of Edinburgh
2003

Abstract

Optimisations for Instruction-level Parallelism (ILP) in synchronous processors can assume deterministic execution times for instructions. However, in asynchronous architectures it is less certain in advance when instructions complete execution and when results become available. The instruction latency depends on a number of factors, including the input data, the type of computation, and contention for architectural resources at run-time. In particular, in micronet-based asynchronous processors, which feature non-linear pipelines and out-of-order completion, instructions would compete for resources and overtake other instructions. Such a behaviour makes it more difficult to consistently predict at compile-time, the optimal order of instruction execution.

This thesis investigates the problem of optimisations for ILP in micronet-based asynchronous processors. A novel scheduler called Penalise True Dependency (PTD) is presented for scheduling instructions within basic blocks, which minimises stalls due to data dependencies and resource contentions. PTD has been extended to perform global optimisations using this metric on techniques such as code motion, code and tail duplication, and block merging, and in the appropriate order to minimise code expansion.

The simulation results for a subset of the `SPEC95Int` benchmarks executing on an instruction set simulator of the micronet-based asynchronous processor demonstrate that the PTD scheduler outperforms traditional scheduling methods such as list schedulers, and has a better algorithmic time complexity.

Acknowledgements

I would like to thank my supervisor Damal K. Arvind for all his encouragement and support throughout this research. His numerous advises and suggestions have had a very positive outcome for this work.

My parents, Salvador and Maria Guadalupe, who have been a constant unlimited support, I thank them for their patience through *all* these years. Los quiero mucho.

I would also like to thank to the members of the MAP group (Lennart Beringer, Robert Mullins, Vinod Rebello, Johannes Schneiders and Christos Sotiriou) for their useful discussions and comments. A pleasure to work with.

I could not forget Paul Coe, Adam Donlin, Jonathan Meddes, Dominic Stanyer and Lawrence Williams for their friendship on and off work.

Many thanks to my friends Pedro, John, Arturo, Alex and many others that made my stay in Edinburgh to be a very enjoyable experience.

Thanks to the School of Informatics from the University of Edinburgh for providing me with the environment and the material to accomplish this research.

I would also like to thank my second supervisor Dr. Tim Hopkins for his support during the early stages of this research.

Finally, I would like to thank Conacyt for the confidence and support they offered me by granting me with the studentship.

Salvador Sotelo

Declaration

This thesis was written by myself. The work and results reported herein are my own except where otherwise stated.

Salvador Sotelo

Table of Contents

List of Figures **5**

List of Tables **8**

Chapter 1 Introduction **11**

1.1 Contributions of the Thesis 12

1.2 Thesis Structure 13

Chapter 2 Background **15**

2.1 Compilers 15

2.1.1 Compiler Optimisations 16

2.1.2 Data Dependencies 20

2.1.3 Control Dependencies 21

2.2 ILP Processor Architectures 22

2.2.1 Pipeline Hazards 23

2.2.2 Scalar and Superscalar Architectures 26

2.2.3 VLIW Architectures 27

2.2.4 EPIC architectures 28

2.2.5 Transport-Triggered Architectures 28

2.3 Asynchronous Control 30

2.4 Summary 32

Chapter 3 Towards Schedulers for Asynchronous Architectures **34**

3.1 Local Scheduling Definitions 35

3.2 Local Scheduling Theory 36

3.2.1 Complexity Issues 40

3.2.2 Types of Scheduling Algorithms 41

3.3 Instruction Scheduling in Synchronous Architectures 43

3.3.1 List Scheduling for Synchronous Platforms 43

3.3.2 Synchronous Model for the Compiler 47

3.3.3 Common Heuristics for List Schedulers 48
3.4 Asynchronous Circuits 50
3.4.1 Introduction 50
3.4.2 Advantages 53
3.4.3 Disadvantages 56
3.4.4 A Compiler Model for Asynchronous Architectures 58
3.4.5 Considerations for the Compiler 61
3.5 Summary 62

Chapter 4 Asynchronous Architectures 64
4.1 Introduction 64
4.2 Review of Asynchronous Architectures 64
4.2.1 AMULET 64
4.2.2 NSR and Fred 66
4.2.3 Caltech Asynchronous Processors 66
4.2.4 Counterflow Architecture 67
4.2.5 SCALP 67
4.3 The Micronet Architectural Model 69
4.3.1 Preliminaries 69
4.3.2 Previous Work 70
4.3.3 Architectural Description 71
4.3.4 Parametric Model 76
4.3.5 Characteristics 79
4.3.6 Event-driven Simulator 80
4.4 Summary 82

Chapter 5 Local Scheduling for Micronet-based Architectures 84
5.1 Introduction 84
5.2 The Influence of Dependencies 85
5.2.1 Data Dependencies 85
5.2.2 The Effects of Resource Dependencies 90
5.2.3 The Combined Effect of Data and Resource Dependencies 90
5.2.4 Applying Penalties to a Schedule 91
5.3 The Penalise True Dependencies (PTD) Scheduler 92
5.3.1 Extending the PTD Measure 97
5.3.2 Safety Conditions for Reducing Penalties 98
5.3.3 Reduction of Resource Penalties 99
5.4 The PTD Scheduler Algorithm 99

5.5 Additional Concepts in the PTD Scheduler . . . 110
5.5.1 Static Memory Disambiguation . . . 110
5.5.2 Subgraphs . . . 113
5.6 Algorithmic Complexity . . . 117
5.7 Discussion . . . 119
5.7.1 Overlapping Penalties . . . 120
5.7.2 Input Sensitivity . . . 120
5.8 Summary . . . 121

Chapter 6 Global Optimisations 124
6.1 Introduction . . . 124
6.2 Related Work . . . 125
6.2.1 Trace Scheduling . . . 126
6.2.2 Superblock Scheduling . . . 126
6.2.3 Hyperblock Scheduling . . . 127
6.2.4 Dominator-path Scheduling . . . 127
6.2.5 Code Motion . . . 127
6.3 Global Scheduling for the Micronet Model . . . 128
6.3.1 Definitions . . . 129
6.3.2 Code Motion for the PTD Scheduler . . . 131
6.3.3 Code Duplication for the PTD Scheduler . . . 135
6.3.4 Safety Conditions . . . 136
6.4 Global Optimiser for the Micronet Model . . . 137
6.4.1 Tail Duplication and Block Merging for the PTD Scheduler 138
6.5 Algorithms . . . 140
6.6 Discussion . . . 145
6.7 Summary . . . 149

Chapter 7 Experimental Results 151
7.1 Introduction . . . 151
7.2 Evaluation Framework . . . 151
7.2.1 SUIF Compiler . . . 152
7.2.2 The Compilation Process . . . 152
7.2.3 Other Schedulers for Comparison . . . 153
7.2.4 Instruction-level Simulator for the Micronet Architecture . 154
7.2.5 Evaluation Process . . . 154
7.3 Benchmarks . . . 155
7.4 Experimental Results . . . 158

7.4.1 Local Optimisations 158
7.4.2 Global Optimisations 186
7.5 Discussion 198
7.6 Summary 201

Chapter 8 Conclusions and Future Work **203**
8.1 PTD Scheduler 204
8.1.1 Penalty Measure 204
8.1.2 Local Optimisations 204
8.1.3 Global Optimisations 205
8.1.4 Performance of the PTD Scheduler 206
8.2 Architectural Model 206
8.3 Future Work 207
8.3.1 Profile Information 208
8.3.2 Other Optimisations 209
8.4 Conclusions 209

Appendix A Published Papers **210**
A.1 Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures 210
A.2 An Improved PTD Scheduler for MAP Architectures 219

Appendix B Description File **227**

Appendix C Comparison of the schedulers **232**
C.1 Local Scheduling 233
C.2 Global Scheduling 235

Bibliography **239**

List of Figures

2.1 Front-end of a typical compiler. 16
2.2 Back-end of a typical compiler. 17
2.3 Movement with compensation code, (a), and without (b). 20
2.4 Classification of an architecture depending on the division of responsibilities between the compiler and the architecture [136], and an extension to it according to [77]. 23
2.5 Pipeline stages, without interlocks, (a), and, with (b). 25
2.6 Synchronous control flow (a), and, asynchronous (b). 31
2.7 (a) Two-phase handshake and, (b) four-phase handshake. 32

3.1 An example of a DAG 37
3.2 (a) Non-preemptive and (b) preemptive schedules. 38
3.3 (a) Non-greedy and (b) greedy schedules. 39
3.4 Representation of the scheduling problem. 40
3.5 (a) An expression-tree and (b) an equivalent DAG representation. 45
3.6 An alternative interpretation for the DAG from Figure 3.1. . . . 48
3.7 New representation of the scheduling problem. 60

4.1 (a) A synchronous pipeline, (b) an asynchronous pipeline and (c) an asynchronous pipeline that exploits spatial parallelism. 71
4.2 Architectural model of the micronet-based datapath. 73
4.3 The micronet operation. 78

5.1 Pipeline execution with true data dependencies: (a) from a memory instruction, and (b) from a non-memory instruction. 88
5.2 Sequence of instructions with penalties. 89
5.3 An example C-code and its inner loop assembly code equivalent. 93
5.4 DAG of the core loop in Figure 5.3. 94
5.5 Makespans of the simulated schedules (y-axis) on a model of the micronet architecture against the penalty measure (x-axis). . . . 95

5.6 Instruction search procedure to reduce penalties in the PTD scheduler. 96
5.7 An example of overlapping penalties. 114
5.8 Basic block from Figure 5.4 decomposed into subgraphs. 115
5.9 Code example in Figure 5.3 after scheduled by the PTD scheduler. 117

6.1 (a) Control flow graph and (b) its control dependence subgraph. 130
6.2 (a) Dominator-tree and (b) postdominator-tree. 131
6.3 Control flow graph with a loop. 132
6.4 Code motion in the PTD Scheduler. 134
6.5 Code duplication in the PTD Scheduler. 136
6.6 Global optimisations applied to the CFG from Figure 6.1 (a). . . 139
6.7 (a) Example of a CFG with a loop and (b) its transformation. . . 140

7.1 Flow of the evaluation process. 155
7.2 Influence of subgraphs and memory disambiguation on the PTD scheduler (1 AU) in terms of percentage improvement in the execution time. 162
7.3 Influence of subgraphs and memory disambiguation on the PTD scheduler (2 AU) in terms of percentage improvement in the execution time. 162
7.4 Influence of subgraphs and memory disambiguation on the PTD scheduler (3 AU) in terms of percentage improvement in the execution time. 163
7.5 Influence of subgraphs and memory disambiguation on the PTD scheduler (4 AU) in terms of percentage improvement in the execution time. 163
7.6 DAG from `livermore` without memory disambiguation. 164
7.7 DAG from `livermore` with memory disambiguation. 164
7.8 Schedule for `livermore` generated with memory disambiguation (a), and, without (b). 165
7.9 DAG of the `fract` benchmark with subgraphs being applied. . . 166
7.10 Schedule generated with (a), and, without subgraphs (b), for a portion of the `fract` benchmark. 167
7.11 Percentage improvement in the issue stalls for 1 AU. 172
7.12 Percentage improvement in the issue stalls for 2 AU. 173
7.13 Percentage improvement in the issue stalls for 3 AU. 174
7.14 Percentage improvement in the issue stalls for 4 AU. 175

7.15 Normalised percentage improvement in the issue stalls for 2 AU. 176
7.16 Normalised percentage improvement in the issue stalls for 3 AU. 177
7.17 Normalised percentage improvement in the issue stalls for 4 AU. 178
7.18 Local scheduler execution performance for the 1 AU configuration. 182
7.19 Local scheduler execution performance for the 2 AU configuration. 182
7.20 Local scheduler execution performance for the 3 AU configuration. 183
7.21 Local scheduler execution performance for the 4 AU configuration. 183
7.22 Simulation results for 1 AU. 191
7.23 Simulation results for 2 AU. 192
7.24 Simulation results for 3 AU. 193
7.25 Simulation results for 4 AU. 194
7.26 The most frequently executed function from `puzzle`. 200

List of Tables

4.1 Latency distribution for the different components in ns. 77

5.1 Degree of the penalties depending of the type of dependency. . . 91
5.2 Static memory disambiguation scheme for the PTD scheduler. . . 113

7.1 Benchmark characteristics. 156
7.2 Distribution of types of instructions in the benchmarks. 157
7.3 Average benchmark compilation times (in seconds). 158
7.4 Standard deviations of the benchmarks' compilation times. . . 158
7.5 Static memory disambiguation statistics. 160
7.6 Percentage reduction in the issue stall by the schedulers (1 AU). 168
7.7 Percentage reduction in the issue stall by the schedulers (2 AU). 168
7.8 Percentage reduction in the issue stall by the schedulers (3 AU). 169
7.9 Percentage reduction in the issue stall by the schedulers (4 AU). 169
7.10 Average issue stall improvements for the four configurations. . . 170
7.11 Number of out-of-order instructions (1 AU). 179
7.12 Number of out-of-order instructions (2 AU). 179
7.13 Number of out-of-order instructions (3 AU). 180
7.14 Number of out-of-order instructions (4 AU). 180
7.15 Performance execution improvement of the local schedulers for the four configurations, with functional units' latencies as defined in Table 4.1. 184
7.16 Performance execution improvement of the local schedulers for the four configurations, with latencies with equal range of values. . . 185
7.17 Performance execution improvement with a memory unit's cache hit:miss ratio of 9:1, and with cache penalty hit:miss ratio of 1:10. 186
7.18 Code motion and code duplication statistics. 187
7.19 Tail duplication/block merging statistics. 188
7.20 Performance execution improvement of code motion for the four configurations. 195

7.21 Performance execution improvement of code duplication for the four configurations. 196
7.22 Performance execution improvement of both code motion and tail duplication for the four configurations. 197
7.23 Percentage reduction in the issue stall by the schedulers for the `puzzle` benchmark (1 AU). 198
7.24 Percentage reduction in the issue stall by the schedulers for the `puzzle` benchmark (2 AU). 198
7.25 Percentage reduction in the issue stall by the schedulers for the `puzzle` benchmark (3 AU). 199
7.26 Percentage reduction in the issue stall by the schedulers for the `puzzle` benchmark (4 AU). 199

C.1 Performance execution improvement for the 1 AU configuration. 233
C.2 Performance execution improvement for the 2 AU configuration. 233
C.3 Performance execution improvement for the 3 AU configuration. 234
C.4 Performance execution improvement for the 4 AU configuration. 234
C.5 Performance execution improvement for the 1 AU configuration. 235
C.6 Performance execution improvement for the 2 AU configuration. 235
C.7 Performance execution improvement for the 3 AU configuration. 236
C.8 Performance execution improvement for the 4 AU configuration. 236
C.9 Performance execution improvement for the 1 AU configuration. 237
C.10 Performance execution improvement for the 2 AU configuration. 237
C.11 Performance execution improvement for the 3 AU configuration. 238
C.12 Performance execution improvement for the 4 AU configuration. 238

List of Algorithms

5.1 ***PTD_scheduler (entry)*** algorithm. 99
5.2 ***PTD_resource_phase (root)*** algorithm. 101
5.3 ***PTD_consecutive_phase (root)*** algorithm. 102
5.4 ***PTD_nonconsecutive_phase (root)*** algorithm. 103
5.5 ***PTD_arrange_left_data (node)*** algorithm. 106
5.6 ***PTD_arrange_right_data (node)*** algorithm. 107
5.7 ***check_left_swap (node, aux)*** algorithm. 108
5.8 ***check_local_move (node, aux)*** algorithm. 109
5.9 ***divide_subgraph (node, pred, size)*** algorithm. 116
6.1 ***code_motion (region)*** algorithm. 142
6.2 ***tail_duplication (region)*** algorithm. 144
6.3 ***move_up (node, source_block, dest_block)*** algorithm. 146
6.4 ***check_global_move(node, aux)*** algorithm. 147
6.5 ***update_best_position (node, aux)*** algorithm. 148

Chapter 1

Introduction

There has recently been a revival of interest in asynchronous computer architectures [161]. Computer architectures have traditionally been synchronous, *i.e.* the components involved in computation and communication are controlled globally by a central clock. Asynchronous architectures, in contrast, sequence the operations using local handshaking protocols [145]. Experimental prototypes of asynchronous processors have been fabricated at the University of Manchester [56][57][63], California [111][113] and Tokio Institute of Technology [121][164]. Notable examples in industry include Phillips' fully asynchronous DCC error corrector [16] and an asynchronous 80C51 micro-controller [59], Sharp's self-timed data-driven multimedia processor [166], and some asynchronous parts in SUN's UltraSPARCIII processor [99].

The execution times of instructions in a synchronous architecture is fixed at the design phase and is expressed in terms of clock cycles. In micronet-based asynchronous architectures, in contrast, the operations proceed at their own speed, which implies that the execution times of instructions would vary in a manner dependent on the data and local delays, and the availability of resources. This poses an interesting problem for the compiler which can no longer assume a deterministic model which has been successfully exploited in instruction scheduling and optimisations for synchronous pipelined architectures [25][67]. For instance, it is now difficult to exploit conditions that cause a datapath to stall, such as data hazards, which are defined in terms of clock cycles, which the compiler uses to reorder instructions to avoid them. The complexity of the task of scheduling instructions to different types of resources is known to be NP-hard [61], and there exists a large body of work on heuristics for scheduling instructions in synchronous architectures [12][34][64][92][100][124][132], but not so for asynchronous ones.

This thesis addresses the issue of efficient scheduling of instructions with uncertain latencies in micronet-based asynchronous architectures. Micronet is a net-

work of entities which compute concurrently and communicate asynchronously. A micronet-based processor [4][6][151] exhibits fine-grained concurrency, both spatial and temporal [5]. The datapath is modelled as a network of functional units, in which each instruction visits the appropriate functional units, and for as long as is necessary to execute that part of the instruction. There are several instructions active at any time, and they compete for functional unit resources, and may even overtake each other. Data consistency is maintained by a register locking mechanism [127] which locks the destination register every time an instruction is issued, and is only released when it is completed. Central to the performance of the architecture is the ability to issue instructions rapidly and keep all the functional units busy.

Generating efficient schedules for such a target is a challenging task. It is uncertain when an instruction will be completed after it has been issued. Also, the order of completion is not known in advance, as the instructions can complete in an out-of-order fashion. The first attempt at a list-based scheduler did not consider any variance in the costs for the functional units and adopted worst-case figures [7].

1.1 Contributions of the Thesis

This thesis has proposed a new scheduling algorithm for asynchronous processor architectures whose instruction latencies are uncertain. The uncertainty is due to instruction-issue stalls caused by data dependencies and resource contention. The scheduling algorithm statically estimates the effects of the instruction stall for a given schedule. Data dependences in consecutive instructions cause the issue unit to stall when waiting for the pending operand to be evaluated. A penalty is assigned based on the parametric cost model for the instruction set. Resource contention occurs when two instructions of the same type are scheduled and there are not enough functional units of that type. A correlation was demonstrated between a higher Penalise True Dependency (PTD) measure and longer execution times of the programs, and vice versa.

The local scheduler which schedules instructions within a basic block is based on the PTD measure. The aim is to minimise the number of penalties within the basic block. The scheduler is prioritised to reduce the higher penalties first, *e.g.* penalties due to load instructions, before dealing with the lower penalties. Once the penalties due to consecutive instructions have been dealt with, the scheduler tackles those due to non-consecutive ones.

The scheduler differs from traditional techniques based on the list scheduler, *i.e.* ones which construct a list of ready instructions. The thesis demonstrates that the complexity of the PTD scheduler is governed by the number of penalties instead of the number of instructions, and its complexity is better than that of the list scheduler.

The thesis also presents a global extension to the PTD scheduler whereby instructions can be moved across basic blocks to improve the penalty measure within local blocks. Global scheduling techniques such as code motion, code and tail duplication and block merging are incorporated within the scheduler.

The local and global optimisation methods were compared with two well-known methods based on the list scheduler. The scheduled programs were executed on a stochastic simulator using instruction set models of the micronet-based processor architecture. The experimental results demonstrated that the PTD scheduler outperformed the other schedulers on issue unit stalls and program execution times.

1.2 Thesis Structure

A description of the remaining chapters of this thesis is presented next.

Chapter 2. This chapter introduces key ideas from three different areas of research which overlap in this thesis: compiler design, ILP architectures and asynchronous hardware design. The compiler flow is described and the compiler optimisations covered in this thesis are located in this flow. The overview of ILP architectures presents their characteristics and limitations. The rôle of asynchronous control in architectural design is introduced. These ideas provide the background for appreciating the contributions in the rest of the thesis.

Chapter 3. The definitions of local scheduling algorithms and their notations are introduced. Scheduling theory and the complexity of the task are reviewed. The revival of the asynchronous style in architectural design is recounted, along with the advantages and disadvantages of such an approach. The influence of an asynchronous architecture on the design of the back-end of the compiler is elaborated.

Chapter 4. This chapter reviews examples of asynchronous architectures in the literature. Next the micronet-based asynchronous architecture is described in detail as this is the target for the schedulers. The modelling of this

architecture in the simulator for evaluating the performance of the compiler optimisations is discussed.

Chapter 5. This chapter proposes an alternative approach to local scheduling for micronet-based asynchronous architectures. Traditional techniques are based on the list-based scheduling algorithm. This chapter introduces the Penalise True Dependency (PTD) measure which statically estimates the effect of issue unit stalls due to data and resource dependencies of different instruction schedules on the execution times of the programs.

Chapter 5 introduces the algorithm of the PTD scheduler and its complexity is analysed. Two methods are described for further improving the scope for parallelism within the basic block.

Chapter 6. This chapter extends the PTD scheduling algorithm so that instructions can be moved across basic blocks to reduce the penalty measure once the local scheduling cannot reduce it any further. The chapter describes the global optimisation techniques such as code motion, code duplication, tail duplication and block merging. Other research in the area of global optimisation is reviewed.

Chapter 7. This chapter presents the framework for evaluating the PTD scheduler on a simulator of the micronet architecture. The benchmarks are scheduled using PTD and two other well-known schedulers, and simulated on different configurations of the micronet architecture. The makespan of the programs are compared against the unscheduled case.

The results are presented separately for local and global optimisation techniques. Data for issue stalls and program execution times are included for local scheduling. For global optimisation, comparisons are made in the case of code motion, tail duplication and the combined effect of both. These results are referred to those of local optimisation techniques.

Chapter 8. This chapter reviews the work presented in the thesis and proposes future work for both the local and global versions of the PTD scheduler.

Chapter 2

Background

The topics covered in this thesis overlap three well-defined areas: compilers, processor architectures and asynchronous circuit design. This chapter provides background information in each of these areas.

2.1 Compilers

A compiler transforms a source program into a target one under precise construction rules. One view of a compiler is that it transforms a high-level specification, *i.e.* a programming language, into a machine-level specification, in a series of steps. The *front-end* of a compiler analyses the input program in three main stages: lexical, syntactic and semantic analyses. The lexical stage recognises and matches lexems to tokens defined by a grammar and filters out unrecognised tokens. Once all the tokens have been identified, the parsing stage builds a hierarchical parse tree according to precise grammatical rules. The final phase analyses the semantic coherence between the identifier types and operators.

The intermediate code, which is an internal representation of the source code (to ease generating the machine code), is produced once the source code successfully passes the three phases. The *back-end* of the compiler reads this intermediate code, performs optimisations and generates code for the target machine. Figure 2.1 depicts the front-end of a typical compiler, while Figure 2.2 shows the three major functions in the back-end of the compiler.

The partition of the compilation process into front- and back-ends allows the same source code to be translated into different machine codes independently of the analyses. Although optimisations can be performed after intermediate code generation, their purpose is to remove redundancies from the source code and any introduced during the generation of the intermediate code. This type of optimisation is not biased by the target platform, but instead prepares the

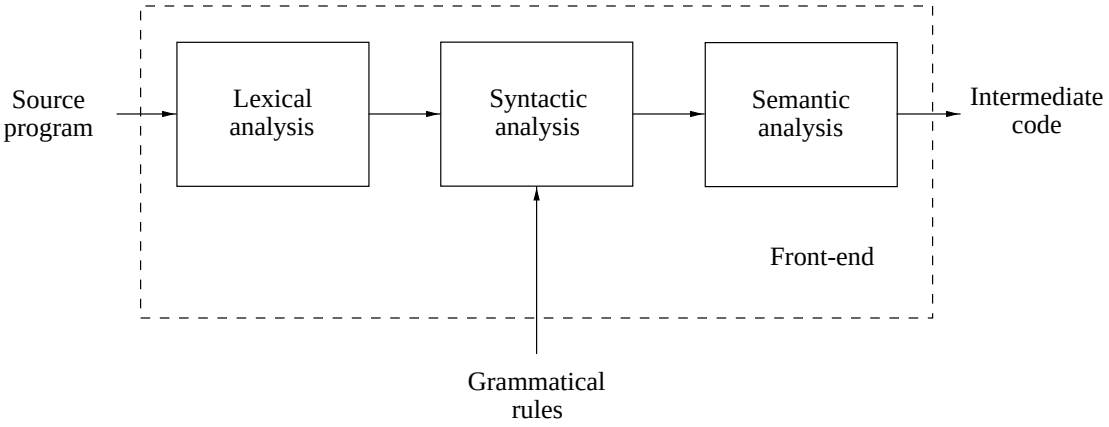


Figure 2.1: Front-end of a typical compiler.

intermediate code for the machine-code generation phase. These are known as platform-independent optimisations and examples include *common subexpression elimination*, *dead-code elimination* and *constant propagation* [2].

In contrast, the optimisations performed after machine-code generation are intended to tune the performance on the target machine by optimising **register allocation** and **instruction scheduling**. The optimisations performed at this level are closely matched to the machine model and relate to the instruction set. The following sections will discuss these so-called Instruction-level Parallelism (ILP) optimisations.

2.1.1 Compiler Optimisations

The nature of intermediate code representation may vary significantly depending on the instruction set and the target architecture. The most common representations include postfix notations, virtual machine representations such as stack-machine representation, graphical representations such as expression-trees and direct acyclic graphs (DAG), and three-address representations [2].

It is the case that ILP optimisations will be driven by the type of intermediate code representations, which in turn is influenced by the particular target architecture. For example, in stack-based environments such as the Java Virtual Machine [104], stack-machine representations mimic a stack-like behaviour in which instruction operands are "pushed" onto the stack and "popped" for execution with the resultant value being pushed back onto the stack. With stack-machine representations, optimisations must be performed to avoid unnecessary

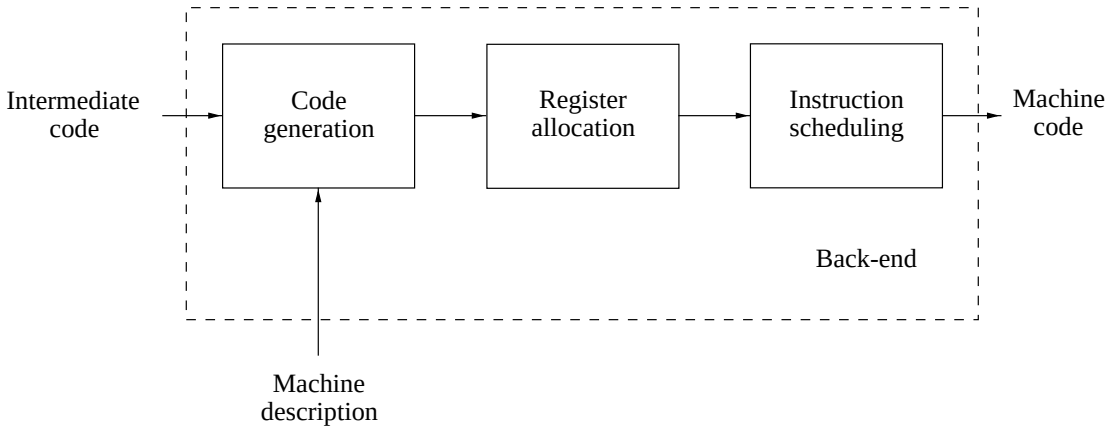


Figure 2.2: Back-end of a typical compiler.

operand push and pop operations [37].

In register-based architectures, however, it is more suitable to use graphical or even three-address schemes to represent the intermediate code. Nowadays, most processors have instruction sets with three operands: one destination register, and two source registers. Optimisations for register-based architectures seek to minimise the number of registers used.

ILP optimisation seek efficient usage of both storage resources (register allocation) and fast execution times through instruction scheduling. The search space of solutions is vast, and moreover, the ordering suggested in Figure 2.2 is not entirely fixed. The interaction between instruction scheduling and register allocation is complex and is a research area in its own right [21]. Performing register allocation before instruction scheduling reduces ILP, because the former tries to reuse systematically the registers (an effect caused by minimising the number of registers). Conversely, if instruction scheduling precedes register allocation, then the lifetimes of the registers may increase, which in turn will require a greater number of them, contrary to the register allocation [23][118][171].

In order to reduce the counter effects of register allocation with respect to instruction scheduling and to evaluate the effectiveness of the latter in an asynchronous target, this thesis will only concentrate on issues regarding instruction scheduling, and will therefore assume the scheme in Figure 2.2.

2.1.1.1 Register Allocation

Register allocation is an optimisation technique to make efficient use of the registers. Registers store intermediate results during a computation and, as such,

are a scarce and costly resource, and therefore limited. Optimising the use of intermediate results reduces the need to store results in memory (which is even slower) — a process called *spilling* code. Hardware solutions tend to increase the number of registers or to include the use of cache mechanisms to reduce storage time and loading back a temporary result into a register.

The task of the software register allocator is to map temporary values, usually called *pseudo-registers*, at the intermediate code level into physical registers, keeping in mind their scarcity. The difficulty of register allocation though, is that different values have different "liveness", *i.e.* the total time that they must be kept *alive* in registers, so they often overlap. The register assignment must be carefully optimised with the aid of *interference graphs* representing the overlapping life-ranges of the pseudo-registers, and the use of the graph colouring algorithm [24]. The complexity of register allocation has been acknowledged in the past resulting in alternative solutions to find optimal and near-optimal results [14].

2.1.1.2 Instruction Scheduling

Instruction scheduling aims to reorder the code output from the generation phase to improve its execution time. The reordering should preserve the semantics of the program while exploiting the architecture to improve performance. Local scheduling (unlike the global one) confines the reordering of instructions to those in the basic blocks. A basic block is defined as the group of instructions delimited by a single entry and a single exit. The instructions in a basic block share the same control properties. The functions in the program are decomposed into basic blocks connected by a control structure which reflects the semantics of the function. Chapters 3 and 5 discuss local ILP optimisations, and Chapter 6 covers global ILP optimisations.

Acyclic optimisations take into account multiple basic blocks within an acyclic region, and instructions are moved to other basic blocks in the program. Cyclic optimisations perform optimisations from different iterations instead of just one. These are described in the following sections.

2.1.1.3 Acyclic Optimisations

Acyclic optimisations can be regarded as a generalisation of local scheduling in which the instruction reordering is not limited to the basic block boundary. The average number of instructions within a basic block is around twenty. Moving instructions between basic blocks increases the scope for ILP. However, in order to maintain the semantics of the program, copies of the instruction called

compensation code, may have to be replicated.

The movement of instructions can take place either with the need for compensation code or without, which is either in the same or opposite direction to the flow of control. These four cases are depicted in Figure 2.3: B_1 to B_6 are basic blocks with B_1 being a fork instruction for B_2 and B_3, and B_6 being the join block for B_4 and B_5. Figure 2.3 (a) shows both cases when code movement requires compensation copies (represented by the dashed arrows) since moving instructions away from the fork (B_1) and the join (B_6) blocks into one of the paths would cause instruction executions to miss in the other path. Instruction movements in Figure 2.3 (b) on the other hand, do not require the addition of compensation code since irrespective of the path taken, instructions will be executed as soon as the control flow arrives at the branch block B_1, or as late as the control flow reaches the join block B_6. In these cases, movement without copies introduces redundant executions if the path taken is the opposite to where the instruction was originally located.

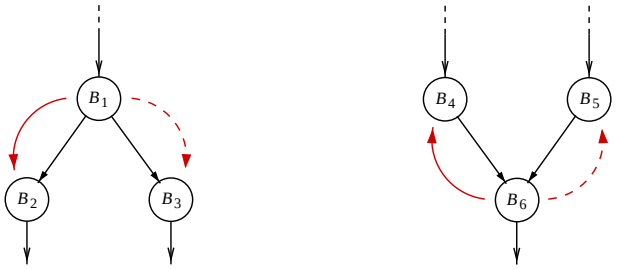
The issue in global acyclic optimisations is that the effectiveness of moving instructions depends on the run-time behaviour of the program. For example, if the blocks shown in Figure 2.3 are not frequently executed, then the benefit due to those movements may be insignificant. In addition, instruction movement with copies can increase the size of the program and the overhead on performance of redundant instructions. The performance issues of acyclic optimisations are discussed in more detail in Chapter 6 (Section 6.2).

2.1.1.4 Cyclic Optimisations

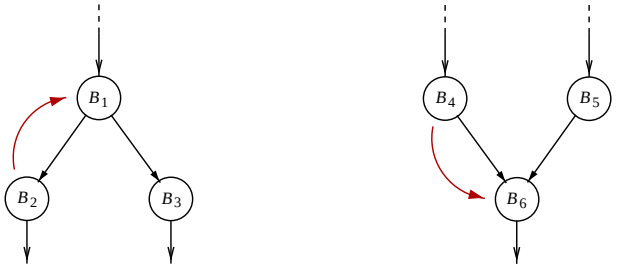
Cyclic optimisations exploit the cycles in the control flow to enable optimisations not only through basic blocks, but also through cycle iterations. These optimisations are driven by programs in which the control flow spends considerable time in the core of the loops.

Data dependence information carried from the front-end of the compiler can help identify instructions that are independent across iterations. These can be grouped together and scheduled, thus providing aggressive optimising characteristics.

Another example of cyclic optimisation is to "unfold" the loop body, *i.e.* carry instructions from subsequent iterations to increase the size of the loop body, with the aim of augmenting ILP. This technique, also called *loop unrolling*, has been well studied [2][103]. Loop unrolling not only increases parallelism, but also reduces redundant branch comparisons and the use of loop indexing variables.



(a)



(b)

Figure 2.3: Movement with compensation code, (a), and without (b).

The unrolling of the loop is limited by the following: there will be greater pressure in the register allocator phase as the registers are overcommitted, and also the expansion of the code is likely to overflow in the instruction cache [40].

Both local and global optimisations are governed by data and control dependencies, which respect the semantics of the program. Data and control dependencies are described in the following sections.

2.1.2 Data Dependencies

Data dependencies must be respected throughout the compilation process. The use of pseudo-registers during code generation, and registers after register allocation, reflect these dependencies. Three different types of data dependencies exist:

True dependencies. The Read-After-Write (RAW), or true dependencies, occur when one instruction requires the contents of another, and must wait until the latter result is written. These dependencies cannot be removed and represent the flow of data during a computation and must be preserved.

An instruction that is truly dependent on another cannot be positioned before it.

False dependencies. False or Write-After-Read (WAR) dependencies occur when one instruction needs to store a result, but this location which is either in memory, a register or a pseudo-register is to be read by another instruction. The former instruction must wait until the current value in the storage location has been read by the second instruction, before committing its result.

If the first instruction is placed ahead of the second one, then the value will be overwritten by the time it is read by the latter, and would therefore be the wrong value.

Output dependencies. Output or Write-After-Write (WAW) dependencies occur when the same destination storage location is due to be written by two different instructions. In which case, the second one has to wait until the previous one has written into the destination.

This is similar to false dependencies, as reversing the order of the instructions will result in the variable being assigned the wrong value.

False and output dependencies can be removed if the name of the destination variable or the destination register is different from the one where it is originally read from in a WAR dependency, or where it is originally written to in a WAW dependency. In other words, if the destination location is *renamed*, then the dependency no longer applies. In some references, false and output dependencies are also termed as *name* dependencies.

2.1.3 Control Dependencies

Control dependencies occur when the execution of an instruction depends on the result of a conditional branch, such as an `if` statement. If the condition is true then the branch is taken and the instruction is executed; otherwise, not.

With control dependencies, the control-dependent instructions cannot be moved out of the `if` section, as this would force the instruction to be executed under any condition. Similarly, an instruction cannot be moved inside the `if` section, as it would only be executed if that path of control was taken, unless a copy of it is placed in the `else` section.

Data dependencies are defined entirely statically, whereas control ones have to be resolved at run-time. Data and control dependencies reflect sequentiality in the program, and their removal, where possible, is an important aspect of ILP.

2.2 ILP Processor Architectures

Instruction-level parallelism architectures, as the name implies, exploit concurrency at the instruction level. An instruction is composed of an opcode type, represented by a unique mnemonic defined in the instruction set, and the operands which include a destination operand and one or more source ones. The datapath of an ILP architecture allows more than one instruction to be active at the same time. An active instruction is one which is located in one of the following steps in its execution: instruction fetch, decode, operand fetch, execution or write-back. The fetch unit fetches the instructions from the instruction cache (or from memory in the absence of one). The opcode is next decoded to determine the instruction destination in the datapath to reserve the appropriate resources. Once the instruction is decoded, its source operand values are retrieved from the registers. In the execution stage, the functional unit executes its operation and outputs the result. This result is written into the destination register in the write-back stage (as determined by the destination operand in the instruction).

This process is repeated for all the instructions. However, data and control dependencies impose restrictions during their execution. One way of characterising ILP architectures is in the way in which ILP parallelism is interpreted, or in other words, how much of the data dependence information is passed from the compiler to be interpreted by the architecture [136]. This characterisation is illustrated in Figure 2.4. *Sequential* architectures such as scalar and superscalar ones, do not interpret any information from the compiler. The relationships between active instructions must be determined by the processor at run-time in order to maintain the correct order of execution. On the other hand, *independent* architectures, such as VLIW ones, rely entirely upon the compiler to provide an independent stream of instructions for execution. They do not implement any relationship analysis at run-time. The control logic for these processors is therefore much simplified. Section 2.2.2 and 2.2.3 describe the sequential and independent architectures in more detail.

In between VLIW and superscalar architectures one can find the EPIC (Explicitly Parallel Instruction Computing) architectures [143]. EPIC architectures share with VLIW architectures in that the compiler is required to identify groups of independent operations to form very long instructions. However, the architecture is now responsible to assign these operations to functional units and coordinate the timing of their execution [142]. In a certain way, EPIC architectures take the best of both VLIW and superscalar architectures.

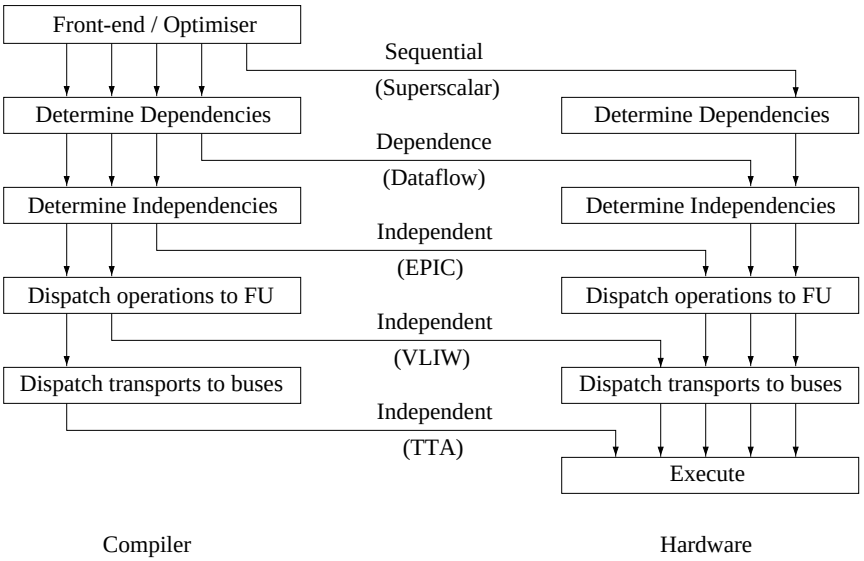


Figure 2.4: Classification of an architecture depending on the division of responsibilities between the compiler and the architecture [136], and an extension to it according to [77].

Another completely different approach to the previous ones is in the case of Transport Triggered Architectures (TTA) [35]. A TTA is based on an interconnection network in order to connect the functional units with the register file. With such a scheme, the compiler for a TTA has even more responsibilities than the one for a VLIW architecture as it can be seen in Figure 2.4, since it has to decide not only the assignment of operations to functional units, but also the paths that such instructions will be require to take within the interconnection network.

EPIC and TTA architectures are explained in more detail in Sections 2.2.4 and 2.2.5, respectively.

2.2.1 Pipeline Hazards

Pipelining is a technique for exploiting concurrency in the temporal domain. In a pipelined architecture several instructions are in flight executing in the different stages: being fetched, decoded, their operands being fetched and being executed, as long as they do not interfere with each other. These architectures require n instructions to fill a n-stage pipelined datapath to achieve maximum throughput and resource utilisation. In such a scheme, if the number of pipeline stages is

increased, then more instructions can be active at any time, and thereby achieving greater ILP. In theory at most, the execution time can be reduced by up to n times when compared to a non-pipelined datapath[1]. Figure 2.5(a) shows a pipeline stream of instructions in a 4-stage pipeline.

However, as mentioned in Sections 2.1.2 and 2.1.3, data and control dependencies enforce sequentiality in the instruction execution order. Therefore, the pipelined architecture must ensure that the correct ordering is preserved by stalling some stages for a period of time, an effect called pipeline hazards. These include *data hazards* due to data dependencies, *control hazards* due to control dependencies and *structural hazards* due to resource conflicts. All of these will restraint the continuous flow of operations in the datapath, causing "bubbles" in the pipeline, and thus increasing the execution time.

In the case of data hazards, the stall is related to the type of data dependency. For example, with a true dependency, the instruction that requires the result of the previous one will not have the result ready for it to be read at the operand fetch stage. For correct operation, the architecture must apply an *interlock* (a "bubble" in the pipeline) to the second instruction, so that it will remain stalled until its operand(s) is/are fetched. Figure 2.5(b) shows a pipeline with two interlocks produced by data dependencies. In the example, instruction I_3 requires the result of instruction I_1; at the time that the operand is to be read by instruction I_3, it has not yet been written back by instruction I_1. Therefore, I_3 must wait for a clock cycle before resuming execution. Similarly, instruction I_4 depends on I_2, and causing another interlock.

For name dependencies, *i.e.* WAR and WAW, interlocks are applied at later stages. In an output dependency, the architecture may only stall the write-back of an instruction to ensure that the previous write-back takes place first.

One hardware solution for solving the problem of data hazards due to true dependencies is to forward the results or bypassing. In addition to writing the result in the register file, the functional unit forwards the result directly to the fetch stage where it is needed, in order to avoid the hardware interlock. However, if the true dependency stands for a load instruction, then the data hazard may not be avoided. A load instruction may incur a cache miss, in which case the time to load the value into the register will be delayed. In pipelined architectures the compiler is partially responsible for avoiding such cases. The compiler must ensure that the pipeline is full of independent instructions, and when that is not

[1]In practice, the speedup is bound by the amount of parallelism in the code and by the clock cycle overhead with large number of pipeline stages [76]. The performance/cost ratio has also a peak in terms of the cost for all the logic stages and the latch and delay costs [84].

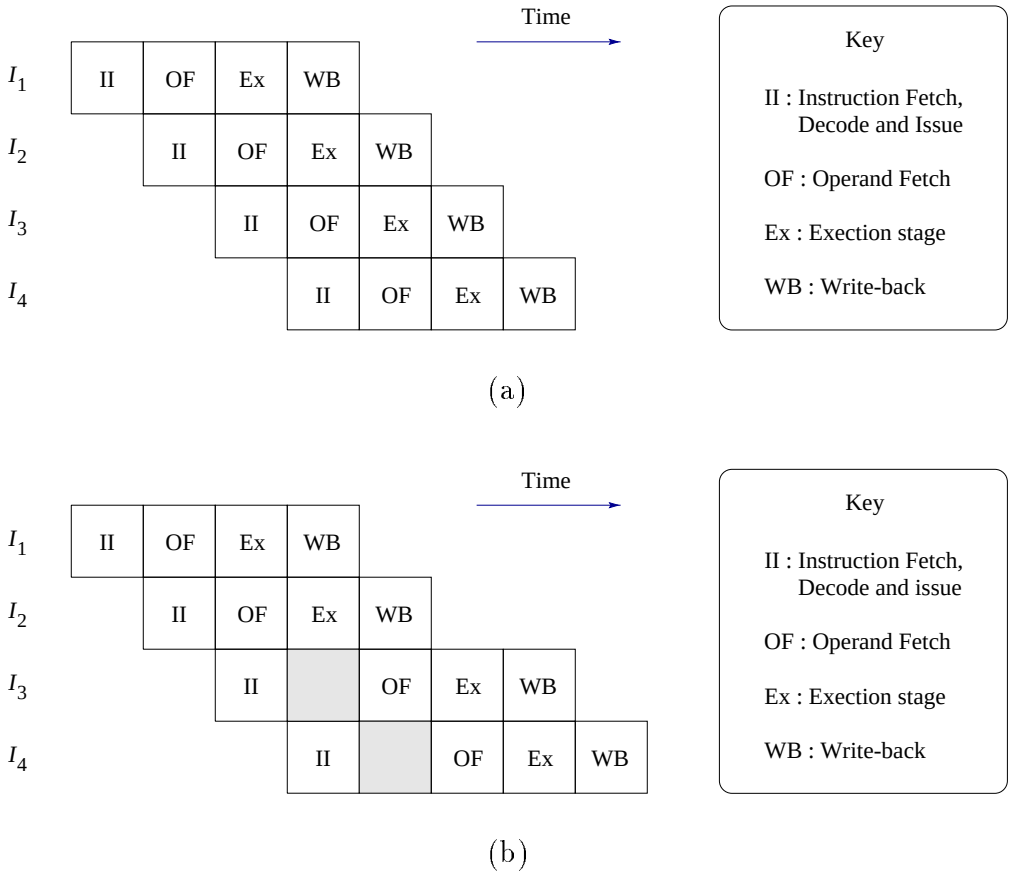


Figure 2.5: Pipeline stages, without interlocks, (a), and, with (b).

possible, it must try and avoid combination of instructions with true dependencies from a load.

Data hazards due to name dependencies on the other hand, can be avoided by renaming the destination register that is common. Register renaming can be implemented either dynamically by the hardware or statically by the compiler. A hardware register renaming scheme consists of logical registers, as seen by the compiler, that are transparently mapped to a greater number of physical registers. If, for example, there are two instructions writing to the same logical register, two physical registers can be allocated to hold the values. A mapping table is used in the decode stage to map logical to physical registers. In software, register renaming can be easily achieved by using a different register name each time. At the software level, the overuse in the number of registers may cause spilling code (as described in Section 2.1.1.1), which might negate any gains due to removing the name dependencies.

A control hazard due to a conditional branch instruction arises because instructions following the branch one cannot start their execution until the outcome of the branch is resolved. Should the branch be taken, the program counter is updated with the new branch address and fetching can be resumed; otherwise, fetching continues with the instructions following the branch. In either case, there is a time delay represented in terms of clock cycles or *delay slots*, until the branch instruction is completed. The delay slots can be filled with control independent instructions such as as "nop" (no-operation).

Another solution is to use branch prediction [149]. If the branch outcome is known by the time a subsequent instruction from the branch needs to be fetched, then the pipeline does not have to be stalled. Branch prediction allows instructions after a predicted branch to be executed speculatively; if the branch was mispredicted then the results have to be backtracked. Branch predictors use a history table where the occurrence of previous branches is stored. The individual number of *hits* and *misses* will decide the likelihood of a branch. The prediction rates achieved nowadays (between 80% and 95% depending on the type of branch prediction and the size of the history table) outweigh the cost of the misprediction penalty, *i.e.* recovering the state of the processor (pipeline and flags) before the misprediction.

The last possible hazard in pipelined architectures is the structural hazard. This occurs when there are resource conflicts and the hardware cannot support the operating conditions for a particular set of active instructions. A resource conflict occurs when an instruction has all its operands ready but there are no functional units available or the buses are busy.

2.2.2 Scalar and Superscalar Architectures

There are two policies for issuing instructions: *in-order*, that issues all the instructions in the same order as they were fetched; and *out-of-order*, that issues instructions not necessarily in the original program order.

When both issue and execution are implemented in-order, then dependent instructions and resource conflicts might stall subsequent instructions. A subsequent and independent instruction will be forced to stall its write-back in order to maintain execution order. Alternatively, if this instruction is allowed to proceed with the help of additional hardware, then the outcome will be in-order issue, but out-of-order execution. A hardware mechanism that allows out-of-order execution is the *scoreboard*. The scoreboard holds dependency records for all the

instructions in flight[2] and guarantees that they are issued as soon as their operands become available; it also checks for data, control and structural hazards. This is achieved by gathering status information about the functional units, the register file and the instructions themselves. All this information serves to determine which instruction can be issued. The penalty incurred by implementing the scoreboard is compensated by the extra parallelism gained from enabling concurrent instructions to avoid being stalled.

These policies and mechanisms also apply to superscalar architectures. A superscalar architecture datapath is capable of issuing more than one instruction per cycle. A n-issue superscalar processor fetches and decodes n instructions at a time. To achieve this the complete datapath consists of n-parallel pipelines. The control logic in such architectures are more complicated due to the checking of dependencies along the different stages in the datapath, and resources such as functional units, buses and register file's write ports must be arbitrated and managed efficiently.

In both scalar and superscalar architectures, out-of-order issue and execution imply dynamic re-arrangements at run-time, since one instruction can overtake another in the case of data hazards or if the latter requires more time to complete. Architectures with dynamic scheduling make less demands on the compiler as shown in [102], although local scheduling certainly contributes to performance in dynamic scheduling.

2.2.3 VLIW Architectures

Very-Long Instruction Word (VLIW) architectures, as the name implies, pack instructions into a single, long instruction word. This means that when a VLI word is fetched, n independent operations can be decoded at the same time, and n operations can be issued concurrently to be executed in parallel. The number of operations per word can vary from 8 in the Multiflow computer [144], to up to 20 in the IBM VLIW processor [117].

One of the principal features of VLIW architectures is that the task of finding independent instructions is performed by the compiler and not by the hardware. The compiler is responsible for grouping independent operations into VLI words. Subsequently, the control logic (fetch and decode stages) in VLIW architectures is simpler since it does not have to identify dependencies between active instructions and perform run-time resource management. The grouping of VLIW instructions

[2]The number of instructions in flight depends on the size of the scoreboard.

by the compiler also specifies the mapping of operations to functional units. A VLIW architecture has neither dynamic scheduling, nor out-of-order execution; rather the instruction issue order is decided statically. When the parallelism in the code is less than the maximum ILP of the architecture, then the compiler must schedule no-operations to fill the vacant slots in the VLI word.

VLIW architectures are targeted at scientific applications where the bulk of the program execution time is spent in core loops with potential for parallelism. For general-purpose applications VLIW architectures does not compare as well as superscalar machines since their efficiency, *i.e.* the ratio of useful instructions to the total number of instructions, decreases with the increase in the number of no-operations.

2.2.4 EPIC architectures

Explicitly Parallel Instruction Computing (EPIC) architectures [143] can be considered as an evolution of VLIW architectures. One of the drawbacks from VLIW architectures is that they are not compatible across different implementations, since compiled code for one implementation with a particular set of functional units and latencies will not run properly in another one with a different set of parameters.

The compiler for an EPIC architecture is required to determine the data dependencies in the code and group concurrent operations into VLIW instructions in a similar manner as in VLIW architectures, but the architecture is responsible for mapping them to functional units and coordinate the start of their execution. This particular characteristic allows the architecture to execute code from another implementation to run without compatibility problems.

An EPIC architecture supports higher levels of ILP through the use of predicated and speculative execution to overcome frequent control transfers and ambiguous memory dependencies [9][148]. These techniques, which have often been used in superscalar architectures, help EPIC architectures to perform well in more general-purpose applications. Since EPIC architectures time the execution of the operations, they are more capable of handling exceptions and interruptions.

2.2.5 Transport-Triggered Architectures

Transport-Triggered Architectures (TTA) [36][79] represent another type of architectures evolved from VLIW ones, but with even more responsibilities given at compile-time and less at run-time [35].

One of the main differences between VLIW and TTA architectures is that the functional units in the latter do not necessarily have dedicated connections to the register file, as in a traditional VLIW machine. Instead, they are connected through an interconnection network with the goal of having a better scalability by reducing the port requirements of the register file [78]. The interconnection network consists of data transport or *move* buses that enable functional units to communicate with each other and with the register file through sockets. The inputs and outputs of each functional unit are connected to the network by input and output sockets, respectively. The input sockets act as data multiplexers, whereas the output sockets act as demultiplexers. A fully connected network implies that every socket is connected to all of the move buses, which simplifies the code generation by the compiler, but with a side effect of impacting the cycle time. The connectivity of the sockets can be tailored so that certain functional units share more paths in common, *e.g.* memory units and arithmetic units, since the former requires an adder to determine the memory address.

A TTA instruction is composed of one or more *move* operations which involve the data transport between two registers. The registers can be separated into *operand*, *trigger* and *result* registers. The moves between these registers represent the movement of data from the register file to a move bus (operand), from one socket of the move bus to the input socket of a functional unit (trigger), which effectively causes a functional unit to start the operation, and from the functional unit to an output socket (result).

It is the task for the compiler to optimise such move operations and schedule them. All these move operations are pipelined to obtain a high throughput, and the scheduler is responsible for that. Some of the optimisations performed by the scheduler are TTA-specific and these include *bypassing*, *operand and socket sharing* and *result move elimination* [77][89]. These optimisations have to deal with the efficient use of the interconnection network.

The compiler for a TTA architecture also has the responsibility of performing resource assignment which is a more complex task than for VLIW ones since not only functional units need to be assigned to instructions, but move buses and sockets to move operations as well. TTA architectures exhibit a dataflow characteristic in that data transports from different move operations must match so that functional units will operate on correct values. This represents an extra challenge for the compiler since there are more resources needed per instruction than in other type of architectures [80].

2.3 Asynchronous Control

The communication of data between components of *synchronous* systems is divided into two parts: one channel is dedicated to the data transmission, mainly through a directional bus from the sender to the receiver, while the other is the control channel represented by a uni- or bi-directional bus used for high-level control. These buses have timing restrictions in that the output data from the first component must be settled and the second component has some period of time to read the correct signals. This timing constraint is co-ordinated by a global signal – the clock – that feeds both components to synchronise them. Figure 2.6 (a) depicts two modules communicating data through a data bus, while control signals from both modules can be sent and received back and forth from a control unit. This scheme is centralised around the control unit and it is via the clock that it dictates the timing operation for all the components in the system. The control unit starts the communication process and regulates the flow of data through the control bus until its completion.

An alternative way of communication between these components is by decentralising the global synchronisation of the previous mechanism, with the removal of the clock. In this approach components communicate with each other via a handshaking mechanism. In this scheme the sender is responsible for the start of the transaction and the receiver responds when it is ready to receive.

Figure 2.6 (b) shows two modules: Module 1 starts a new transaction with a *request* signal and awaits an *acknowledge* signal from the receiver (Module 2) before sending the data. Request/acknowledge mechanisms are convenient for asynchronous communication since delays are prone to vary. Request and acknowledge signals can be active during positive or negative edge transitions. When positive and negative edge transitions are treated equally during a handshake, then the handshake is called *two-phase*. If only positive-edge transitions are used in the handshake, it is called *four-phase* [161]. The two-phase and four-phase handshakes are shown in Figure 2.7. The term "two-phase" stems from the fact that two events take place: the first phase is represented by the sender requesting transfer of data (1), and the second phase by the actual transfer of the data (2), as depicted in Figure 2.7 (a). Similarly, four events take place in the case of the four-phase: (1) the sender starts the transaction, (2) the receiver acknowledges, (3) the sender stops sending the data, and (4), the receiver finishes the handshake, as shown in Figure 2.7 (b). The dashed lines represent a signal to the sender that the receiver has started or completed the transaction, and thus it can proceed to

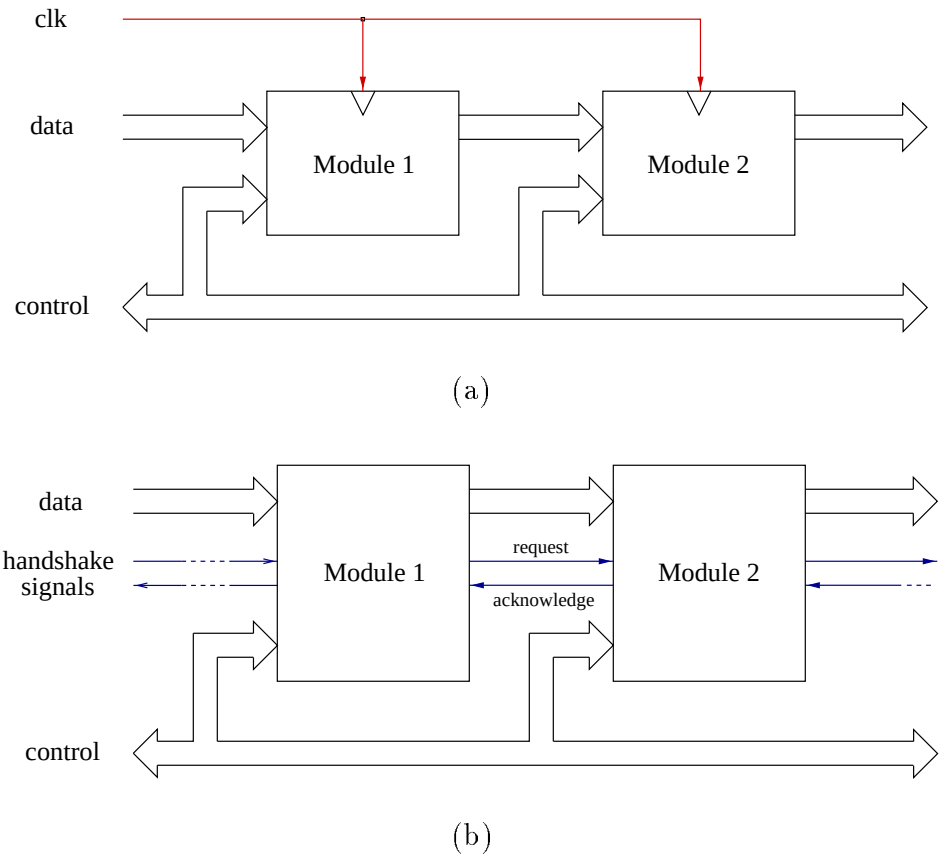


Figure 2.6: Synchronous control flow (a), and, asynchronous (b).

complete the transaction or start a new one.

It is obvious that there are twice the number of transactions in a four-phase handshake than a two-phase one. The problem with the two-phase handshake approach is that circuit implementations require larger – and therefore slower – gates. Usually XOR gates are required as opposed to AND and OR gates as used in four-phase designs [161]. Another characteristic found in circuits with four-phase handshakes is that the second half of the handshake (events 3 and 4) can be concurrent with the computation. This is advantageous considering that transactions spend most of the time in computation rather than communication. Four-phase circuits can achieve higher performances and lower costs than two-phase implementations using level-sensitive technologies such as CMOS [55].

The handshake examples in Figure 2.7 are called *bundled-data* handshakes and assume that the data is available in the bus prior to the control signal from the sender, or in other words, it is assumed that the delay from the request is longer

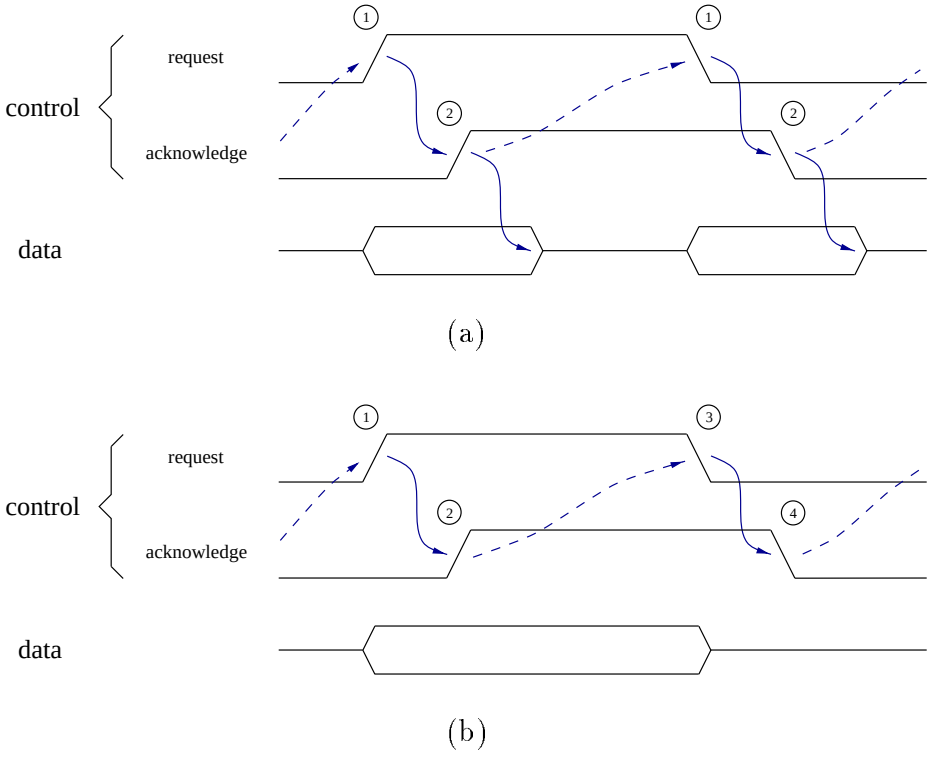


Figure 2.7: (a) Two-phase handshake and, (b) four-phase handshake.

than the delay from each of the data bus signals. This assumption violates the *delay-insensitive* model where no timing restrictions are applied [172][173]. With delay-insensitive circuits signal delays are assumed to be unbounded, therefore a valid data bit is required in order to distinguish between a no-change signal from a delayed one. The extra line for each of the data bus signals ensures that a transition from the previous data to the actual data has taken place. One line can be used to represent the previous value while the other line represents the change in transition. With such schemes, the request signal by the sender is not necessary.

2.4 Summary

This chapter has reviewed background concepts in three major areas: compilers and the rôle of code optimisation, classification of ILP architectures such as superscalar and VLIW ones, and asynchrony as a method of circuit and system design.

The following chapters build upon these areas. Work described in Chapters 3, 5 and 6 is concerned with code optimisation, covering local scheduling in Chapters 3 and 5 and global scheduling in Chapter 6. These are targeted for an asynchronous scalar architecture whose model of operation is described in detail in Chapter 4.

Chapter 3

Towards Schedulers for Asynchronous Architectures

Progress in silicon technology in the 70's had resulted in the emergence of faster and more complex processors as epitomised by the Complex Instruction Set Computers (CISC). Compilers for high-level programming languages had matured to exploit the hardware capabilities. The emergence of VLSI in the early 80's saw however, a re-evaluation of processor architectures and a greater interest in the interaction between the compiler and architectures, as evidenced by Reduced Instruction Set Computers (RISC).

This interaction between the compiler and the hardware has been an important consideration in the design of high-performance systems. The concept behind early RISC architectures was to redefine a reduced instruction set resulting in a fast stream of short-cycle instructions, instead of a shorter stream of more complex instructions, as experienced in the CISC approach. Immediate effects would be locality in the memory hierarchy, faster throughput, and hardware simplicity [125]. The advantage of using a bank of restricted number of registers to store intermediate results, instead of continuously loading and storing them in the memory demanded the optimisation of their usage, which was the responsibility of the compiler.

One of the objectives of the back-end of a compiler is to convert intermediate code into assembly instructions as defined in the instruction set of the processor. This task, however, involves operations that require special attention: these are register allocation, resource mapping and instruction scheduling. Each is a non-trivial problem and their individual interactions have been studied in the context of RISC architectures [23][131]. In particular, scheduling techniques have matured considerably for synchronous architectures. The early work in embedded systems, where highly optimised code must meet tight deadlines due to timing restrictions

in order to perform a task, stamped a strong impression in what code optimisation can achieve in co-operation with the hardware [157].

One of the motivation of this thesis lies in the fact that very little work has been undertaken to optimise code for asynchronous processors [7]. It is our belief that the scheduling assumptions for synchronous targets differ considerably from those for an asynchronous target, which will influence the scheduling mechanism. The key is to identify particular features of an architecture and characterise them in the compiler. Later in the chapter we will show how this is modelled by the compiler in some synchronous systems, and the difficulties to do so under asynchronous behaviour, given its unique properties.

3.1 Local Scheduling Definitions

The local scheduling model which is considered in this thesis is based upon instructions which are restricted in some way and executed in specific functional units with a corresponding assigned cost. The six-tuple $(\mathfrak{I}, \prec, \mathfrak{R}, \mathfrak{T}, \mathfrak{L}, \mathfrak{E})$ is used to represent the graph $\mathfrak{G}$, and is defined as follows:

$\mathfrak{G}$: $\mathfrak{G}$ is a graph defined as $\mathfrak{G}\ (\mathfrak{I}, \prec, \mathfrak{T}, \mathfrak{R}, \mathfrak{L}, \mathfrak{E})$.

$\mathfrak{I}$: $\mathfrak{I} = \{I_1, I_2, \ldots, I_n\}$ is the set of n instructions of a basic block to be executed. In early scheduling work this set was referred to as the set of tasks.

$\prec$: $\prec$ is an irreflexive partial order which specifies the set of precedence constraints in $\mathfrak{I}$. For two instructions I_x and $I_y \in \mathfrak{I}$, $I_x \prec I_y$ implies that instruction I_x must finish its execution before instruction I_y can start. In addition, the subset $\ll \in \prec$ denotes that for three instructions I_x, I_y and I_z, where $I_x \ll I_y$, there is no I_z such that $I_x \prec I_z \prec I_y$.

$\mathfrak{T}$: $\mathfrak{T} = \{T_1, T_2, \ldots, T_t\}$ represents the set of different types, t, of functional units contained in the architecture.

$\mathfrak{R}$: $\mathfrak{R}$ is the set of resources or functional units. For every instruction I_i, with $1 \leqslant i \leqslant n$, there is at least one functional unit associated with it. That is, $\mathfrak{R} = \{R_{1\,T_a}(I_i), \ldots, R_{p\,T_a}(I_i)\}$, with p being the number of functional units and $1 \leqslant a \leqslant t$.

$\mathfrak{L}$: $\mathfrak{L}$ is the set of execution times or *latencies* for the set $\mathfrak{I}$ and is defined as $\mathfrak{L} = \{L_1(R_{j\,T_a}), \ldots, L_n(R_{j\,T_a})\}$ with $1 \leqslant j \leqslant p$ and $1 \leqslant a \leqslant t$.

These values will depend upon architectural characteristics as will be shown in Sections 3.3.2 and 3.4.4 for the synchronous and the asynchronous approaches, respectively.

$\mathfrak{E}$: $\mathfrak{E}$ is the set of communication costs for every instruction for propagating results. It is defined by $\mathfrak{E} = \{E_1, E_2, \ldots, E_e\}$, if e is the total number of edges and can be considered proportional to the number of instructions n. Like $\mathfrak{L}$, the values of $\mathfrak{E}$ depend upon communication delays in the architecture, but the number e is related to $\prec$, where $|\prec| = e$.

If there is a path from instruction I_x to instruction I_y, where I_x and $I_y \in \mathfrak{I}$ and $I_x \prec I_y$, I_x is called a *predecessor* of I_y and instruction I_y is called a *successor* of I_x. If $I_x \ll I_y$, then I_x is the *immediate predecessor* of I_y and I_y is the *immediate successor* of I_x.

The partial order $\prec$ is acyclic. It has no transitive or redundant edges which implies that it cannot represent loops. The graph $\mathfrak{G}$ is therefore a Directed Acyclic Graph, or a DAG for short. A graphical representation of a DAG is given in Figure 3.1, where nodes (instructions) from $\mathfrak{I}$ are connected to their successors and predecessors via directional edges $\prec$. Such a DAG is a basic block, as shown in Figure 3.1[1], if it has single entry and exit nodes.

If two instructions I_x and $I_y \in \mathfrak{I}$ are not related in $\prec$, *i.e.* $I_x \not\prec I_y$ and $I_x \not\succ I_y$, then they are *independent*, *i.e.* $I_x \parallel I_y$. By inspecting Figure 3.1 we can observe that instructions I_2 and I_3 are independent, for example. The order in which these instructions are selected for execution is called the *schedule* of $\mathfrak{G}$, *i.e.* $S_\mathfrak{G} = [\ldots, I_x, I_y, \ldots]$. For instructions that are "placed" consecutively in the schedule $S_\mathfrak{G}$, such as I_x and I_y, we define $I_x \triangleright I_y$, if I_x is placed before I_y.

3.2 Local Scheduling Theory

The completion of the schedule $S_\mathfrak{G}$ implies the completion of the execution of all the instructions ($\mathfrak{I}$) when mapped to functional units ($\mathfrak{R}$) of particular types ($\mathfrak{T}$), while respecting the partial order $\prec$. The time taken to complete the execution of all the instructions is called the *completion time* or the *makespan* of execution,

[1]The example in Figure 3.1 shows a DAG with $n = 9$ instructions and their execution times ($\mathfrak{L}$). The set $\prec$ is defined by $\prec = \{ I_1 \prec I_2, I_1 \prec I_3, I_2 \prec I_4, I_2 \prec I_5, I_4 \prec I_7, I_4 \prec I_8, I_5 \prec I_7, I_5 \prec I_8, I_3 \prec I_6, I_6 \prec I_9, I_7 \prec I_9, I_8 \prec I_9 \}$. The latencies ($\mathfrak{L}$) and communication costs ($\mathfrak{E}$) are defined as $\mathfrak{L} = \{ 1, 2, 1, 2, 2, 4, 2, 2, 1 \}$ and $\mathfrak{E} = \{ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0 \}$, respectively, with $e = 12$.

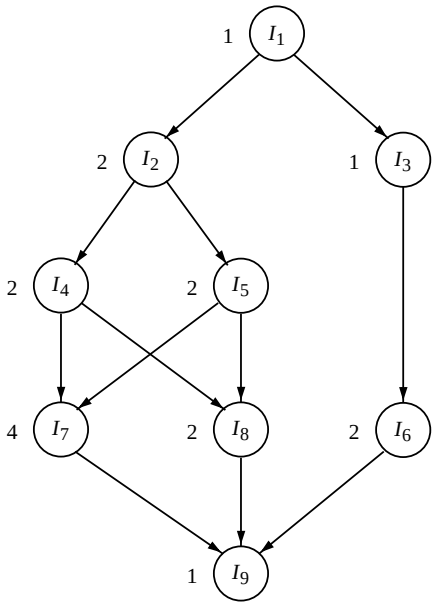


Figure 3.1: An example of a DAG

denoted as w. The term makespan has traditionally been considered as the total time to complete tasks or jobs in job-shop scheduling theory [11][34][130]. The scheduling problem can be treated in different ways depending on the scheduling goals. The most common ones are to minimise the completion time, the number of functional units or the functional unit idle time (or maximise functional unit utilisation) [67]. Other goals include minimising the *mean flow time*[2], which is described in [34].

Throughout this thesis the goal of scheduling is to minimise the completion time w. The absolute minimum completion time is termed as the optimal solution w_o. A schedule may have more than one optimal solution, although the aim is to find at least one.

Scheduling for uniprocessor architectures is usually performed using the non-preemptive and work-greedy approaches. In the non-preemptive approach, once an instruction has started its execution, it cannot be stopped and resumed in another functional unit from the point of suspension. It can be expensive in hardware to stop an instruction and resume its execution from a restartable state. Figure 3.2 shows an example of a non-preemptive and preemptive schedules with

[2]Mean flow time is the average completion time for n instructions. The lower this value is, the less time (on average), resources like memory or cache, will need to keep values in use. Flow time is the sum of the completion times of the individual instructions.

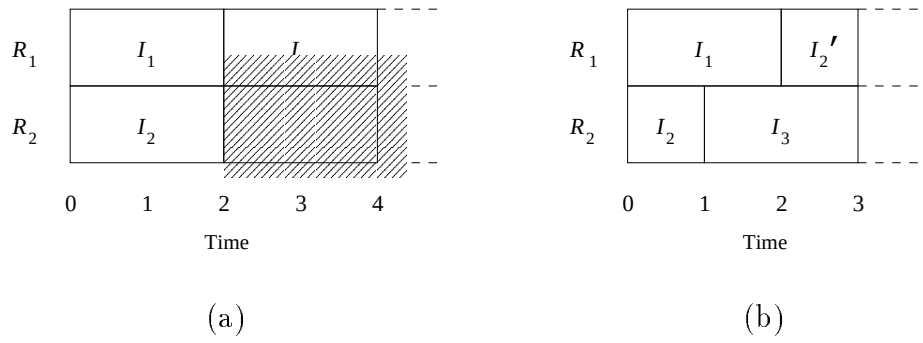


Figure 3.2: (a) Non-preemptive and (b) preemptive schedules.

Gantt charts. Preemptive techniques can produce better schedules than non-preemptive ones, assuming that the instruction-switching overhead is not very costly. The strategy usually adopted in non-preemptive scheduling follows the work-greedy approach. This means that if a functional unit is idle at some point and there is a ready instruction that can be executed by it, then this instruction is assigned to that unit. In practice, the resources are kept busy, but it does not necessarily lead to optimal results. Figure 3.3 depicts scheduling examples for the DAG in Figure 3.1, where assuming a greedy approach could not obtain the optimal schedule, whereas by keeping functional unit R_1 idle for one unit of time, the optimal schedule is achieved. Given that the completion time w represents the total time, *i.e.* the sum of busy and idle times, then maximising the functional unit utilisation should minimise the idle time. Hereafter, we will only discuss non-preemptive and greedy scheduling techniques — the ones mostly commonly used in uniprocessors.

In some cases such as in real-time scheduling, it is necessary to introduce deadlines to instructions, or to a subset of instructions, by when they must meet timing constraints. In a *hard* real-time system [116] all the instructions must meet their deadlines with no exception. In order to do so, it may be necessary to stop one instruction to allow another to resume its execution and meet its deadline, with the use of preemptive algorithms. In *firm* real-time systems, scheduling takes either earliest-deadline-first or smallest-slack-time approaches. In both cases, an accumulative value gets incremented every time an instruction misses its deadline [98].

Another important idea is that of *deterministic scheduling* that are applied to problems that are fully deterministic, *i.e.* all the information governing the

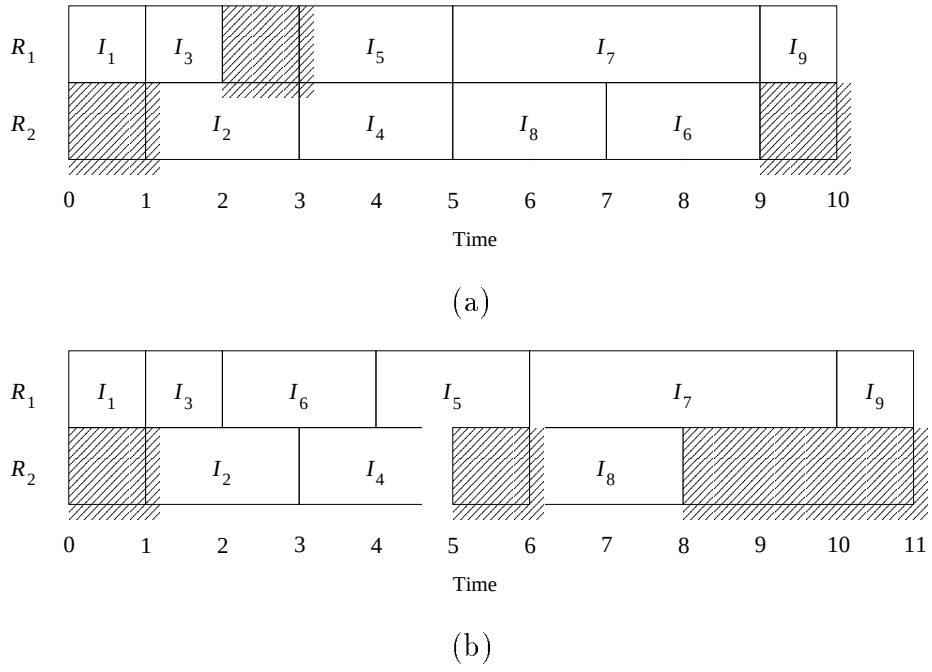


Figure 3.3: (a) Non-greedy and (b) greedy schedules.

scheduling decision is known in advance. That is, the sets $\mathfrak{T}$, $\mathfrak{R}$, $\mathfrak{L}$ and $\mathfrak{E}$ are known, and fixed, before the scheduling process and can be thought of as target-specific input parameters. The sets $\mathfrak{I}$ and $\prec$, on the other hand, can be considered as problem-dependent inputs of the scheduling problem, as shown in Figure 3.4.

In fact, in early scheduling work [25][34][67], *task scheduling* did not contemplate the sets $\mathfrak{T}$ and $\mathfrak{E}$. Most of the problems had a combination of one or more functional units – usually called processors – as part of the set $\mathfrak{R}$, and single or multiple unit duration times for set $\mathfrak{L}$. In majority of the cases, these processors were identical, with no allowance for different types and their execution periods were integer-based only. $\mathfrak{E}$ was regarded as a set of null, weighted edges like the DAG in Figure 3.1. Early research was mainly dedicated to showing that a class of scheduling problems had optimal solutions [34]. Failing that, the approach was to *bound* the scheduling solution (belonging to a class of problems) within a constant margin of the optimal, in order to evaluate the effectiveness of a scheduling algorithm. The constant margin determined how close the proposed solution was to the optimal. The rationale was that greedy techniques cannot be worse than

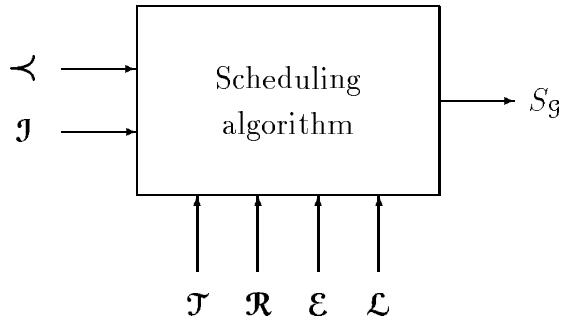


Figure 3.4: Representation of the scheduling problem.

the optimal by a constant factor[3].

3.2.1 Complexity Issues

It has been noted that the complexity of the scheduling-length problem for n instructions with a non-preemptive approach for a general p-functional unit processor[4] is NP-complete [34][170]. By extending this problem by having variable lengths in $\mathfrak{L}$ and the addition of different types $\mathfrak{T}$, the scheduling problem becomes NP-hard [61]. Searching through an exponential space to find an optimal schedule is an expensive task that has surprisingly not always been avoided. Massalin [114] exhaustively executes all possible schedules to get to the optimal; in [10] the compiler generates a small subset of *good* schedules that are filtered using a machine model, and then simulated to discard the sequences with lower performance results. Although the exponential set of schedules is not considered for execution, the compiler must perform an exponential search through the filtered set to gather the best ones. Another example is in real-time scheduling [106], which also considers as an option, exponential-time algorithms such as *exhaustive-enumeration* and *branch-and-bound*. They inspect all legal schedules to calculate their costs[5]. The first schedule with a non-positive cost, *i.e.* every instruction meets its deadline, is returned. The branch-and-bound algorithm is a modifica-

[3]If w' is the makespan for a particular solution and w_o represents the makespan of the optimal solution, $w'/w_o \leqslant c$, with $c \geqslant 1$, expresses the goodness of an algorithm. If $c = 1$ then the solution is as good as the optimal.

[4]At least $p \geqslant 3$. For $p = 2$ the problem complexity is polynomial, if and only if, all values of $\mathfrak{L}$ are single unit and all the functional units $\mathfrak{R}$ are identical ($\mathfrak{T} = 1$ and $\forall E_i \in \mathfrak{E}, E_i = 0$).

[5]The cost represents the total sum of *delayed*-times when an instruction misses its deadline. The delay-time is the amount of time-slots from the instruction's deadline.

tion of exhaustive-enumeration that aims to reduce the running-time. It sets an upper-bound cost, and every time the algorithm finds a schedule with a lower cost, the upper bound is updated. Once again, the algorithm terminates as soon as a non-negative cost is found.

Another scheduling example is found in [32], where a solution tree for deriving optimal schedules is generated by keeping track of all partial schedules. In order to reduce the solution space, *equivalence* and *dominance* relationships between partial schedules are deduced, and nodes from the tree are eliminated as early as possible. Two instructions, I_i and I_j, are *equivalent* when they can be interchanged in the schedule without affecting the length of the makespan, and *dominant* when instruction I_i can always be scheduled no later than I_j. The algorithm starts by creating the root node of the tree. The root node consists of all the instructions with no predecessors at the start of the scheduling process. Then, all the possible combinations that can occur (mapping instructions to types and functional units), are allocated to successors nodes from the root. The equivalence and dominance relations help to reduce the excessive growth of child nodes. The tree is constructed until all the instructions have been scheduled; each of its paths being a valid schedule. The final schedule is generated simply by parsing the solution tree.

However, performing these expensive computations may not only require exponential time to terminate, but may also need exponential resources which may be restricted. Secondly, finding an optimal solution for one architecture family may be sub-optimal for another, even with small changes. Thus, heuristic-based approaches have become a viable option to get an approximation of the optimal solution with a reasonable, polynomial, complexity time and reasonable amount of resources. The use of heuristics helps to capture the little differences in the architectures to get a more general, sub-optimal solution, without the need to recompile a program every time. Some of these heuristics are discussed in greater detail in Section 3.3.1.

3.2.2 Types of Scheduling Algorithms

3.2.2.1 Simulated Annealing

Simulated annealing is a stochastic approach to complex combinatorial optimisation problems based on Metropolis's algorithm. In [28], [95] and [106], this optimisation technique is the core of a scheduler for embedded systems, but can also be found in diverse applications ranging from VLSI block placement and global routing [150], to the airline crew scheduling problem [46], to mention just

a few. The algorithm follows a probabilistic distribution that converges to a local minimum close to the absolute optimal solution, and its behaviour has a pattern which avoids local minima.

The core of the scheduler works as follows: first, an initial schedule is randomly generated and values for initial and final temperature, T_i and T_f are set. The algorithm then sets a reference temperature T to T_i and this is compared to T_f. Then, while T stays beyond the final value of T_f, the core-loop of the scheduler is repeatedly executed. The loop consists of a random perturbation of the actual schedule (the initial schedule at start) called the new schedule, which produces a new temperature, T_{new}. If Δ_T, *i.e.* $T_{new} - T$, is positive, then the schedule is updated and replaced by the actual one; otherwise, it would be updated following a probability distribution that defines the acceptance criteria of solutions ($C = e^{-\Delta_T/T}$). Finally, the temperature T is reduced gradually, by a *cooling factor* (α), resulting in a decreasing exponential distribution.

There are important issues for efficiently using annealing for scheduling purposes. The first one is the random choice of both an appropriate initial schedule and the perturbation function. The initial schedule and the perturbation function must return "reasonably" good schedules. The second, is the choice of a suitable rate to decrease the temperature through the use of α. A slow rate tends to result in optimal solutions, but at the cost of more iterations.

3.2.2.2 Level Scheduling

Level scheduling was originally proposed in [82], as a solution to the problem of assigning products to different operation lines, a variant of the minimum-length-schedule problem. The algorithm allocates the same priority to the products from the same hierarchical level in the DAG, but products from higher levels get higher priorities. The outcome of this prioritisation scheme is a topological sort. The problem has an optimal solution when all the operations take a single unit of time and there are only two *processors* or processing units. In the case of greater than two processors, the solution is bounded within the optimal [67]. For example, if there are three processors, the ratio between the solution and the theoretical optimal solution, w_{LS}/w_o, is 1.5.

3.2.2.3 List Scheduling

List scheduling uses a priority list to order in advance the set of instructions $\mathfrak{I}$, by respecting $\prec$. This list is then scanned sequentially in decreasing order, to assign an instruction to an available functional unit. The difference between algorithms,

and therefore priorities, resides in the classes of heuristics that are considered. Once the instruction is assigned to a functional unit, it is removed from the list and the cycle continues. This behaviour leads to a lack of preemptions in the algorithm, and is therefore a non-preemptive approach[6]. If multiple functional units of the same type are ready, then the priority list gives preference according to the order in which instructions are placed. The algorithm terminates once all the instructions have been assigned to functional units, *i.e.* the priority list is empty.

List scheduling has traditionally been used for scheduling in uniprocessors. In particular, the priority list is filled at the start with instructions with no predecessors, usually called "ready" instructions, from which the heuristic will decide the best candidate to be scheduled next. As soon as an instruction is removed from the *ready* list, all of its successors become available, and they are included in the list for consideration in the following cycle.

3.3 Instruction Scheduling in Synchronous Architectures

3.3.1 List Scheduling for Synchronous Platforms

The list scheduling technique has been used in innumerable synchronous scheduling applications. The following are important work in the area of local scheduling.

Scheduling with no Hardware Support for Interlocks 1 - One of the earliest examples of list scheduling in a synchronous uniprocessor is found in [75]. The MIPS processor [91] does not support interlocks, so the compiler is responsible for characterising the pipeline constraints and re-scheduling the code to avoid them, and in some cases with the help of no-operations (nop) instructions.

An alternative register allocation scheme was also conceived for the lack of hardware support for interlocks. Register allocation for an interlocked-pipelined architecture does not necessarily have to avoid a combination of instructions with a Read-After-Write dependency (in such a case the datapath would introduce an interlock). But for a MIPS processor, in the register allocation phase, the use of registers must be modified in order to avoid such hazards systematically.

[6] *List scheduling* is therefore a subset of preemptive schedules.

Register allocation is an NP-complete problem in its own right [124]. This approach to scheduling has to deal not only with the scheduling problem but with the register allocation one as well. The combination of both problems is shown to be NP-hard, though with the use of no-operations, the problem can be reduced to NP-complete [75].

Due to the nature of the problem, the approach is not meant to be optimal, but to select the shortest legal schedule from a subset which is generated. The algorithm finds the shortest legal schedule in polynomial time, *i.e.* in $\theta(n^4)$ time [74].

Scheduling with no Hardware Support for Interlocks 2 - Results in [132], on the other hand, show that with some restricted set of pipeline constraints in *Delayed-load* architectures such as the MIPS, optimal scheduling and register allocation can be achieved in linear time. The necessary conditions though, include that all instructions are executed in one clock cycle and that all destination registers in load instructions cannot be accessed until one cycle has elapsed ($delay = 1$). If they are accessed before one cycle, *i.e.* just after the load, a pipeline interlock occurs.

Another simplification is the use of expression-trees rather than DAGs. An expression-tree differs from a DAG in that each node must either be a symbol or a parameter (an address or a number), whereas in a DAG, each node represents an instruction with its own opcode and operands. The complexity of dependencies in a DAG ($\prec$) makes register allocation particularly difficult. With the use of binary expression-trees, the task is much simpler. Expression-trees help implement register allocation and instruction scheduling in polynomial time; the complexity is proportional to the size of the tree when the value of *delay* is one. Figures 3.5 (a) and 3.5 (b) show examples of an expression-tree and a DAG for the assignment `m = A[1];`.

For cases where the *delay* is greater than one, optimal scheduling results are maintained such that the shortest schedule is obtained, but the register allocation results are no longer optimal. Such a schedule does not guarantee the use of minimum number of registers. In [100], the binary tree algorithm is used as a heuristic to schedule instructions with arbitrary delay slots when a DAG is used. The run-time complexity of the scheduler is $\theta(n)$.

Scheduling with Hardware Support for Interlocks - The work by Gibbons and Muchnick [64] differs from the previous two cases in that it is targeted

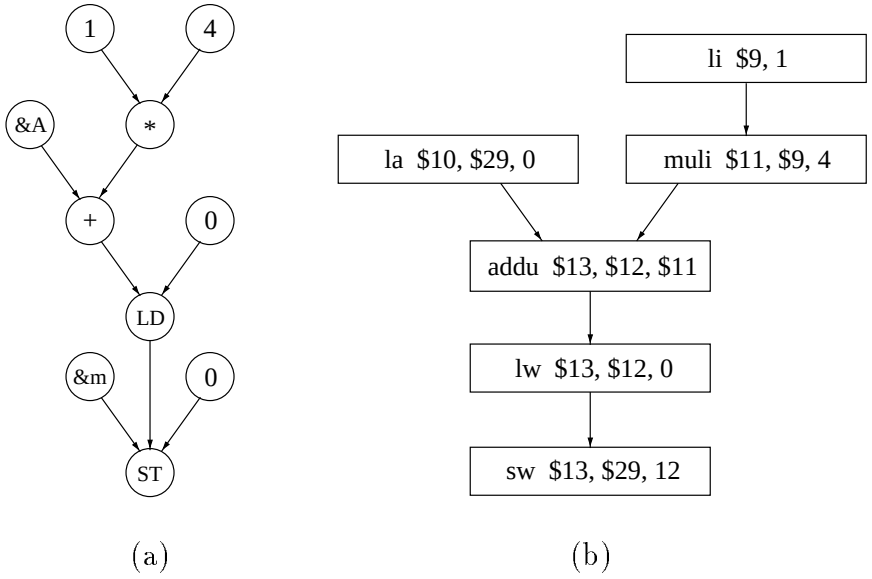


Figure 3.5: (a) An expression-tree and (b) an equivalent DAG representation.

at architectures with hardware support for interlocks. In deeply pipelined architectures, an optimal schedule is one that causes the minimum number of stalls in the pipeline. In particular, a scheduler has to be able to perform well under various implementations with different sets of interlocks. The resulting schedule might not be optimal for a particular set, but will still perform well overall. The algorithm uses two main criteria for selecting a candidate instruction from the ready list: the candidate must not cause an interlock with a previous scheduled instruction, and it has to be the most likely instruction to interlock the instruction after it.

The algorithm was implemented without lookahead in order to maintain a low run-time complexity. Lookahead is a property of an algorithm in that it looks for "near future" features, *i.e.* choosing ready instructions which will trigger ready candidates in the future. The scheduler considers some heuristics to mimic lookahead properties at the moment of choosing an instruction from the ready list. These include the number of immediate successors, the longest path from the candidate to the leaves of the DAG, and whether the candidate would cause an interlock with its immediate successors. The scheduler has a run-time complexity of $\theta(n^2)$.

Rank Algorithm - Another well known work in scheduling RISC architectures is presented in [124]. Palem and Simons use the latency information from

every instruction I_i $(1 \leqslant i \leqslant n)$, together with the *deadline*, to define the rank of I_i, called $rank\,(I_i)$. The scheduling algorithm uses this rank to construct the list and then schedules it in a greedy fashion. The deadline is a sufficiently large figure by when the instructions are guaranteed to have been executed. An example of a deadline time is $n\,(k+1)$, where k is the maximum possible latency. The time complexity of the scheduler based on the rank algorithm is $\theta(en + e\ log\ n)$, where e is the number of edges in the DAG.

This work was extended in [101] to include the presence of deadlines and release-times for embedded applications and real-time systems. Due to the nature of the overhead involved in computing the deadlines, the run-time complexity becomes $\theta\big(n^3\,\alpha(n)\big)$. $\alpha(n)$ is the inverse of the Ackermann function, and should be considered a small constant as n increases [69].

Balanced Scheduler - The balanced scheduler [92] is another example of list scheduling that introduces the concept of measuring *load-level parallelism* to the algorithm. The reason for this is that the latency of a load is not always constant due to a possible cache miss in the memory hierarchy; therefore, waiting for a value from memory can lead to undesirable stalls in the pipeline. Architectures with non-blocking loads allow other instructions to be executed concurrently with a load instruction; hence, it is important to pad an appropriate number of non-load instructions for every load one.

One of the differences with the other list schedulers is that the balanced scheduler does not allow a free instruction to be inserted in the ready list until its predecessors have exhausted their expected latencies. The heuristics used by the balanced scheduler to select a candidate from the ready list are the priority, *i.e.* the weight based on the load-level parallelism, the maximum priority of its successors, the largest difference between consumed and defined registers (to monitor register pressure), and the number of successors of the candidate if it were to be scheduled.

The balanced scheduler is the first example in the literature that considers latencies that vary at run-time. The results show, however, that if the latency gap between the cache hit and cache miss grows, it becomes harder to compensate the effect of a miss by inserting independent instructions, and performance degrades considerably. The run-time complexity of the balanced scheduler is $\theta\big(n^2\,\alpha(n)\big)$.

3.3.2 Synchronous Model for the Compiler

The RISC experience has shown that a simple instruction set offers several advantages for the compiler. Firstly, simple instructions run faster by using pipelining, and secondly, the majority of the instructions have the same execution time, so the compiler does not have to implement a complicated algorithm in order to distribute long latency instructions amongst those with short latencies.

In early synchronous RISC architectures, instruction sets included simple instructions that normally execute at the rate of one instruction per cycle through the use of pipelined datapaths [125]. Examples of these include the MIPS processor from Stanford University, and the RISC I and RISC II processors from University of California at Berkeley. In the early schedulers all instructions are considered to take the same amount of time to execute [75]; in [100] and [132], all the pipeline stages of the delayed-load architecture take only one clock cycle to complete. Furthermore, the rank algorithm in [124], and its extension in [101], consider all latencies to be either zero or one unit to achieve optimal results.

The "uniform" latency assumption simplifies the compiler model described in Section 3.1. For the set $\mathfrak{L}$, we now have : $\forall L_i \in \mathfrak{L},\ L_i = 1$. This means that the scheduler can assume that after choosing a ready instruction and removing it from the ready list, it can be considered as executed. The next step of the algorithm is to update all its immediate successors and insert them in the ready list. These iterations of the algorithm have an implicit, discrete timing. If the latencies are integer values, then the scheduler must wait for a number of "cycles" until the completion of the instruction before activating its successors [174]. An important point to note is that, in some approaches, the values from the latencies are used as part of the set $\mathfrak{E}$, instead of $\mathfrak{L}$ [10][19]. This means that the set $\mathfrak{L}$ becomes empty ($\mathfrak{L} = \emptyset$), with the assumption that the communication costs are neglected. The DAG in Figure 3.1 would become a DAG with delay values assigned to the edges as depicted in Figure 3.6. If a node has multiple immediate successors, then the value is broadcast to all its edges. The ability to integrate the hardware latencies directly into the model has been an important factor in the success of these schedulers.

However, more recent RISC architectures with organisations containing a number of functional units in their datapaths, a higher degree of pipelining, and different levels in the memory hierarchy are more difficult to model [10]. It is recognised that latency variations of memory accesses lead to a degradation in the quality of scheduling. The balanced scheduler [92] is the first example of a scheduler in which the latencies are not considered fixed, because of variations in

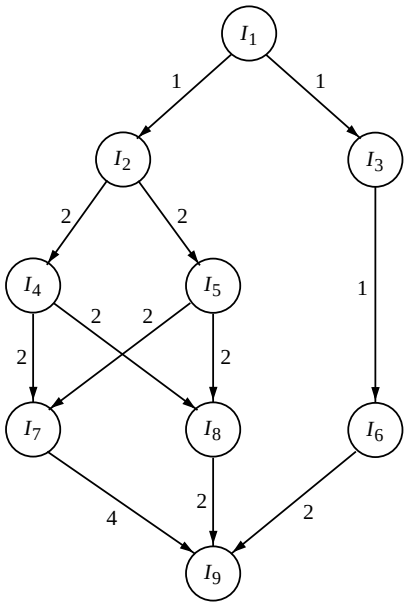


Figure 3.6: An alternative interpretation for the DAG from Figure 3.1.

the memory hierarchy. Two machine cases are presented: one RISC-type machine with a cache ratio bounded between two and five clock cycles, for a hit and a miss respectively, and an interconnect-based network whose memory latency is defined by a normal distribution with both its mean and standard deviation ranging from two to five clock cycles. It is worth mentioning that, for the sake of evaluating the scheduler, an unbalanced configuration[7], where not enough load-level parallelism can be found, was selected. In some benchmarks, the degradation was so significant that the scheduler performed worse than a list scheduler implementation without the load-level information.

Even with these run-time variations in latencies, computational models for synchronous architectures can still capture the principal features effectively with the use of heuristics. The next section presents some of the most common heuristics used in list schedulers.

3.3.3 Common Heuristics for List Schedulers

The effectiveness of list schedulers depends on a crucial decision that is taken during the execution of the algorithm: instructions from the ready list must be chosen according to heuristics such that the most "important" instructions are

[7]A mean value of 30 clock cycles and a standard deviation of 5 clock cycles.

selected first. These heuristics are usually combined to a weighted sum that can be tuned to improve the scheduler. There are a number of possible factors that can be used for a heuristic. The key is to find which factors affect the performance for a particular architecture. This section describes some of the commonly-used criteria:

Critical path. The critical path is one of the most commonly used heuristics. It describes the longest path from the entry point to the exit point of a basic block. The critical path is an upper bound for any scheduler. If the edges of the DAG are not zero, then the critical path is the one with the greatest collective weight from the entry node to the exit node. For example, the critical paths in Figure 3.6 are I_1, I_2, I_4, I_7, I_9 and I_1, I_2, I_5, I_7, I_9.

Number of successors. The number of successors describes how much the result of a node is needed by its successors, *i.e.* the importance of an instruction's result. The more successors an instruction has, the earlier this result must be resolved in order that its successors become ready as soon as possible.

This heuristic can be interpreted on the immediate number of successors or the total number of successors. The immediate number of successors can be used to eliminate ties in the main heuristic.

Number of predecessors. The number of predecessors describes the number of parents of a node. An instruction with many predecessors represents a synchronisation point, thus reducing concurrency and must be scheduled as late as possible, so that all its predecessors produce their data as early as possible. The immediate number of predecessors can be applied in the same way as the immediate number of successors.

Distance to the leaves of the DAG. This is the distance in terms of the number of edges, from an instruction to any sink in the DAG (all the sink nodes are joined to the exit node as mentioned in Section 3.1). This heuristic is used in [64].

Number of operands. The purpose of this metric is to identify register pressure. More operands in an instruction implies more releases of registers as soon as these operands are read. This is relevant as releasing registers means other temporary values can be assigned to them and spilled code can be avoided. This heuristic is subject to the number of registers in the architecture in question.

Resource usage. This factor helps to track the instructions when there are limited number and types of functional units. When an instruction is selected from the ready list, its functional unit is recorded, so that the scheduler recalls which types have been recently used. Depending on past selections, the scheduler uses this information to choose instructions so that run-time contentions for resources are reduced. This heuristic is subject to the number and types of functional units.

The work in [13] uses genetic algorithms to tune a large set of heuristics for an instruction scheduler targeted at three different synchronous machines. The statistical data show that the critical path is indeed the most beneficial heuristic, and that the number of successors and predecessors heuristics are not as effective. The results also show that the distance to the leaves of the DAG and the number of operands heuristics are not as useful, at least for the machines which were tested.

3.4 Asynchronous Circuits

3.4.1 Introduction

In recent years, there has been a revival of interest in asynchronous circuits. This is in part due to serious problems that are beginning to affect the design and implementation of high performance synchronous systems, which will be aggravated in the future, as the clock period shrinks. Among these, the clock skew and power consumption remain crucial issues in the design of future synchronous architectures [109].

Clock skew represents the small differences in the arrival times of the clock at different parts of the integrated circuit. The problem with clock skew is that it has an effect on a small, and crucial, window of time where the clock edge must take place. This period of time is comprised of the *setup time* (the maximum time that the output values of the combinational logic from the previous state have to be stable), and the *hold time* (which is the minimum time that it takes for the input nodes to be charged at the present state). If the clock edge does not take place within this window, either the values from the previous state will not have time to settle and cause a setup violation, or there will not be enough time to charge the new entries from the current state, and thereby incur a hold violation.

There are several physical reasons which cause these variations in the arrival times such as temperature, technology process, threshold voltage, and signal

propagation delays in conjunction with the routing and the topology of the clock signal. The importance of the clock distribution in high-speed and high-density implementations is that the clock load must be equally balanced all over the chip. The implications of the clock routing topology are decisive in the trade-off between several metrics such as clock skew, clock load, the maximum clock frequency, and the power consumption of the clock buffers. But other issues will contribute to the pressure on the clock skew with the projected scaling trends [146]: firstly, it is known that clock frequencies allow around twenty *fanout-of-four*[8] (FO4) inverter delays per clock cycle, but this figure will be reduced to around five at the feature size of $0.05\,\mu m$ technology and clocking at speeds of 10 GHz, according to the SIA projections [1][81]. The clocking overhead will use a significant part of the cycle time making conventional flip-flop schemes more difficult to design [52]. Secondly, the ever increasing operating frequencies will tighten the timing restrictions, *i.e.* the window within which the clock transition must take place will narrow. And thirdly, the continuous increase in the die size will naturally lengthen the clock wires, and thereby producing longer delays. Signal delays are governed by electromagnetic wave propagation and are directly proportional to the wire length[9], so this tendency of longer paths for the clock will also have major effects on the clock skew.

The second main cause of concern in synchronous systems is their power consumption. The power consumption at the device level for CMOS logic is proportional to the operation frequency (f), the total output capacitance (C_T), the supply voltage (V_{DD}), and to the *short-circuit current* (I_{sc}) and the *leakage current* (I_{leak}) [175], as defined by Equation 3.1.

$$P = fC_T{V_{DD}}^2 + (V_{DD} - 2V_{th})^3 I_{sc} + V_{DD} I_{leak} \tag{3.1}$$

Among these three terms, the one that dominates is the first term, called the *switching current*. In recent years, the supply voltage (V_{DD}) has consistently been reduced, but it is reaching a limit as V_{DD} closes the gap with the threshold voltage V_{th} in deep sub-micron technologies. There are clear indications that lowering the supply voltage requires lowering the threshold voltage, and low threshold voltages lead to significantly large subthreshold leakage currents [88]. It also has been shown in [43], that at a feature size of $0.18\,\mu m$ there are difficulties

[8]FO4 is a delay metric to estimate circuit speeds independent of the process technology. The FO4 delay is the time for an inverter to drive four copies of itself.

[9]The delay of a wire is quadratically proportional to its length and independent of its width. Widening the wire will reduce its resistivity but will proportionally increase its capacitance.

to lowering the supply voltage and the threshold voltage below $1.0\,V$ and $0.3\,V$ respectively, without loss in speed. Scaling analysis shows that V_{th} seems to be limited at $0.3\,V$ for room temperature of CMOS circuits [38]. This suggests that there is a potential lower bound for the supply voltage, while clock frequencies will maintain their relentless increase. The actual trend in microprocessors (for the last 15 years) is that clock frequency increases 30% per year [52]. The power consumption therefore might become strictly related to the switching frequency: a prediction that by the year 2006 the devices will operate at frequencies around 4 GHz, and consuming more than 170 Watts [22][146].

Even though the clock can be gated in synchronous designs [134], it is not always straight-forward to find a suitable condition to shut down the clock and it is likely that some parts of the clock tree will still be switching, therefore consuming power [177]. Moreover, it has been shown that in high-performance processors, the clock circuitry, *i.e.* generation, drivers, distribution tree and loading, represents up to 40% of the power consumption in high-performance processors [168], and between 15% and 45% in more generic synchronous designs [128]. Asynchronous designs on the other hand, will only consume power when being active. There have been several examples where the asynchronous design is often larger than its synchronous counterpart, but with the advantage of having considerable power savings [16][93][141], and even some of them having no impact in terms of area [140][165].

There are other reasons that lead us to believe that future VLSI circuits will find it difficult to continue with the trends described above. All these reasons have motivated research into asynchronous circuits and systems, so underlying their differences is the first step in understanding them.

In synchronous design there are two main assumptions: all signals are binary and the time is discrete [26][72]. The former has permitted Boolean algebra not only to express in mathematical terms combinational circuits, but to help methodologically their realisation and optimisation; the latter means that hazards and feedback can be ignored to some extent.

In the asynchronous domain, the only assumption that is held is that all signals are binary; time is no longer considered discrete. This difference is the basis for several positive features not found in clocked circuits. This section looks at some of the benefits and drawbacks that result from this property.

All these implications are relevant when designing a scheduler with an asynchronous architecture in mind. Later, we will describe the principal features that such a scheduler should embody, and how best to model the behaviour of the asynchronous target.

3.4.2 Advantages

The main advantages of asynchronous circuits are summarised as follows:

Low power. As asynchronous circuits do not use a clock for synchronisation, their components only consume power during useful operations or transactions. During the rest of the time, they remain in a quiescent state, when only leakage current is consumed[10]. It is believed, however, that more signal transactions take place during activity, but they only occur in areas involved in the computation.

Synchronous circuits however, consume power even when they are not performing any useful operation. An alternative solution is to gate the clock (which effectively turns the clock off) in areas that are not used frequently, but this has been pointed out to be the source of other problems during synthesis and verification, because modified clocks generally generate glitches. The other concern when gating the clock is due to current variation. The switching variation from different blocks toggling on and off strains the power delivery mechanism [168].

Furthermore, the patterns seen in synchronous circuits over recent years when technology scales down (frequency doubling, supply voltages scaling down 30%, capacitance growing from 30% to 35% and die size growing around 25%) show that the main limitation for performance and integration in future technologies will be the power dissipation and power delivery [22][168]. Even reducing the power supply does not help enough to reduce the power consumption of today's processors.

So far, several examples of asynchronous implementations found in the literature have presented low power consumption characteristics [113][126][167]. Some of them have shown power savings with respect to comparable synchronous implementations.

Average case instead of worse case. One of the main advantages of asynchrony is that components do not need to wait after they complete a transaction; they can proceed immediately to the next operation if requested. The speed therefore will depend on the *average* speeds of all the entities. In order to increase the overall speed, one should analyse their "standard deviation" over time: since there is a collection of different speeds, one should

[10]For CMOS circuits, the leakage current can be neglected when compared to the current consumed in active mode [17].

look at the frequency of operation of the slower components, and depending on their number of occurrences, those with a higher figure, should be improved. Doing this will actually increase the average speed of the circuit, *i.e.* reducing its standard deviation.

In the synchronous approach the clock speed is determined by the speed of the slowest component, *i.e.* the *worst-case*. To increase the speed of such a system, all the slower components must be able to operate faster altogether, resulting in a new clock frequency which is determined by the improved speed of the slowest element(s).

No clock skew. The lack of a clock in asynchronous systems means that there is no clock skew. When the clock signal is propagated, the differences in the arrival times impinge upon the behaviour of the circuit; if the clock frequency is incremented, there is a higher probability for variations in the clock skew, so special attention is paid to the clock routing and buffering. This is a major concern in synchronous designs nowadays, due to the increasing clock operating frequencies. The removal of the clock skew problem helps asynchronous circuit design to relax the *global* timing demands.

Automatic adaptation to physical properties. In synchronous design, the physical environment of the circuit such as temperature, power supply and fabrication specifications, needs to be taken into account so that the circuit should work under the worst possible operating conditions. It must operate within a safety margin in order to guarantee its functionality in case of variations, so typical values cannot be used. The term, worst-case, applies in the same way as before.

On the other hand, asynchronous circuits are more tolerant to physical variations. Since there are no critical timing requirements to match a specific clock speed, circuits may have different delays corresponding to a particular variation and will run as fast as their operation conditions will allow. Their functionality, and more importantly, their correctness, will be maintained in any case.

Low noise and low emission. The downside effects produced by clocking at very fast speeds can be often found in power lines where noise is induced. This is relevant if the circuit includes analog or RF circuits, since the noise caused by the clock could interfere in their operation. The high-frequency harmonics induced could be confused for a proper signal. An example of

these negative effects are found in analog-to-digital converters, where fluctuations in the power supply caused by noise can lead to wrong voltage reference levels, and therefore, to wrong conversions. In such cases, these implementations require expensive circuitry for filtering the noise generated.

On the other hand, the levels of noise in an asynchronous circuit are better tolerated [17]. In [59] for example, an asynchronous version of the 80C51 micro-controller is presented in which the level of noise is substantially reduced compared to its synchronous counterpart. In [126], a low-noise, low-power, self-timed DSP is described. It is characterised by substantial reductions of noise and electromagnetic interference (EMI) emissions.

Locality. Asynchrony supports a modular approach to system design. The absence of a clock isolates the different components that communicate locally, from the rest of the system. These could be replaced, expanded or removed without having a side-effect on the rest of the system. Furthermore, components can be fully designed and optimised independently. Again, in a synchronous platform where balancing the clock load, optimising and routing the clock are important design issues, any of the previous actions would require global modifications to the circuit, *i.e.* re-computing the clock speed and revising both the clock load and the clock routing scheme. The whole optimisation process has to be performed in a global manner.

Locality also helps to expose fine-grain concurrency. Components in a datapath only communicate with neighbouring components in order to perform a computation. Other components, independent of this computation, can operate freely without the need for synchronisation. This model exposes a finer degree of concurrency [5][6][7]. This will be discussed in more detail in Section 4.3.1.

Globally asynchronous - locally synchronous. An area where asynchrony might have an immediate impact on synchronous systems will be to replace the global clock with an asynchronous protocol, to benefit the clock skew and the overall power consumption. The principle is to have a collection of synchronous components that communicate with each other asynchronously — the so-called Globally Asynchronous Locally Synchronous (GALS) systems [30]. A design methodology for GALS systems is presented in [73]. The methodology aims to find an optimum balance between partitioning a chip in large synchronous blocks with low asynchronous communication overheads. Results show that up to 70% of power consumption can be

reduced and the asynchronous communication circuitry can be neglected, when an ASIC is partitioned into 52 blocks, covering a die size of $163\,mm^2$ with over 3 million gates.

3.4.3 Disadvantages

Despite the aforementioned positive attributes of asynchronous circuits, there are some well-known disadvantages that serve as motivation for researchers, but more importantly, explain why asynchrony has so far eluded wider acceptance.

Design Difficulty. At first sight asynchronous circuits are "difficult to design". Designers have used the clock to enforce the sequentiality of events. A sufficiently large clock period was adequate to group combinational and latching sections to store the outputs and hold the present state. This scheme is simple enough and has had great success.

Now that asynchrony is again an active topic of research, the freedom of not having global timing restrictions has exposed the challenges in designing circuits without a clock. It has enabled numerous and different styles to describe asynchronous circuits [26], — from *speed-independent* to *delay-insensitive* circuits, which all have to deal with the sequentiality of signals and avoid hazards and races (non-determinism) to ensure the correct behaviour. The designer needs to be able to specify the behaviour of the circuit and the values of signals at every moment of time — a difficult process.

Completion detection. One of the drawbacks of asynchronous circuits, not found in their synchronous counterparts, is the generation of completion signals. Since there is no timing reference, it is not obvious how the completion signal of an operation can be generated. This depends on the circuit itself, but it has to be ensured that it is not generated neither too early, nor too late. An early completion signal would cause the following stage to read the wrong data, and a delayed completion signal would degrade the access time.

Tools. The design of synchronous systems has developed a large body of mature tools, which helps in design, simulation, synthesis, routing and verification. These tools have allowed us to cope with the large scale integration of systems with increasingly fast growth in size and number of gates, and shorter spans of design cycles due to time-to-market demands.

Just recently, there has been a surge in the availability of research tools and methodologies for asynchronous design, varying from high-level description languages to hardware verification and synthesis techniques [135]. Although their increasing use can be found in numerous examples [16][63], they are still immature when compared to the infrastructure available for synchronous design. Although these tools produce working designs, the performance results achieved in terms of speed are inadequate, as reported in [159]. Directions for future research in CAD tools can be found in [158].

Testing. In synchronous designs, testing has been developed around the clock: testing wrappers are built-in around components, and when a testing mode is set, testing vectors are applied to the circuit. In the testing mode, one can proceed with the flow of operation and stop it at any point by disabling the clock signals. The state of the system at that point in time can be checked. *Online* testing is a technique that consists of a circuit checker that analyses illegal states representing illegal outputs. The testing overhead is considerable but is compensated by the effectiveness of this technique.

On the other hand, testing an equivalent asynchronous circuit is a difficult problem [15]. The main reasons that make this task particularly difficult stem from the fact that, first, there is a larger presence of state-holding elements at the point where the test generation seems practically impossible; secondly, the inability to "freeze" the state of the circuit and "single step" from it; thirdly, the larger overhead of logic to be paid around the asynchronous circuitry, and lastly, due to the difficulties in detecting hazards and races [83], that have to be avoided in the first place.

These difficulties are challenging, but there are examples where testing can be implemented under control with some degree of success. *Self-checking* circuits offer an interesting property only seen in self-timed circuits. During the handshake of an asynchronous transaction a request from the sender initiates the process and the receiver must eventually acknowledge back the sender, when ready to receive. If the process of this handshake is never completed (an indefinite wait), it is likely due to a fault in the circuit. Self-checking circuits halt when faulty, a characteristic that makes them known as fully testable circuits [83][129].

Test generation is another complex task in asynchronous design, not only because during testing hazards could be introduced (a problem not found when testing synchronous circuits), but because testing for a particular fault

may require a series of input patterns that need to be fed up to the known state where the input vector will then exercise that fault.

Testing still represents a challenging area in the synchronous domain and will continue to be so with the emergence of the recent system-on-chip (SOC) methodologies [178]. This means that the challenge will be even greater for testing entire asynchronous systems.

Performance measurement. Measuring the speed performance of an asynchronous circuit is not a trivial task compared to its synchronous counterpart. With clocked circuits this task involves measuring the length of the critical path and then counting the number of clock cycles. For an asynchronous circuit, the time it will take to complete a task will depend on its delays (related to hardware issues) and on the input data (related to software issues). This means that the performance measures are prone to be variable. (The reasons of this are explained in more detail in the next section). The performance metric used must be based on the average measure.

An alternative method for measuring the performance of asynchronous microprocessors is the MIPS/Watt metric. This metric measures the performance by showing how fast a processor runs, at the expense of its power consumption. To counter the criticisms that asynchronous microprocessors are not as fast as their synchronous counterparts, this metric shows that performance can be related to power consumption, and that a design could be thermally more efficient for its performance. In [113], an asynchronous implementation of a MIPS R3000 microprocessor is checked at different supply voltages to compare its performance per wattage. It is shown that one can trade a higher speed at a higher voltage, and thus a higher power consumption, for a lower supply voltage with lower power dissipation, but with lower performance overall.

3.4.4 A Compiler Model for Asynchronous Architectures

As discussed in Section 3.3.2, compilers, and importantly, schedulers for synchronous targets have benefited from the regular streams in the instruction execution. The method for forcing all events to occur with the clock, and eventually the execution of the instructions, eases the scheduling model described in Section 3.1. The behaviour of the architecture is captured in $\mathcal{G}$, giving the algorithm full control for accurate optimisations.

However, the assumptions made in the synchronous model do not hold for an asynchronous target, simply because events do not have precise timings, *i.e.* they do not have a pre-defined time to resolve. Values from set $\mathfrak{L}$ are variable and are certainly not integers. They are governed by a number of different factors: (1) At the physical level, the time it takes for an asynchronous datapath to complete a task depends upon its design, its physical layout and its operating conditions.

(2) The dynamic behaviour when executing a sequence of instructions ($\mathfrak{I}$) is based on the number and type of resources; sets $\mathfrak{R}$ and $\mathfrak{T}$, respectively. If, for example, instructions from this sequence belong to the same type and there is only one functional unit available of that type, then structural hazards will occur and introduce additional delays which affect the execution times. The dynamic influence of these delays comes from a tight relationship between both the sequence of instructions and the architecture configuration (number and types of functional units and the connectivity between them).

(3) Another important dynamic effect in asynchronous designs is that the time it takes to process an action depends on its input data as well. A typical example is found in asynchronous adders where the time to complete an addition is proportional to the number of one-plus-one's, say m (number of ones from one operand added to the ones from the other), because carry signals must propagate m stages to allow the addition to complete [94][151].

And (4), statically, the latencies are dependent upon the instructions themselves and their ordering. The opcode of the instruction defines which type of functional unit will execute the instruction; the operands determine from which registers the data will be read or written accordingly.

Finally, the ordering in which they are scheduled is decisive in the way events will take place in order to avoid stalls due to data dependencies during execution.

This series of interdependencies can be illustrated in Figure 3.7. The figure depicts a different representation of the scheduling problem from the scheme shown in Figure 3.4.

The dashed lines in the figure represent the latency information that is only available at run-time and not statically. The solid lines represent set of information that are fully available before scheduling takes place. The feedback from $S_{\mathfrak{g}}$ does not mean that the output schedule is being fed back, but to illustrate that the ordering in the schedule needs to be processed in combination with all aspects explained above, and that will determine the final values of the latencies. The dashed line leaving the block where $\mathfrak{L}$ is determined implies only that these values are not available at compile-time.

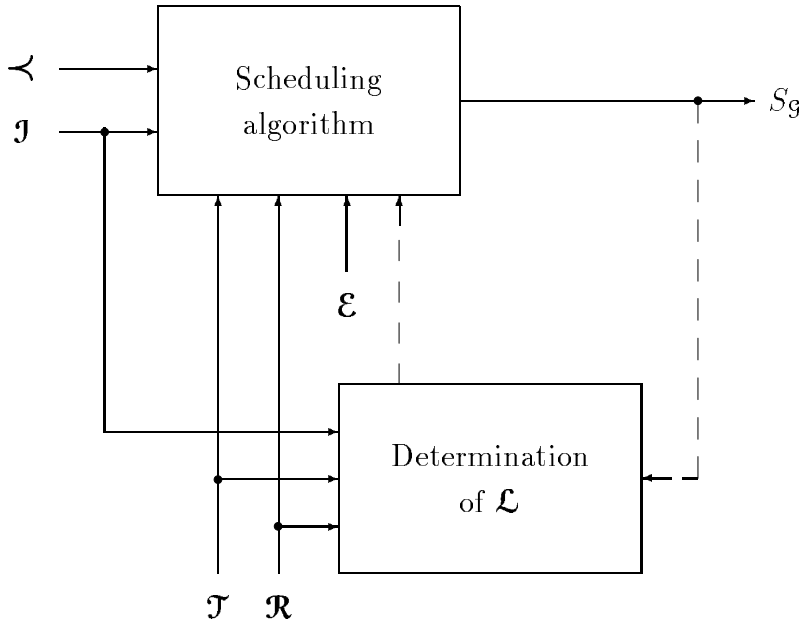


Figure 3.7: New representation of the scheduling problem.

Given all the factors and combinations involved in defining the latencies values, they are found to be bounded within a best and a worst case. The best case is when there are optimal operating conditions, minimum delays coming from the inputs, minimum dynamic delays and minimum processing time of data. In the adder example, the best case is when adding zero to zero. Conversely, the upper bound, *i.e.* the worst case, is when the operation is performed under the worst operating conditions, with maximum delays from the inputs and maximum processing of data. In the case of the adder (the case of adding maximum numbers together), *i.e.* numbers with maximum number of ones, the carry has to propagate to its maximum. The data is mostly responsible for the operating range of the adder. Its delay function is therefore, an ascendent curve directly proportional to the amount of one-plus-ones m, starting from the best case and bound at the worst case [62].

Finally, asynchronous circuits require an additional time to become ready to operate when they complete an operation. During this period of time an asynchronous component settles and goes to an initial state whereby it can start a new operation. In synchronous circuits this period of time is included in the cycle time, which is calculated from the slowest component. However, the *cycle* time in asynchronous circuits is particular to each component. Its duration is mainly

due to the implementation of the circuit, but is also affected by the physical conditions explained earlier. The cycle time of a component can be considered to be a fraction of its *latency* time, but cannot necessarily be ignored.

3.4.5 Considerations for the Compiler

As has been mentioned before, the scheduling problem becomes NP-hard when pipeline stages are no longer equal, even if the basic block consists of independent instructions [61][124]. Having different types of units and variable latencies poses a challenging optimisation problem for the compiler. The number of variables involved in the determination of the latency values is large and diverse enough to integrate into a single computational model. The complexity of such a model combined with the complexity of the scheduling task seems to have a intractable solution. In order to simplify the problem, the compiler has to make some assumptions without removing the principal features of asynchrony: the operating conditions can be assumed to be fixed to some extent, and the delays between neighbouring components can be neglected since they should be placed together in the physical layout.

The dynamic variations due to data dependencies, data processing and resource contention cannot be simplified because they are only resolved at run-time, which makes it difficult to parametrise them. However, they can be decomposed in terms of costs. Some operations or events may be more costly in terms of delay than others. In this way, priorities can be incorporated into the scheduler. It has been shown that in delayed-load architectures without hardware support for interlocks, the scheduler must ensure that there cannot be two data dependent instructions scheduled consecutively. A dedicated heuristic can be implemented in such a way that this case should never happen.

In [7], data dependencies (RAW, WAR and WAW), are categorised by the amount of stall they induce in an asynchronous architecture and prove to be a good mechanism for consideration as a heuristic. Capturing the impact of different costs instead of the actual latency values seems an attractive approach to the problem. Detailed explanations will be presented in Chapter 4.

However, one could not presume to find optimal solutions for asynchronous architectures. As mentioned before, the scheduling problem expects to obtain near-optimal solutions given its complexity. The assumptions made for asynchronous circuits may lead, to some degree, to relatively *good* solutions, but the non-deterministic nature of the problem makes it impossible to always achieve optimal results. Furthermore, a schedule may have different optimal values over

different runs due to the dynamic characteristics explained earlier. It is expected that several runs of the same scheduled code will have different makespans. Hardware considerations in [7] include different costs for different functional units based upon SPICE-level simulations, but latencies are fixed figures during the simulation process. Although the ratios between costs and latencies are representative of the architecture, they do not reflect the variability as a result of input data and resource contentions at run-time.

From the scheme shown in Figure 3.7, one can parameterise the number and types of functional units ($\mathfrak{T}, \mathfrak{R}$). The cost involved in the communication of results ($\mathfrak{E}$) can be associated to the delay function of the register file after SPICE simulations. As for $\mathfrak{L}$, the scheduler may use a *range* of values covering a maximum and minimum latencies in the parametrised model, in order to apply different costs to combination of instructions at compile-time. All this information will serve to capture the computation model for the scheduler.

3.5 Summary

Scheduling techniques for uniprocessors have matured enormously since the problem was first approached, and has been helped by previous scheduling research in other domains such as management science and operations research. These studies have focused on the optimisation concerning people, equipment and raw materials. These problems have dealt basically with integer numbers, and it is this reason that enabled the subtle transition to scheduling code in instruction set architectures.

List scheduling has become a *de facto* solution for the "p-functional unit processor, n-instructions" problem. The sparse diversity of heuristics offers an appropriate method to overcome the constraint's complexity and is able to tune scheduling optimisations for different target specifications. However, these specifications rely upon regular and synchronous behaviours. The scheduling examples described in this chapter show the tight relationship between the back-end of the compiler and the architecture. This relationship is greatly responsible for the success of scheduling in synchronous architectures in achieving good performances.

The challenge for scheduling asynchronous architectures though is apparent. Despite all the advantages of asynchronous circuits described in this chapter, the property of exploiting average case delays is in fact the primary reason for the difficulties in scheduling. The result of having ranges in the latencies for the set $\mathfrak{L}$,

instead of single point values, raises the question whether traditional scheduling techniques would be as effective for asynchronous architectures, and at first sight reduces expectations for achieving optimal solutions.

The next chapter will present an overview of some well-known asynchronous architectures, including the Micronet-based asynchronous architecture. Descriptions and characteristics of the Micronet architectural approach will be presented along with the behaviour of its model. The scheduling schemes proposed later in this thesis will be targeted towards such an architecture, and will be evaluated on a simulator using a detailed model of the micronet architecture.

Chapter 4

Asynchronous Architectures

4.1 Introduction

The benefits of asynchronous circuits, as explained in Chapter 3, have triggered a revival of interest in asynchronous architectures and their design. Early research investigated the feasibility of large asynchronous systems by "porting" contemporary synchronous processors in order to build up credibility and confidence. Experiences from this research in asynchronous systems [42][137][145] helped to understand that asynchronous architecture design needed to develop alternative methods in order to ease the design process and to exploit the advantages of asynchronous circuits [15][72][110].

This chapter presents an overview of contemporary asynchronous architectures, and is not meant to be an exhaustive list. The chapter also describes the Micronet [4], an asynchronous architecture in which not only temporal parallelism, but also spatial parallelism is exploited. The micronet model used in this thesis and its functionality and characteristics are detailed. The architecture has been modelled in a stochastic event-driven simulator to evaluate the potential of code scheduling in asynchronous architectures.

4.2 Review of Asynchronous Architectures

4.2.1 AMULET

The AMULET group at Manchester University has developed three asynchronous processor implementations based on the ARM (synchronous) processors. The first of them, the AMULET1 [56], is an asynchronous micropipelined version of the ARM6 microprocessor. The implementation used a two-phase, bundled-data communication protocol. Its register bank is accessed through the use of a register-lock FIFO buffer to allow for multiple, pending write operations and

maintaining data coherence. The width of the register-lock FIFO buffer is equal to the number of registers, whereas the size of the buffer (its height) defines the number of pending write operations. Each location consists of a bit that indicates which register is to be written. Control logic ensures that at most one column attempts to write to a register. The datapath of the AMULET1 includes an ALU, a shifter, a multiplier and a memory unit, so as to be able to execute the ARM instruction set. The results showed that the AMULET1 was slightly larger in size and consumed more power, when compared with an ARM6 implementation using the same technology process. However, the ARM6 is a compact design and a highly efficient commercial processor in terms of performance per Watt, so this comparison was not entirely fair. AMULET1 did prove the feasibility of a large-scale asynchronous architecture.

The AMULET2 [57] improved on the AMULET1 in several aspects. The AMULET2 used the four-phase handshake scheme, which is faster and more power efficient; a data-forwarding mechanism reduced pressure on the register file and a branch prediction mechanism reduced the percentage of prefetched instructions that were discarded when a branch was taken. The AMULET1 was reported to have an average of three discarded instructions per branch. With branch prediction, this average was reduced to one. The AMULET2 design and manufacturing costs were equivalent to that of a similar clocked processor. Furthermore, the AMULET2 also demonstrated the potential for power efficiency (in terms of MIPS/Watt) and better EMI characteristics [58].

The AMULET3 [63] is the most recent implementation in this series. It offers similar performance and functionality as the ARM9TDMI microprocessor. The AMULET3 introduced new mechanisms and improved upon several aspects of AMULET2. For example, a Thumb decoder was incorporated for full compatibility with the Thumb instruction set, and a reorder buffer was incorporated at the write-back stage. The reorder buffer replaced the register-lock FIFO buffer used in previous AMULET designs. It enables data forwarding to be more dynamic and flexible. The result to be forwarded is stored in the reorder buffer until needed. The forwarding event can take place in parallel with the register write-back. The reorder buffer shortens the path for results normally written and read immediately via the register file, thus reducing the response time for results to be available. If the instruction results do not need to be forwarded, then they are written back in order. The AMULET3 performs favourably against the ARM9, in terms of power consumption, performance and size (they were both implemented in the same $0.35\,\mu m$ CMOS process).

4.2.2 NSR and Fred

NSR [138] and Fred [139] were asynchronous processors designed at the University of Utah which feature decoupled datapaths to hide the latency of slow operations, such as memory instructions. Pipeline stages communicate via variable-length FIFO buffers which allow instructions to proceed at their own pace. They also allow other instructions not to be held up by slow ones, such as memory or branch operations. However, the main disadvantage of FIFO buffers in between stages is the latency delays introduced into the pipelines which lowers the throughput.

Both processors issue one instruction at a time, but have out-of-order execution. They use register-locking schemes to preserve data consistency, similar to the ones used in the AMULET1 and AMULET2 processors. However, the NSR processor is a 16-bit implementation, whereas the Fred processor is a 32-bit one, with more functional units and is based on the Motorola 88100 instruction set.

4.2.3 Caltech Asynchronous Processors

The Caltech Asynchronous Processor was the first VLSI implementation of an asynchronous microprocessor [111]. It uses a 16-bit RISC-like instruction set and consists of three functional units: an ALU, a memory and a program counter. Its register file consisted of 16 registers that could be accessed concurrently. Measurements on the prototype [112] demonstrated the potential for wide operating conditions, *i.e.* testing was successful at a broad range of supply voltages (from 20 V to 0.35 V), and the ambient temperature varying from room temperature (300 °K) down to 77 °K.

The second Caltech processor was an asynchronous implementation of the MIPS R3000 [113]. The asynchronous MIPS processor considered architectural features not covered in the first design. These included caches, precise exceptions, register forwarding, branch prediction and the branch delay slot. Although the asynchronous MIPS processor was compatible with the MIPS instruction set, the datapath was not a straight synchronous to asynchronous pipeline conversion. The execution stage of the datapath was decomposed to allow for out-of-order execution, through the use of multiple functional units and a register unit. The register unit consisted of a register file, a register lock, and execution and bypass buses. The register file had two read and write ports which could operate concurrently. Results could be written either solely to the register file or written to both the register file and forwarded to a functional unit, if the following instruction requires the result.

Performance, as defined by $E\tau^2$ (where E is the average energy per instruc-

tion and τ is the average instruction execution time), compares favourably against other synchronous and asynchronous implementations. Furthermore, as the voltage is independent of this metric, it can be adjusted to select either high performance and a higher power consumption operation or a lower performance with a lower power consumption operation.

4.2.4 Counterflow Architecture

The counterflow pipeline processor (CFPP) [153] is different in that the instructions and results flow in opposite direction in the pipelines, in order to perform data-forwarding. In the instruction pipeline, instructions flow towards the register file, whilst results flow in the opposite direction towards the instruction fetch stage, in the results pipeline. Each instruction carries *binding* information about its source and destination operands. Each binding consists of a register name, a data value and a *validity* bit. This validity bit indicates whether an instruction should be cancelled by a trap or a branch. The instruction pipeline has several stages, each associated with a different functional unit.

Once a result is committed, the instruction binding is passed into the result pipeline. The result binding flows in the opposite direction in order to "meet" the instruction that requires that result. At every stage, instruction and result bindings are compared for a register name match. If there is a match, then the result is copied to the instruction binding.

However, the asynchronous counterflow datapath presents limitations mainly in two aspects. Firstly, the throughput of the pipeline is limited by the amount of control in each stage. The coordination between the instruction and result pipelines to compare their corresponding bindings requires arbitration. An arbiter-based mechanism is used to decide whether an instruction or a result can proceed to the next stage, in their corresponding direction. This depends on the state of the stage and its neighbouring stages, *i.e.* whether the stages are busy or not. Secondly, average-case execution cannot be fully exploited in the datapath because instructions from one pipeline need to 'synchronise' with results from the other pipeline. The speed in which instructions and results advance is therefore adjusted to the speed of the slower pipeline.

4.2.5 SCALP

SCALP was a superscalar asynchronous low-power processor [47]. The SCALP processor issued more than one instruction at a time. Instructions were encoded previously by the compiler with opcode, destination and functional unit specifiers.

The functional unit specifier indicated which functional unit was assigned to the instruction for execution, and the destination specifier denoted where the result had to be forwarded. The compiler, therefore, removed the tasks of identifying data dependencies and performing dynamic resource allocation, in the same way as compilers for VLIW architectures do. This information allowed the simplification of the issue unit for distributing instructions to functional units in parallel [48]. A crossbar switch was responsible for connecting the issue unit to multiple functional units.

In the SCALP architecture the concept of data-forwarding was taken beyond conventional data-forwarding schemes. Since all instructions embed the destination of their results, these were immediately redirected to their corresponding functional unit's queues after execution. This scheme has similarities to a data-flow machine, although control is not governed by the data, but by control logic. The SCALP processor treated the register bank as another functional unit. It was only used for medium-term storage, *i.e.* when a result was not immediately required by a following instruction. Short-term storage, *i.e.* consecutive dependent instructions, was substituted by data-forwarding. A result router was responsible for the distribution of results to the input queues of the functional units.

The programming model for a compiler targeted at the SCALP processor required several considerations. Both the code generation and the instruction set had to be tailored for such a data-forwarding mechanism. One typical example that differs from traditional programming models was when an operand was used more than once in a computation. Normally, in a register-based processor, one load operation would retrieve the value from memory into a register so that it could be used as many times as was needed (one producer to multiple consumers). With the dedicated data-forwarding scheme of SCALP (one producer to one consumer), the operand must be duplicated the number of times it will be needed, in order to be multiplely forwarded. Thus, in the code generation phase, duplicate instructions were inserted after the load instruction to avoid the use of multiple loads.

The compiler has to produce code to exploit the potential of explicit data-forwarding, and also not introduce deadlocks and non-determinism in the architecture. Deadlocks could occur if instructions sent multiple results and no instruction would consume them, whereas non-determinism would occur, if for example, two functional units sent results of the same operand to the same functional unit at the same time.

4.3 The Micronet Architectural Model

4.3.1 Preliminaries

The speed of synchronous pipelines is determined by the speed of the slowest stage, and the throughput is proportional to the length of the pipeline, *i.e.* the number of *active* instructions at a time. Figure 4.1 (a) shows a synchronous pipeline that exploits temporal parallelism. The pipeline shows four pipeline stages. The shadows represent the activity of the stages, while the white spaces represent idle times. It can be seen that resource efficiency is degraded when the latencies of the stages are not well balanced. The throughput of synchronous pipelines is dictated by the cycle time, which is determined by the speed of the slowest stage.

The speed of asynchronous pipelines is characterised by the average-case speed of their components[1]. Figure 4.1 (b) shows an asynchronous pipeline that offers the same amount of temporal parallelism, but exploits the actual delays, thus resulting in a more efficient resource utilisation. Micropipelines, as described by Sutherland [161], are representative of this type of asynchronous pipelines. In such pipelines only different instruction stages can operate concurrently. For example, the execution stages of two different instructions cannot overlap. The average throughput of such pipelines is limited by the stage with the slowest average throughput.

The work in [6] has proposed an asynchronous model of operation that not only exploits temporal parallelism, but spatial parallelism as well. Figure 4.1 (c) shows such a pipeline, in which stages from different instructions can overlap. From the example shown in the figure, it can be seen that the execution stages from two instructions take place concurrently at any time. In such pipelines further resource utilisation can be achieved. Furthermore, a degree of elasticity is exposed that allows for all the stages to operate concurrently. It has to be noted that in both asynchronous pipelines (Figures 4.1 (b) and 4.1 (c)), the self-timed protocols have been omitted for the sake of clarity.

[1] In fact, research has shown that, asynchronous pipelines operate closer to worse-case rather than average-case [68]. The reason is that in asynchronous pipelines, the maximum throughput requires a receiver to be always ready when its sender is going to transmit data, but also that by the time the receiver has completed its operation, its own receiver will also have to be ready. A continuous flow of data in this manner is described as *wave pipelining* [27][71]. In wave pipelines, the maximum throughput is determined by the difference between the fastest and the slowest component, as opposed to being determined by the slowest one, as in conventional pipelines.

4.3.2 Previous Work

4.3.2.1 An Asynchronous Network of Micro-operations

A *micronet* is a network of entities which compute concurrently and communicate asynchronously without centralised control [4]. This network of entities can be regarded as a generalisation of micropipelines. In micropipelines, instructions *propagate* at their own pace, and their execution times are bounded by the speed of the slowest stage. This effect has been presented as analogous to one-dimensional wave propagation. Such pipelines may exploit further temporal parallelism by relaxing the synchronous control, but the parallelism is limited by regular structures. For example, in a micropipeline datapath there cannot be more active instructions than the number of stages in the pipeline.

In contrast, the micronet model is limited by the number of functional units. The execution of an instruction consists of executing several micro-operations. Micro-operations communicate with each other only when necessary. An instruction only uses the micro-operations required for its execution. This enables other instructions to use resources within the same stage, if they are not used. For example, an instruction that requires only one operand, and therefore one *read* bus, will leave the other read bus and its entity available. The second *read* bus and its entity can be used concurrently by another instruction that also has just one operand. In this manner, fast instructions are able to overtake slower ones. In fact, this model of operation enables the competition for resources. The micronet datapath offers a finer-level of concurrency than the level of concurrency offered by micropipelines [5]. Simulations had demonstrated the following features of the micronet architecture: variable instruction execution due to the type of instruction and the availability of operands; the control overhead in a micronet is hidden by the concurrent operations inherent in the model; the concurrent execution of micro-operations of different instructions in the same datapath. In effect a scalar micronet datapath can exploit both temporal as well as spatial parallelism.

4.3.2.2 VLSI Implementation

The implementation feasibility of the micronet model has recently been studied in [151], in which a transistor-level VLSI implementation of a micronet-based asynchronous processor is presented. The VLSI micronet implementation is a 32-bit scalar processor that contains a register file with two read ports and one write port, and three functional units, namely an arithmetic unit, a memory interface unit and a branch unit.

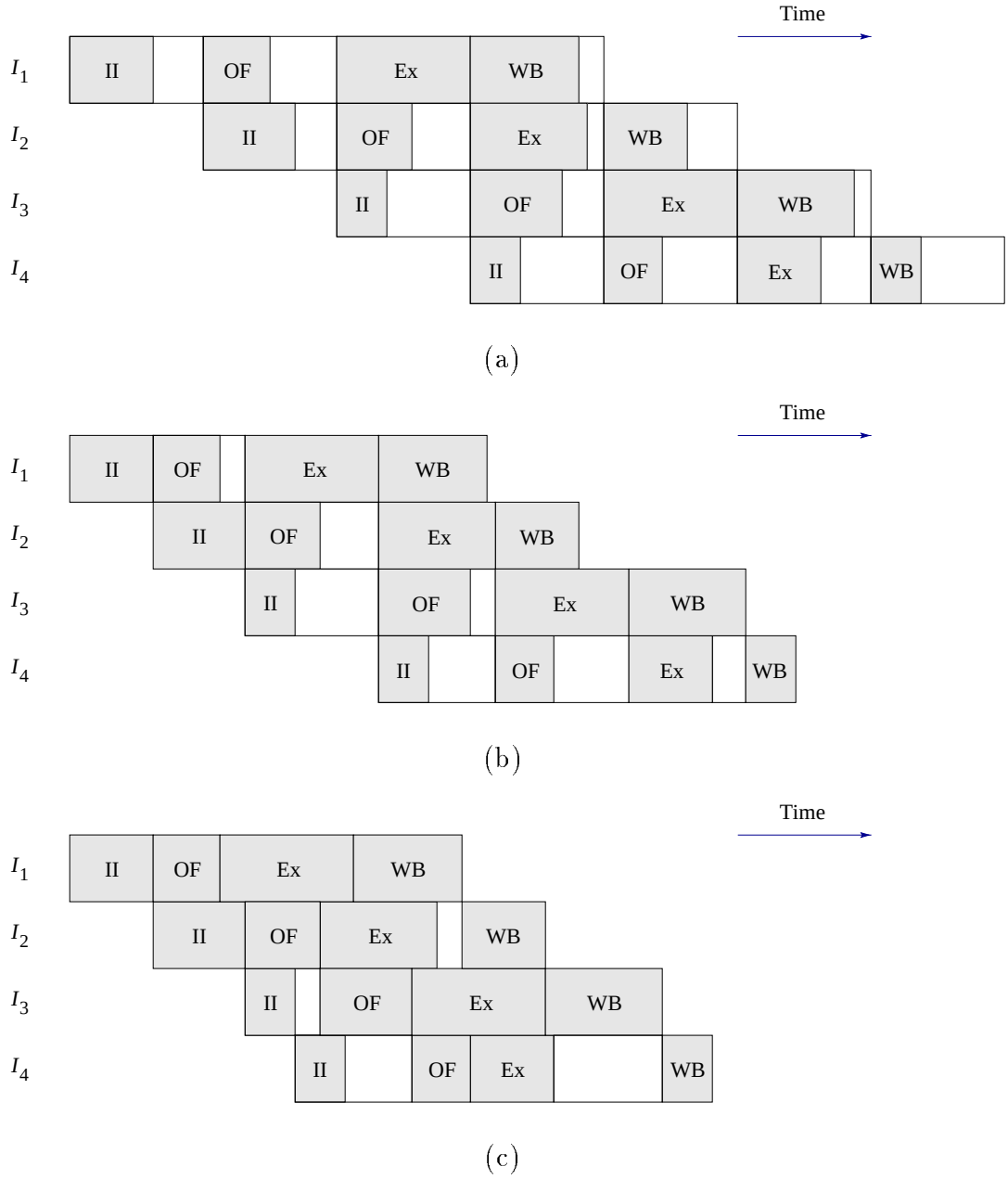


Figure 4.1: (a) A synchronous pipeline, (b) an asynchronous pipeline and (c) an asynchronous pipeline that exploits spatial parallelism.

4.3.3 Architectural Description

The model of the micronet architecture used throughout this work is shown in Figure 4.2. The micronet processor is composed of an issue unit, an operand fetch unit, a set of functional units and a write-back unit. The functional units include

a memory unit, an arithmetic unit, a logical unit and a floating-point unit. The architecture can be configured to have more than one instance of the arithmetic, logical and floating point units. The typical operations of an arithmetic unit are integer addition, substraction, multiplication and division, but it may also include immediate and address *loads* (a load operation writes a single address value directly into a register). The logical unit performs bitwise operations such as bit shifting, bitwise comparisons and logical operations, *i.e.* AND, OR, XOR and NOT. The floating-point unit performs similar arithmetic operations on floating-point numbers. The memory unit loads from, and stores values into, the memory through a data cache. The memory unit includes an internal adder for calculating the effective memory address. Each functional unit is independently connected to both the operand fetch unit and the write-back unit, to allow instructions with different types to be executed concurrently. Furthermore, each functional unit is assigned a pair of *read* buses and a *write* bus. During a computation, temporary results are stored in a multiported register file.

The order of events during execution is the following. Initially, the issue unit fetches instructions from the instruction cache and issues them to the operand fetch unit. The operand fetch unit reads the operand values from the register file through the pair of *read* buses and hands them over to the corresponding functional unit. When the functional unit completes its execution, the result is sent to the write-back unit. Finally, the write-back unit stores the results in the register file via a *write* bus.

The issue unit issues one instruction at a time in an in-order manner. It is responsible for issuing instructions as soon as their operands become ready, *i.e.* their operands have been written into the register file. If the operands are ready, then the issue unit inspects if both *read* buses and a functional unit corresponding to the appropriate instruction type are available. Depending on the availability of these resources, the issue unit issues or stalls the instruction. If the resources are available then the instruction is issued, otherwise the instruction is conditionally issued, so that it proceeds up to the resource that is busy, and stalls. The outcome of this scheme provides for different cases depending on the availability of these resources. These are described as follows:

- If there are no functional units of the type required which are ready, but the *read* buses and the operands are available, then the instruction will be issued and the operand fetch unit will proceed normally, but will stall immediately after reading the operands. The instruction will remain stalled until a functional unit becomes ready to execute the operation.

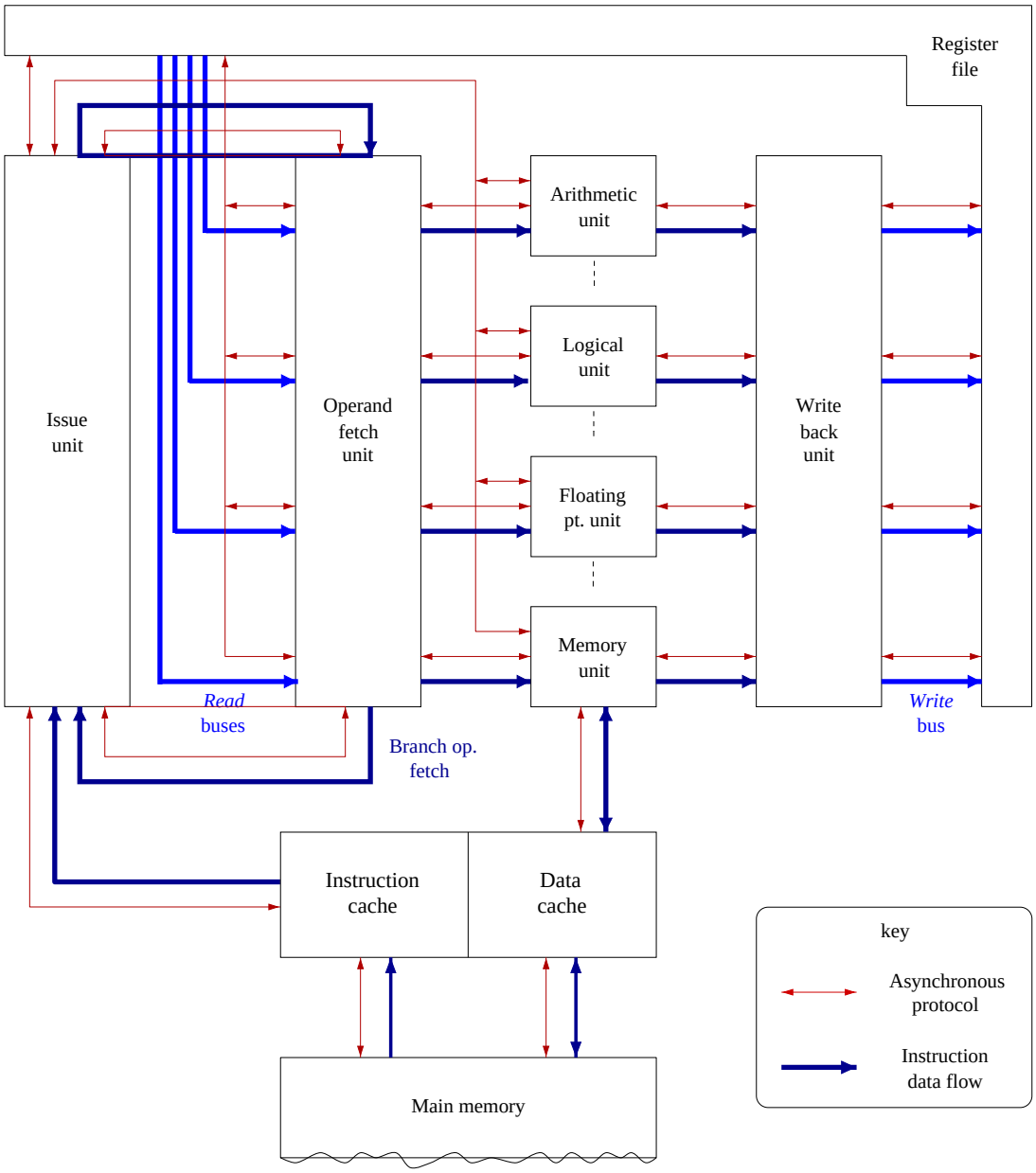


Figure 4.2: Architectural model of the micronet-based datapath.

- If any of the source operands of the instruction are not ready (data requirement), or the *read* buses are not available (resource requirement), the instruction will not be issued. The instruction will remain stalled until all of its operands are ready and there are *read* buses available. Since the architecture issues instructions in-order, when these data and resource requirements are not available, the issue unit will remain stalled.

When a functional unit completes the execution of an instruction, it will send the result, only if there is a *write* bus available. If no *write* buses are ready, then the functional unit stalls and remains in the "busy" state until it can deliver its result to the write-back unit.

4.3.3.1 Data Coherence

The register file contains a bank of registers with one or more read ports and one write port. The register contents are retrieved from one of the read ports, whereas results are written to the registers via the write port. Read and write accesses can take place in parallel if they refer to different register locations. If a register has more than one read port, then its contents can be read concurrently.

The mechanism for ensuring data coherence during instruction execution using a register file is based on the register *locking* scheme [127]. The concept of locking registers has been a common solution in many asynchronous implementations [56][57][139]. The VLSI implementation of the micronet-based processor [151] also uses the register locking approach. The locking scheme is required to guarantee correct data operation in the presence of asynchronous accesses to the register file. The mechanism consists of a device that contains an individual *lock* bit per register. The lock bit indicates that a register is yet to be written by a pending instruction. In the micronet model, if the lock bit is set, it is implied that the register cannot be read from, or written to, by any subsequent instruction. Conversely, if the lock bit is unset, any instruction[2] can read the contents of the register until the register is locked again.

Since there is no way of knowing how long it will take for a register to contain valid data, the issue unit cannot issue instructions that depend on a locked register, *i.e.* if there is a RAW or a WAW dependency. In either case, the issue unit remains stalled until the value of the operand is available, *i.e.* the register is unlocked. The register locking mechanism ensures that locked registers cannot be accessed (read or written) by any instruction before their results are committed.

When the issue unit proceeds to issue an instruction, its destination register is locked. This register remains locked throughout the duration of the instruction's execution, and only after the write-back unit commits the result will the register be unlocked. The write-back of a result will cause a pending instruction waiting for that result to update its status. As mentioned before, the issue unit checks for the availability of operands; if the unlocked register was the only operand that

[2]As many instructions as the number of read ports.

was being awaited for, then the destination register of the stalled instruction is locked similarly, and the instruction can be issued and will proceed as long as its resources are available (buses and functional unit).

Maintaining data coherence for memory accesses is more complicated than the register locking mechanism. The architecture shown in Figure 4.2 does not consider more than one memory in the interest of simplicity. If multiple memory units were available, that would allow for concurrent memory operations. Implementing concurrent memory operations introduces the possibility that loads may overtake stores and vice-versa. This means that every time such a case should arise, the memory locations being referenced would have to be *disambiguated*, *i.e.* compared and proved to be different, in order to avoid violating memory dependencies. Such dependencies occur when loads and stores refer to the same memory location. If a store precedes a load in the code, it means that there is a true dependency (RAW), whereas if a load precedes a store, it represents an anti-dependency (WAR). If two stores refer to the same memory address, it means that there is an output dependency (WAW).

With the current architecture (one memory unit with in-order issue) dynamic memory disambiguation is not required, simply because loads and stores execute in-order. Implementing run-time memory disambiguation in synchronous processor architectures poses substantial hardware overheads [53]. In an asynchronous design, run-time memory disambiguation may restrict the performance even further because memory references need to be synchronised in order to be disambiguated.

4.3.3.2 Write-back Operation

When a functional unit completes the execution of an instruction, the only potential contention in the write-back operation is the need for a *write* bus. Since the register has been previously locked, it is guaranteed that none of the other functional units will attempt to write to the same register. This means that it is safe to write to the register as soon as a *write* bus is available. The architecture shown in Figure 4.2 assigns a *write* bus to every functional unit, thus removing this resource contention.

4.3.3.3 Control-flow Operations

The execution of control-flow instructions in the architecture shown in Figure 4.2 is performed by the issue unit. The processor's program counter (PC) is updated depending on the type of *control* instruction, *i.e.* `jump`, `call`, `return` or `branch`.

When a `jump` instruction is issued, the PC is updated with the address specified by its operand. The issue unit then resumes instruction fetch from the new PC. A `call` instruction causes the PC to be updated in the same way as `jump` instructions do, with the difference that the PC is saved to the stack memory, before its value is updated. A `return` instruction simply restores the PC from the stack memory. Finally, `branch` instructions modify the PC depending on the result of register comparisons. The result of such comparisons is written into a register. Once the result of the comparison is committed, the issue unit issues the `branch` instruction and the operand can be fetched. The operand fetch unit returns the value of the register back to the issue unit (*c.f.* Figure 4.2), where the PC is updated depending upon the value. In the case of a *branch-if-true* `branch` instruction, the PC is updated if the register contents are not a zero value, *i.e.* usually one. In the case of a *branch-if-false* `branch` instruction, if the register contains the value zero, then the PC is updated. A `branch` instruction is regarded as a conditional `jump` instruction.

This concept of "test-and-branch" is similar to the branch *decoupling* scheme of the NSR processor [138]. The difference is that the outcome of the test in the NSR processor is stored in a flag bit rather than a register, but the branching mechanism waiting for the flag to be set is similar. Instructions can be scheduled between the test and the branch in order to avoid the hazard.

4.3.4 Parametric Model

4.3.4.1 Configuration Description

The description of the architecture may be parameterised in order to be able to include different types of components and various connectivities between them. A configuration file describes the architecture, *i.e.* the type and number of functional units, the number of *read* and *write* buses, the latencies of the various components and the instruction set. The instruction set is defined by specifying the instruction types that can be executed by the various types of functional units. For each group of instructions, *read* buses, a functional unit and a *write* bus are assigned. These resources will be required during the execution of any of the instructions belonging to a particular group. In this way, functional units can be "specialised" to perform specific operations. Instructions from each group are described by their opcode and the number and type of operands they require.

In asynchronous architectures the completion time of a functional unit depends on both static and dynamic factors, as discussed in Section 3.4.4. The static factor

Component Type		Latency time		Cycle time	
		Minimum	Maximum	Minimum	Maximum
Issue unit	(IU)	1.00	2.00	0.50	1.00
Read buses	(RF)	2.00	4.00	0.50	1.00
Write buses	(RF)	2.00	4.00	0.50	1.00
Arithmetic unit	(AU)	4.00	8.50	0.50	1.00
Logical unit	(LU)	2.00	7.00	0.50	1.00
Floating point unit	(FU)	6.00	8.00	0.50	1.00
Memory unit	(MU)	10.00	20.00	0.50	1.00

Table 4.1: Latency distribution for the different components in ns.

is based on the type of functional unit. The type of functional unit operation determines the range of delays. For each architectural component its latency and cycle time must be specified whose operating range is bounded between a minimum and a maximum value. This range is set to model dynamic factors such as the input data.

The range of latency and cycle times for the different architectural components is shown in Table 4.1. Some of these values are based on SPICE simulations from a prototype of a micronet datapath in $0.7\,\mu m$ CMOS process technology [4]. These include the issue unit, the *read* and *write* buses and the arithmetic unit. The values from the table will be used systematically throughout this thesis. The range of latencies in the table attempts to reflect values with a reasonable variance, so that operations are not considered to complete in fixed times. The variance represented in the table reflects delays due to all the possible aspects that affect the latency as described in Section 3.4.5, *i.e.* from data-related ones to process and temperature variation. Moreover, the relative latencies imply that some operations are more costly than others in terms of delay. This configuration presents arithmetic and logical units which are relatively faster than the memory unit, since a memory operation may include an addition, and the actual process of loading from, or storing into, memory is relatively slow. Similarly, the issue unit is faster than any other component in order to model a considerably fast single-issue architecture in which resources are kept busy [4]. The register file has been partitioned into separate components: the access times of the *read* buses and the access times of the *write* buses. The table also shows cycle times that

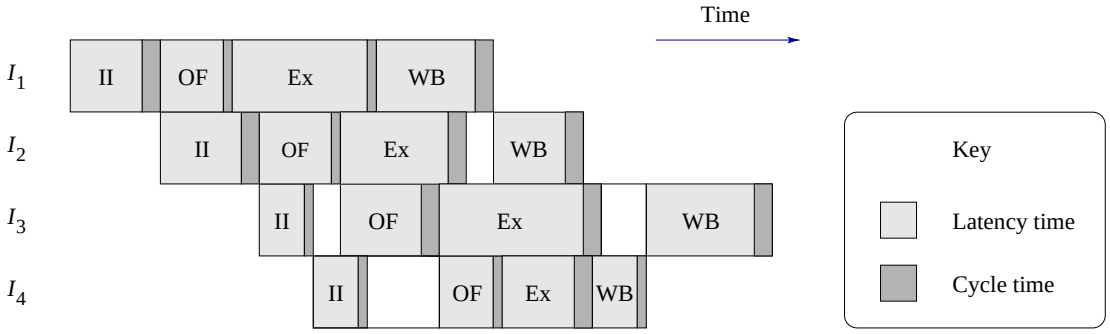


Figure 4.3: The micronet operation.

are relatively smaller than latency times. Cycle time represents the delay for components to become ready again after the completion of an operation.

An example of the asynchronous pipeline of the micronet architecture is shown in Figure 4.3 which incorporates both latency and cycle times.

4.3.4.2 Components Distributions

The latency distribution of the different architectural components are defined depending on the nature of their type. For example, the memory unit has a bimodal distribution which attempts to simulate the cache behaviour, *i.e.* either a cache hit or a cache miss. The minimum latency time represents a cache hit and the maximum latency time represents a cache miss. The distribution is equally balanced so that cache hits and cache misses have the same probability[3].

The arithmetic unit has a linear distribution starting from the minimum latency value towards the maximum latency. Previous research in asynchronous adders [94] demonstrated that for a random set of input data, 50% of the additions can be completed with delays close the minimum. For the rest of the additions, the completion times are incremented significantly towards the maximum latency. This behaviour, of course, depends on the implementation, but a continuous linear distribution is normally expected [62][151].

The rest of the components have been modelled with uniform distributions bounded within minimum and maximum latencies and cycle times as specified in Table 4.1.

[3]This is a pessimistic assumption considering that cache hit:miss ratios achieved nowadays can be as high as 95% for some benchmarks.

4.3.4.3 Instruction Set

The instruction set used for the micronet model is based on the MIPS instruction set [91]. Appendix B shows an example of the configuration file that contains the parametric description of the architecture. The description file contains the number and types of functional units and their latencies in nanoseconds. The description of the instruction set includes the assignment of instructions to functional units and buses for execution.

4.3.5 Characteristics

The architectural model described in Section 4.3.3 in conjunction with the latency distribution from Table 4.1 offers interesting characteristics. The micronet-based model presents a scalar architecture that features a fast in-order single issue unit, and a write-back stage where results are committed fully out-of-order. Having a fast single issue unit models a processor capable of issuing more than one instruction at a time, without the additional hardware cost of superscalar designs [45][48]. Single instruction issue also restrains the potential growth of complexity and size of asynchronous superscalar issue units.

Out-of-order write-back schemes avoid the need to reorder write-back events when results are ready to be committed. Reordering these events to maintain in-order write-backs introduces synchronisation, which reduces the benefits of average-case execution. The use of both out-of-order execution and out-of-order write-back exploits more parallelism.

These features give the asynchronous micronet model some characteristics similar to VLIW and superscalar architectures. To sustain a fast issue rate, streams of independent instructions must be available. VLIW and superscalar architectures sustain a fast issue rate by issuing multiple independent instructions. VLIW architectures issue more than one instruction at a time, because the code has previously been analysed and scheduled by a compiler. As a result, the control section of a VLIW architecture is much simplified. Superscalar architectures, on the other hand, require significant hardware control to perform dynamic scheduling, when the compiler is unable to provide independent instructions.

The micronet architecture shares with VLIW architectures the characteristic of not having to perform dynamic scheduling, which is expensive in terms of hardware complexity, and it shares with superscalar architectures the need to prevent hazards at run-time. The micronet model shares with both architectural schemes, the need to identify independent instructions in order to issue as fast as possible.

This model can be regarded as if the issue unit were "dealing" instructions to the group of functional units. To a certain extent, VLIW architectures operate in the same manner, but with the difference being that the multiple issue process takes place in parallel.

Another similarity between a micronet and a VLIW architecture is that data-forwarding is not implemented. Once an instruction is executed, its result is committed into the register file. Although in principle data-forwarding can reduce an instruction's execution time, it requires that the operand fetch stage is synchronised with the write-back stage. This synchronisation will inevitably slow down the faster stage, *i.e.* the fetch stage, when two dependent instructions are fetched and issued one after the other. In such cases this synchronisation will take place, whether or not data forwarding is implemented. However, if the instructions are not scheduled one after the other, or there is more than one instruction waiting for the result of the first instruction, then the write-back stage of the first instruction could be held up unnecessarily[4].

In the micronet model, not having data-forwarding allows instructions to execute as fast as possible. As soon as their requirements are fulfilled (operands and functional unit), instructions will run to completion without synchronisations.

This is the fundamental difference with VLIW architectures. In the VLIW approach, the compiler uses aggressive compilation techniques to expose ILP to utilise the functional units. When the compiler cannot provide enough independent operations in one cycle, bubbles, *i.e.* no-operations, fill the empty slots of the VLIW instruction word. In the micronet architecture on the other hand, the introduction of no-operations is impractical as the issue unit would be spending time processing instructions that do not contribute to the execution of the program. Thus, the micronet compiler must schedule independent instructions in such a way that the issue unit does not stall, or stalls minimally if it does. The goal of the scheduler is therefore to maintain a fast instruction issue rate.

4.3.6 Event-driven Simulator

A stochastic event-driven simulator for the micronet architecture had already been implemented [97]. It works by executing assembly-level instructions compiled from source programs. During instruction execution, the simulator creates events

[4]Forwarding in asynchronous architectures does not have straightforward solutions, and it raises specific synchronisation issues. The SCALP architecture [47], for example, is located at the other end of the scale, where most of the communication is based through the use of data-forwarding. The complexity of the datapath is quite considerable. Since results can be required by any functional unit, a result router connects the outputs to the input buffers.

for the different datapath components. These events are dynamically created depending on an instruction's component requirements. An instruction's type is used to decide its execution path through the micronet datapath. During execution, events from neighbouring components communicate when the data is ready to be transfered from one stage to the other. For example, the issue unit generates an event to the operand fetch stage unit when an instruction is issued. Then, when the instruction is ready for execution, the operand fetch unit will generate an event to the appropriate unit. When the functional unit completes the execution of the instruction, a write-back event is generated. The write-back event emulates the write-back stage, and writes the result back to the register file and unlocks the register.

Every time an event is generated, both the latency and the cycle times are dynamically associated with it depending on the instruction type. The instruction type also determines which random distribution needs to be selected. The latency time denotes the time when data becomes available, whereas the cycle time represents the time when that particular stage can restart its operation. The startup time of an event is based on the latency time of another event that has been previously generated. When an issue event has completed for example, the simulator will assign to the next event, *i.e.* the operand fetch, a startup time which is equal to the latency of the issue event[5]. The next issue event, however, starts only when the current issue event expires its cycle time. Therefore, the start time of the new issue event corresponds to the cycle time of the current event, *i.e.* the one which is about to finish.

When the event is created and its set of timings are assigned (startup, latency and cycle times), the event is inserted into an event list, which is ordered by event startup times. Each event is processed according to the simulation time. It is possible that two events occur concurrently, *i.e.* have the same time stamp, but are processed sequentially. The simulation finishes when the event list is empty.

The simulator uses a global time variable that holds the time of the event that is currently in process. An event that cannot proceed, because for example due to the lack of available resources or because a register is locked, must delay its startup time until the resources become available or the register is unlocked. Of course by that time, many other non-related events may take place. When an event needs to be stalled, the simulator creates a new event with the same

[5]The simulator does not assign additional time for handshake delays during communication, with the premise that handshakes take considerably smaller time than latency or cycle delays.

attributes as the former, *i.e.* the same unit and instruction information, updated startup time, and the old event is deleted.

When a program is to be simulated, memory instructions must be *"pre-loaded"*, so that load and store instructions refer to a common and global memory map. This means that global variables and data structures are pre-assigned to fixed memory locations used by the simulator. These fixed memory locations belong to a *memory* map. All program references to global locations are modified accordingly. For example, a global variable `max_value` in a load instruction (for example `lw $17, max_value`) will be modified to `lw $17, num`, where `num` corresponds to the particular memory location pre-loaded, which will be used at run-time. In this way, values from global variables can be initialised so that the simulator begins instruction execution with the correct data.

The simulator actually executes the instructions, hence does not need an execution trace. It is effectively a data-driven simulator in which the data are responsible for taking the correct paths during the execution of a program. At the end of a simulation, the simulator produces real results, *i.e.* both the memory and registers hold the correct data.

The simulator's level of abstraction allows for the interaction between events in the micronet datapath to be captured. It is implemented at such granularity so that the simulation speed is not compromised. If all control handshaking was explicitly modelled, the simulation speed would be significantly slower.

4.4 Summary

Different asynchronous architectures have been developed to investigate feasibility, power efficiency and performance. The AMULET group at Manchester University has implemented three asynchronous versions of synchronous ARM processors, achieving different goals. The AMULET1 proved the feasibility of a large design comparable in size to the ARM6 processor; in the AMULET2, architectural improvements were incorporated such as data-forwarding and branch prediction that increased prefetch efficiency, and the AMULET3 core included Thumb-instruction execution compatibility, the use of interrupts and a write-back reorder buffer, that overall helped perform favourably against the ARM9TDMI processor.

The asynchronous processors from Caltech have shown the potential of performance and adaptation to wide physical conditions in asynchrony. The counterflow architecture has been proposed as an alternative approach to data-forwarding,

although synchronisation at every stage appeared to limit throughput and parallelism. The SCALP processor has demonstrated the complexity involved in superscalar asynchronous design, particularly in the distribution of instructions from the issue unit to the functional units, and the distribution of results back to the functional units.

This chapter has also described the micronet asynchronous architecture which distributes control in order to exploit both temporal and spatial parallelism. The micronet architectural model presented features single instruction issue and out-of-order write-back. The architecture is capable of issuing one instruction at a time at a fast rate. This characteristic attempts to mimic multiple instruction issue without the expense of implementing it.

The variability of instruction latencies presents a challenging problem for the compiler. A compiler is required to schedule the code in order to minimise issue stalls due to data and resource dependencies. The next chapter introduces a novel instruction scheduling approach for asynchronous architectures. A local instruction scheduler targeted at the micronet model described in this chapter is next presented.

Chapter 5

Local Scheduling for Micronet-based Architectures

"If a processor exposes the variations in actual memory reference latency to the compiler through non-blocking load instructions, instruction scheduling becomes more complicated" [92].

5.1 Introduction

The previous chapter described the model of an asynchronous micronet-based processor. Its basic characteristics are that it issues one instruction at a time at a very fast rate and writes the results back in an out-of-order fashion. The datapath of a micronet-based processor exploits fine-grain temporal and spatial parallelism by executing instructions on a network of microagents that communicate asynchronously.

The ability of these microagents to communicate independently between them, allows instructions to be executed as fast as possible without unnecessary global synchronisations. Furthermore, microagents of different instructions within the same *pipeline* stage can execute concurrently. The maximum number of active instructions in a micronet is limited by the number of microagents as opposed to the number of pipeline stages as in micropipelines. Instructions that do not require resources from a pipeline stage are able to skip the stage and thus enabling them to overtake one another.

The asynchronous nature of the processor provides the benefits of average-case execution, *i.e.* exploits the actual delays of the functional units. The delays due to the functional units depend on several aspects. These include the nature of the input data, the type of the functional unit and the particular order of execution of the instructions, since their order might introduce resource contentions at run-

time.

The combination of all these characteristics pose a challenging problem for the compiler, not only because instructions do not have fixed completion times, but because the order in which instructions are executed cannot always be enforced. This implies that the compiler does not have an accurate model to consistently predict when results will be available. This makes efficient code scheduling difficult since typical scheduling heuristics (as used for synchronous targets) rely on deterministic and timing-precise execution models. This knowledge is necessary to decide whether instructions dependent on a result can become ready to be scheduled, as has been described in Chapter 3.

The performance of the micronet processor is dependent on maintaining a high instruction issue rate to maintain high resource utilisation rates. The stream of instructions must be scheduled in such a way that the issue unit stalls for the minimum possible time.

This chapter presents a novel technique for implementing local instruction scheduling, as specified in Chapters 2 and 3, which can be applied to an asynchronous processor based on the micronet model. The proposed scheduling technique is not based on the list scheduler [64] which is traditionally used for scheduling synchronous ILP architectures. The technique in this thesis is based on identifying and producing a measure of the cost of true data dependencies in the code. This measure is then used to arrange the instructions in order to minimise the cost of the dependencies, and therefore, minimise the stalls in the instruction issue unit.

5.2 The Influence of Dependencies

5.2.1 Data Dependencies

Data dependencies impose a serialisation in the execution of instructions. ILP architectures are characterised by a single thread of control in which instruction-level parallelism can be exploited. However, data dependencies restrain the scope of parallelism. In particular, true dependencies or Read-After-Write (RAW) dependencies, require the completion of an instruction before its result can be used by its dependent instructions.

The cost of the effect of true dependencies in an architecture can be addressed either in hardware or software. In hardware, for example, the data-forwarding mechanism was conceived to avoid the penalty of having to write to the register bank, and some time later, reading the result from it. Data-forwarding is a very common solution in synchronous designs. However, in asynchronous architectures

this solution may be less effective because it may introduce undesired synchronisations, as explained in Section 4.3.5.

In software, the method for minimising the effect of true dependencies is through the use of instruction scheduling. Independent instructions are placed in between the producer and the consumer instructions in order to hide the cost incurred at run-time.

Schedulers for synchronous architectures rely on an accurate knowledge of the penalty in the presence of true-dependent instructions. The penalty in synchronous architectures is expressed in terms of number of clock cycles. The compiler can decide the number of 'clock cycles' that the dependent instructions should be distanced by.

In contrast, the schedulers for asynchronous architectures do not have such a precise model of execution. This implies that the penalty of true dependencies cannot be precisely fixed, and therefore, it is unclear how one decides the number of independent instructions that are required to minimise the cost of the dependency. The balance should be right: too few instructions may not reduce completely the effect of the stall, and too many instructions may deprive independent instructions for reducing the stalls in other parts of the code. The next section proposes a method for measuring the cost of true dependencies for the asynchronous micronet model.

5.2.1.1 True Data Dependencies in the Micronet Model

Although the latencies of asynchronous functional units are not fixed, they are bounded within minimum and maximum values, based on best and worst cases, respectively. The actual value of the latency depends on several static and dynamic elements as discussed in Section 3.4.4. Among the elements contributing to the latency value, only the static ones are available to the compiler. These elements can be divided into two groups: the first one relates to the code itself, that covers the ordering of instructions and their data dependencies; the second one covers the parametric model of the architecture, which includes the number and types of functional units and their minimum and maximum latency values. The dynamic elements which contribute to the latency are related to the input data, and to run-time variations such as resource contentions and operating conditions.

In the micronet model, in order to issue instructions, the issue unit must check that the operands and the necessary resources are available. Otherwise, the instruction will not proceed, and the issue unit will remain stalled until the requirements are fulfilled. The total amount of time due to the stall depends on

the degree of conflict during the instruction execution. For example, the minimum stall produced will be when the operands and the *read* buses are available, but the functional unit of the required type is unavailable. In this case, the instruction will be issued and its operands will be fetched, but it will stall until the functional unit becomes free. In contrast, the maximum stall will take place when the operands of an instruction are unavailable, *i.e.* they are locked. In such cases, the issue unit stalls until they are unlocked.

The exact stall times cannot be determined statically, although the cost of the different stalls relates to the type of dependency. In synchronous architectures, data dependencies do impinge on the cost of the stall, but this cost will be constant in terms of clock cycles [76].

In the micronet datapath, the stall due to a true dependency is directly related to the completion time of the instruction that produces the result. (The completion time of the instruction is the sum of the times to fetch its operands, to operate the functional unit, and to write the result in the register file). Since the issue unit has to wait until the register is unlocked, before issuing the dependent instruction, the execution time of the producer instruction is related to the type of its functional unit. The relative differences between the range of latencies of different functional units can distinguish an instruction type to be *slower* than another, and therefore produce a longer stall.

In general, memory operations are slower than any other register-based operation, so one should expect that the cost of true dependencies from memory operations is higher than the cost from other non-memory true dependencies. However, generalising this assumption to any case may be difficult to guarantee, since latencies have a range of operation and the range of latencies may be overlapped within different functional units. Figure 5.1 shows the execution stages of two true-dependent instructions I_1 and I_2. In Figure 5.1 (a), instruction I_2 depends on a memory instruction I_1, whereas in Figure 5.1 (b), instruction I_2 depends on a non-memory instruction I_1'. It can be seen that even with variations of delay in the stages from instructions I_1 and I_1', the functional unit dominates the completion time.

5.2.1.2 Penalising True Data Dependencies

Given that true dependencies contribute significantly to stall-times of the issue unit, a mechanism to assign penalties to them has the potential to discriminate between schedules. The penalty provides a measure to evaluate the accumulative

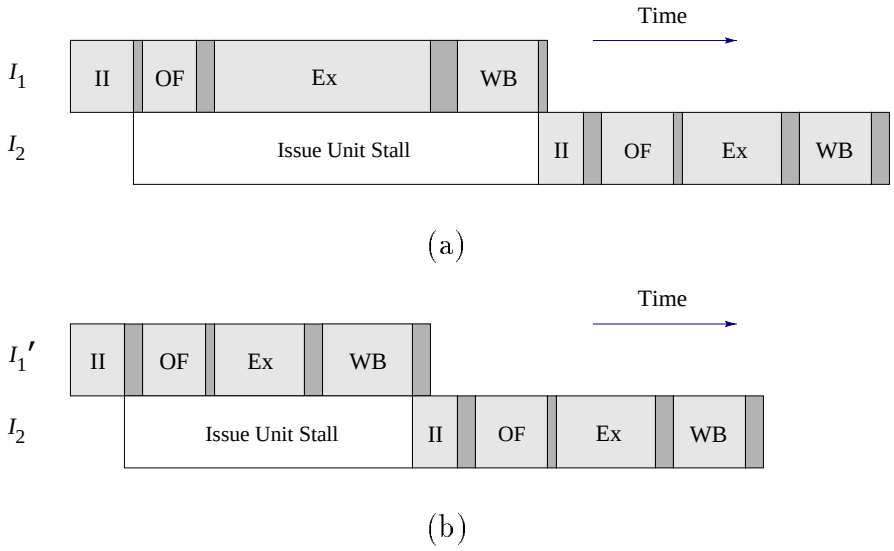


Figure 5.1: Pipeline execution with true data dependencies: (a) from a memory instruction, and (b) from a non-memory instruction.

stall incurred by the issue unit at run-time.

The *measure of penalty* consists of the accumulative cost incurred by true data dependencies. For a fully sequential code, the penalty measure should return a high value of measure as every true dependency is assigned a penalty. For a fully parallel code, the measure of penalty should return a zero measure, given that there are no data dependencies.

Given that the delay-cost from a true data dependency caused by a memory instruction is higher than that due to a true dependency caused by an arithmetic operation, the penalty for the memory instruction is higher than the penalty assigned to the arithmetic one. Figure 5.2 (on the left) shows a sequence of unscheduled instructions, in which an address is obtained (instruction I_5) in order to load a value from memory (instruction I_6), which is required by another instruction (instruction I_7). The penalty P_b assigned to instruction I_6 represents a higher penalty than the penalty P_a assigned to instruction I_5, in order to reflect better the effect on stalls. The penalties are applied to the instructions rather than to the dependencies (arcs) because in this way the producer of a result can be recorded, even if independent instructions are placed in between the producer and the consumer.

The method to establish the degree of cost due to data dependencies is performed by comparing the relative latency times from the different types of the

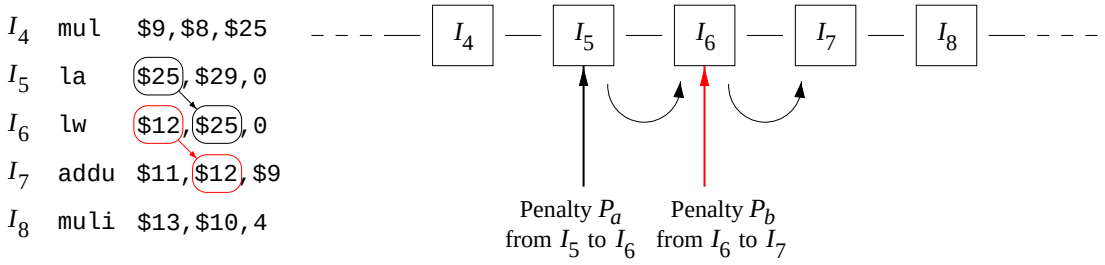


Figure 5.2: Sequence of instructions with penalties.

functional units. The difference in the range of latencies provides a guideline of the difference in delay-costs. A greater difference in latencies corresponds to a greater effect on the stall of the issue unit.

Among the true dependencies, the ones between the compare-and-branch instruction are particularly costly. This instruction represents a change in the control flow and acts as a synchronisation point for all the instructions that precede it in the basic block. Typically, this synchronisation point implies that less parallelism is available when the issue approaches the branch. In a loop for example, it is important to capture the cost of the true dependency on the branch with independent instructions. Otherwise, the execution of the loop will have to be stalled in every iteration until the branch is resolved.

5.2.1.3 Other Data Dependencies

WAR dependencies are not penalised as they do not cause stalls to the issue unit. If one instruction depends on another via a WAR dependency, by the time the register of the former instruction is locked, the latter instruction has been issued (in-order issue). The operand fetch of the latter instruction takes place much earlier than the time the register has to be written by the former instruction.

In WAW dependencies, the penalty of stalling the issue unit is the same as for true dependencies, because the dependent instruction cannot attempt to lock its destination register when it is already locked. However, the occurrence of these type of dependencies after the code generation phase is rare[1].

[1] For all the benchmark programs described in Chapter 7 only one instance of WAW dependencies was found.

5.2.2 The Effects of Resource Dependencies

The scheduling problem is based on selecting a schedule that minimises the number of stalls of the issue unit during the execution of instructions. These stalls can be caused either by data dependencies or by resource contentions. So even scheduling independent instructions may produce stalls because the instruction sequence could exhaust the resources of the architecture at run-time.

The sequence of instructions must be scheduled in such a way that the types of instructions match the resources in the architecture. For example, if there are three functional units of a particular type, then three instructions of that type can be issued in succession without causing stalls (assuming that there are no data dependencies). However, a fourth instruction of the same type will be expected to stall until one of the functional units become available. In this case, to avoid resource contention, the type of the fourth instruction that is scheduled must be different.

5.2.2.1 Penalising Resource Dependencies

When more than one instruction of the same type is scheduled consecutively and in the absence of functional units of the appropriate type, a penalty is imposed to the latter instructions. For instance, given that the delay from memory instructions is relatively larger than the ones due to other instruction types and that there is only one memory unit in the architecture, the scheduling of two consecutive memory instructions produces a considerable stall in the datapath. The amount of stall caused by consecutive memory instructions is comparable to that due to true dependencies. Although the instruction can be issued and the operands can be fetched (something that cannot happen with true dependencies), the wait delay until the memory unit completes its operation is non-trivial. For other types of functional units, this situation might not be as important for two reasons. Firstly, the non-memory functional units may be replicated, so scaling the architecture can be a solution. Secondly, the latencies of non-memory instructions are smaller, and the effect on the stall is proportionally lighter.

5.2.3 The Combined Effect of Data and Resource Dependencies

Table 5.1 shows the scheme for applying penalties to the different cases of dependencies. The values shown in the table represent the penalties for the latency distribution in Table 4.1. True dependencies due to memory loads incur the

Types of dependencies	Consecutive instructions	Separated by one inst.
True dependency with a load instruction	3	1
True dependency with a branch instruction	2	0
Any other true dependency	1	0
Resource dependency from a memory instruction	1	0

Table 5.1: Degree of the penalties depending of the type of dependency.

longest stall, and are therefore assigned the highest penalty. The cost-effect of a memory load is such that even when an independent instruction separates the load and its successor, a single penalty has to be applied.

The penalty due to true dependencies with branch instructions is assigned the next level of cost. The resulting stall is comparable to the other true dependencies since the penalty is the result of a compare instruction. However, the synchronisation nature of branch instructions makes these true dependencies more important to reduce. The rest of the data dependencies are treated at the same level.

The penalties in Table 5.1 attempt to both characterise the delay-cost of the dependencies, and provide an ordering that prioritises the penalties that ought to be reduced by the scheduler.

5.2.4 Applying Penalties to a Schedule

The scheme to penalise the dependencies defined in Section 5.2.3 is evaluated in this section. Figure 5.3 shows a fragment of a C program and its equivalent MIPS-like assembly code of the inner loop. The figure on the right shows the penalties applied to the true dependencies according to the scheme listed in Table 5.1, and the overall penalty measure for this particular schedule. The penalty measure of a schedule is determined by the sum of the individual penalties, resulting in a total of 12 units for the schedule shown in Figure 5.3. The assembly code generated prior to the scheduling phase produces many penalties due to several consumer instructions being scheduled immediately after the producer. This is a common feature after performing code generation, and before instruction scheduling.

The relationship between the instructions of the loop core is displayed by the DAG in Figure 5.4. The solid lines connecting the nodes represent true

dependencies, while the dashed lines represent other dependencies. For instance, the dependency between instructions I_8 and I_{14} represents a WAR dependency, while the one between instructions I_{13} and I_{16} only specifies that I_{13} should be issued before I_{16}.

The core of the loop has been exhaustively scheduled to determine the one with the minimum penalty measure. Furthermore, every schedule has been simulated executing its instructions on the micronet architecture, to relate the overall penalty measure to the makespan. Even such a small example (16 instructions) has a considerable number of possible combinations — 1,567,742 valid schedules. Each one was simulated on a model of the micronet architecture with latency distributions from Table 4.1[2]. The makespans of the valid schedules were plotted as a function of the measure, as shown in Figure 5.5. Each point in the graph represents the simulated makespan of a schedule in nanoseconds.

The distribution of the makespans shown in the figure is characterised by an ascendent pattern: as the measure increases, so does the makespans of the schedules. Ideally the distribution should be a strict monotonic function, so that there would be no overlapping regions between neighbouring penalties. In practice however, the overlaps between the schedules of neighbouring penalties are tolerable for considering this measure as the basis for a heuristic for a scheduler for micronet-based asynchronous processors.

The schedule displayed in Figure 5.3 has a penalty measure of 12 which is close to the maximum penalty measure of 14. On the other hand, the minimum penalty measure for this group of instructions is 0. Figure 5.5 shows that the optimal schedule is indeed located in the section with the minimum penalty measure.

5.3 The Penalise True Dependencies (PTD) Scheduler

The PTD scheduler is a novel approach for performing local scheduling of instructions, that is based on minimising the penalty measure of a basic block. The penalty measure is used as a metric for statically categorising the goodness of a schedule.

The PTD scheduler differs from traditional mechanisms such as the list scheduler. The list scheduler constructs a list of ready instructions and selects the best candidate based upon heuristics (the list is initiated with instructions from the top-level of Figure 5.4 which do not have predecessors). Once an instruction is

[2]The configuration of the micronet architecture consisted of an arithmetic unit, a memory unit and a logic unit.

```
                                        li      $9,0
                                        li      $16,10
                                L5.main:
                                        muli    $123,$8,4
                                        la      $122,$29,32       1
                                        addu    $121,$122,$123
main() {                                muli    $127,$8,4
                                        la      $126,$29,32       1
 int i, j, n = 10;                      addu    $125,$126,$127    1
 int x[10];                             lw      $87,$125,0
                                        muli    $131,$9,4
  for (i = 0; i < n; i++)               la      $130,$29,32       1
    for (j = 0; j < n; j++)             addu    $129,$130,$131    1
      x[i] = x[i] * x[j];               lw      $92,$129,0        3
}                                       mul     $86,$87,$92       1
                                        sw      $86,$121,0
                                        addui   $9,$9,1           1
                                        slt     $136,$16,$9       2
                                        bt      $136, L5.main
                                L7.main:                          12
                                        addui   $8,$8,1
```

Figure 5.3: An example C-code and its inner loop assembly code equivalent.

removed, *i.e.* scheduled, its immediate successors become ready. This allows them to be inserted in the ready list for selection. The process of choosing an instruction and updating the ready list repeats itself until all the instructions of the basic block have been scheduled.

In contrast, the PTD scheduler analyses the schedule based on the PTD measure. If the measure returns a zero value, then the code is not modified. Otherwise, the scheduler traverses the schedule to evaluate optimisations on every instruction that is penalised. In order to reduce the penalty, the scheduler must find an independent and unrelated instruction to place in between the producer and the consumer instructions.

A schedule, as shown on the top of Figure 5.6, can be regarded as an "horizontal sequence" of instructions which are executed in order from left to right. When a penalised instruction is encountered, an independent instruction, and preferably an unrelated one at that, is searched on both sides of the penalised instruction starting from the left side of the penalised instruction, and if a *candid-*

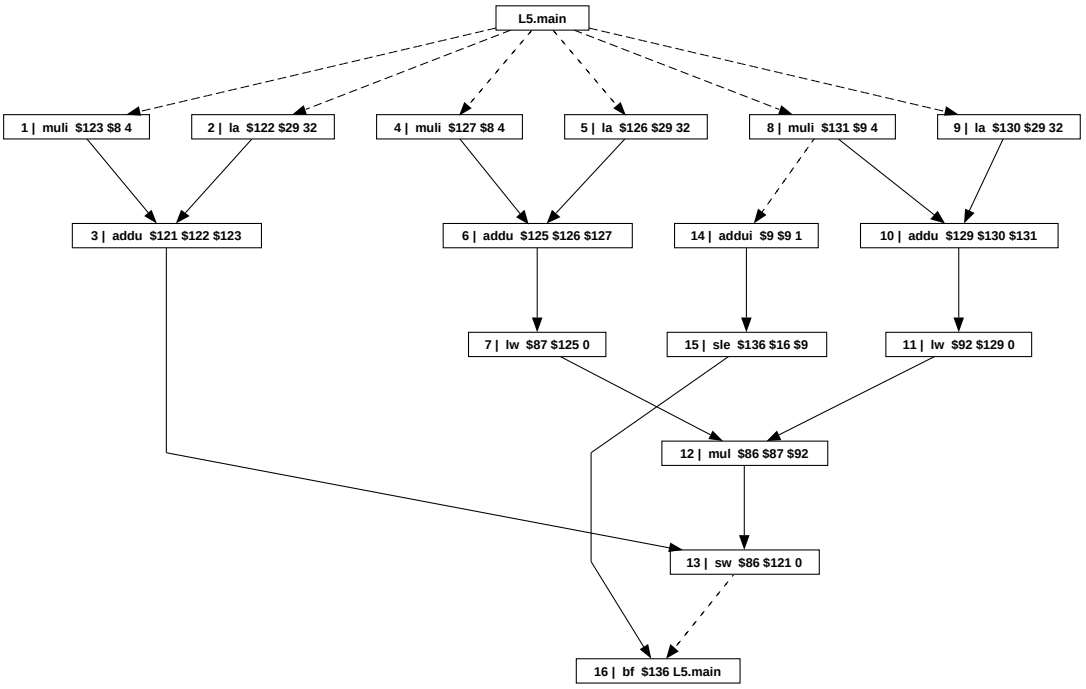


Figure 5.4: DAG of the core loop in Figure 5.3.

ate instruction cannot be found, then the search switches to the right side of the penalised instruction. The schedule in Figure 5.6 is representative of the DAG in Figure 5.4; Figure 5.6 (a) shows the traversal on the left side starting with the immediate neighbour instruction I_5, and Figure 5.6 (b) the traversal on the right side of the penalty.

There are two necessary conditions for an instruction to be considered as a candidate for movement ahead of the penalised instruction. Firstly, the instruction in question has to be independent of the penalised instruction, and secondly, the instruction has to be independent of all the instructions scheduled in between the candidate and the penalised instruction. These conditions are necessary to preserve the semantics of the code and are known as the *valid* conditions. They only allow valid movement of instructions in which the order of execution is preserved, but the performance of the outcome of such movements has to be analysed further. The *safety* conditions, *i.e.* the rules to ensure that the movement of an instruction improves the quality of the schedule are discussed in Section 5.3.2.

From Figure 5.6 (a), instruction I_5 fails to comply with the first condition since it is data-dependent on instruction I_6, and is therefore not a valid candidate. Similarly, instruction I_6 depends on instruction I_4, and, therefore, I_4 cannot

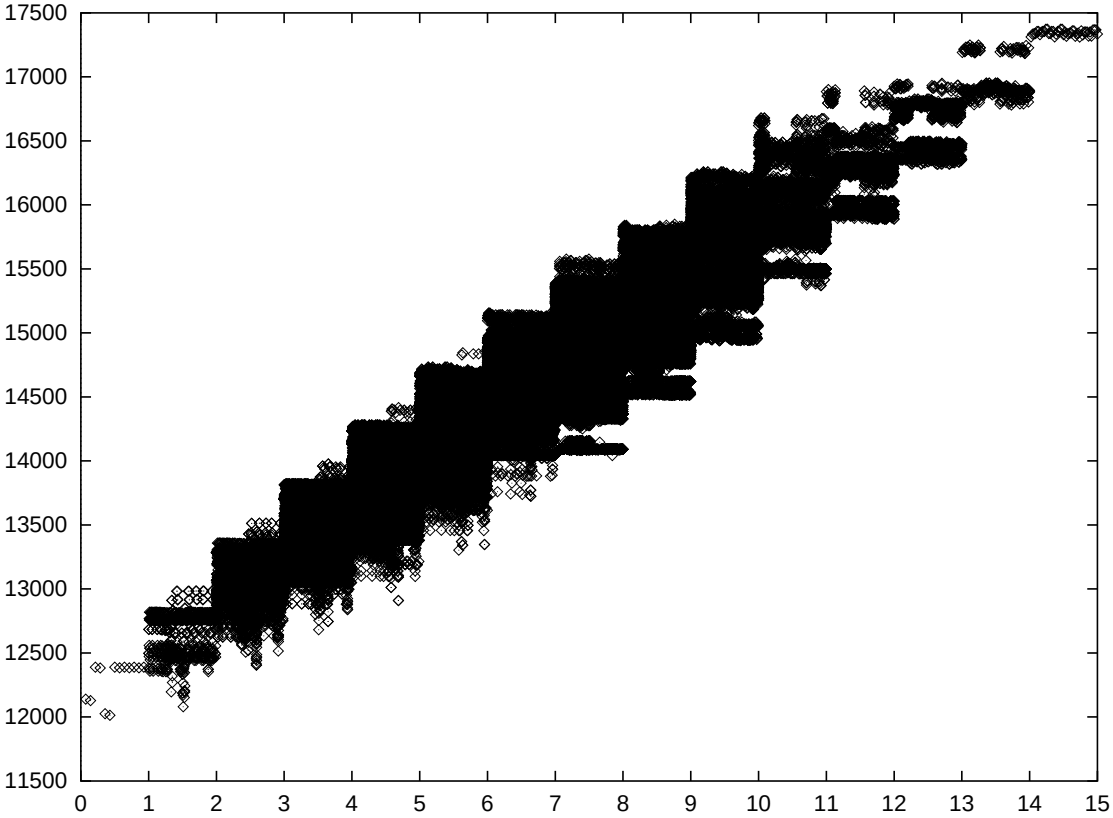
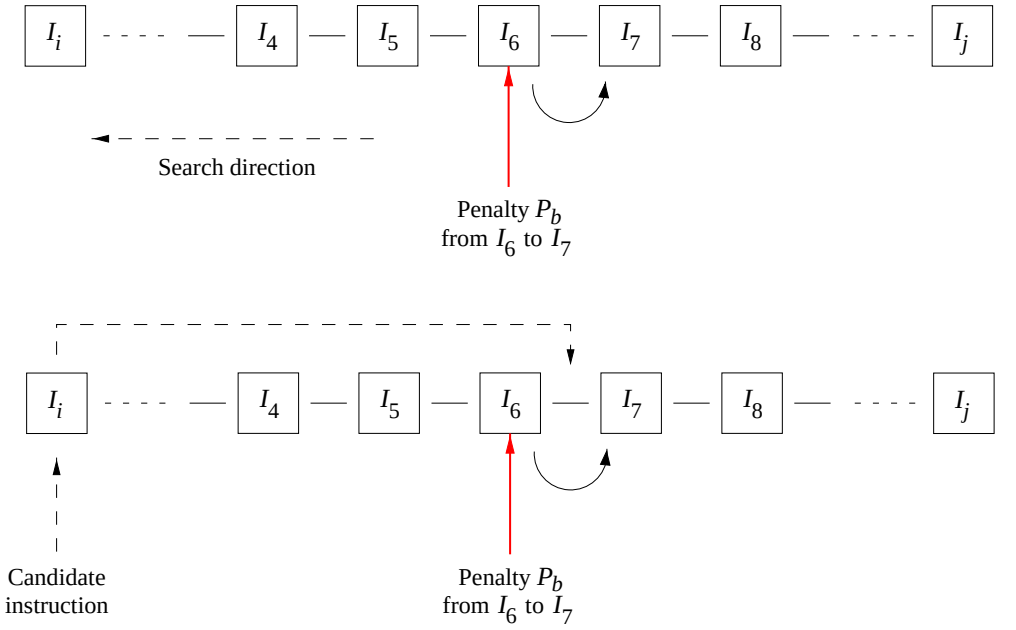


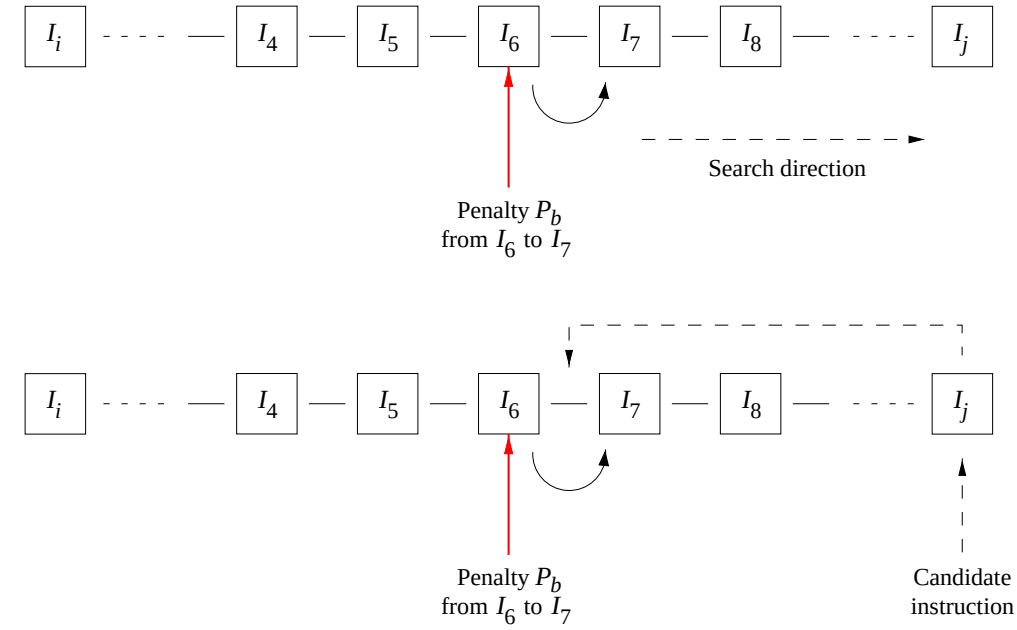
Figure 5.5: Makespans of the simulated schedules (y-axis) on a model of the micronet architecture against the penalty measure (x-axis).

be a candidate either. An instruction I_i is a candidate for movement, if it is independent of all the instructions from I_{i+1} to I_6. If an independent instruction cannot be found that would satisfy both conditions in the left side of the penalty, then the search is continued on the right side. From Figure 5.6 (b), the first instruction to be considered is instruction I_8 since instruction I_7 is dependent on the penalised instruction. The candidate instruction I_j must be independent of all instructions from I_7 to I_{j-1}.

The main reason for searching candidate instructions starting from the left hand side of the penalty is because there is a higher likelihood of finding one faster. The exit of the basic block, which is located at the right hand side, can be seen as a synchronisation point, so it may offer fewer options. Conversely, the entry of the basic block at the left hand side may have a wider scope for finding a candidate sooner.



(a) Left search for a valid candidate.



(b) Right search for a valid candidate.

Figure 5.6: Instruction search procedure to reduce penalties in the PTD scheduler.

A first pass over the basic block attempts to reduce penalties with the higher penalty values according to Table 5.1, which corresponds to data dependencies due to loads. The second pass reduces penalties due to branch instructions in the basic block, if it ends with such an instruction. And, in the final pass, the scheduler reduces the remaining penalties with a value one.

Such a priority scheme aims to reduce penalties with the higher delay-costs to the issue unit first. Even in the case when a basic block does not have enough parallelism to reduce all the penalties, the dependencies causing the higher stalls are more likely to be reduced. For the two penalties shown in Figure 5.2, the one due to the load instruction (I_6) is handled in the first pass, while the data dependency due to the other instruction (I_5) is reduced in a subsequent pass.

5.3.1 Extending the PTD Measure

Placing an independent instruction between two consecutive instructions that share a true dependency will certainly reduce the stall, but is not removed entirely. For some types of dependency, it may be necessary to place more than one instruction in between the pair. The penalty measure was extended so that dependent instructions at a distance of more than one instruction can also be penalised.

The delay-cost of true dependencies when there are independent instructions placed in between the producer and consumer instructions is lower than the delay-cost of consecutive true-dependent instructions. When two true-dependent instructions are scheduled consecutively, no other instruction can be issued so the amount of stall is considerable. On the other hand, if one or more instructions separate the pair of dependent instructions, then they can execute during the period when waiting for the result to be written by the producer instruction. The intervening instructions allow the issue unit to continue the issuing process, although the consumer instruction may still have to be stalled, albeit for a shorter period of time. The penalties applied to *non-consecutive* dependent instructions are therefore smaller than those for *consecutive* penalties as listed in Table 5.1.

The PTD scheduler makes a distinction between consecutive and non-consecutive penalties to prioritise the order of their reduction. Given the degree of stalls caused by consecutive dependencies, consecutive penalties are given a higher priority than non-consecutive ones. A first pass reduces consecutive penalties, while a second pass reduces the others. This is equivalent to improving the schedule in larger steps first, followed by finer optimisations.

5.3.2 Safety Conditions for Reducing Penalties

The quality of a schedule depends on safety conditions when reducing the penalties. When moving an instruction, it is necessary to check that doing so does not introduce another penalty to the instructions around the source, and to those around the destination. This is to guarantee that after each transformation the penalty measure is always reduced.

When a candidate instruction is found, a penalty comparison of the schedule before and after the movement is performed. The movement is permitted only if the quality of the schedule is improved, *i.e.* the penalty measure is reduced after the movement. The safety condition is defined by the following equation:

$$\sum P_{after} < \sum P_{before} \tag{5.1}$$

The above equation must hold for consecutive and non-consecutive penalties in their respective passes. Since consecutive and non-consecutive penalties cannot be compared because the stalls produced by them are different, the penalty measures must be treated separately. Equation 5.1 is split into two equations for consecutive and non-consecutive penalties, respectively

$$\sum P'_{after} < \sum P'_{before} \tag{5.2}$$

$$\sum P''_{after} < \sum P''_{before} \tag{5.3}$$

Equation 5.2 represents the behaviour of a greedy algorithm: the movement of a candidate instruction is permitted as long as it reduces the penalty measure. However, should the penalty measure from consecutive dependencies (P') stay constant after the potential movement, then that due to non-consecutive dependencies (P'') is used as a second criteria to allow the movement, if the penalty measure due to it is reduced. This case is represented by the following equation

$$\sum P'_{after} = \sum P'_{before} \quad \wedge \quad \sum P''_{after} < \sum P''_{before} \tag{5.4}$$

The penalty measurement for a specific movement is centered around the candidate and penalised instructions. This means that only the penalties around these instructions are involved in the analysis, and not the entire basic block.

5.3.3 Reduction of Resource Penalties

The candidate for movement is an instruction that not only has to comply with both the valid and safety conditions, but should also be of a different instruction type than the penalised instruction. The process of finding the best candidate may therefore require further search, since more conditions have to be met.

In order to reduce the extensive restriction when reducing the penalties, the scheduler has been partitioned to reduce penalties due to resource contentions first and penalties due to data dependencies after.

The reduction of the penalties due to resource contentions is a similar process to that explained in Section 5.3. The difference being that the safety conditions for the candidate instruction only require to address the difference of the instruction types.

5.4 The PTD Scheduler Algorithm

The PTD scheduler has been partitioned into three phases. The first phase operates on resources penalties, the second phase manipulates penalties from consecutive dependencies and the third phase deals with penalties from non-consecutive dependencies. The decision to reduce the resource penalties first is not only to reduce the stalls produced by resource contentions, but also to introduce a certain degree of 'randomness' to the other scheduling phases. This is explained in more detail in Section 5.7.2. Algorithm 5.1 shows the top-level structure of the PTD scheduler.

Algorithm 5.1 ***PTD_scheduler (entry)*** algorithm.

```
root = first_instruction (entry)

PTD_resource_phase (root)
PTD_consecutive_phase (root)
PTD_nonconsecutive_phase (root)
```

The structure of the three functions in Algorithm 5.1 is similar, but they call different routines with different parameters to perform the movement of the instructions. These functions are shown in Algorithms 5.2, 5.3 and 5.4. The functions receive as their operand the root instruction of the basic block. The first step is to compute the PTD measure (lines 1); if the measure is positive,

then the core of the scheduling process is repeated. Since it is not possible to know *a priori* the value of the minimum penalty measure which would imply a knowledge of the optimal schedule, the algorithm must endeavour to reduce any penalty. This process is repeated as long as the penalty measure is reduced. Conversely, if the measure stays constant after a pass, then it is assumed that there are no more reductions possible, and the loop is terminated. The decision to stop the algorithm when no further reductions of the PTD measure can be made ensures the termination of the algorithm. This is represented by lines `15`, `16`, `29` and `30` in the three algorithms, together with lines `43` and `44` in Algorithms 5.3 and 5.4.

The difference between these algorithms is based on the type of penalties that are being reduced. Algorithm 5.2 calls functions *PTD_arrange_resource*, whilst Algorithms 5.3 and 5.4, call the function *PTD_arrange_data*. These functions are responsible for performing the instruction movements while respecting all data dependencies. The *PTD_arrange_resource* routine is restricted to move instructions that reduce resource penalties, while the *PTD_arrange_data* routine has the responsibility of reducing penalties due to data dependencies.

The difference between functions *PTD_consecutive_phase* (Algorithm 5.3) and *PTD_nonconsecutive_phase* (Algorithm 5.4) is the way in which the second parameter of the function *PTD_measure* is specified. The parameter is used to differentiate the number of neighbouring instructions which are checked around the candidate and penalised instructions. The function *PTD_measure* not only computes the penalty measure of a schedule, but penalises the instructions as well, according to its type, the distance between the producer and consumer instructions, and the scheduling phase.

Algorithms 5.2, 5.3 and 5.4 are characterised by a series of `repeat` loops (lines `3`, `17` and `31`). In these inner loops the two functions *PTD_arrange_left* and *PTD_arrange_right* are responsible for the instruction movements to reduce the penalties on either sides — the right hand side is called if the left hand side search cannot perform a reduction, as explained in Section 5.3. The different loops are targeted at different penalties, in order to prioritise them. The order of the loops corresponds to the degree of the penalty; higher penalties such as those from load instructions are targeted first, while instructions with lower penalties are treated last.

The functions *PTD_arrange_left_data* and *PTD_arrange_right_data* that perform the instruction movements are displayed in Algorithms 5.5 and 5.6 respectively. Both functions receive the penalised instruction as their parameter. An

Algorithm 5.2 ***PTD_resource_phase (root)*** algorithm.

```
1: measure = PTD_measure (root, resource_phase)
2: if measure > 0 then
3:   repeat
4:     node = root
5:     last_measure = measure
6:     while node ≠ NULL do
7:       if penalty_resource (node) == 3 then {Penalties memory inst.}
8:         PTD_arrange_left_resource (node)
9:       end if
10:      if penalty_resource (node) == 3 then {Penalties memory inst.}
11:        PTD_arrange_right_resource (node)
12:      end if
13:      node = next(node)
14:    end while
15:    measure = PTD_measure (root, resource_phase)
16:  until measure == last_measure
17:  repeat
18:    node = root
19:    last_measure = measure
20:    while node ≠ NULL do
21:      if penalty_resource (node) == 1 then {Penalties other types.}
22:        PTD_arrange_left_resource (node)
23:      end if
24:      if penalty_resource (node) == 1 then {Penalties other types.}
25:        PTD_arrange_right_resource (node)
26:      end if
27:      node = next(node)
28:    end while
29:    measure = PTD_measure (root, resource_phase)
30:  until measure == last_measure
31: end if
```

Algorithm 5.3 ***PTD_consecutive_phase (root)*** algorithm.

```
1:  measure = PTD_measure (root, first_phase)
2:  if measure > 0 then
3:    repeat
4:      node = root
5:      last_measure = measure
6:      while node ≠ NULL do
7:        if penalty_consecutive (node) == 3 then {PENALTIES LOAD INST.}
8:          PTD_arrange_left_data (node)
9:        end if
10:       if penalty_consecutive (node) == 3 then {PENALTIES LOAD INST.}
11:         PTD_arrange_right_data (node)
12:       end if
13:       node = next(node)
14:     end while
15:     measure = PTD_measure (root, first_phase)
16:   until measure == last_measure
17:   repeat
18:     node = root
19:     last_measure = measure
20:     while node ≠ NULL do
21:       if penalty_consecutive(node) == 2 then {PENALTIES BRANCH INST.}
22:         PTD_arrange_left_data (node)
23:       end if
24:       if penalty_consecutive(node) == 2 then {PENALTIES BRANCH INST.}
25:         PTD_arrange_right_data (node)
26:       end if
27:       node = next(node)
28:     end while
29:     measure = PTD_measure (root, first_phase)
30:   until measure == last_measure
31:   repeat
32:     node = root
33:     last_measure = measure
34:     while node ≠ NULL do
35:       if penalty_consecutive (node) == 1 then {PENALTIES OTHER INST.}
36:         PTD_arrange_left_data (node)
37:       end if
38:       if penalty_consecutive (node) == 1 then {PENALTIES OTHER INST.}
39:         PTD_arrange_right_data (node)
40:       end if
41:       node = next(node)
42:     end while
43:     measure = PTD_measure (root, first_phase)
44:   until measure == last_measure
45: end if
```

Algorithm 5.4 ***PTD_nonconsecutive_phase (root)*** algorithm.

```
1:  measure = PTD_measure (root, second_phase)
2:  if measure > 0 then
3:    repeat
4:      node = root
5:      last_measure = measure
6:      while node ≠ NULL do
7:        if penalty_nonconsecutive (node) == 3 then {DISTANCED 1 INST.}
8:          PTD_arrange_left_data (node)
9:        end if
10:       if penalty_nonconsecutive (node) == 3 then {DISTANCED 1 INST.}
11:         PTD_arrange_right_data (node)
12:       end if
13:       node = next(node)
14:     end while
15:     measure = PTD_measure (root, second_phase)
16:   until measure == last_measure
17:   repeat
18:     node = root
19:     last_measure = measure
20:     while node ≠ NULL do
21:       if penalty_nonconsecutive (node) == 2 then {DISTANCED 2 INST.}
22:         PTD_arrange_left_data (node)
23:       end if
24:       if penalty_nonconsecutive (node) == 2 then {DISTANCED 2 INST.}
25:         PTD_arrange_right_data (node)
26:       end if
27:       node = next(node)
28:     end while
29:     measure = PTD_measure (root, second_phase)
30:   until measure == last_measure
31:   repeat
32:     node = root
33:     last_measure = measure
34:     while node ≠ NULL do
35:       if penalty_nonconsecutive (node) == 1 then {DISTANCED 3 INST.}
36:         PTD_arrange_left_data (node)
37:       end if
38:       if penalty_nonconsecutive (node) == 1 then {DISTANCED 3 INST.}
39:         PTD_arrange_right_data (node)
40:       end if
41:       node = next(node)
42:     end while
43:     measure = PTD_measure (root, second_phase)
44:   until measure == last_measure
45: end if
```

auxiliary pointer, `aux1`, is used to identify candidate instructions. It traverses the schedule, starting from the penalised instruction towards the entry of the schedule, in the case of the *PTD_arrange_left* algorithm, or towards the exit of the schedule in the *PTD_arrange_right* algorithm. A second auxiliary pointer, `aux2`, traverses the schedule in the opposite direction towards the penalised instruction.

Since the auxiliary pointer, `aux1`, starts its route from the neighbour of the penalised instruction, there is a possibility that the neighbour instruction is a candidate itself. The `if` statement in line `7` in both the *PTD_arrange* algorithms evaluates the condition of the neighbouring instruction. The *independent* function determines whether the neighbouring instruction has dependencies with the penalised instruction (valid condition). The dependency check is performed using an n `x` n matrix of relationship between the instructions of a basic block. The *check_left_swap* and *check_right_swap* functions determine whether the movement of the neighbouring instruction reduces the penalty measure of the basic block (safety condition). The purpose of the function *check_subgraphs* (located in lines `7` and `18`) is explained in more detail in Section 5.5.2.

If the conditions are met, then the candidate and the penalised instructions are swapped and the penalties around these instructions are updated (lines `8`, `9` and `10`). Otherwise, `aux1` is required to search for a candidate instruction further away from the penalised instruction which is performed within the `while` loop, starting at lines `12`, in Algorithms 5.5 and 5.6.

Pointer `aux2` is used to check the dependency (valid conditions) within the inner `while` loop located between lines `14` and `17`. The variable `ind_nodes` holds the collective status of the candidate being checked against all the instructions scheduled up to the penalised instruction. The `while` loop is aborted as soon as one of the instructions is found to be dependent on the candidate instruction, or `aux2` reaches the penalised instruction `node`; otherwise, it advances to the next instruction.

If the candidate is independent of all the instructions scheduled between itself and the penalised instruction, then its safety condition is checked. The function, *check_local_move*, in lines `21` in both algorithms performs the penalty analysis. If there is a reduction, then the movement is allowed and the candidate instruction is moved (line `28`). Similarly, the penalties are updated and the penalised instruction is cleared. In the case of a consecutive penalty from a load instruction, the penalty is not fully cleared but reduced from 3 units to 1 unit, according to Table 5.1.

The functions, *PTD_arrange_left_resource* and *PTD_arrange_right_resource*, which are omitted for brevity, have the same structure as the functions shown in Algorithms 5.5 and 5.6. The differences lie in the way the safety conditions are checked when calling the functions *check_left_swap*, *check_right_swap* and *check_local_move*.

The routine *check_left_swap* in Algorithm 5.7 evaluates the state of the penalty measures before and after the movement. The variables *before_swap* and *after_swap* hold the penalty values before and after the instruction movement. The variables *consecutive* and *nonconsecutive* hold the penalty values of the first criteria, *i.e.* the value of the consecutive penalties, and the second criteria, respectively.

The function *penalty_data(inst1, inst2, closeness)* used in lines `1` to `4` returns the value of the penalty between two instructions *inst1* and *inst2*. The function treats the instruction *inst1* as being scheduled before *inst2*. The parameter *closeness* is used to define the distance between *inst1* and *inst2*. The penalty returned by the function depends on whether the instructions are consecutive or not, and null is returned if the instructions are independent.

As explained in Section 5.3.2, a safe movement is one which strictly reduces the penalty measure. Line `5` of Algorithm 5.7 compares the two consecutive measures: if the penalty measure is reduced then the swap is permitted; if equal (line `8`), then the non-consecutive comparison is considered. If the second criteria is smaller after the movement (line `9`), then the swap proceeds.

The function *check_right_swap* has the same behaviour and the same structure as the *check_left_swap*, but has been omitted for the sake of brevity.

The other algorithm for performing instruction movements is shown in Algorithm 5.8: the function *check_local_move* has a similar construction to the *check_swap* functions, but requires more consideration. The first `if`-section (lines `1-4`) specifies the penalty conditions before and after the movement, when the candidate is positioned at the entry of the basic block; the second one (lines `5-8`) specifies the opposite case, when the candidate is positioned at the exit of the basic block, and the last (lines `9-12`) deals with any other case. The variables *before_move* and *after_move* hold the state of the penalties before and after.

Algorithm 5.5 ***PTD_arrange_left_data (node)*** algorithm.

```
1: aux1 = prev(node)
2: ind_nodes = 1
3: cond_out = 0
4: if aux1 ≠ NULL then
5:    aux1 = prev(aux1)
6: end if
7: if independent (prev(node), node) and
       check_left_swap (node, aux1) and
       check_subgraphs (node, aux1) then
8:    update_node (node)
9:    update_node (aux1)
10:   swap (prev(node), node)  {SWAPS THE PREVIOUS INSTRUCTION.}
11: else
12:   while aux1 ≠ NULL and cond_out == 0 do
13:      aux2 = next(aux1)
14:      while aux2 ≠ next(node) and ind_nodes ≠ 0 do
15:         ind_nodes = ind_nodes and independent (aux1, aux2)
16:         aux2 = next(aux2)
17:      end while
18:      if ind_nodes ≠ 0 and check_subgraphs (node, aux1) then
19:         ind_nodes = 0
20:      end if
21:      if ind_nodes ≠ 0 and check_local_move (node, aux1) then
22:         ind_nodes = 0
23:      end if
24:      if ind_nodes ≠ 0 then
25:         cond_out = 1
26:         update_node (node)
27:         update_node (aux1)
28:         move_ahead (aux1, node)  {MOVES THE CANDIDATE INSTRUCTION.}
29:      end if
30:      ind_nodes = 1
31:      aux1 = prev (aux1)
32:   end while
33: end if
```

Algorithm 5.6 ***PTD_arrange_right_data (node)*** algorithm.

```
1:  aux1 = next(next(node))
2:  ind_nodes = 1
3:  cond_out = 0
4:  if aux1 ≠ NULL then
5:    aux1 = next(aux1)
6:  end if
7:  if independent (next(node), next(next(node))) and
        check_right_swap (node, aux1) and
        check_subgraphs (node, aux1) then
8:    update_node (node)
9:    update_node (next(node))
10:   swap (next(node), next(next(node)) {SWAPS THE FOLLOWING TWO INST.}
11: else
12:   while aux1 ≠ NULL and cond_out == 0 do
13:     aux2 = prev(aux1)
14:     while aux2 ≠ node and ind_nodes ≠ 0 do
15:       ind_nodes = ind_nodes and independent (aux1, aux2)
16:       aux2 = prev(aux2)
17:     end while
18:     if ind_nodes ≠ 0 and check_subgraphs (node, aux1) then
19:       ind_nodes = 0
20:     end if
21:     if ind_nodes ≠ 0 and check_local_move (node, aux1) then
22:       ind_nodes = 0
23:     end if
24:     if ind_nodes ≠ 0 then
25:       cond_out = 1
26:       update_node (node)
27:       update_node (aux1)
28:       move_ahead (aux1, node) {MOVES THE CANDIDATE INSTRUCTION.}
29:     end if
30:     ind_nodes = 1
31:     aux1 = next(aux1)
32:   end while
33: end if
```

Algorithm 5.7 ***check_left_swap (node, aux)*** algorithm.

```
1: consecutive_before_swap =
       penalty_data (aux, next(aux), consec) +
       penalty_data (aux, node, consec) +
       penalty_data (node, next(node), consec) +
       penalty_data (node, next(next(node)), consec)
2: consecutive_after_swap =
       penalty_data (aux, node, consec) +
       penalty_data (aux, next(aux), consec) +
       penalty_data (next(aux), next(node), consec) +
       penalty_data (next(aux), next(next(node)), consec)
3: nonconsecutive_before_swap =
       penalty_data (prev(aux), next(aux), not consec) +
       penalty_data (prev(aux), node, not consec) +
       penalty_data (aux, node, not consec) +
       penalty_data (next(aux), next(node), not consec) +
       penalty_data (next(aux), next(next(node)), not consec) +
       penalty_data (node, next(next(node)), not consec)
4: nonconsecutive_after_swap =
       penalty_data (prev(aux), node, not consec) +
       penalty_data (prev(aux), next(aux), not consec) +
       penalty_data (aux, next(aux), not consec) +
       penalty_data (node, next(node), not consec) +
       penalty_data (node, next(next(node)), not consec) +
       penalty_data (next(aux), next(next(node)), not consec)

5: if consecutive_after_swap < consecutive_before_swap then
6:   return 1 {First criteria is smaller after the movement.}
7: else
8:   if consecutive_after_swap = consecutive_before_swap then
9:     if nonconsecutive_after_swap < nonconsecutive_before_swap then
10:      return 1 {Second criteria is smaller after the movement.}
11:    else
12:      return 0 {Second criteria is greater after the movement.}
13:    end if
14:  else
15:    return 0 {First criteria is greater after the movement.}
16:  end if
17: end if
```

Algorithm 5.8 ***check_local_move (node, aux)*** algorithm.

```
1: before_move =
       penalty_data (prev(prev(aux)), aux, not consec) +
       penalty_data (prev(aux), aux, consec) +
       penalty_data (prev(aux), next(aux), not consec) +
       penalty_data (aux, next(aux), consec) +
       penalty_data (aux, next(next(aux)), not consec) +
       penalty_data (prev(node), next(node), not consec) +
       penalty_data (node, next(node), consec) +
       penalty_data (node, next(next(node)), not consec)
2: after_move =
       penalty_data (prev(prev(aux)), next(aux), not consec) +
       penalty_data (prev(aux), next(aux), consec) +
       penalty_data (prev(aux), next(next(aux)), not consec) +
       penalty_data (prev(node), aux, not consec) +
       penalty_data (node, aux, consec) +
       penalty_data (node, next(node), not consec) +
       penalty_data (aux, next(node), consec) +
       penalty_data (aux, next(next(node)), not consec)

3: if after_move < before_move then
4:   return 1 {PENALTY MEASURE IS SMALLER AFTER THE MOVEMENT.}
5: else
6:   if after_move = before_move then
7:     return 0 {PENALTY MEASURE IS EQUAL AFTER THE MOVEMENT.}
8:   else
9:     return -1 {PENALTY MEASURE IS GREATER AFTER THE MOVEMENT.}
10:  end if
11: end if
```

5.5 Additional Concepts in the PTD Scheduler

5.5.1 Static Memory Disambiguation

Memory instructions restrict the available parallelism as they may introduce implicit data dependencies through memory locations. A data dependency exists when two memory instructions refer to the same location. For instance, a load cannot be scheduled ahead of a store if they both refer to the same address, in the case when the store is scheduled first. Similarly, a store cannot be moved ahead of a load if they refer to the same memory address, if the load is scheduled ahead of the store. Two stores have to be scheduled in order, if they refer to the same address.

The process of determining if two memory instructions access the same memory location is called *memory disambiguation*. Memory disambiguation can be implemented either at run-time [41][53][123], called dynamic memory disambiguation, or at compile-time [31][105], called static memory disambiguation, or a combination of both [60].

Dynamic memory disambiguation schemes keep track of the memory instructions in the order in which they are decoded. When a memory instruction is to be issued, its address must be compared to the addresses of all previously decoded memory operations, to check whether the address has been referenced. However, as the number of entries grow, the hardware would become slow and complex.

Static memory disambiguation on the other hand, has the flexibility of affording more aggressive algorithms to disambiguate the memory references. The memory disambiguation problem is often related to a form of integer linear programming, *i.e.* of finding an integer solution to a set of linear equalities. This process can be implemented at different levels in the compiling framework. At a higher level for example, when performing the data dependence analysis, memory references need to be disambiguated in order to apply high-level optimisations and transformations [115][133]. At the back-end of the compiler, memory disambiguation is required to identify more independent instructions, and thereby increase parallelism.

Ideally, the scheduler should discard all the data dependencies from the memory instructions that do not share the same address. The problem with static memory disambiguation is that it is not always possible to disambiguate memory references. When the scheduler cannot determine whether two memory instructions refer to the same location in memory, the memory instructions have to be assigned a data dependency in order to enforce the order of execution.

Static memory disambiguation mechanisms analyse the address expression of memory instructions. Memory instructions are characterised by having two operands: the data operand which holds the value that is to be loaded or stored, and the address operand, which holds the address in the memory where the data is to be loaded or stored. The address expression depends on the nature of the addressing mode. The following list describes the most common addressing modes [76].

Immediate. `Mem[` a `]`

The constant a represents the exact address of the store or the load.

Register. `Mem[` R_1 `]`

R_1 specifies the value of the address — R_1 "points" to that memory location.

Displacement. `Mem[` $R_1 + a$ `]`

R_1 specifies a base address and a is an offset to that base address. It is used when indexing arrays, *e.g.* `x[i + 1]`.

Indexed. `Mem[` $R_1 + R_2$ `]`

The address is the result of the addition of the contents of registers R_1 and R_2, in a similar way to the displacement mode.

Indirect. `Mem[Mem[` R_1 `]]`

The memory location pointed by R_1 contains the actual memory location being referred. This addressing mode is found in the case of double indexing, *e.g.* `x[y[i]]`.

For most of these addressing modes, the address expression is transformed to one consisting of a base address and an offset. Only in the immediate addressing mode is the address transformed into a constant value reference. Memory instructions at the assembly level use one register to hold the base address and another to hold the offset value. In the immediate addressing mode, only one register is used to hold the immediate value representing the address.

The memory instructions from the code in Figure 5.3 can be used as an example to describe how the memory references are compared. The address of the the memory operations is expanded in order to get the memory expression. The three memory instructions and their memory expressions are shown below:

```
(1)  lw $87,$125,0      lw $87,($29 + 32) + ($8 * 4) + 0
(2)  lw $92,$129,0      lw $92,($29 + 32) + ($9 * 4) + 0
(3)  sw $86,$121,0      sw $86,($29 + 32) + ($8 * 4) + 0
```

From the memory expressions, it is clear that instruction (1) and instruction (3) are dependent since they have the same expression (`($29 + 32) + ($8 * 4) + 0`). However, the disambiguation from instruction (2) against (3), refers to solving the equation `$8 = $9`. In such cases, the scheduler assumes a conservative approach and assigns a dependency, since the values of registers `$8` and `$9` are not known at compile-time, but may have the same value at some point during execution.

The static memory disambiguation scheme which was incorporated into the PTD scheduler is shown in Table 5.2. The table displays all the cases with memory instructions for both the base-offset and immediate types. The first two columns show the different combinations of address expressions, for a pair of memory instructions[3]. The third column shows the resultant equalities of the address expressions from both instructions. In the table, x and y represent registers, which are considered as variables; a and b represent constants with dissimilar values and are not zero. The last column shows the outcome of the memory disambiguation scheme for all the possible cases.

There are three possible outcome of two memory references: the first outcome is when the memory instructions have the same address expression, in which case there is a data dependency irrespective of the values of the registers and the type of the expression. The second outcome is when the memory references can be disambiguated, *i.e.* the memory address is not the same. In these cases, the address expressions can be disambiguated because they differ by a constant value. Constant values do not change during execution, so if the register used as the base address is the same, and the offset constants differ, then it can be safely assumed that the address is not the same. Thus, the memory disambiguation mechanism does not apply a data dependency to the instructions. The third outcome is when the memory addresses cannot be disambiguated because different registers are part of the expression, as in the example above. This case is generalised, when more than two registers are involved in either of the address expressions.

[3]If both memory references are load instructions, it implies that there is no dependency between them.

Instruction 1	Instruction 2	Equality	Outcome
x	x	$x = x$	Data dependency
x	y	$x = y$	Not disambiguated
x	a	$x = a$	Not disambiguated
a	a	$a = a$	Data dependency
a	b	$a \neq b$	Disambiguated
x	$x + a$	$x \neq x + a$	Disambiguated
$x + a$	$x + a$	$x + a = x + a$	Data dependency
$x + a$	$x + b$	$x + a \neq x + b$	Disambiguated
$x + a$	$y + a$	$x + a = y + a$	Not disambiguated
$x + \ldots$	$y + \ldots$	$x + \ldots = y + \ldots$	Not disambiguated

Table 5.2: Static memory disambiguation scheme for the PTD scheduler.

This is represented in the last case of the table. The results and statistics from the memory disambiguation mechanism used for the PTD scheduler are shown in Section 7.4.

5.5.2 Subgraphs

The PTD scheduler searches for candidate instructions starting from the instruction neighbouring the penalised one, which reduces the average search time. However, this can lead to a mix of instructions such that no further reductions can be made. Figure 5.7 shows a list of instructions before and after scheduling. Figure 5.7 (a) depicts a schedule with two penalties due to consecutive data dependencies. Figure 5.7 (b) shows the same code after the first scheduling phase. The penalties were reduced by swapping instructions I_2 and I_3. The resulting penalties become non-consecutive.

If no independent instruction can be moved to reduce these penalties during the second scheduling phase, then they might never be removed. For instance, instructions I_2 and I_3 cannot be moved away since they reduce the penalties due to instructions I_3 and I_1. If instruction I_2 is moved further away from instruction I_1, then the non-consecutive penalty due to I_3 becomes a consecutive one. Similarly, if instruction I_3 is moved further away from I_4, then the penalty due to I_1 becomes consecutive. This is thanks to the safety conditions explained earlier. Figure 5.7 (b) shows that overlapping penalties may produce under-optimised code.

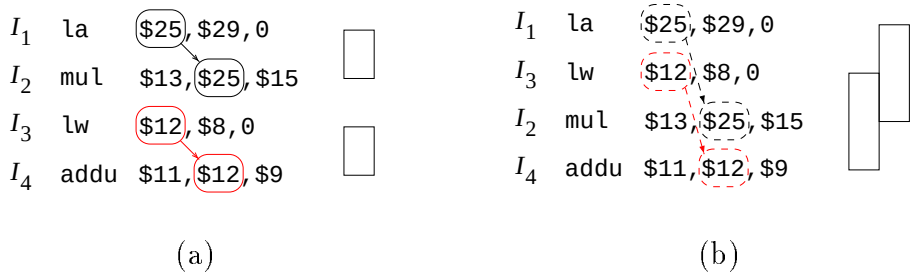


Figure 5.7: An example of overlapping penalties.

A solution to this problem is to mask the scope for searching candidate instructions within the basic block. A basic block can be regarded as a group of instructions that are related in an ordered way to perform a computation. The basic block can be divided into subcomponents (subgraphs) that perform part of the overall computation. For example, two separate subgraphs can be identified in a memory operation. The first one involves the computation of the address, and the second one involves the computation of the actual data that is to be loaded from or stored at that address. This can be seen in the store instruction (I_{13} : `sw $86,$121,0`) in the code example in Figure 5.3. Figure 5.8 displays the *DAG* from Figure 5.4 which has been decomposed into three subgraphs: the address is computed in the first subgraph (A), formed by instructions I_1, I_2 and I_3; the second subgraph (B) is dedicated to computing the data, as formed by instructions I_4 to I_{12}; and the third subgraph (C) is responsible for updating the array index which is checked by the branch instructions I_{15} and I_{16}. The node numbering reflects the same order as in the schedule in Figure 5.3 and the highlighted arcs represent the penalties from consecutive data dependencies.

Without the subgraphs, the PTD scheduler might reduce the consecutive penalties due to instructions I_9, I_{10} and I_{11} with instructions from the group I_4, I_5, I_6 and I_7. These movements may reduce the penalties due to instructions I_5 and I_6 as well, at the expense of creating overlapping penalties that will be difficult to reduce in subsequent scheduling phases.

The subgraph B forbids the selection of a candidate instruction that belongs to the same subgraph, when reducing penalties. The candidates can only be selected from the other two subgraphs.

The selection and size of the subgraphs is particularly important. If the size of the subgraph is too small, then there is a greater likelihood of producing

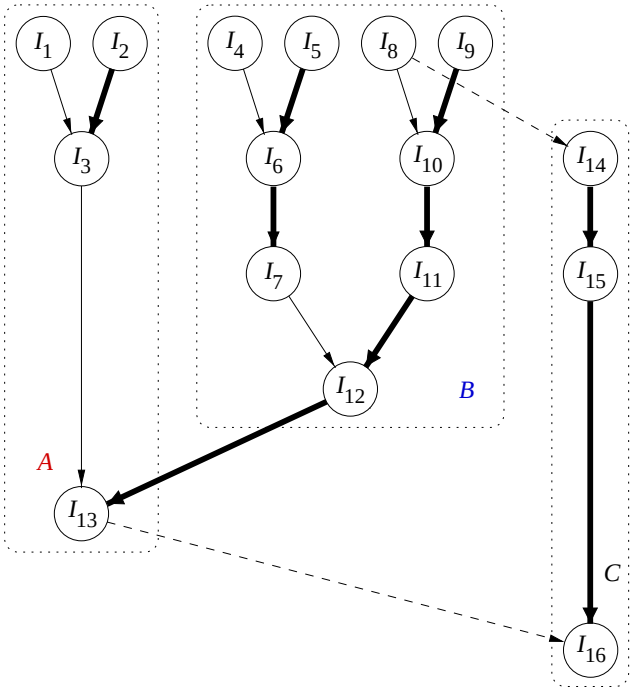


Figure 5.8: Basic block from Figure 5.4 decomposed into subgraphs.

overlapping penalties. If the size of the subgraph is too large, then it becomes more difficult to find a suitable candidate. The granularity of the subgraphs plays an important rôle for the scheduler. This can be exemplified in the *DAG* example in Figure 5.8. If the subgraph B is divided into two subgraphs, one formed by instructions I_4, I_5, I_6 and I_7, and the other formed by instructions I_8, I_9, I_{10}, I_{11} and I_{12}, then the scheduler will try to intermix their penalties so that overlapping penalties will occur.

The mechanism for selecting subgraphs defines the whole basic block as one subgraph and recursively divides it into several subgraphs. The subgraph separation depends on the *relative parallelism* of each instruction. The relative parallelism of an instruction is defined as the number of predecessors divided by the *level* of the instruction, terms that are defined by true data dependencies. The level represents the critical path relative to the instruction. This ratio suggests that for a bigger value, there are more instructions per level (less sequentiality), which can be considered as a measure of concurrency: greater the value of this measure, bigger is the subgraph. Since the algorithm starts with a subgraph the size of the basic block, this metric defines which instruction must delimit a new subgraph.

Algorithm 5.9 ***divide_subgraph (node, pred, size)*** algorithm.

```
1: parallel = (float) predecessors (pred) / (float) level (pred)
                                  {LEVEL(PRED) IS EQUAL TO LEVEL(NODE) - 1}
2: if size < 23 then
3:   if predecessors (node) ⩾ 6 and parallel > 1.5 then
4:     create_subgraph (node) {INST. SEPARATED INTO ANOTHER SUBGRAPH.}
5:   end if
6: else
7:   if predecessors (node) ⩾ 10 and parallel > 2.0 then
8:     create_subgraph (node) {INST. SEPARATED INTO ANOTHER SUBGRAPH.}
9:   end if
10: end if
```

Algorithm 5.9 shows the function *divide_subgraph* that separates the basic block into subgraphs. The routine is called for every predecessor of an instruction in the basic block. The function receives the instruction, its predecessor and the size of the basic block. Line 1 of the algorithm computes the relative parallelism of the predecessor. Line 2 selects which ratio value has to be selected: basic blocks with a small number of instructions require a lower ratio threshold, otherwise, the algorithm might not divide the subgraph. A basic block with a size lower than 23 instructions creates a new subgraph if the ratio is greater than 1.5 (line 3); if the basic block has a size bigger than 23, then the ratio must be greater than 2 (line 7). These values have been customised through experimentation. The rest of the condition of lines 3 and 7 specifies that a minimum number of predecessors are required, otherwise it will not be necessary to identify a new subgraph.

For instruction I_{13} of Figure 5.8 the *parallel* variable holds a value of 2.25 (9 predecessors with a level of 4) corresponding to the arc of predecessor I_{12}, and 1.5 (3 predecessors with a level of 2) corresponding to the arc of predecessor I_3. The ratio of the predecessor I_{12} is adequate to consider a subgraph, while the ratio of the predecessor I_3 is not sufficient.

Figure 5.9 shows the scheduled output of the PTD scheduler for the code in Figure 5.3. The label marks beside the instructions illustrate to which subgraph each instruction belongs. The instructions are scheduled in such a way that the label marks are mixed as if they were different threads. Masking the instructions with the subgraphs method helps to achieve this.

The figure also shows the PTD measure after scheduling the code which corresponds to the minimum measure that this code exhibits. The makespan of this

```
L5.main:
   B    muli    $127,$8,4
   B    la      $126,$29,32
   B    muli    $131,$9,4
   B    la      $130,$29,32
   B    addu    $125,$126,$127
   A    la      $122,$29,32
   B    addu    $129,$130,$131
   B    lw      $87,$125,0
   A    muli    $123,$8,4
   B    lw      $92,$129,0
   C    addui   $9,$9,1
   A    addu    $121,$122,$123
   B    mul     $86,$87,$92
   C    slt     $136,$16,$9
   A    sw      $86,$121,0
   C    bt      $136, L5.main
L7.main:                          0
```

Figure 5.9: Code example in Figure 5.3 after scheduled by the PTD scheduler.

particular schedule is 12085.067 ns is one which is among the best schedules in the makespan distribution in Figure 5.5.

The subgraphs checking between the candidate and the penalised instruction is performed by function *check_subgraphs* in lines 7 and 18 in Algorithms 5.5 and 5.6. The function performs the checking in a way similar to the dependency checking: a n x n matrix relates the different subgraphs for every instruction. The function returns null if two instructions belong to the same subgraph, otherwise, a one is returned.

The order of checking the instruction in these functions is as follows: data dependency check, subgraph check and safety condition check. The instruction is moved if it satisfies all these conditions.

5.6 Algorithmic Complexity

The structure of the PTD scheduler is different from traditional techniques because the algorithm is driven by the penalties in the code. The complexity of the PTD scheduler is derived next.

The `while` sections in the functions *PTD_resource_phase* (Algorithm 5.2), *PTD_consecutive_phase* (Algorithm 5.3) and *PTD_nonconsecutive_phase* (Algorithm 5.4) traverse the basic block stopping at every penalty. These sections have a complexity of $\theta\left(et^2 + (n-e)\right)$, where n is the number of instructions in the basic block, and e is the number of penalties. The term et^2 corresponds to the time spent searching for a candidate instruction and checking its dependencies when the functions *PTD_arrange_left* and *PTD_arrange_right* are called. The term $n-e$ covers the instructions that are not penalised.

The `repeat` loops (lines `3-16`, `17-30` and `31-44` in Algorithms 5.2, 5.3 and 5.4) ensure that at least one scheduling pass is performed. The `repeat` sections continue until there are no reductions in the penalty measure. The number of times these sections are repeated is denoted by c. Thus, the above term becomes $\theta\left(cet^2 + c(n-e)\right)$.

However, the parameters c, e and t are not general. Basic blocks have different number of penalties (e) of a given type and particular number of retries (c) for each basic block. Similarly, penalties have different instruction distances (t) when searching for a candidate. For our purposes, these factors will be considered general in the interest of clarity and simplification.

The `repeat` block is replicated eight times in the three scheduling phases. Therefore, the complexity of the PTD scheduler is

$$\theta\left(8cet^2 + 8c(n-e) + 8nc + 3n\right) \tag{5.5}$$

The terms $8nc$ and $3n$ represent the computation of the penalty measure throughout the three scheduling phases, both inside and outside the `repeat` sections, respectively.

Equation 5.5 has a few important simplifications. Observations of the scheduling process show that the number of times the `repeat` sections are looped is not greater than three or four. Therefore, c can be considered to be a small constant ($c = 2, 3, 4$). Since the search for candidate instructions start from the instruction neighbouring the penalised one, it is expected that the search (on either side) does not reach $n/2$. Furthermore, if the candidate is found on the left hand side of the penalised instruction, the search on the right hand side is not necessary. Thus, the factor t can be considered to be $t \ll n$. As for the term e, it is often the case that there are not as many penalties of the same type as there are instructions, resulting in $e < n$.

The upper and lower bounds of Equation 5.5 are defined by two opposite scenarios. The upper bound is represented by a pure sequential code. It takes

place when there are as many penalties as instructions ($e = n$), there are no candidates found for all of those penalties ($t = n - 1$), and there can only be two retries ($c = 2$). The upper bound of Equation 5.5 is therefore

$$\theta \left(16\, n\, (n-1)^2 + 19\, n \right) \tag{5.6}$$

However, since the same type of penalties cannot affect more than one `repeat` section, only one term is governed by $cn\,(n-1)^2$, while the other seven share the term $c\,n$ ($e = 0$). The former term dominates the rest of the terms, so the equation has an upper bound of n^3. The upper bound in practice becomes

$$\theta \left(2\, n\, (n-1)^2 + 33\, n \right) \tag{5.7}$$

The lower bound of Equation 5.5 is represented by a purely independent code. In this case, there are no penalties ($e = 0,\ \Rightarrow t = 0$) and no retries ($c = 0$). The complexity is therefore

$$\theta \left(3\, n \right) \tag{5.8}$$

The `if` conditions before the `repeat` sections avoid any scheduling attempt if there are no penalties. Only the initial PTD measures take part in the complexity. The lower bound of the PTD scheduler is therefore of the order of n.

If all the constants are removed from the Equation 5.5, then the complexity of the PTD scheduler becomes

$$\theta \left(et^2 + n - e \right) \tag{5.9}$$

In normal conditions, however, the parameters e and t have particular values with respect to n. As the algorithm progresses, the number of penalties are reduced, so e becomes $e \ll n$. Similarly, t is much smaller than n ($t \ll n$) since in general, the candidate is meant to be found from a close neighbour. Therefore, as the algorithm progresses in normal conditions, the Equation 5.5 is found to be of the order of n.

5.7 Discussion

The nature of the PTD scheduler differs from traditional scheduling techniques. It offers interesting properties and has different characteristics. These are discussed in the following section.

5.7.1 Overlapping Penalties

The priorities given to the different penalties by the PTD scheduler are not only meant to target more expensive stalls first, but also to manage the amount of penalties in the code. During the second scheduling phase, penalties from data dependencies due to different distances between producers and consumers have to be compared. On many occasions the movements are disallowed because of the intermix of instructions and the overlapping of penalties. It has to be ensured that the penalty measure is always reduced after a movement.

By strictly reducing penalties, the PTD scheduler moves through the scheduling space towards the minimum penalty measure. However, the overlapping penalties constrains this search. If they cannot be reduced, it implies that there is no independent instruction in the basic block which when placed in between the penalised instructions, reduces the penalty measure.

In order to overcome the effect of overlapping penalties when a penalty cannot be reduced, the algorithm is required to perform two or more instruction movements. The scheduler must be capable of identifying an instruction that could be moved to a place where the penalty measure stays constant, and from there, evaluate if there is a second instruction that could be placed in the desired position and reduce the penalty measure. If this combination of instructions fails to satisfy the safety conditions, then the transformations are reversed and the process of testing other combinations has to continue.

The problem with this approach is that the number of transformations cannot be known in advance. Furthermore, the complexity of performing a two-instruction transformation is of the order of $\theta(\, t_1^2 + t_2^2\,)$, where t_1 is the distance from the first instruction to its final position, and t_2 is the distance from the second instruction to the penalised instruction. In the worst case, the instruction has to be checked against all the instructions in the basic block. This case implies that $t_1 = n$, resulting in a complexity of the order of $\theta(\, n^2\,)$.

The solution for this task seems more complex than the scheduling algorithm itself. Overlapping penalties is a characteristic of the PTD scheduler. The concept of subgraphs incorporated into the scheduler helps to attenuate their effect.

5.7.2 Input Sensitivity

Another characteristic is that the PTD scheduler is sensitive to the initial order of the instructions. This means that the scheduler is non-deterministic to $\prec$,

unlike the list scheduler. The latter produces the same output independently of the initial order, given that the data dependencies between the instructions are preserved.

However, the PTD scheduler is driven by the penalties in the code and therefore is susceptible to their form. Since the penalties are applied in accordance with the position of the instructions in the schedule, different initial schedules will be approached by the scheduler uniquely. Although higher penalties are handled before penalties with lower ones, the algorithm has to choose among the penalties that share values. This decision leads to different 'paths' during the transformations which result in different schedules.

Given that the PTD scheduler is sensitive to the initial order, it is convenient to *shuffle* its order, before applying the scheduling process. This is in response to the amount of penalties that can be produced by the compiler. An analogy can be drawn with simulated annealing where the initial code is randomly generated.

The first scheduling phase (*PTD_resource_phase*) is considered to be the random factor introduced into the initial schedule. This phase only performs instruction movements to reduce penalties due to resource contentions, and the number of movements is not considerable. However, these transformations serve to produce a better initial schedule.

5.8 Summary

The micronet-based asynchronous processor described in Chapter 4 requires instruction scheduling with the aim of minimising the issue unit stalls due to data dependencies and resource contentions. The processor features an in-order issue unit and out-of-order write-back. Its datapath is characterised by instructions that run as fast as their requirements are fulfilled, and that may overtake other instructions and compete for resources. The functional units operate within different range of latencies depending on several static and dynamic factors. This model presents particular problems to the scheduler, because the dynamic behaviour of the instruction execution makes it difficult to consistently predict the time when results become available.

This chapter has presented a novel and alternative way of performing local scheduling for the micronet-based asynchronous processors. The new method is not based on traditional techniques such as the list scheduling. In contrast, the PTD scheduler is based on a scalar measure that quantifies the amount of stall incurred in the issue unit by data and resource dependencies.

Consecutive instructions with data and resource dependencies are penalised because the issue unit cannot issue the dependent instruction until its operand becomes available, or because consecutive instructions share a common functional unit and there are not enough instances of that type. The penalties are given different degrees in order to reflect different amounts of stall to the issue unit. Data dependencies are more costly in terms of delay; therefore, consecutive data dependencies are assigned higher penalties.

The PTD scheduler performs instruction scheduling within the basic block with the aim of reducing these penalties. The different degrees of penalties help to prioritise the order in which the scheduler reduces them. Valid conditions are introduced to avoid dependency violations, while safety conditions have been defined in order to improve the overall state of the schedule every time an instruction is moved.

The scheduler is divided into several scheduling stages to give priorities to the different spacing of the penalties. Penalties from consecutive data dependencies are treated in a first pass since the delay produced by them is the most significant. Non-consecutive penalties are treated in the following pass, after consecutive penalties have all been reduced.

The PTD scheduler uses concepts such as memory disambiguation and subgraphs to improve the scope for parallelism and the quality of the instruction movements. Memory disambiguation is a well known technique for discarding data dependencies between memory operations when their addresses are not the same. This procedure not only reduces the data dependencies between memory instructions, but those inherited by all their successors. This dependency reduction increases the scope for ILP within basic blocks that can be exploited by the scheduler.

The concept of subgraphs in a basic block was introduced as a heuristic to mask the instructions when selecting candidates to reduce penalties. The PTD scheduler moves independent instructions to reduce a penalty that are located closer to the penalised instruction. The long-term effect due to this practice is that the penalties become overlapped and cannot be reduced further. These overlapping penalties restraint instruction movements in accordance with the safety conditions. Subgraphs are composed by a group of instructions from the basic block involved in a part of the computation. The PTD scheduler is forced to search for candidate instructions from different subgraphs in order to reduce the penalty. This mechanism attenuates the effect of overlapping penalties.

The complexity of the PTD scheduler is based on the number of penalties, and is of the order of $et^2 + n - e$. Since the number of penalties (e) decreases as the scheduler progresses, its value becomes smaller when compared to the number of instructions n. In the long-term, the complexity can be considered to the order of n. This results in a better complexity than those of list schedulers, which are of the order of n^2.

The PTD scheduler offers an alternative method for implementing local instruction scheduling which is tuned to the requirements of asynchronous architectures. The results of the local PTD scheduler are presented in Chapter 7.

The next chapter presents a global extension to the PTD scheduler so that instructions from different basic blocks can be moved across in order to reduce further the penalties.

Chapter 6

Global Optimisations

6.1 Introduction

In the previous chapter a new technique was presented for performing local scheduling which is targeted at micronet architectures. The PTD scheduler is based on penalising the effect of data and resource dependencies which stalls the issue unit of the processor. The objective of the scheduler is to minimise the penalties which are reduced by inserting independent and unrelated instructions in between the dependent ones. The scheduler terminates when the number of penalties cannot be reduced any further.

However, there are two reasons why the schedule for the basic blocks can potentially be under-optimised. Firstly, the mechanism for reducing the penalties introduces an effect called overlapping penalties. This takes place when a group of penalties that are in close proximity restrict the movement of an instruction to reduce another penalty. The safety conditions that guarantee instruction movements to strictly reduce the penalty measure, are often forced to avoid the reduction of a pending penalty as a result of these overlapping penalties. The second reason is the limited amount of ILP that can be found within basic blocks. It is well known that basic blocks may not offer enough parallelism to maintain high levels of resource utilisation, and in the case of the PTD scheduler, the parallelism may not be adequate to minimise the stalls in the issue unit.

It is common practice to search for independent instructions beyond the basic block in order to improve their ILP. Instructions from different basic blocks can fill the scheduling "gaps" produced by basic blocks with insufficient ILP. Global scheduling techniques exploit higher levels of ILP to those obtained by local scheduling. A wider scope of the program is considered for regrouping the instructions in order to distribute the parallelism. Global information about the program such as basic block structure and frequency of instruction execution is

used to combine instructions from different basic blocks.

The advantage of global optimisations is apparent, but there is considerable scope for arbitrary decisions where heuristics may be globally optimal, but locally suboptimal. The penalty measure offers the possibility of a single metric for use in both local and global decisions. A global scheduler capable of moving instructions across different basic blocks to reduce the remaining penalties in the local scheduler represents a natural extension.

In this chapter, the scope of the PTD scheduler for searching candidate instructions is extended beyond basic blocks. A global version of the PTD scheduler allows movement of instructions beyond the basic block boundary in order to reduce penalties. The global scheduler is also based on the PTD measure which is used as a reference metric to perform global code motion. Basic blocks with a resultant positive measure are eligible for global code motion. The context of code motion within this chapter represents global movement of instructions without the need for copies or speculation. When code motion cannot be performed due to data dependencies, code duplication (code motion with copies) is applied, in an effort to reduce the penalties left after local scheduling.

This chapter describes some underlying concepts for applying code motion without copies or speculation. The safety conditions introduced previously for reducing the penalty measure are extended to handle code motion and code duplication. Code motion and code duplication, along with their generalisation in tail duplication and block merging are also explained. Finally, a description of a global PTD scheduling algorithm is presented.

6.2 Related Work

Global scheduling is an optimisation technique that involves instruction movement and scheduling across multiple basic blocks, with the aim of reducing the program execution time. Global program representations that include data and control relationships allow schedulers to group instructions from different basic blocks. When moving instructions over different basic blocks the semantics of the program must be preserved. When an instruction is moved into another basic block that is located in a different path of the program, it has to be ensured that complementary measures are taken to respect the overall meaning of the program.

There are several approaches to forming regions larger than basic blocks, which can be categorised under either cyclic or acyclic optimisations. Cyclic optimisations include instructions from different iterations of the program to increase the

parallelism. Conversely, acyclic optimisations only include basic blocks within the same acyclic region. The optimisations discussed in this chapter are strictly acyclic. A region is defined as a series of related basic blocks as a result of the compilation of a function.

Global scheduling can also be divided into two types based on whether the transformations are based on profile information — the recorded run-time behaviour of a program, for a particular set of inputs. The inputs must be carefully selected with representative data. The profile information is annotated on all the basic blocks of the program with their percentage of execution, for the particular set of inputs. Given the frequency of execution for each basic block, the frequencies of all possible paths can be determined.

A selection of global acyclic scheduling techniques, including profile-based transformations, are described in this section.

6.2.1 Trace Scheduling

Trace scheduling [44][51][105] is a global optimisation technique that considers a sequence of basic blocks as a trace. The selection of basic blocks is performed upon the most likely trace – often called the *on-trace* – of the program. The information is retrieved from several runs of the program operating on typical data. The resultant trace is scheduled with a list scheduler as a large single basic block.

If an instruction is moved across basic block boundaries, then one or more copies of compensation code may be required in the *off-trace*. If the instruction is moved above a join or below a fork, then compensation copies are inserted into the off-trace paths. This process of inserting compensation code is called *bookkeeping*. One of the side-effects is an explosion in code size, but policies exist to limit the generation of compensation code [54].

6.2.2 Superblock Scheduling

Superblock scheduling [85] is a variation of trace scheduling. The main difference is that the traces do not have side entrances. Profile information is used to select the most frequent traces. After these traces have been generated, a technique called *tail duplication* is performed in order to remove the side entrances [29]. The resulting traces are called *superblocks*. After all the superblocks have been defined, the Superblock Scheduler uses list scheduling techniques to optimise the superblocks.

The benefits due to optimisations performed along the frequent paths, such as the on-trace and the superblock, are often made at the expense of the non-frequent paths.

6.2.3 Hyperblock Scheduling

Hyperblocks [108] are similar to superblocks in that the control flow can only enter from the top, but can leave from different exits. The difference between a superblock and a hyperblock is that the latter is based on predicated execution. Predicated execution refers to the conditional execution of instructions depending upon the value of a boolean source operand. If the operand, also called a *predicate*, is true, then the instruction is executed normally; otherwise, it is treated as a no-operation (nop) instruction. The advantage of hyperblocks over superblocks is that hyperblocks contain instructions from more than one path of control when there exist multiple, important paths [86].

6.2.4 Dominator-path Scheduling

Dominator-path scheduling [163] is a global scheduling technique similar to trace scheduling in that several basic blocks are treated as a single block. The main difference is that the blocks are selected from the dominator-path of the region, and not from traces. This path is selected from the dominator tree[1] with the help of heuristics or profile information.

Another difference with trace scheduling is that dominator-path scheduling uses this dominator analysis to avoid the use of compensation code, which is a significant concern. Once the blocks are selected, they are scheduled with a list scheduler.

6.2.5 Code Motion

All the previous examples of global scheduling are characterised by the basic blocks being considered as the unit of transformation. The frequently-encountered basic blocks are grouped as a single *meta*-block in which local scheduling can exploit more parallelism. However, the effectiveness of these techniques depend on grouping frequent paths that outweigh the degradation of other paths that are less frequent. Trace-based scheduling techniques though are less effective when targeting programs with paths that are evenly frequent.

[1]The dominator tree [2] represents the *dominance* set between basic blocks. One block is said to dominate another, if the former is executed, and then eventually, the latter has to be executed. The concept of dominance is explained further in Section 6.3.1.

Other types of global scheduling consider the instruction as the unit of transformation. Instead of grouping basic blocks, instructions are individually moved to other basic blocks. Considering single instructions as the unit for the transformations provides a finer granularity in the ILP improvement.

The term *code motion* has been overloaded. It has been used to describe the motion of expressions in the intermediate code, and also to describe the motion of assembly instructions after code generation. At the intermediate code level, code motion is applied to eliminate redundancies in the code, such as constant propagation and partial redundancy elimination [33][96], or to distribute coarse-grain parallelism [3].

At a lower level, the term code motion is used to describe the movement of assembly instructions within a scheduler. A global scheduler is presented in [20], in which *useful* instructions are moved beyond basic blocks within an acyclic region. Useful instructions represent instructions that can be moved without the need for compensation code, or speculation (*c.f.* Section 2.1.1.3). The movement of useful instructions has the characteristic of being profile-independent, since the movements do not compromise any of the paths that the control flow may take in between the source and destination blocks. The scheduler does not consider code duplication in any of the code motion transformations.

Another example of code motion of instructions is found in [107], in which the scheduler is targeted to work with or without the use of profile information. The code motion within this scheduler does consider limited cases of code duplication.

6.3 Global Scheduling for the Micronet Model

Global schedulers have used the list scheduling algorithm for optimising the order of instructions in the meta-blocks. The ones described previously usually define instruction markers to specify the boundaries of the original basic blocks. These markers range from compiler directives to instruction identifiers. The list scheduler interprets these markers so that the heuristic can be tailored for scheduling instructions from more than one basic block. For instance, the instructions within a meta-block may no longer have the same weight and priorities as instructions in local scheduling do. There are several reasons behind this, some of them being particular to the global scheduler itself. One reason being that an instruction from the new meta-block may not have the same number of execution as the others [87][163], or the same control properties, in the case of a speculative movement. Another reason is that by moving one instruction beyond its basic block

causes the life of its registers to be prolonged[2] — an important issue if register allocation is performed after global scheduling [122].

A global version of the PTD scheduler differs from those described in Section 6.2, since the underlying scheduling method is not based on the list scheduler; instead, the global PTD scheduler is based on the penalty measure. The next section defines the terms used to explain the heuristics for code motion in the global PTD scheduler. The global PTD scheduler with code motion, but without compensation code, is described in Section 6.3.2, and with code duplication is described in Section 6.3.3. The global PTD scheduler does not consider either speculative movement of instructions or the use of profile information.

6.3.1 Definitions

The word *region* is another overloaded term. For example, in [70] and [3], a program is divided into regions that are composed of code statements in order to perform region optimisations. In [90] programs are divided into control regions, *i.e.* sections of the program with the same control dependencies.

The term region, as described in this thesis, defines a group of basic blocks containing assembly instructions with a single entry and multiple exits. They are the result of compiling the functions of a program. A region is defined as part of a program in which the basic blocks are strongly connected [162].

Let A and B be two basic blocks in a region. A is said to *dominate* B (A *dom* B), if block A appears in all the paths from the entry of the region to block B. Similarly, B is said to *post-dominate* A (B *post* A), if block B appears on all the paths from block A to the exit of the region [2]. Figure 6.1 (a) depicts a *control flow graph* (CFG) of a region with its entry and exit nodes. It can be seen that B_1 dominates all the other basic blocks, as it appears in all the paths from the entry of the region up to all of the blocks. Similarly, block B_7 post-dominates the other blocks since it appears in all the paths from these blocks to the exit of the region.

If both conditions hold for blocks A and B, *i.e* A dominates B and B postdominates A (A *dom* $B \wedge B$ *post* A), then it is said that A is *equivalent* to B [20]. This is an important characteristic because it means that should A be executed, then B will definitely be executed. It can be seen in Figure 6.1 (a), that whichever path is taken after executing block B_1, the flow of control will eventually arrive at block B_7. In practice, the equivalence condition implies that blocks A and

[2]If the instruction is moved against the flow of control, the life of the destination register is prolonged. Conversely, the source register is prolonged when the instruction is moved along the control flow.

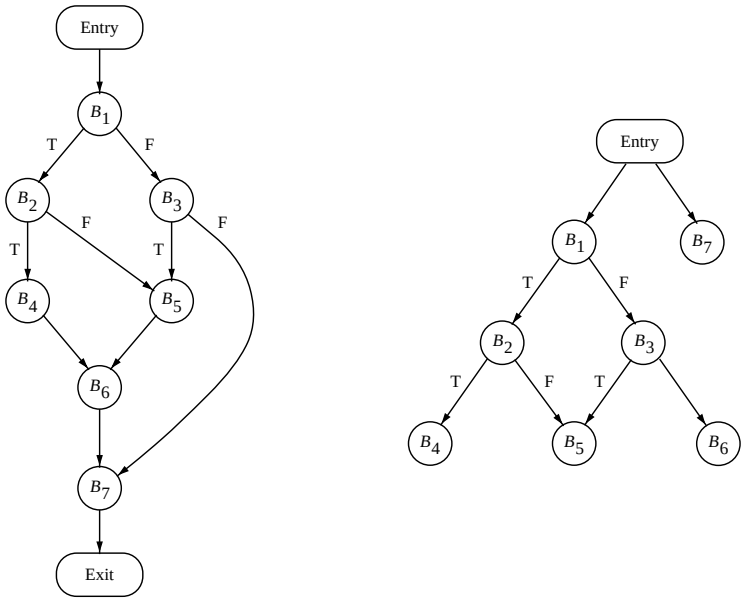
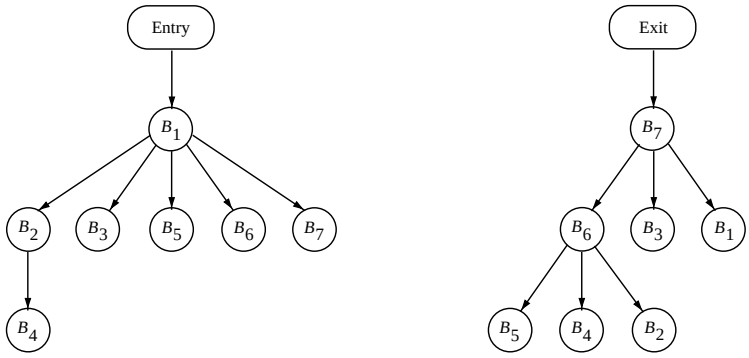


Figure 6.1: (a) Control flow graph and (b) its control dependence subgraph.

B can be executed in parallel, as long as their data dependencies are respected. Therefore, instructions from equivalent blocks can be moved without the need for copying instructions, and without the risk of speculation. Blocks A and B are said to be *control-independent*. In the example, only blocks B_1 and B_7 fullfill the equivalence definition.

Figure 6.1 (b) shows the *control dependence subgraph* of the control flow graph of Figure 6.1 (a). The control dependence subgraph shows the control flow dependencies between the basic blocks. It is shown that basic blocks B_1 and B_7 are control independent, as are blocks B_4, B_5 and B_6. However, blocks B_4, B_5 and B_6 are not equivalent. Although block B_6 postdominates blocks B_4 and B_5, blocks B_4 and B_5 do not dominate block B_6. This is shown graphically in Figure 6.2. Figure 6.2 (a) shows the dominator tree, and Figure 6.2 (b) shows the postdominator tree of the control flow graph in Figure 6.1 (a).

When there are back entries as in loops in the region, it must be ensured that the equivalent blocks are contained within the same *sub-region*. If a loop is added to the control flow graph of Figure 6.1 (a), then the basic blocks, B_1 and B_7, are no longer equivalent. Figure 6.3 depicts the new control flow graph with a back entry. Although B_1 still dominates B_7 and B_7 postdominates B_1, B_1 may be



(a) (b)

Figure 6.2: (a) Dominator-tree and (b) postdominator-tree.

executed more often than B_7. For the graph in Figure 6.3, basic blocks B_1 and B_7 are not equivalent.

6.3.2 Code Motion for the PTD Scheduler

The local PTD scheduler attempts to minimise the penalty measure on each basic block by performing local movement of instructions. The local optimisation serves as a monitor of how much parallelism can be exploited by the micronet architecture (or how much stall to the issue unit is caused by true dependencies and resource contentions). If the ILP in the basic block is adequate, then the penalty measure will be proportionately reduced at the end of the local scheduling process.

The natural extension to the local PTD scheduler is to allow instructions from different basic blocks to be moved, in order to reduce the penalty measure if it has not been totally cleared. Within the PTD scheduler, the penalty applied to an instruction can be regarded in two ways: one view is that the penalty indicates that another instruction has to be moved in order to reduce it (as in the local scheduler), and the other is that the instruction itself must be moved, in order to reduce its penalty. The global scheduler assumes the latter view. Since the search space for independent instructions can grow considerably when compared to a basic block, it is more feasible to move away the cause of the penalty (the penalised instruction) rather than searching for independent instructions.

With this assumption, penalised instructions are considered *candidate* instruc-

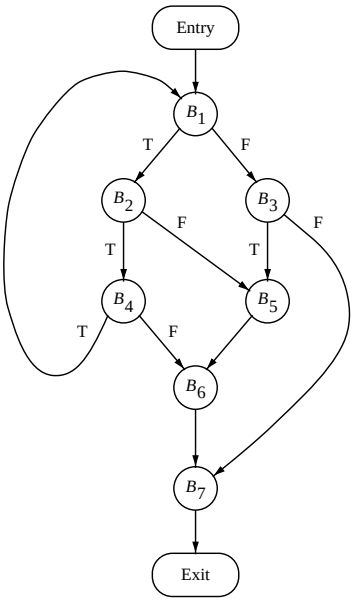


Figure 6.3: Control flow graph with a loop.

tions for global movement. The target destination for the code movement is restricted by the equivalence definition. Code motion is only allowed to equivalent blocks.

The equivalence definition enables code motion beyond basic blocks but without speculation or code duplication. This definition determines which group of instructions are executed and under what static *control flow* conditions. Thus, the task is reduced to deciding under which *data* conditions instructions can be moved.

The data conditions are governed by the penalty measure. Since each basic block has its own penalty measure, a *global measure* can be regarded as the total amount of stall caused to the issue unit within the region. The local scheduler minimises the penalty measure on every basic block, so the global effect is to reduce the collective measures.

When the penalties cannot be completely reduced, the global scheduler's task is to distribute instructions within equivalent nodes. To do so, the global scheduler has to ensure that both the data dependencies and the safety conditions are respected. For the global scheduler, the safety conditions defined for the local scheduler are expanded to consider instructions from two basic blocks, *i.e.* the *sender* and the *receiver* block.

The global extension of the PTD scheduler is applied after the local scheduling.

The global scheduler scans the regions of the program in topological order, that starts at the entry of the region and finishes at the exit. The global scheduler is restricted to moving instructions against the flow of control (without considering the back entries from loops). Since the candidate instructions for code motion are the penalised instructions, there would be very few occasions in which they could be allowed to move, if the direction of movement was in favour of the flow of control. This is because the likelihood of their successors being located on the trajectory of the movement is high. Another slight advantage of moving instructions against the flow of control is that only the life of one register is lengthened, as opposed to two.

When the scheduler stops at a basic block with a positive penalty measure, the penalised instructions become candidates to be moved outside the basic block. The receiver block is selected among the set of equivalent blocks. The set is traversed in reverse order, *i.e.* from the closest equivalent block to the furthest. The penalised instructions are selected from the start of the block towards the exit. The penalised instruction is checked against the preceding instructions in the basic block in order to guarantee that the instruction can leave the block. The next step is to check the data dependency with the instructions from the basic blocks positioned in between the sender and receiver block, if any. This checking also includes memory disambiguation in the case of load instructions. If there is a function call within these blocks, then the dependency checking is also performed in that region.

Figure 6.4 shows the region pictured in Figure 6.1 (a), with the final instructions from block B_1 and the first ones from block B_7. The global PTD scheduler traverses the region from blocks B_1 to B_7. The instructions shown in basic blocks B_1 and B_7 represent the code after local scheduling. Although B_1 is equivalent to B_7, the instructions cannot be moved in the direction of the control flow, from B_1 to B_7. When the scheduler stops at block B_7, the penalised instructions I_1 and I_2 become candidates for code motion. Since both instructions are located on top of the basic block, and are independent, they are allowed to leave basic block B_7. Both instructions are checked for data dependencies with the instructions from basic blocks B_2 to B_6. In the example, it is assumed that instructions I_1 and I_2 do not have data dependencies with any of the instructions from blocks B_2 to B_6.

When a candidate is able to be moved to another basic block, the selection of its final position becomes important. Since local scheduling has been performed previously at the receiver block, its instructions hold the penalties, if any, depending on the ILP in the block. The ideal situation is to reduce a penalty at

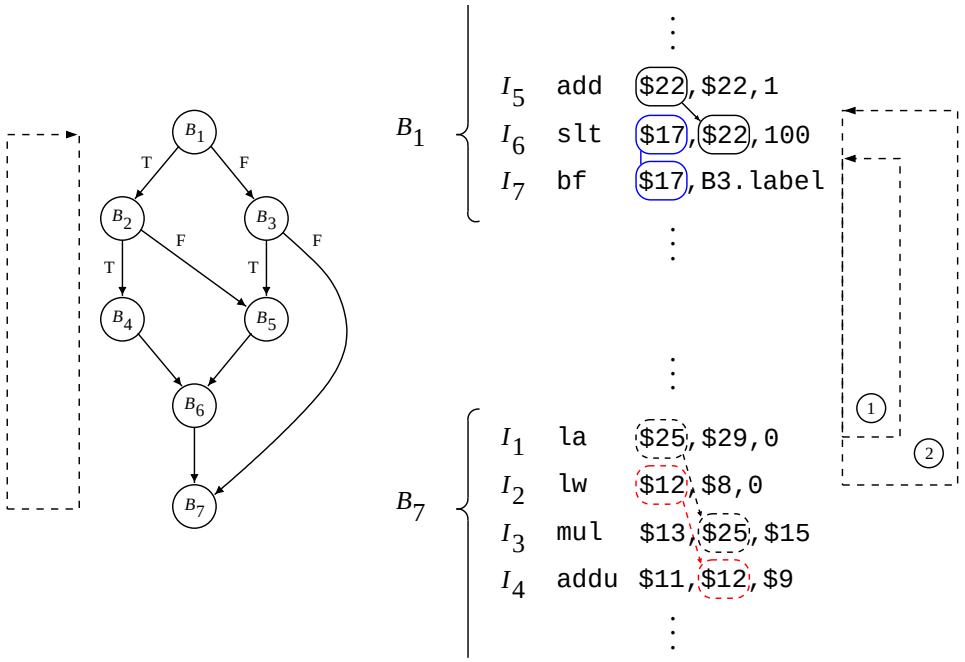


Figure 6.4: Code motion in the PTD Scheduler.

the receiver block and match a different type of functional unit to those of the instructions that share a penalty. The safety conditions of the global scheduler are responsible for deciding the final position of the candidate instruction. (The safety conditions are explained in Section 6.3.4).

In Figure 6.4, instruction I_1 is the first candidate to leave its basic block. Instruction I_1 is placed in between instructions I_6 and I_7 to reduce the penalty due to the true dependency. The type of instruction I_1 is different from those of instructions I_6 and I_7, to reduce any possibility of resource contentions. Similarly, instruction I_2 is moved in between I_5 and I_6 to separate the true dependency. The load instruction also differs from the types of instructions I_5 and I_6.

Since it is not known in advance how many candidate instructions will be amenable for code motion, when a candidate is moved it has to be placed at the site of the highest penalty in the receiver block. This position is represented by the penalty between the set and branch instructions (I_6 and I_7) in basic block B_1. When instruction I_1 is moved the penalties at the receiver block are updated, so when instruction I_2 is moved, the highest penalty is assigned to instruction I_5.

This procedure works well when the instructions moved into the receiver basic block are independent, as in the example in Figure 6.4. In contrast, if the can-

didate instructions are dependent, then the second instruction cannot be placed before the first one, in the case when the second highest penalty is located before. Due to this, when the first instruction is to be moved, the second-best position is always stored. When the second instruction is moved, the first one is relocated to the second-best desired place, so that the second instruction could fill the original place.

The code motion algorithm terminates when there are no penalised instructions left in the sender block, or when the remaining penalised instructions cannot leave the block due to data dependencies.

6.3.3 Code Duplication for the PTD Scheduler

Data dependencies often restrict the code motion of an instruction from its basic block to an equivalent basic block. In an attempt to reduce the penalty of the instruction, it is still preferable to perform code duplication. Instead of moving the instruction to an equivalent basic block, the instruction is copied and moved to the immediate predecessor blocks.

When there is code duplication, the expansion of the code is always a concern. The advantage of performing code duplication only when code motion cannot be called keeps the expansion of code to a minimum.

The decision for selecting instructions for code duplication follows the same principle as that in code motion. When there is a data dependency in the trajectory to the equivalent block that avoids the movement, the receiver block is replaced by the predecessor blocks instead. The copies of the instruction are individually placed in the best position at each of the receiver blocks. The safety conditions, which are described in the next section, are also the same as in the case of code motion.

Figure 6.5 shows the same control flow graph of Figure 6.1 (a), this time in an attempt to reduce the penalties in basic block B_6[3]. Since B_6 does not have equivalent nodes and its predecessors have only one successor (B_6 post-dominates both B_4 and B_5), the penalised instructions I_1 and I_2 are copied and moved into blocks B_4 and B_5. First, instruction I_1 from block B_6 is moved into block B_5. The instruction is placed at the highest place to reduce the penalty of instruction I_1 in B_5. This is because the type of instruction is not relevant since all the instructions involved (I_1, I_2 and I_3 from B_5, and I_1 from B_6) are arithmetic instructions. Similarly, instruction I_1 is copied to the highest position to reduce the penalty of instruction I_5 in block B_4, since all the instructions (I_5, I_6, I_7 and

[3]Instructions from B_7 are ineligible for code duplication since B_3 has a 'side-exit' (block B_5).

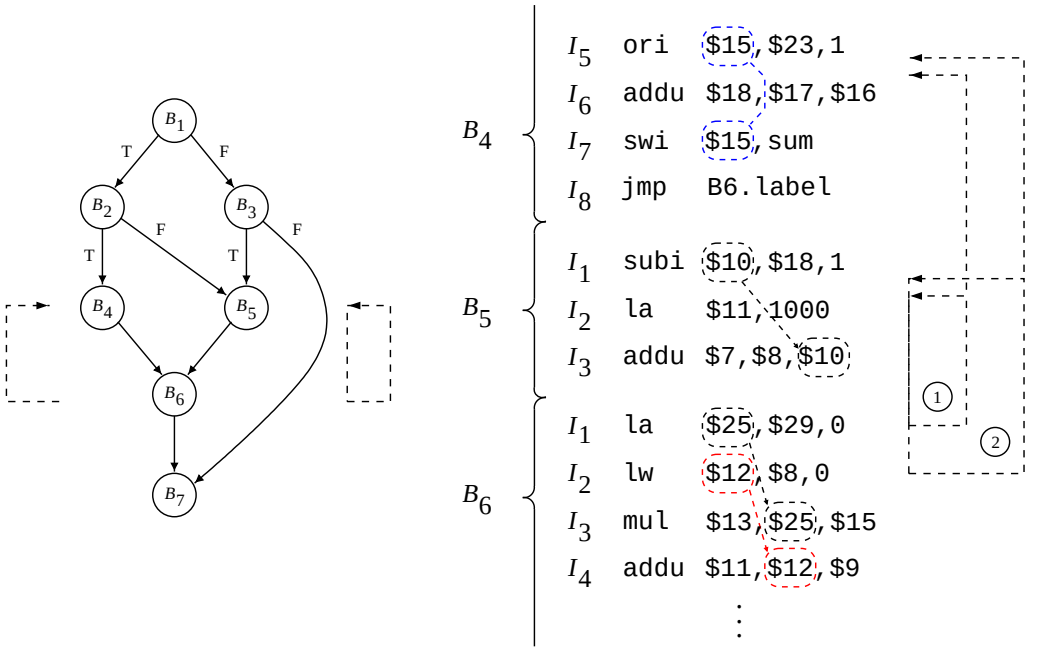


Figure 6.5: Code duplication in the PTD Scheduler.

I_8 in block B_4) have different types.

By the time the second instruction I_2 from block B_6 has to be copied, the penalties would have been updated at the receiver blocks. Instruction I_2 is moved to the highest position to reduce the penalty in block B_5. This time, the type of instruction is more important since the penalty due to data dependency has been reduced previously by distancing I_1 and I_3 in block B_5 with instruction I_1 from B_6. In the basic block B_4, the load instruction is moved before the recently-copied instruction I_1. This is also to distance the memory instructions (I_7 from B_4 and I_2 from B_6) in the basic block.

6.3.4 Safety Conditions

The safety conditions for the global scheduler are an extension of the one for local scheduler presented in the previous chapter. The difference rests in the need for strictly reducing not only the consecutive and non-consecutive penalties, but also instructions have to alternate ones from the different available types.

A candidate instruction for code motion is allowed to leave the source block if it complies with the valid conditions, *i.e.* it does not have a data dependency with any instruction that belongs to basic blocks located in the trajectory of the

movement. When the candidate instruction is allowed to move to an equivalent basic block, then the first safety condition is to guarantee that the penalty measure is reduced in the source block. This is defined by the following equation

$$\sum P(s)_{after} < \sum P(s)_{before} \tag{6.1}$$

The penalty measure after the transformation $P(s)_{after}$ considers a source block without the candidate instruction, a difference with the safety condition for local scheduling. If Equation 6.1 holds, then the safety condition is evaluated at the destination block.

$$\sum P(r)_{after} < \sum P(r)_{before} \tag{6.2}$$

Similarly, the penalty measure after the transformation on the destination block ($P(s)_{after}$) considers the candidate instruction as being introduced, whereas the penalty measure before code motion does not include it. If Equation 6.2 holds, then the code movement is allowed.

Among the instructions that can be moved, ones with certain registers are not considered as candidates for code motion or code duplication, if there is a function call in the trajectory of the movement. Instructions that reference registers `$29` and `$2` are not allowed either to be copied or moved. Register `$29` contains the stack pointer, while register `$2` is used to pass the *static link* (return value) when there are nested procedure calls [91]. Instructions that refer to the stack, such as loads and stores, may modify the value of a memory location through a memory address stored in the stack (indirect addressing mode, *c.f.* Section 5.5.1). The memory disambiguator may not be able to detect these references, and thereby causing a data dependency violation. The restriction over instructions using register `$2` is because the data dependencies from standard libraries cannot be detected since their code is not annotated after compilation (*c.f.* Section 7.2.2).

6.4 Global Optimiser for the Micronet Model

As mentioned in Section 6.2.5, the type of global optimisations are governed by the unit of transformation. Global scheduling in our case is a fine-grain optimisation and considers the instruction as the unit of transformation. Other methods such as region scheduling use the basic block as the unit for performing coarse-grain optimisations.

Region transformations have been studied at different levels in the compiler. At the intermediate code level, region transformations are usually implemented to remove redundancies in the code (*c.f.* Section 2.1). However, they can also be applied to distribute the parallelism in the code [70]. The list of region transformations includes *move*, *copy* and *merge* region. The *copy* transformation is used to remove unconditional branches. It is also known in the literature as *tail duplication* or *node splitting*. Tail duplication refers to the duplication of a node (a basic block or a region) and its edges, when the node has more than one predecessor. This transformation was originally applied to break cycles of dependencies in order to generate better code for parallel machines. It helps to reduce communication and synchronisation costs [50]. Mueller and Whalley use *code replication* to both remove unconditional branches (`jmp` instructions) [119], and to avoid conditional branches as well [120]. The resultant code contains simplified control flow that benefits vector and parallel compilers.

The *merge* transformation joins two regions (or basic blocks) by removing the unconditional branch. This transformation is performed when the regions share the same set of control dependencies, *i.e.* one region is equivalent to another, and the former has only one exit and the latter has only one entry. These nodes are said to be *collapsible* if these conditions are met.

6.4.1 Tail Duplication and Block Merging for the PTD Scheduler

Tail duplication along with block merging represent the same concept of code duplication as defined in Section 6.3.3 but with a different granularity. The difference being that the whole basic block is copied, instead of copying instructions on a one-by-one basis.

Tail duplication and code merging are incorporated into the PTD scheduler with the aim of increasing the scope for ILP. The conditions for applying these techniques require that after the transformation the size of the merged basic block is incremented. In other words, it is required that the basic block that is to be duplicated has at least two predecessors, and at least one of them has at most one successor (no conditional branches), so that the block can be merged. Additionally, the predecessors' basic blocks should not have to end with a `call` instruction, since this type of unconditional branch avoids block merging. (Function inlining [39] is an optimising technique to overcome this limitation and merges code from the caller and callee functions).

The result of applying tail duplication and block merging to the control flow

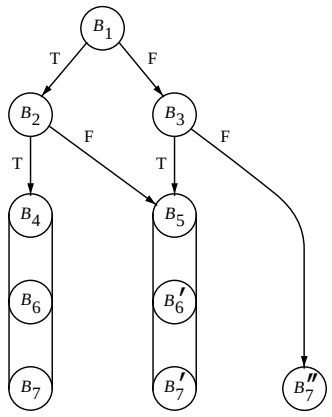


Figure 6.6: Global optimisations applied to the CFG from Figure 6.1 (a).

graph shown in Figure 6.1 (a) is displayed in Figure 6.6. It can be seen that basic blocks, B_6 and B_7, were duplicated and merged with blocks B_4 and B_5, respectively, to form two larger basic blocks. One is formed with basic blocks B_4, B_6 and B_7 and another is formed with blocks B_5, B_6' (a copy of B_6) and B_7' (a copy of B_7), respectively.

Basic blocks B_7'' (another copy of B_7) and B_4 cannot be merged with blocks B_3 and B_2 respectively, because they have two successors (conditional branch). For the same reason, B_5 cannot be duplicated because neither of its copies would be merged with blocks B_2 and B_3 due to their conditional branches.

Another example of a control flow graph including a loop is shown in Figure 6.7. Figure 6.7 (a) displays a control flow graph and Figure 6.7 (b) shows its transformations. Tail duplication can be normally applied to basic block B_5 even if its conditional branch represents a loop. Blocks B_5 and B_7 were duplicated for merging with basic blocks B_4 and B_6, respectively. Block B_6 cannot be duplicated because all its predecessors (B_2, B_5 and B_5') have conditional branches.

Tail duplication as well as code duplication offer the advantage of increasing the ILP available in the code. However, their indiscriminate use leads to code expansion. The code expansion due to tail duplication has an upper bound of the order of $p\,2^p$, where p is the number of *if-then* statements that appear in the region [70]. The conditions for applying tail duplication only if there is an increase in the size of the basic block, help to limit the expansion of code. The control flow graph examples of Figures 6.6 and 6.7 show that not all the candidates for tail duplication are duplicated.

Although tail duplication and block merging are very similar to code duplica-

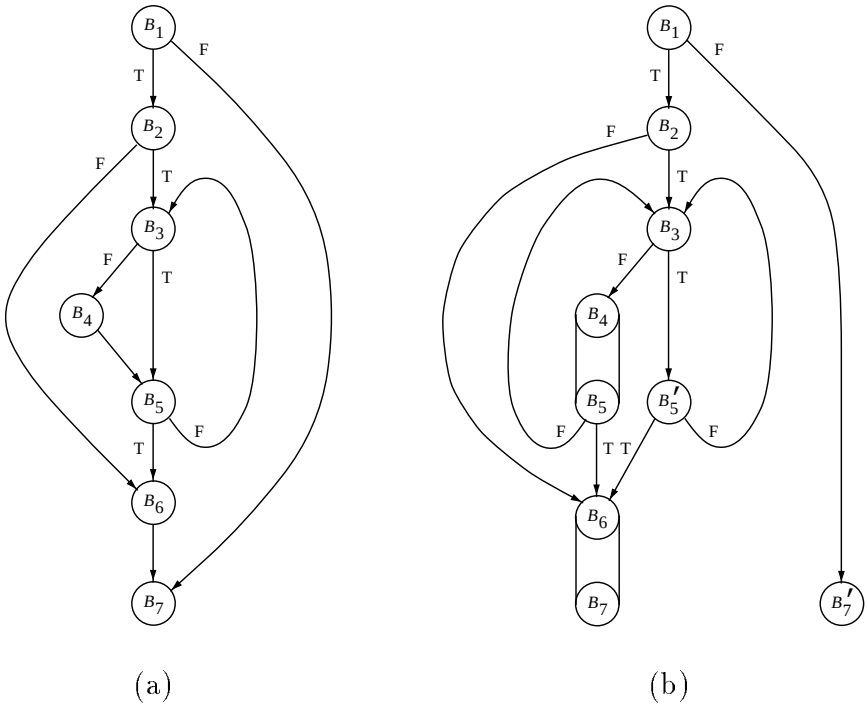


Figure 6.7: (a) Example of a CFG with a loop and (b) its transformation.

tion, the decisions to tune the merged basic blocks after the transformation differ substantially. Since it is a complete basic block being merged with another, it becomes more practical to apply local scheduling to the resultant basic block. Once tail duplication and block merging have been performed in the region, the data dependencies of the modified blocks are updated and local scheduling is applied again.

Tail duplication is performed in a bottom-up order. This represents a natural extension to the control flow graph since tail duplication folds its leaves as in a tree.

6.5 Algorithms

All the global optimisations described in this chapter are applied after local scheduling has been performed. The function *code_motion* in Algorithm 6.1, performs code motion and code duplication, whereas function *tail_duplication* in Algorithm 6.2, performs tail duplication and block merging. These functions identify the basic blocks that require further optimisations to reduce their penalty measures. They traverse each region and stop at every basic block that

has a non-zero penalty measure. The region is traversed in control-flow order in the *code_motion* function, and in reverse control-flow order in the *tail_duplication* function.

Line `1` in both algorithms sets the starting basic block (*block*). For code motion, the starting block is represented by the entry block of the region, while for tail duplication it is represented by the exit of the region. The `repeat` loop contained within lines `2` and `46` in Algorithm 6.1 and lines `2` and `33` in Algorithm 6.2, traverse the whole region.

The penalty measure and the number of equivalent nodes for code motion, or the number of predecessors for tail duplication, are computed to know whether the basic block requires further penalty reductions and identify the destination blocks. This is checked in lines `5` and `7` in Algorithms 6.1 and 6.2, respectively.

In function *code_motion*, a penalised instruction can potentially be moved to any of the equivalent basic blocks. Line `7` selects from one of the equivalent basic blocks, under the `for` loop in line `6`. Line `8` obtains the basic block identifier (*id*), which is compared against the equivalent block identifier (*which_blk*) in line `9`, to ensure that the latter represents one that was previously scheduled, and therefore, against the flow of control. The variable *equiv_block* holds the equivalent block (line `13`), and variable *node* is assigned the entry instruction of the basic block (lines `12` and `10` from the algorithms).

The two inner `while` loops in both algorithms traverse the basic block in an attempt to move any penalised instruction. The first `while` loop is dedicated to consecutive penalties, while the second one is dedicated to non-consecutive penalties.

The function *free_for_motion* in lines `16` and `31` in the *code_motion* algorithm, searches for data dependencies with respect to *node* from the current basic block (*block*) to the equivalent basic block (*equiv_block*). Also, if a function call is located within these blocks, then data dependency checking is carried out in that function against the registers used by the penalised instruction *node*. If it is a memory instruction, then memory disambiguation is also performed on any memory reference located in these basic blocks. The instruction is allowed to be moved, if there are no data dependencies (valid condition).

move_up in lines `17` and `32` of Algorithm 6.1 is the function responsible for moving the instruction to the equivalent basic block. In contrast, if there are data dependencies and the instruction cannot be moved, then it is evaluated for code duplication. This is performed by the function *validate_code_dup* in lines `19` and `34`. This function ensures that there are no call functions ending the predecessor

Algorithm 6.1 ***code_motion (region)*** algorithm.

```
1:  block = entry (region) {START FROM THE REGIONS' ENTRY BLOCK.}
2:  repeat
3:    measure = PTD_measure (block, first_phase)
4:    dest_blocks = equiv (block) {RETURNS THE NUM. EQUIV. BLOCKS.}
5:    if measure and dest_blocks then {PENALTY & DESTINATION BLOCKS.}
6:      for j = 0 to j < dest_blocks and measure do
7:        which_blk = get_equiv_id (block, j)
8:        id = get_block_id (block)
9:        if which_blk < id then {RESTRICT MOVEMENT IN COUNTERFLOW.}
10:         pred1 = pred (block, 1)
11:         pred2 = pred (block, 2)
12:         node = first_instruction (block)
13:         equiv_block = equiv (block, j)
14:         while node ≠ NULL and measure > 0 do
15:           if penalty_consecutive (node) > 0 then {CONSEC. PENALTY.}
16:             if free_for_motion (node, block, equiv_block) then
17:               move_up (node, block, equiv_block) {IT IS CODE MOTION.}
18:             else {IT IS CODE DUPLICATION.}
19:               if validate_code_dup (node, block, pred1, pred2) then
20:                 move_up (duplicate (node), block, pred1)
21:                 move_up (node, block, pred2)
22:               end if
23:             end if
24:           end if
25:           node = next (node)
26:         end while
27:         node = first_instruction (block)
28:         measure = PTD_measure (block, second_phase)
29:         while node ≠ NULL and measure > 0 do
30:           if penalty_nonconsecutive (node) > 0 then {NON-CONSEC.}
31:             if free_for_motion (node, block, equiv_block) then
32:               move_up (node, block, equiv_block) {IT IS CODE MOTION.}
33:             else {IT IS CODE DUPLICATION.}
34:               if validate_code_dup (node, block, pred1, pred2) then
35:                 move_up (duplicate_inst (node), block, pred1)
36:                 move_up (node, block, pred2)
37:               end if
38:             end if
39:           end if
40:           node = next (node)
41:         end while
42:       end if
43:     end for
44:   end if
45:   block = next (block)
46: until block == NULL
```

basic blocks. The functions *free_for_motion* and *validate_code_dup* also restrict any instruction for code motion, if the stack pointer (register `$29`) is referenced.

The two function calls *move_up* in lines `20` and `21`, and again in lines `35` and `36`, represent the duplication of the instruction. A duplicated copy of the instruction is moved in the first call, while in the second call the original instruction is moved. The variables *pred1* and *pred2* hold the predecessors of the current basic block (lines `10` and `11` in Algorithm 6.1, and lines `8` and `9` in Algorithm 6.2), where the instructions are being destined.

The main difference between Algorithms 6.1 and 6.2 lies in the validation checking for the potential transformations. The function *validate_tail_dup* in lines `13` and `24` from Algorithm 6.2, checks that at least the current basic block can be merged with one of the predecessors (the predecessor must have only one successor and it must not end with a call). The function also checks that the predecessors are not empty, and that both the sender and receiver blocks have the same *static count* of loop iterations. The static count of loop iterations is maintained by incrementing a variable for every basic block when there is a back entry from a loop.

The function *move_up* in Algorithm 6.3 is responsible for performing code motion, which when called it is assumed that the instruction has to be moved with no exceptions. The function then has to decide the best possible position at the particular time, for the instruction being moved into the basic block. (This search also records the second-best position for possible future considerations.)

The variable, *aux_node*, is positioned at the last instruction of the destination basic block (line `1`). From this position, the data dependency checking is performed in order to know the highest position of the incoming instruction (parameter *node*). The variable, *move_node*, holds the position of the destination node.

Data dependencies and memory disambiguation are checked in lines `2` and `17`, and in lines `7` and `29` (Algorithm 6.3), respectively. If there is a data dependency, then the instruction is moved to the end of the basic block, otherwise, the candidate instruction is checked with the rest of the instructions in the basic block (within the `while` loop in lines `16` and `40`). The `while` loop stops either when a data dependency is found or the start of the basic block is detected.

If there is a data dependency with an instruction that had been previously moved as a result of code motion (line `20`), then this instruction is moved back to its second-best position, only if this is located earlier in the schedule (line `21`). The idea is to place this instruction as early as possible in the schedule, since it

Algorithm 6.2 ***tail_duplication (region)*** algorithm.

```
1:  block = exit (region) {START FROM THE REGIONS' EXIT BLOCK.}
2:  repeat
3:    measure = PTD_measure (block, first_phase)
4:    if pred (block) == 2 then {RETURNS THE NUM. PRED. BLOCKS.}
5:      dest_blocks = 1
6:    end if
7:    if measure and dest_blocks then {PENALTY & DESTINATION BLOCKS.}
8:      pred1 = pred (block, 1)
9:      pred2 = pred (block, 2)
10:     node = first_instruction (block)
11:     while node ≠ NULL and measure > 0 do
12:       if penalty_consecutive (node) > 0 then {CONSEC. PENALTY.}
13:         if validate_tail_dup (block, pred1, pred2) then
14:           move_block (duplicate_block (block), pred1)
15:           move_block (block, pred2)
16:         end if
17:       end if
18:       node = next (node)
19:     end while
20:     node = first_instruction (block)
21:     measure = PTD_measure (block, second_phase)
22:     while node ≠ NULL and measure > 0 do
23:       if penalty_nonconsecutive (node) > 0 then {NON-CONSEC. PEN.}
24:         if validate_tail_dup (block, pred1, pred2) then
25:           move_block (duplicate_block (block), pred1)
26:           move_block (block, pred2)
27:         end if
28:       end if
29:       node = next (node)
30:     end while
31:   end if
32:   block = prev (block)
33: until block == NULL
```

is not known in advance how many dependent instructions will be moved later on. The actual position is stored at *move_node* in order to place the incoming instruction at the place where the previously moved one was located (line 22).

If there are no data dependencies, then the global safety conditions (function *check_global_move* in Algorithm 6.4) defined in Section 6.3.4 are applied to the instruction referenced by *aux_node* (line 35). If these conditions hold, a finer heuristic (function *update_best_position* in Algorithm 6.5) is applied in order to obtain the best and second-best positions (line 36).

6.6 Discussion

The global scheduler presented in this chapter uses well-known optimising techniques such as code motion and code duplication, and tail duplication to improve fine-grain parallelism, and tail duplication and block merging to improve coarse-grain parallelism. However, this differs from other approaches in the way decisions are taken based on the notion of penalties which are applied to instructions.

In the case of code motion, for example, instructions are moved not only to reduce the penalties applied to them, but also to remove penalties located in the destination blocks. This idea is different from the global scheduler presented in [20]. This is basically a *global list scheduler*, where the ready list consists of instructions from a basic block and its equivalent blocks. If there are too many ready-instructions, then two heuristics are applied to each instruction to help selection. The *delay heuristic* obtains the maximum cumulative delay from any of the paths of the successors to the exit of the basic block; the *critical-path heuristic* computes a measure of time required to complete the execution of all the successors. The priorities of the global scheduler when code duplication is considered [18] also ensure that code expansion is minimised. However, the main difference with the PTD scheduler is that the global scheduler performs tail duplication to unscheduled code.

The global scheduler in [107] is also based on the list scheduler and considers the critical-path length, the critical resource usage and the register pressure as parameters in the heuristic. The critical-path length information is derived from local and global components. While the local component takes into account only the local distances to the leaves of the DAG, the global component considers the cumulative paths from all basic blocks to the exit of the region. Other global schedulers such as the trace-based ones also use the list scheduler for the final rearrangement. However, the candidate instructions from the ready list are selec-

Algorithm 6.3 ***move_up (node, source_block, dest_block)*** algorithm.

```
1:  aux_node = last_instruction (dest_block)
2:  if not independent (node, aux_node) then {CHECK FOR DATA DEPEND.}
3:     ind_nodes = 0
4:  end if
5:  if ind_nodes ≠ 0 then {CHECK FOR MEMORY DISAMBIGUATION.}
6:     if is_store (node) and is_memory (aux_node) or
          is_memory (node) and is_store (aux_node) then
7:        if not disambiguated (node, aux_node) then
8:           ind_nodes = 0
9:        end if
10:    end if
11: end if

12: if ind_nodes ≠ 0 then
13:    aux_node = prev (aux_node)
14: end if

15: move_node = aux_node
16: while aux_node ≠ NULL and ind_nodes ≠ 0 do
17:    if not independent (node, aux_node) then {CHECK FOR DATA DEPEND.}
18:       ind_nodes = 0
19:    else
20:       if previously_moved (aux_node) then
21:          if get_secondpos (aux_node) < get_actualpos (aux_node) then
22:             move_node = aux_node
23:             move_ahead (aux_node, secondpos (aux_node))
24:          end if
25:       end if
26:    end if
27:    if ind_nodes ≠ 0 then {CHECK FOR MEMORY DISAMBIGUATION.}
28:       if is_store (node) and is_memory (aux_node) or
             is_memory (node) and is_store (aux_node) then
29:          if not disambiguated (node, aux_node) then
30:             ind_nodes = 0
31:          end if
32:       end if
33:    end if

34:    if ind_nodes ≠ 0 then
35:       if check_global_move (node, aux_node) then
36:          move_node = update_best_position (aux_node)
37:       end if
38:    end if
39:    aux_node = prev (aux_node)
40: end while

41: move_ahead (node, move_node)
42: update_block (source_block)
43: update_block (dest_block)
```

Algorithm 6.4 ***check_global_move(node, aux)*** algorithm.

```
1: consecutive_before_move =
       penalty_data(prev(aux), next(aux), not consec) +
       penalty_data(aux, next(aux), consec) +
       penalty_data(aux, next(next(node)), not consec)
2: consecutive_after_move =
       penalty_data(prev(node), node, not consec) +
       penalty_data(aux, node, consec) +
       penalty_data(aux, next(node), not consec) +
       penalty_data(node, next(aux), consec) +
       penalty_data(node, next(next(aux)), not consec)
3: nonconsecutive_before_move =
       penalty_data(prev(prev(aux)), next(aux), 3) +
       penalty_data(prev(aux), next(aux), not consec) +
       penalty_data(prev(aux), next(next(aux)), 3) +
       penalty_data(aux, next(aux), consec) +
       penalty_data(node, next(node), consec) +
       penalty_data(node, next(next(node)), not consec)
4: nonconsecutive_after_move =
       penalty_data(prev(prev(aux)), node, 3) +
       penalty_data(prev(aux), node, not consec) +
       penalty_data(prev(aux), next(aux), 3) +
       penalty_data(aux, node, consec) +
       penalty_data(node, aux, consec) +
       penalty_data(aux, next(node),consec)
5: unit_before_move =
       penalty_unit(aux, next(aux)) +
       penalty_unit(prev(node), node) +
       penalty_unit(node, next(node))
6: unit_after_move =
       penalty_unit(aux, node) +
       penalty_unit(node, next(aux)) +
       penalty_unit(prev(node), next(node))

7: if consecutive_after_move ⩽ consecutive_before_move or
      nonconsecutive_after_move ⩽ nonconsecutive_before_move then
8:    return 1 {Penalty measure is smaller after the movement.}
9: else
10:   return 0 {Penalty measure is greater after the movement.}
11: end if
```

Algorithm 6.5 ***update_best_position (node, aux)*** algorithm.

```
1: penalties = number_penalties (aux)
2: if consecutive_before_move > consecutive_after_move then
3:   return aux
4: else
5:   if penalties > 1 then
6:     diff_consecutive = consecutive_before_move − consecutive_after_move
7:     if diff_consecutive ⩾ max_diff_consecutive then
8:       if diff_consecutive > max_diff_consecutive then
9:         max_diff_consecutive = diff_consecutive
10:      end if
11:      return aux
12:    else
13:      second_best = aux
14:    end if
15:  else
16:    if penalties ⩾ max_penalties then
17:      diff_nonconsecutive = nonconsecutive_before_move −
                               nonconsecutive_after_move
18:      if penalties > max_penalties then
19:        max_penalties = penalties
20:        if diff_nonconsecutive > 0 then
21:          return aux
22:        end if
23:      end if
24:    end if
25:  end if
26: end if
```

ted from the common path (on-trace) of the program, which may consider several basic blocks.

Tail duplication and block merging are considered as profile-independent optimisations. This means that they are not dependent on the data input to the programs. Different input data may not only exercise different paths of the program, but different frequencies as well. Trace-based techniques make use of this to guide the optimisations, but may require compilation when the input data changes the frequency of execution.

There is, however, an observation regarding the frequency of the paths with the optimisations discussed in this chapter. If, for example, the path $B_1 - B_3 - B_7''$ in the control flow graph in Figure 6.6 is executed ninety percent of the time, then the optimisations performed in blocks B_4, B_5, B_6 and B_7 will only benefit ten percent of the time the region is executed. For the control flow graph of Figure 6.7, it may appear that with the presence of a loop, the likelihood of the frequency execution of the basic blocks contained within the loop could be greater. Thus, the optimisations performed to blocks B_4, B_5, B_6 and B_7 may also have a greater impact on the program execution.

The global optimisations presented in this chapter are purely static. The performance improvement of these techniques is subject to the actual behaviour of programs, defined by their input data.

6.7 Summary

Compiler optimisations have to be selected depending on the degree of parallelism supported by the architecture. Aggressive optimisation techniques such as trace-based scheduling are oriented to VLIW architectures in which high levels of ILP can be exploited and data hazard detection is not supported. With the aid of profile information these techniques identify the most common paths during execution. Instructions are copied into the on-trace path in order to increase the parallelism exploitable by the local scheduler. Compensation code is often required to preserve the semantics of the program in the not-so-frequent paths. The performance gain in the on-trace path outweighs the degradation incurred in the off-trace path.

Micronet-based architectures require a fast instruction issue rate in order to maintain resource utilisation in the datapath. Global optimisation techniques that operate without speculation and without an excessive use of compensation code were investigated for the micronet architecture.

Chapter 5 described a novel method for performing local scheduling. The PTD scheduler assigns penalties to instructions that stall the issue unit either due to true data dependencies or resource contentions. The local scheduler reorganises the order of instructions in order to minimise the number of penalties, with the aim of minimising the effect of the stalls on program execution. The overall effect of the local scheduler is that the individual local penalty measures are reduced. These measures may not be completely reduced either because of lack of parallelism in the program or due to overlapping penalties in the code.

In this chapter, the local scheduler has been extended to allow movement of instructions beyond basic blocks. After local scheduling, the task of the global scheduler is to move the penalised instructions left in the basic block to other basic blocks within the same region. Instead of looking for independent instructions to reduce a penalty as in the local scheduler, the penalised instruction becomes a candidate for global motion. This decision reduces the scope of the search space.

Within the global scheduler, instructions can be moved to basic blocks that have the same control flow characteristics called *equivalent* basic blocks, as long as their data dependencies are respected. If they cannot be moved to any of the equivalent basic blocks, then the instructions are duplicated and moved to the parent basic blocks. This order in the scheduling process attempts to minimise the number of instances of code duplication.

Code motion and code duplication provide a fine granularity of optimisation since instructions are moved on an individual basis. Tail duplication, in conjunction with block merging on the other hand, represents a generalisation of code duplication, where coarse-grain parallelism can be exploited. This optimisation requires the local PTD scheduler to be applied after the transformation, since two groups of instructions are merged.

The optimisations described in this chapter do not require the use of profile information, which makes the optimised code independent of the input.

The next chapter describes the experimental framework for evaluating the local and global optimisations of the PTD scheduler. It includes a description of the compilation process and the benchmarks selected for the evaluation. The chapter draws comparisons with two well-known schedulers for both local and global scheduling, on the basis of issue stall reduction and performance execution over a set of benchmarks.

Chapter 7

Experimental Results

7.1 Introduction

The PTD scheduler performs local and global scheduling based on a static measure, which corresponds to the potential stall imposed on the issue unit by data and resource dependencies. The local scheduler attempts to minimise the stalls within basic blocks. The global scheduler reduces the penalties further after local scheduling, by relocating the penalised instructions to other basic blocks within the same region.

This chapter presents the experimental framework for investigating the effectiveness of both local and global optimisations using the PTD scheduler. Comparisons were carried out between the PTD scheduler and two well-known methods — the list scheduler [64] and a balanced scheduler [92].

The compilation framework for the work in this thesis is based on the SUIF compiler [155]. It provides a flexible implementation environment for schedulers aimed at micronet targets. The schedulers were exercised by benchmarks programs derived mainly from the `SPEC95` [152] benchmark suite.

The performance results presented in this chapter is divided into two sections: results based on the local optimisations as described in Chapter 5, such as comparisons of compilation times, reductions in issue unit stalls and execution times; and, those based on global optimisations, as described in Chapter 6.

7.2 Evaluation Framework

There exist a handful of compiler environments that provide the necessary infrastructure for performing custom transformations and analysis. Trimaran [169] is one which is oriented towards ILP optimisations. Trimaran is the product of the IMPACT group, a consortium consisting of the University of Illinois, the

CAR group at HP Research Laboratory and the ReaCT-ILP group at New York University. SGI Pro64 [147] is a suite of optimising compiler development tools resulting from a joint project between the compiler group at SGI and the CAPSL compiler team at the University of Delaware. The SUIF compiler [155] is yet another such compiler framework from Stanford University. Although the Trimaran framework suits our purposes, at the time of starting this research, SUIF was the only framework available.

7.2.1 SUIF Compiler

The SUIF (Stanford University Intermediate Format) is a compiler development framework from Stanford University [176]. It provides the necessary infrastructure to perform optimisations ranging from high-level transformations to dataflow optimisations. These transformations can be performed progressively and interchangeably over multiple passes, over structures represented as abstract syntax trees (AST) [156]. The ASTs can be converted into a series of sequential lists of instructions oriented for the back-end of the SUIF compiler, which supports code generation, local scheduling and register allocation.

7.2.2 The Compilation Process

During compilation a number of intermediate code optimisations are performed ahead of the code generation phase, which are part of the built-in optimisations included in the SUIF compiler, such as *no_struct_copy*, *no_sub_vars*, *no_call_expr*, *no_index_spill*, *copy_prop* (copy propagation) and *ivar* (induction variable detection) [154].

During the code generation phase, register allocation is called with the following options: *infinite temporary* registers and *finite saved* registers. The temporary registers represent the results that are local to a procedure call that have a single definition and are used within the same region. The *saved* registers represent ones that are preserved across procedure calls [91]. With *infinite* number of temporary registers, a new register is allocated each time a result is generated. This configuration is used in order to minimise the WAR dependencies introduced by reusing the registers (the task of the register allocator), an effect that reduces considerably the ILP.

The results presented in [8] (Appendix A) were obtained from compilation with both temporary and saved register models as finite, *i.e.* register allocation was performed before local scheduling. As a result, the ILP reported was noticeable limited by the WAR dependencies; the overall ILP gain of performing instruction

scheduling after register allocation was hindered by the systematic reuse of the registers from the latter. This is the main reason as to why setting an infinite number of registers in order to evaluate the scheduling capabilities of the PTD scheduler can be considered as a valid assumption. The order in which register allocation and instruction scheduling are performed affects the schedulers being evaluated in a similar way since the ILP within basic blocks stays constant, and therefore does not represent an advantage for the PTD scheduler.

All the scheduling results presented in this chapter are obtained with an infinite number of temporary registers being assumed.

7.2.3 Other Schedulers for Comparison

7.2.3.1 The Gibbons and Muchnick (GM) Scheduler

This is a well-known example of a list scheduling algorithm proposed originally for scheduling instructions in pipelined architectures [64]. The ready instructions in this scheduler are prioritised on the basis that the candidate instruction will not cause an interlock with the previous one, and that given a choice, the candidate instruction is more likely to interlock with instructions after it.

7.2.3.2 The Balanced Scheduler

The second scheduler for comparison is the Balanced scheduler [92], which was originally devised to take account of unpredictable memory access latencies. The idea is to compute weights for load instructions based on the number of available independent instructions. The instructions are scheduled, as in a traditional list scheduler, with independent instructions being distributed behind loads to buffer against unpredictable memory accesses. This idea is generalised to micronet-based architectures in which all the instructions have unpredictable latencies. The priority for ready instructions is based on a weighted sum of values derived from heuristics tailored to the micronet architecture. These include whether the instruction uses the same resources as the previous scheduled one, the number of immediate successors of the instruction and the length of the longest path from the instruction to the leaves of the DAG. The heuristics also include the number of source registers which are freed should the instruction be scheduled, which effectively takes account of register pressure.

During the code generation phase, the local scheduler as part of the SUIF compiler was not used since it is tailored towards *synchronous* MIPS processors. Furthermore, the GM scheduler represents an equivalent implementation of a

list scheduler and has been extended to optimise instructions in micronet-based architectures.

7.2.4 Instruction-level Simulator for the Micronet Architecture

The simulator for the micronet architecture, as described in Section 4.3.6, is an event-driven stochastic simulator [97] that reads and executes assembly instructions generated from the SUIF compiler. Each instruction is associated with a handful of events that emulate the necessary stages in the datapath for its execution. The events are created dynamically and their latency depend on the type of instruction and on resource contentions at run-time. The events from neighbouring stages communicate with each other asynchronously.

A configuration file enables the simulator to emulate a number of stages for each group of instructions, *e.g.* arithmetic, logical or memory. The simulator models a scalar architecture with a single issue unit which issues instructions in-order as soon as the instructions' operands and resources become available. When instructions are issued, they progress at their own pace. This allows for instructions to be overtaken, and since the write-back stage is not reordered, they can commit their results out-of-order.

7.2.5 Evaluation Process

Figure 7.1 shows the overall view of the evaluation process. Firstly, C programs are compiled using SUIF; secondly, a loader program converts the resulting assembly code so that the global memory references and labels fit into a global referencing scheme, and lastly, this output is fed into the instruction-level simulator of the micronet architecture [97] for evaluation. This path is considered to be the *base* case since the code is not scheduled after code generation. In a different path, the output from SUIF is fed into a scheduling phase, which is performed using one of the following: GM, Balanced or the PTD scheduler. The scheduled output is loaded for simulation, as before.

The output of the simulator provides comprehensive information about the execution of the instructions, which includes the number of instructions executed, the total time of the simulation, the stall time of the issue unit, the values of both the registers and memory, and the time spent in each resource.

The set of benchmarks were each scheduled by the three schedulers. The benchmarks were each simulated five times and their makespans were averaged. All the comparisons presented in this chapter were normalised against the base case.

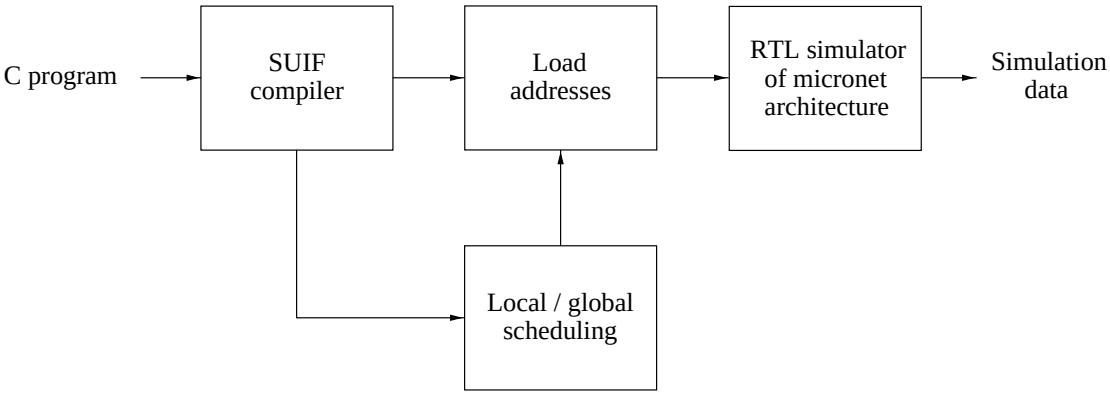


Figure 7.1: Flow of the evaluation process.

The simulations were performed on architectures consisting of one memory unit, one logical unit, one floating point unit and one, two, three and four arithmetic units (referred to as 1 AU, 2 AU, 3 AU and 4 AU, respectively). The architectures also scale the number of register read buses to support the parallelism introduced by increasing the number of functional units. The latencies for the functional units are the same shown in Table 4.1. The cache model used for the memory unit describes a bimodal distribution, *i.e.* a cache hit:miss ratio of 2:1 and a latency hit:miss ratio of 1:2.

7.3 Benchmarks

The benchmarks were a basket of programs drawn from the `SPEC95` suite [152], including `compress`, `li`, `go` and `m88k`, from the `livermore` loops [49], and miscellaneous ones, such as integer matrix multiplication `intmm`, `puzzle`, and `fract` which simulates a variation of the Mandelbrot fractal set[1]. Table 7.1 gives some basic characteristics of the benchmarks, such as the numbers of MIPS instructions executed, functions and the basic blocks generated by the compiler.

The `SPEC95` benchmarks were modified to execute the relevant subset of their full specification. For example, the `compress` benchmark would normally compress and decompress twenty-five times, a random set of data of 1.4 million characters. The benchmark was modified to operate only three times instead, over a set of fourteen thousand characters. The `go` benchmark, which is based upon

[1]The equation was taken from the *hydra* fractal [65], and its graphic representation can be seen in [66].

Benchmark	Functions	Basic Blocks	Instructions	Initialisation phase
`intmm`	6	11	196,074	12.000 %
`livermore`	16	48	1,601,672	84.504 %
`fract`	70	505	3,392,216	2.421 %
`li`	386	2,228	15,207,508	2.007 %
`puzzle`	20	137	16,055,217	0.509 %
`compress`	18	128	16,269,122	20.847 %
`go`	396	8,880	18,242,718	7.906 %
`m88k`	259	3,841	34,323,842	7.504 %

Table 7.1: Benchmark characteristics.

the game, *Many faces of Go*, was modified to reduce the size of the board from 51 to 4. The `m88k` benchmark, a simulator for the M88100 Motorola microprocessor, was modified to reduce the total number of M88100 instructions to ten thousand from around 13.6 million. Finally, the `li` benchmark is a lisp interpreter and the workload used was a lisp implementation of the *queens* problem.

The main reason for these modifications was to shorten the simulation time. The simulator is data-driven (as opposed to trace-driven), and its speed is considerably slower, since all the instructions are being executed. However, truncating the length of the simulation can be misleading as the execution times may be dominated by the initialisation sections of the programs. Initialisation sections could either be very regular, offering scope for more parallelism, or they could have extensive I/O operations, resulting in a much slower pattern of execution. The last column of Table 7.1 shows the percentage of instructions dedicated to initialisation phase for each benchmark. These values have been gathered after simulating all the programs exclusively with their initialisation functions enabled. Only the initialisation section of the `livemore` benchmark consumes a significant percentage of the program. The consideration to reduce the total number of instructions executed in the `SPEC95` benchmarks was chosen carefully to maintain low initialisation percentages.

Further modifications to the benchmarks were required to remove all instances of dynamic allocation of memory (with the exception of the `go` and `compress` benchmarks which do not allocate any variables dynamically). All the variables that were normally allocated dynamically were redefined to be static, removing

Benchmark	Arithmetic	Logic	Memory	Floating	Branch
`intmm`	70.11 %	5.13 %	20.04 %	0.00 %	4.72 %
`livermore`	73.75 %	6.60 %	13.35 %	0.00 %	6.30 %
`fract`	26.40 %	8.22 %	19.64 %	42.85 %	2.89 %
`li`	21.50 %	23.30 %	41.21 %	0.05 %	13.94 %
`puzzle`	52.08 %	22.12 %	11.43 %	0.00 %	14.37 %
`compress`	38.52 %	26.64 %	24.56 %	3.10 %	7.18 %
`go`	46.29 %	20.76 %	18.87 %	0.00 %	14.08 %
`m88k`	36.07 %	24.62 %	27.86 %	0.00 %	11.45 %
Average	45.59 %	17.17 %	22.12 %	5.75 %	9.37 %

Table 7.2: Distribution of types of instructions in the benchmarks.

the overhead of a dynamic memory manager and simplifying the implementation of the simulator.

Another minor modification was to include the source code from standard libraries (`stdlib`, `stdio`, `string`, `setjmp`, `stdarg` and `varargs`) in the benchmark programs before compilation. Normally, this code is only added during the linking process. However, in SUIF even with the 'static compilation' flags enabled, the code derived from the standard libraries could not be generated.

The outputs from the benchmarks when compiled using SUIF and simulated on the MAP simulator, were compared to the outputs from the same benchmarks when compiled using **cc** under Unix; the outputs were corroborated to confirm correct compilation and execution.

The distribution of the instructions types for the benchmarks is listed in Table 7.2. Arithmetic instructions dominated followed by memory instructions. Some of the benchmarks required floating-point instructions; for instance, the `compress` benchmark uses them for the random selection of the input data, and the `li` benchmark uses a few for comparisons. The `fract` benchmark cannot be considered to be a truly integer benchmark as a major proportion of its instructions were floating-point ones; the core body of the benchmark performs floating-point operations over *complex* numbers.

The benchmarks can be divided into two types: those dominated by the loop sections, *e.g.* the `intmm`, `livermore` and `fract` benchmarks, and others, which are dominated by frequent changes in the control flow, *e.g.* `puzzle`, `go` and `li` benchmarks.

Benchmark	Total lines	GM. Time	Bal. Time	PTD Time	Percen.
`intmm`	151	0.0836	0.0839	0.0680	22.94 %
`livermore`	1915	3.7947	3.8388	2.7817	36.42 %
`fract`	5336	3.5940	3.5588	2.1920	62.35 %
`li`	16832	8.7857	8.1169	4.4349	83.02 %
`puzzle`	936	0.7781	0.6369	0.4711	35.19 %
`compress`	1236	0.9757	0.7300	0.5494	32.87 %
`go`	83838	65.2829	60.1321	44.0446	36.53 %
`m88k`	34089	22.0481	20.6159	12.6411	63.09 %

Table 7.3: Average benchmark compilation times (in seconds).

Benchmark	GM. Time	Bal. Time	PTD Time
`intmm`	0.00013	0.00046	0.01183
`livermore`	0.06702	0.03591	0.11787
`fract`	0.03355	0.01517	0.16728
`li`	0.10623	0.06615	0.09527
`puzzle`	0.16317	0.00607	0.01297
`compress`	0.24775	0.00344	0.02311
`go`	5.96753	2.90291	0.18597
`m88k`	0.63226	0.06168	0.10197

Table 7.4: Standard deviations of the benchmarks' compilation times.

7.4 Experimental Results

7.4.1 Local Optimisations

7.4.1.1 Complexity

The PTD scheduler has the characteristic of reducing the number of iterations as the penalties decrease (*c.f.* Section 5.6). This compares favourably with traditional techniques in which the number of iterations is constant and proportional to the number of instructions.

The compilation of the benchmarks were timed for comparison. Table 7.3 lists the average of five compilation times (in seconds) of the scheduling sections in the three schedulers. (This was obtained using the `gethrtime` function from the `time.h` standard library). The last column represents the percentage improve-

ment of the PTD scheduler against the faster of the other two schedulers. It is observed that the compilation times of PTD are on average 39% better than the faster of the other two schedulers, with notable improvement of more than 60% for the `fract` and `m88k` benchmarks, and reaching a peak of 83% for the `li` benchmark. Table 7.4 lists the standard deviation of the compilation times for the three schedulers. The figures shown in this table reveal that the compilation times do not have a considerable range of variation.

7.4.1.2 Static Memory Disambiguation

A static memory disambiguation scheme was proposed in Section 5.5.1 in order to reduce unnecessary data dependencies due to memory instructions, and the statistics from this scheme are shown in Table 7.5. The memory references shown in the second column represent the number of memory instructions that are liable for disambiguation, *i.e.* after discarding all the comparisons between loads and any other memory references that already have a data dependency. The last three columns show the results of applying the memory disambiguation scheme within basic blocks. The last column shows that most of the memory instructions are disambiguated. In fact, on average, only around 8% of the memory references do have a data dependency (third column). The table also shows that the proportion of memory references that cannot be disambiguated due to multiple use of registers in the address expressions is quite low, at around 2% (fourth column).

The memory disambiguation mechanism can potentially improve the parallelism by not only reducing unnecessary dependencies being applied to memory operations, but also by removing all the data dependencies that are introduced to their successors. In the absence of memory disambiguation, memory instructions are by default, assumed to be dependent. Consequently, other instructions that are dependent upon them, also become dependent. By analysing the addresses, the memory disambiguation removes a considerable proportion of these dependencies which are propagated by references.

The increase in parallelism achieved by applying the memory disambiguation scheme is discussed in Section 7.4.1.4.

7.4.1.3 Subgraphs

The idea of subgraphs was introduced in Section 5.5.2. Recall that partitioning a basic block into a group of subgraphs aims to guide the selection of independent instructions when reducing the penalty. The practice of selecting instructions

Benchmark	Memory references	Data dependency	Cannot be disambiguated	Successful disambiguation
`intmm`	21	14.29 %	0.00 %	85.71 %
`livermore`	1,366	3.95 %	5.93 %	90.12 %
`fract`	2,998	8.97 %	0.47 %	90.56 %
`li`	9,704	3.70 %	1.44 %	94.86 %
`puzzle`	156	1.92 %	0.00 %	98.08 %
`compress`	614	23.61 %	0.33 %	76.06 %
`go`	22,911	3.32 %	2.09 %	94.59 %
`m88k`	16,052	4.73 %	2.40 %	92.87 %
Average	6,727	8.06 %	1.58 %	90.36 %

Table 7.5: Static memory disambiguation statistics.

within the same subgraph may lead to overlapping penalties. The movement of the closest independent instruction tends to mix the data dependencies in such a way that no further reductions can be made.

The PTD scheduler is now forced to find an independent instruction from another subgraph to reduce the penalty. This reduces the number of instances of overlapping penalties. However, the decision to either choose, or not to, an independent instruction from one subgraph over another, leads to different *scheduling paths*. A scheduling path refers to the steps of progressively improving a schedule until its minimum penalty measure is reached. Although the scheduler is conceived to obtain a schedule with the minimum penalty measure, but due to these decisions, one scheduling path can result in a considerably better schedule than another.

The evaluation of this heuristic is based on performance simulations which is presented in the next section.

7.4.1.4 Performance Comparison due to Memory Disambiguation and Subgraphs

The performance comparison of the memory disambiguation scheme and the idea of subgraphs as applied to the PTD scheduler is displayed in Figures 7.2 to 7.5, for one to four AU configurations, for the eight benchmarks.

The graphs show the performance improvement in execution time of the PTD scheduler when unaided (first column), with either one of subgraphs or memory

disambiguation schemes being applied (second and third columns, respectively), and finally, with both schemes (fourth column). The results show that, in general, applying the subgraphs heuristic and memory disambiguation results in better performances for the four configurations.

The figures show that memory disambiguation helps to improve the performance of the PTD scheduler by reducing dependencies, and exposing parallelism. The most noticeable improvements are in the cases of the `compress`, `fract` and `li` benchmarks. On average, this scheme improves by 3% to 4% when compared to the unaided cases.

However, there are a few examples in which the combined use of both the schemes does not result in the best performance. For example, the `livermore` benchmark performs better (except for the 1 AU case) when the subgraphs scheme alone is applied. This benchmark, in particular, spends a considerable amount of time in the initialisation phase, as shown in Table 7.1. This section features three basic blocks with similar characteristics, in which three arrays are initialised. The DAG of one of these basic blocks is depicted with (Figure 7.7) and without (Figure 7.6) memory disambiguation.

The basic block has three store operations that initialise the arrays. In the absence of memory disambiguation, the memory operations, which are marked in the figures, have to be considered to be dependent, as shown in Figure 7.6. Conversely, when aided by the scheme, they are found to be independent — referring to three different arrays, `x`, `y` and `z`, as shown in Figure 7.7.

The reason why the PTD scheduler performs better when not applying memory disambiguation, lies in its critical path. In Figure 7.6, the critical path is $I_1 - I_3 - I_5 - I_{10} - I_{15} - I_{19}$, which is imposed by the dependencies between the memory instructions.

Once the dependencies between the stores are released thanks to memory disambiguation, it is less clear which is the critical path. Figure 7.7 shows that there are three candidates in the DAG: $I_1 - I_3 - I_5 - I_{19}$, $I_6 - I_8 - I_{10} - I_{19}$, and $I_{11} - I_{13} - I_{15} - I_{19}$.

The schedule, when generated with the memory disambiguation enabled, still has an untouched non-consecutive penalty from I_3 (`addu $398,$399,$400`) to I_5 in one of the critical paths (as can be seen in Figure 7.8 (a)). In contrast, although there are two penalties which are untouched in the schedule generated unaided by memory disambiguation, as shown in Figure 7.8 (b), these non-consecutive penalties are not located in any of the critical paths mentioned earlier.

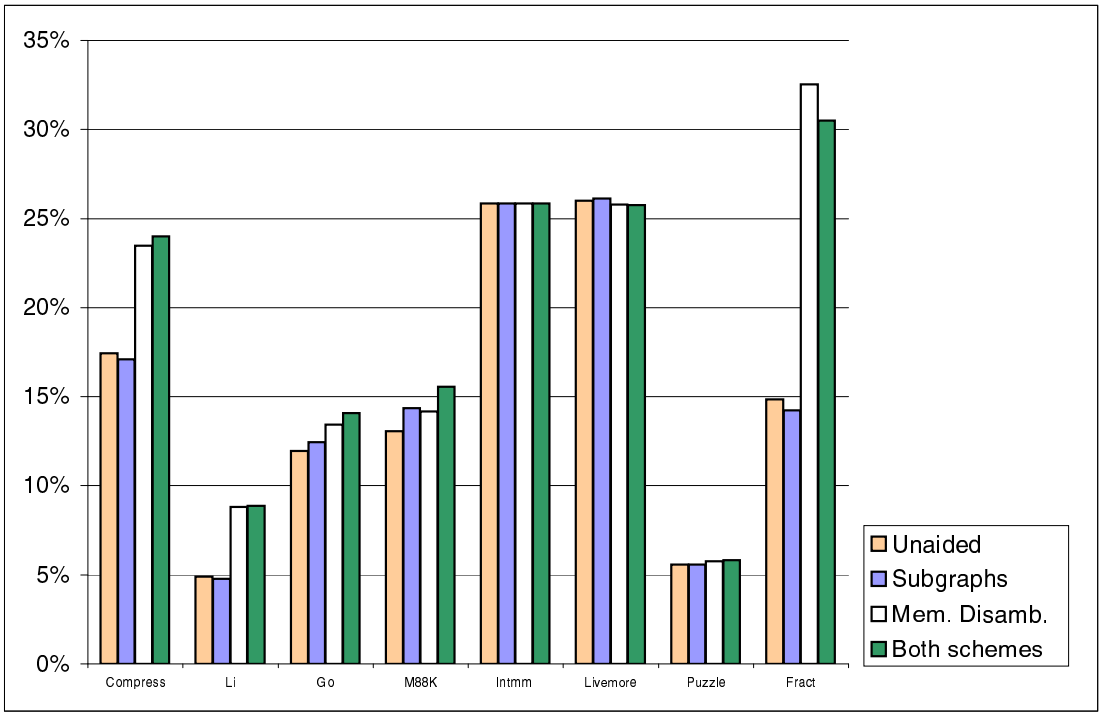


Figure 7.2: Influence of subgraphs and memory disambiguation on the PTD scheduler (1 AU) in terms of percentage improvement in the execution time.

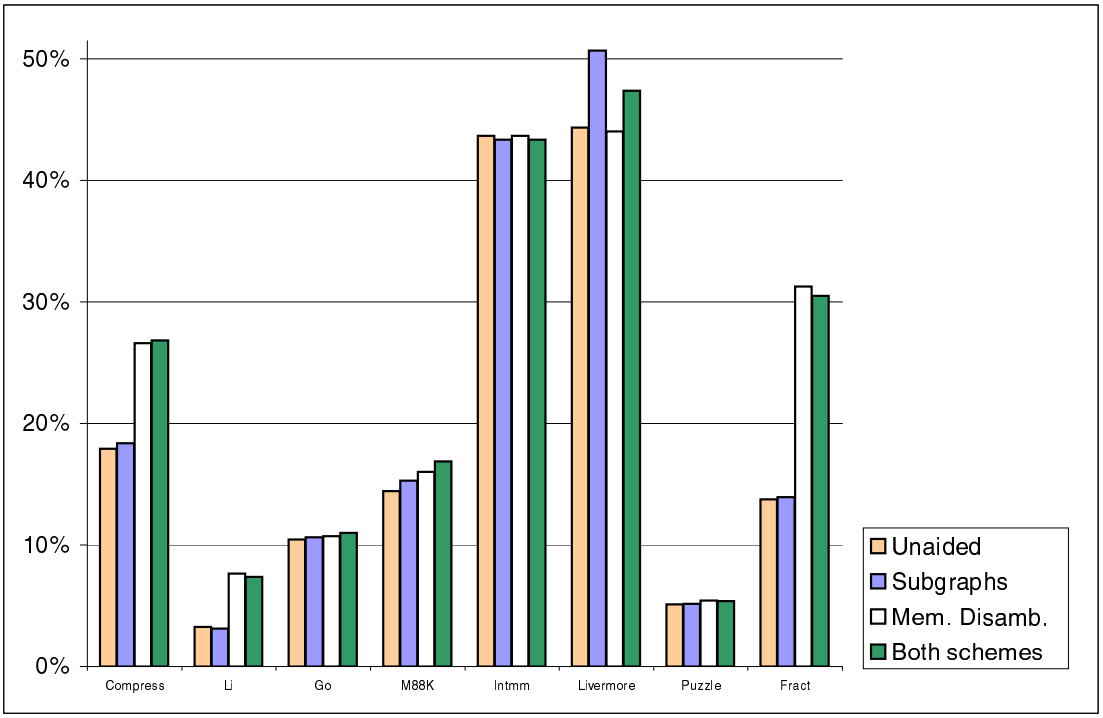


Figure 7.3: Influence of subgraphs and memory disambiguation on the PTD scheduler (2 AU) in terms of percentage improvement in the execution time.

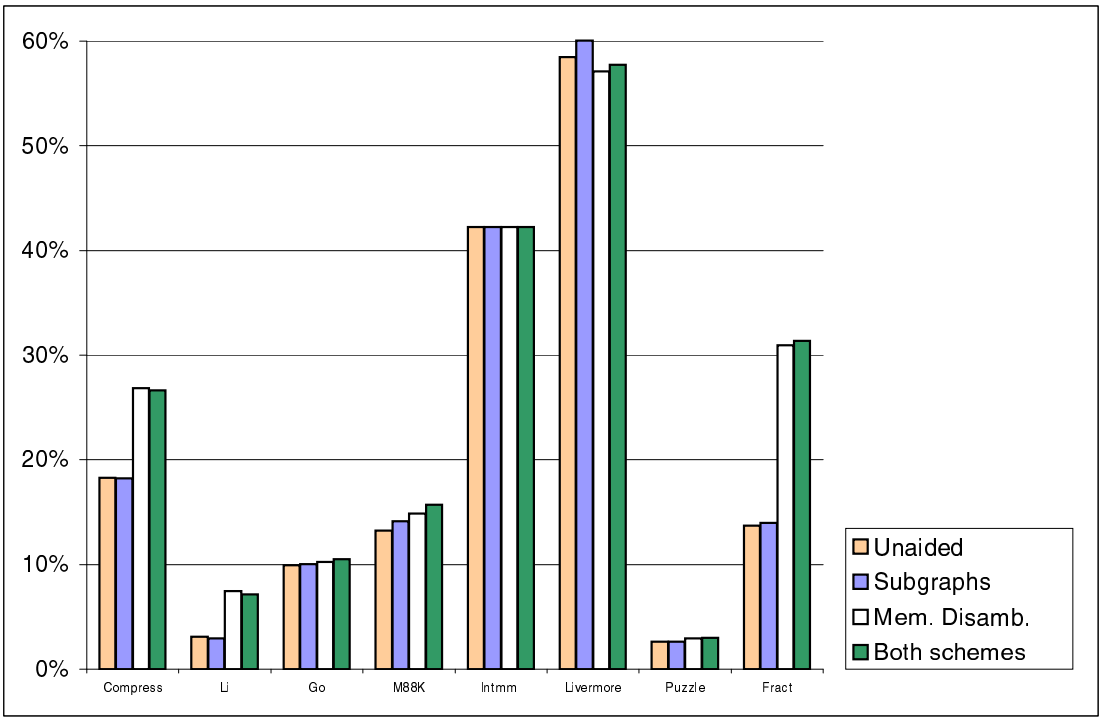


Figure 7.4: Influence of subgraphs and memory disambiguation on the PTD scheduler (3 AU) in terms of percentage improvement in the execution time.

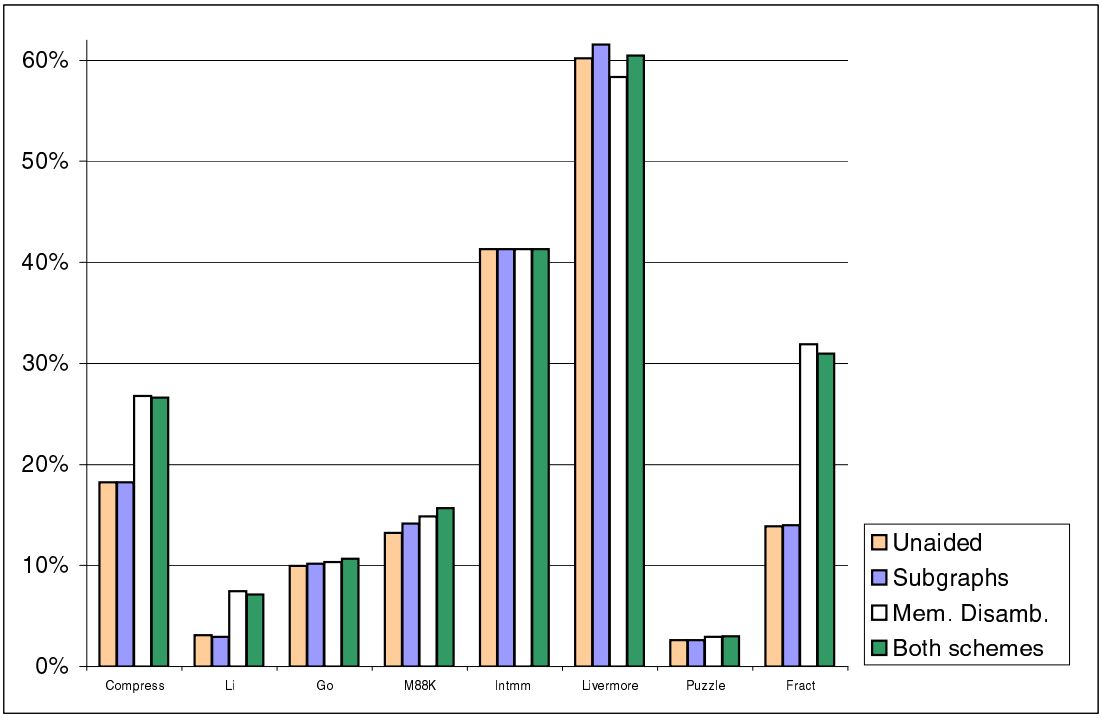


Figure 7.5: Influence of subgraphs and memory disambiguation on the PTD scheduler (4 AU) in terms of percentage improvement in the execution time.

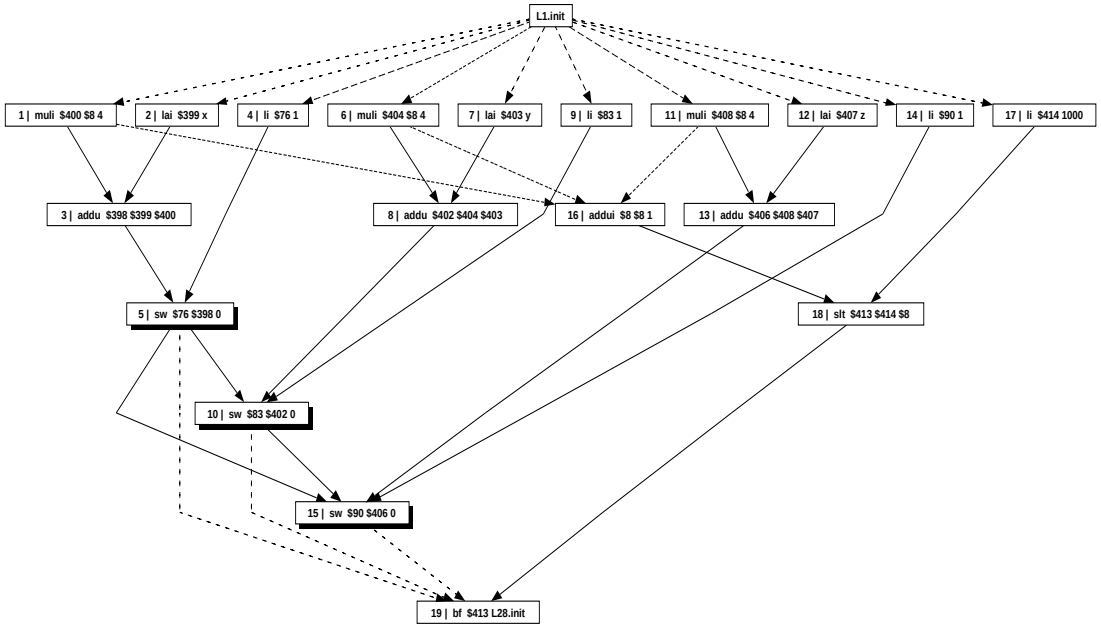


Figure 7.6: DAG from `livermore` without memory disambiguation.

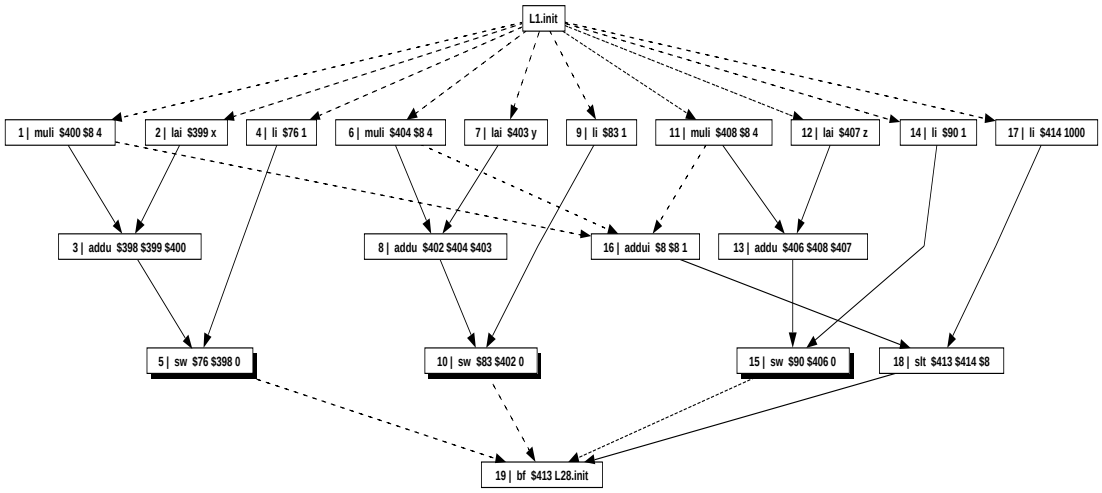


Figure 7.7: DAG from `livermore` with memory disambiguation.

Given that the DAG in Figure 7.7 has several critical paths, more penalties need to be removed. Any priority scheme applied to the penalties would give preference to one critical path over another. On the other hand, the DAG in Figure 7.6 has only one critical path. The increase of data dependencies upon the memory instructions does seem to constraint the movement of instructions. The evidence is that more penalties are left unremoved. However, they are less costly since they are not on the critical path.

The results in Figures 7.2 to 7.5 also show that the `fract` benchmark per-

```
L1.init:
        lai     $403,y
        li      $76,1
        muli    $408,$8,4
        lai     $407,z
        li      $83,1
        muli    $404,$8,4
        lai     $399,x
        addu    $406,$408,$407
        li      $90,1
        muli    $400,$8,4
        addu    $402,$404,$403
        addui   $8,$8,1
        li      $414,1000
        sw      $90,$406,0
        addu    $398,$399,$400   1
        sw      $83,$402,0
        slt     $413,$414,$8     2
        sw      $76,$398,0
        bf      $413, L1.init
```

(a)

```
L1.init:
        muli    $400,$8,4
        lai     $399,x
        li      $76,1
        muli    $404,$8,4
        lai     $403,y
        addu    $398,$399,$400
        li      $83,1
        muli    $408,$8,4
        sw      $76,$398,0
        addu    $402,$404,$403
        lai     $407,z           1
        addui   $8,$8,1
        li      $414,1000
        addu    $406,$408,$407
        li      $90,1            1
        sw      $83,$402,0
        slt     $413,$414,$8     2
        sw      $90,$406,0
        bf      $413, L1.init
```

(b)

Figure 7.8: Schedule for `livermore` generated with memory disambiguation (a), and, without (b).

forms better when the subgraphs heuristic is not applied to the PTD scheduler. To explain this case, a basic block from the core of the `fract` benchmark was analysed. The DAG, which is displayed in Figure 7.9, shows that the basic block is dominated by memory instructions. The marked instructions in the figure show the main subgraph.

As the scheduler does not allow for memory instructions to be moved to reduce penalties, it is more difficult to find a good candidate when the majority of instructions are of the aforementioned type. Moreover, the subgraph introduces more constraints when searching for a candidate, which further limits the possibilities for reduction.

Figure 7.10 (a) shows the code produced by the PTD scheduler when aided by the subgraphs heuristic. The code has two non-consecutive penalties left unreduced (I_8 and I_{11}), as opposed to one (I_8), in the code produced unaided by subgraphs (Figure 7.10 (b)).

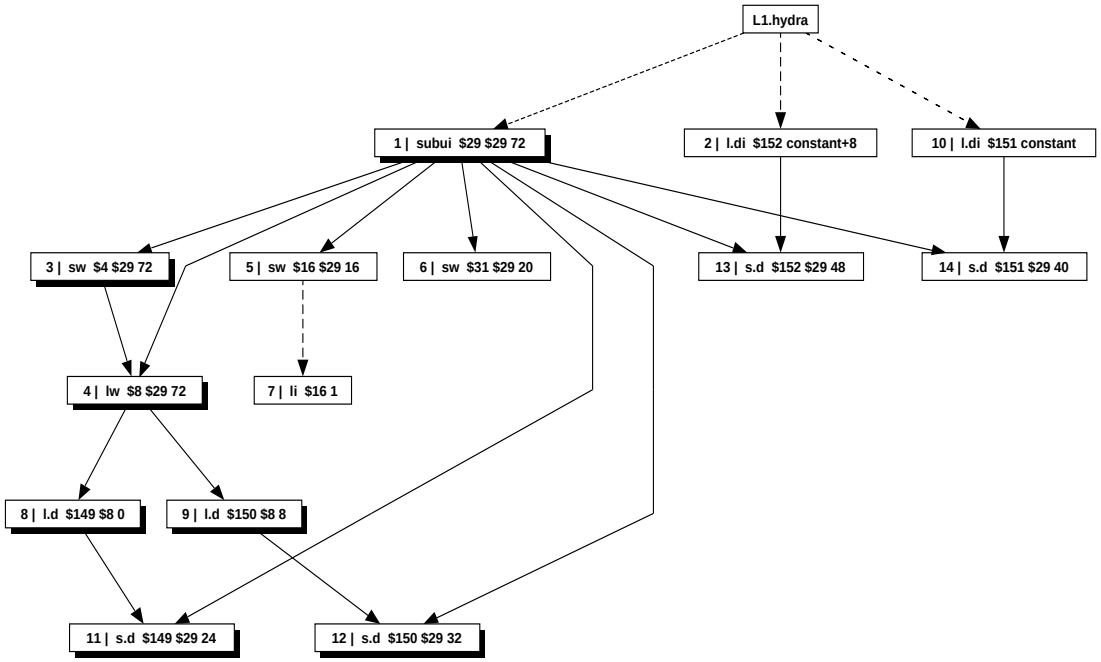


Figure 7.9: DAG of the `fract` benchmark with subgraphs being applied.

These examples show that a lower penalty measure with equally distanced penalties do not necessarily produce a better schedule, and hence a shorter stall in the issue unit. Furthermore, since these basic blocks are executed considerably more often than the rest of the program, the effects of the penalties are magnified, and hence, the program execution time is lengthened.

7.4.1.5 Issue Stall

The central idea in the PTD scheduler is to minimise the instruction issue stall and thereby maximise the issue rate. The penalties in a schedule relate to the amount of stall in the issue unit due to data dependencies and resource contentions.

Tables 7.6, 7.7, 7.8 and 7.9, which correspond to 1 AU, 2 AU, 3 AU and 4 AU configurations respectively, show the reduction of time spent on issue stalls achieved by the three schedulers compared to the base case, *i.e.* the unscheduled code. They represent the percentage improvement with respect to the unscheduled code simulated with one, two, three and four arithmetic units.

It can be observed that the PTD scheduler outperforms the other two schedulers for the 1 AU and 2 AU configurations, and is competitive in a majority of the cases, for the 3 AU and 4 AU configurations.

```
L1.hydra:
        subui   $29,$29,72
        l.di    $151,constant
        sw      $4,$29,72
        lw      $8,$29,72
        sw      $16,$29,16
        sw      $31,$29,20
        l.di    $152,constant+8
        l.d     $149,$8,0        1
        l.d     $150,$8,8        1
        li      $16,1
        s.d     $149,$29,24
        s.d     $150,$29,32
        s.d     $151,$29,40
        s.d     $152,$29,48
```

(a)

```
L1.hydra:
        subui   $29,$29,72
        l.di    $152,constant+8
        l.di    $151,constant
        sw      $4,$29,72
        lw      $8,$29,72
        sw      $16,$29,16
        l.d     $149,$8,0        1
        l.d     $150,$8,8
        sw      $31,$29,20
        s.d     $149,$29,24
        s.d     $151,$29,40
        s.d     $150,$29,32
        li      $16,1
        s.d     $152,$29,48
```

(b)

Figure 7.10: Schedule generated with (a), and, without subgraphs (b), for a portion of the `fract` benchmark.

The effect of overlapping penalties in the PTD scheduler is demonstrated by lower improvement in the percentage reduction of stalls, as the architecture is scaled. It can be seen that in some cases, such as `puzzle`, `compress` and `go`, the reduction of issue stalls cannot achieve the same performance figures as the other schedulers.

Figures 7.11 to 7.14 expand the results from the aforementioned tables and show the percentage improvement in the issue stall when broken down by its causes. These causes are divided into ones due to general data dependencies (`Data`), data dependencies due to a branch instruction (`Branch`), resource contention of the read buses (`Bus`) and resource contention due to a functional unit (`Rsc`).

The figures show that the three schedulers are successful in reducing the issue stalls due to data dependencies (`Data` and `Branch`), — the bars always lie in the positive half of the axis. This reflects that unscheduled code frequently stalls the issue unit due to data dependencies, where an instruction consumes the result from a previous one immediately.

Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	26.08 %	22.17 %	33.38 %
`livermore`	24.48 %	28.68 %	34.27 %
`fract`	28.47 %	34.41 %	39.40 %
`li`	8.55 %	9.73 %	10.67 %
`puzzle`	6.39 %	3.65 %	7.07 %
`compress`	24.06 %	25.35 %	30.12 %
`go`	11.46 %	11.60 %	17.54 %
`m88k`	14.54 %	15.17 %	19.35 %
Average	18.00 %	18.85 %	23.98 %
Geo. Mean	15.85 %	15.58 %	20.79 %

Table 7.6: Percentage reduction in the issue stall by the schedulers (1 AU).

Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	59.00 %	50.45 %	63.88 %
`livermore`	67.23 %	57.57 %	77.23 %
`fract`	31.61 %	37.13 %	40.32 %
`li`	7.82 %	8.59 %	8.97 %
`puzzle`	8.12 %	2.15 %	6.76 %
`compress`	35.40 %	36.51 %	35.70 %
`go`	13.75 %	14.39 %	14.13 %
`m88k`	19.36 %	20.03 %	21.47 %
Average	30.29 %	28.35 %	33.56 %
Geo. Mean	22.88 %	19.51 %	24.52 %

Table 7.7: Percentage reduction in the issue stall by the schedulers (2 AU).

As the code is optimised, the issue unit stalls shift from data dependencies to resource contentions (`Bus` and `Rsc`), because the functional units and their buses become busier. This is reflected in the graphs by their negative effect on the issue stalls, when compared to the base case. In Figure 7.11 the stall due to buses is still considerable, although the overall improvement is positive. As the architecture scales (Figures 7.12, 7.13 and 7.14) the stalls due to `bus` contentions become less important; in fact, the simulations with the 4 AU configuration in Figure 7.14 show that these stalls are practically negligible.

Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	66.63 %	58.03 %	63.24 %
`livermore`	102.74 %	91.08 %	100.84 %
`fract`	32.01 %	37.90 %	41.65 %
`li`	7.76 %	8.41 %	8.70 %
`puzzle`	3.90 %	2.55 %	3.73 %
`compress`	37.18 %	38.30 %	35.61 %
`go`	14.27 %	14.85 %	13.54 %
`m88k`	19.41 %	20.20 %	20.15 %
Average	35.49 %	33.91 %	35.93 %
Geo. Mean	22.61 %	21.71 %	23.19 %

Table 7.8: Percentage reduction in the issue stall by the schedulers (3 AU).

Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	68.06 %	58.90 %	62.21 %
`livermore`	113.46 %	101.36 %	107.05 %
`fract`	32.02 %	37.91 %	41.05 %
`li`	7.71 %	8.40 %	8.69 %
`puzzle`	3.87 %	2.53 %	3.73 %
`compress`	37.31 %	38.47 %	35.60 %
`go`	14.38 %	14.88 %	13.70 %
`m88k`	19.28 %	20.05 %	20.09 %
Average	37.01 %	35.31 %	36.52 %
Geo. Mean	22.93 %	22.01 %	23.30 %

Table 7.9: Percentage reduction in the issue stall by the schedulers (4 AU).

The stalls due to functional units contentions (`Rsc`) are also reduced as the architecture scales; however, their reduction is not as clear cut as bus contentions. This is because the scaling is confined only to arithmetic functional units. Benchmarks with more memory instructions such as the `li` cannot take advantage of the larger amount of parallelism in the architecture. The `fract` benchmark has the same limitation, since it is dominated by floating point instructions (as shown in Table 7.2). When the `fract` benchmark was simulated with four floating point units, the average reduction in the issue stalls due to resource contentions went down from -17.91% (one FPU) to -9.80% (four FPU), in the case of the PTD scheduler.

Conf.	Scheduler	Data	Branch	Bus	Rsc.	Cumul.
1 AU	GM.	37.33 %	2.56 %	−3.13 %	−18.76 %	18.00 %
	Bal.	34.32 %	2.49 %	−2.80 %	−15.16 %	18.85 %
	PTD	41.43 %	3.46 %	−8.61 %	−12.30 %	23.98 %
2 AU	GM.	40.68 %	3.09 %	4.99 %	−18.47 %	30.29 %
	Bal.	35.22 %	3.10 %	4.94 %	−14.91 %	28.35 %
	PTD	41.29 %	2.52 %	4.00 %	−14.25 %	33.56 %
3 AU	GM.	43.14 %	3.70 %	1.39 %	−12.74 %	35.49 %
	Bal.	38.97 %	3.51 %	1.42 %	−9.99 %	33.91 %
	PTD	39.94 %	2.87 %	1.48 %	−8.36 %	35.93 %
4 AU	GM.	43.62 %	3.99 %	−0.08 %	−10.52 %	37.01 %
	Bal.	39.74 %	3.71 %	−0.07 %	−8.07 %	35.31 %
	PTD	39.60 %	3.14 %	−0.17 %	−6.05 %	36.52 %

Table 7.10: Average issue stall improvements for the four configurations.

The `livermore` benchmark, on the other hand, is the benchmark with the highest number of arithmetic instructions. The large negative percentages for the three schedulers shown in Figure 7.14 represents scope for further improvement. In fact, when the benchmark was simulated with an additional arithmetic functional unit, the percentage of issue stall is reduced from −11.22% (with four arithmetic units) to −10.18%[2], in the case of the PTD scheduler.

A distinctive characteristic of the PTD scheduler is that it does not reduce the issue stall as much by data dependencies compared to the other two schedulers, but it reduces the issue stalls due to resource contentions significantly more than the two. This pattern can be seen clearly in Figure 7.12 with the `compress` benchmark. Both the GM scheduler and the Balanced scheduler have a significant effect on the issue stalls due to data dependencies (`data`), by almost 50%. The PTD scheduler does not show the same increase (almost 40%), but it reduces the resource stalls (`rsc`) to around −1%, whereas the other two schedulers are less effective in reducing to around −10%.

This can be explained as follows. On one hand, the effects of the residual overlapping penalties restrains the PTD scheduler in improvements to stalls due to data dependencies. On the other hand, the PTD scheduler tackles resource

[2]The remaining stalls are due to memory operations only.

dependencies by applying penalties to consecutive instructions of the same type when there are not enough functional units of that type. The net result is that the overall percentage improvement of the issue stall due to applying the PTD scheduler compares well with the other two.

Table 7.10 gives a summary of the average issue stall improvements for the set of benchmarks. The table reinforces the pattern that considerable improvements are achieved on the stalls due to data dependencies (`Data` and `Branch` columns), but degradation in stalls due to resource contentions (`Bus` and `Rsc.` columns). However, the important factor is the net percentage reduction in the execution time thanks to the improvement in the issue stalls. The last column shows the cumulative average improvement for the set of benchmarks, which is the sum of the averages; the PTD scheduler outperforms the others by a small margin in all but one case.

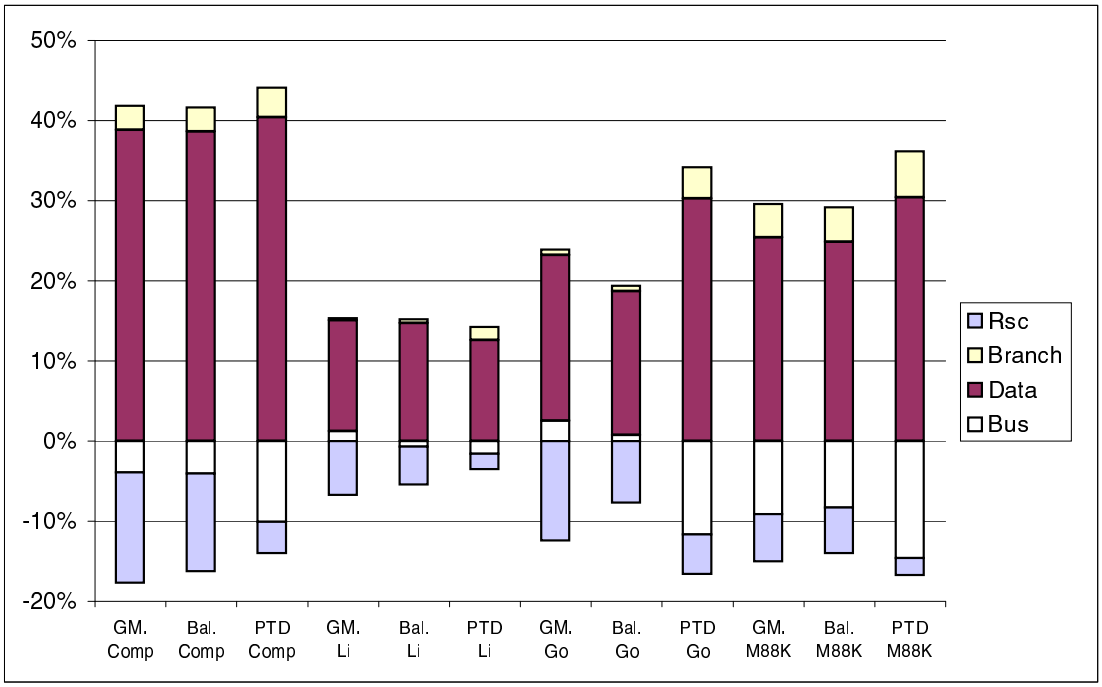
The data between Figures 7.11 to 7.14 cannot be compared as the figures in each graph are normalised against different base cases. The data from the 2 AU, 3 AU and 4 AU configurations is normalised against the 1 AU configuration, so that they can be related. This is represented in Figures 7.15 to 7.17 to show the percentage of improvement in the issue stalls for configurations which are normalised against the very base case, *i.e.* unscheduled code simulated with the 1 AU configuration.

These graphs confirm the limitations suffered by the `li` and `fract` benchmarks, where there are not enough resources (memory and floating-point units, respectively). Both benchmarks sustain a negative percentage in the issue stall due to lack of appropriate functional units as the architecture scales.

The `go` benchmark, on the other hand, presents an interesting scaling pattern. For greater than two arithmetic functional units (Figures 7.16 and 7.17), the benchmark features a degradation in the `data` stalls which is not perceived in the 2 AU configuration (Figure 7.15). This is due to the limited parallelism in the benchmark. In contrast, the `bus` and `rsc` stalls improve as the architecture scales.

The `puzzle` benchmark exhibits a similar behaviour in the absence of sufficient parallelism. It is recursive in nature which explains the high number of branch instructions in Table 7.2. An increase in the number of arithmetic functional units does not result in a corresponding improvement in the issue stalls.

In contrast, the loop-oriented benchmarks such as `intmm` and `livermore` show significant improvement as the architecture is scaled from the 1 AU towards the 4 AU configurations, as can be seen in Figures 7.11 to Figure 7.17.



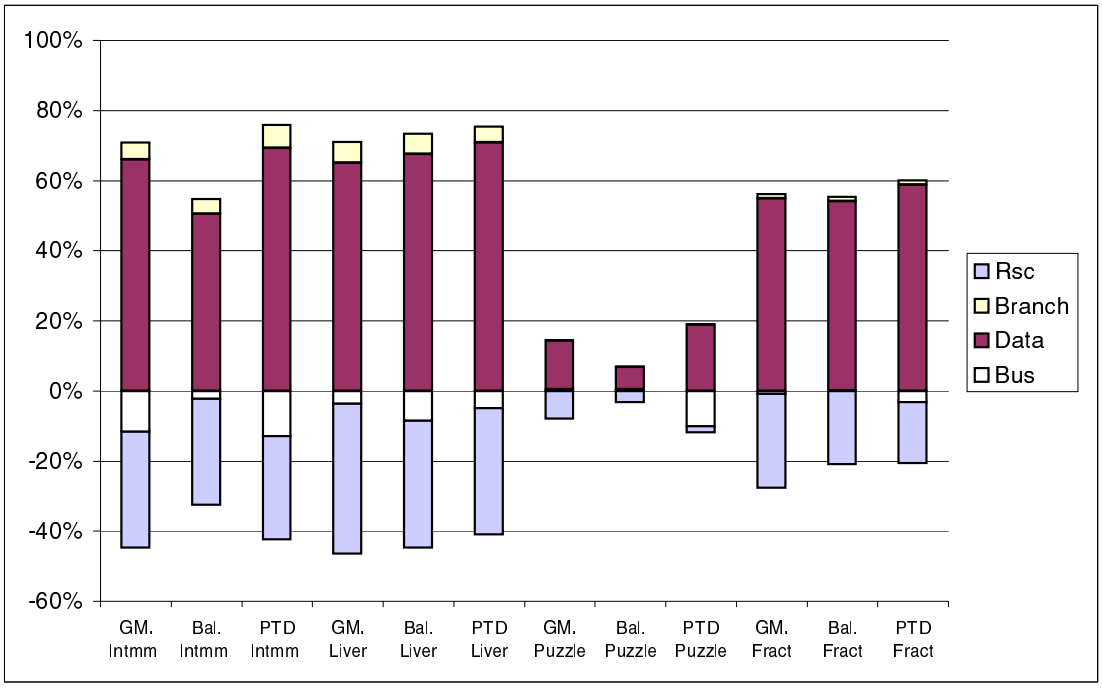
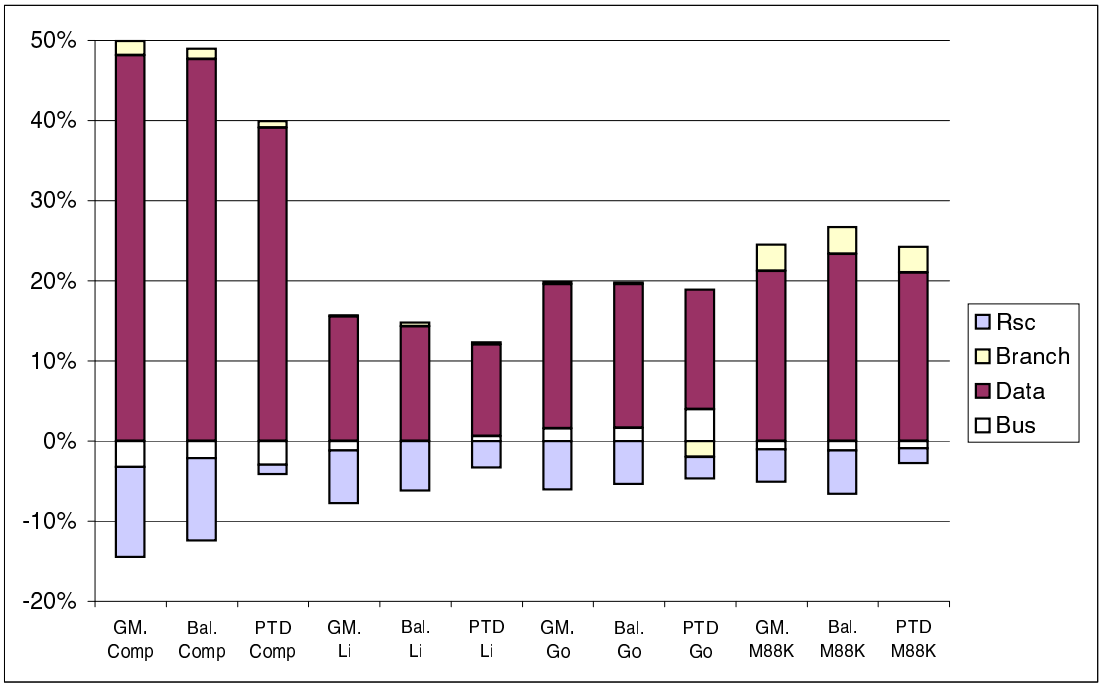


Figure 7.11: Percentage improvement in the issue stalls for 1 AU.



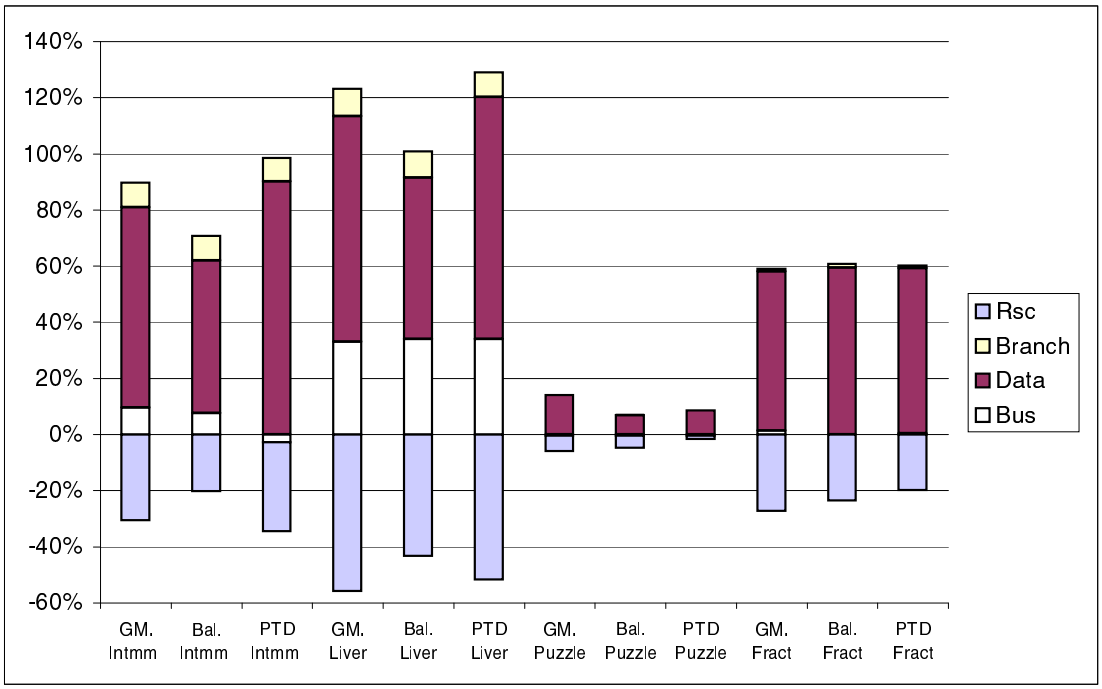
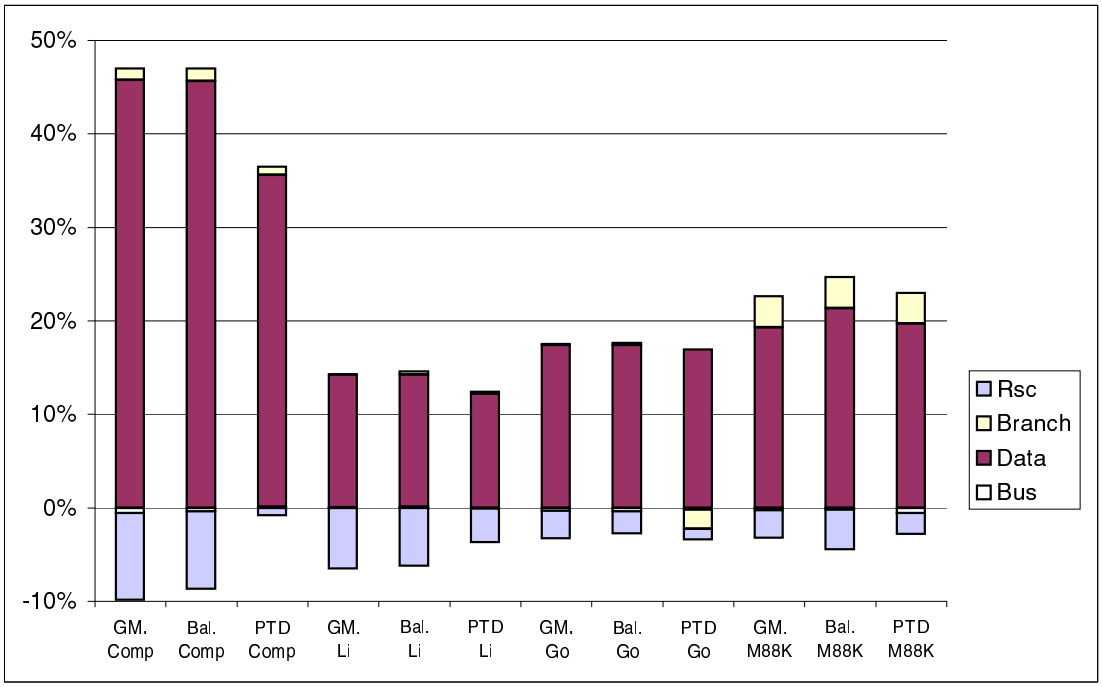


Figure 7.12: Percentage improvement in the issue stalls for 2 AU.



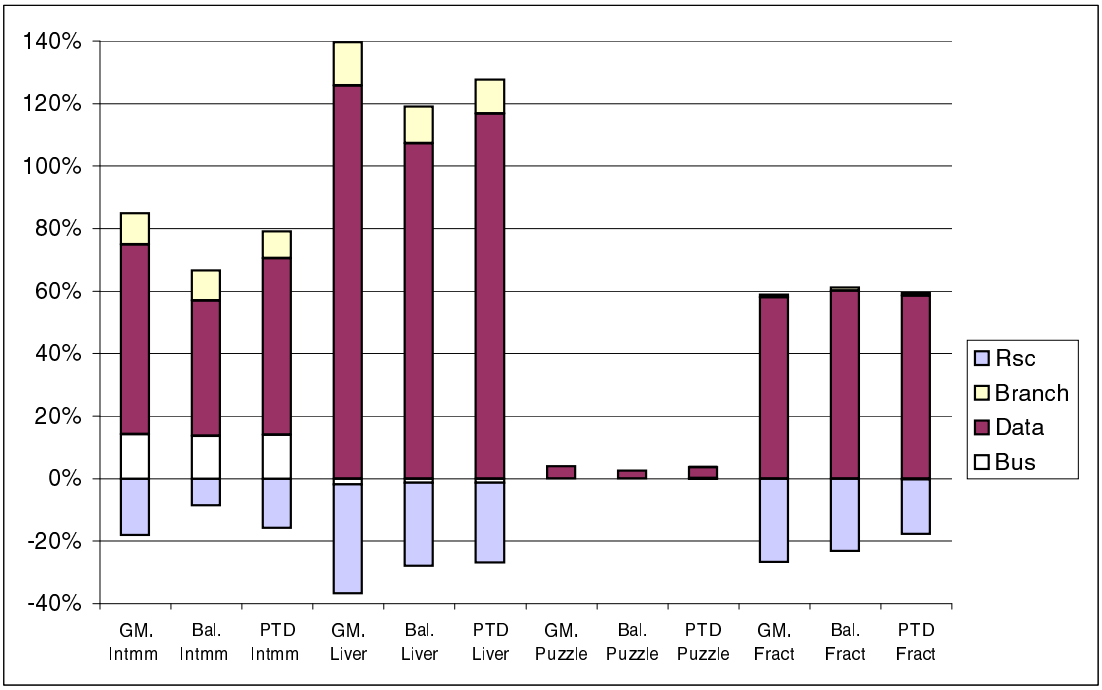
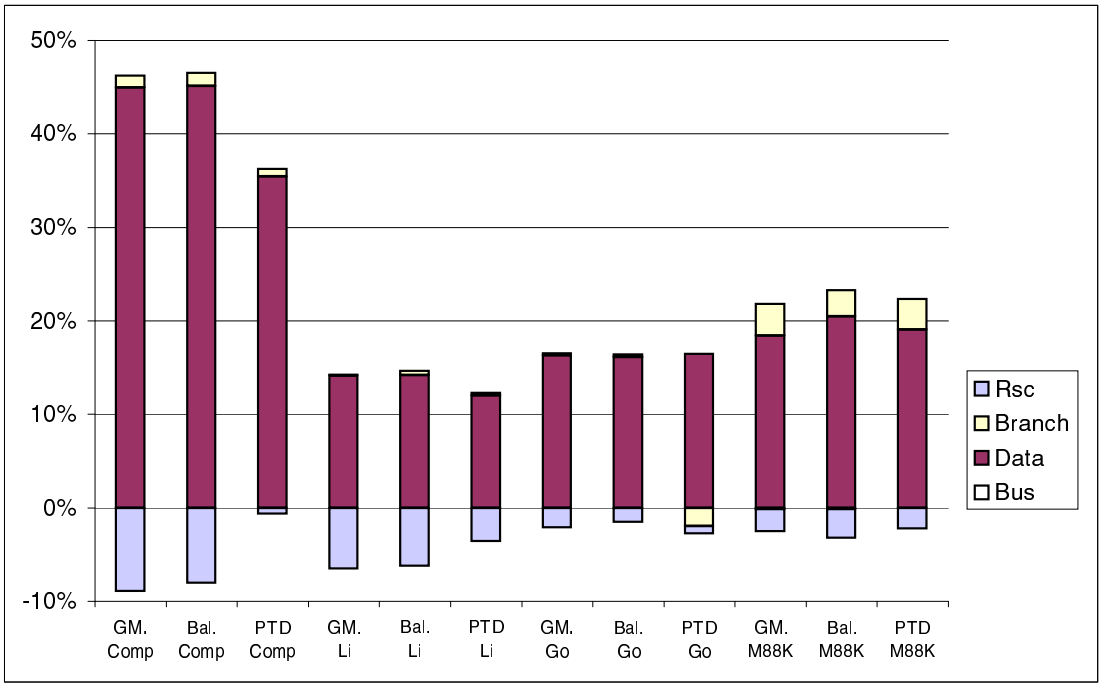


Figure 7.13: Percentage improvement in the issue stalls for 3 AU.



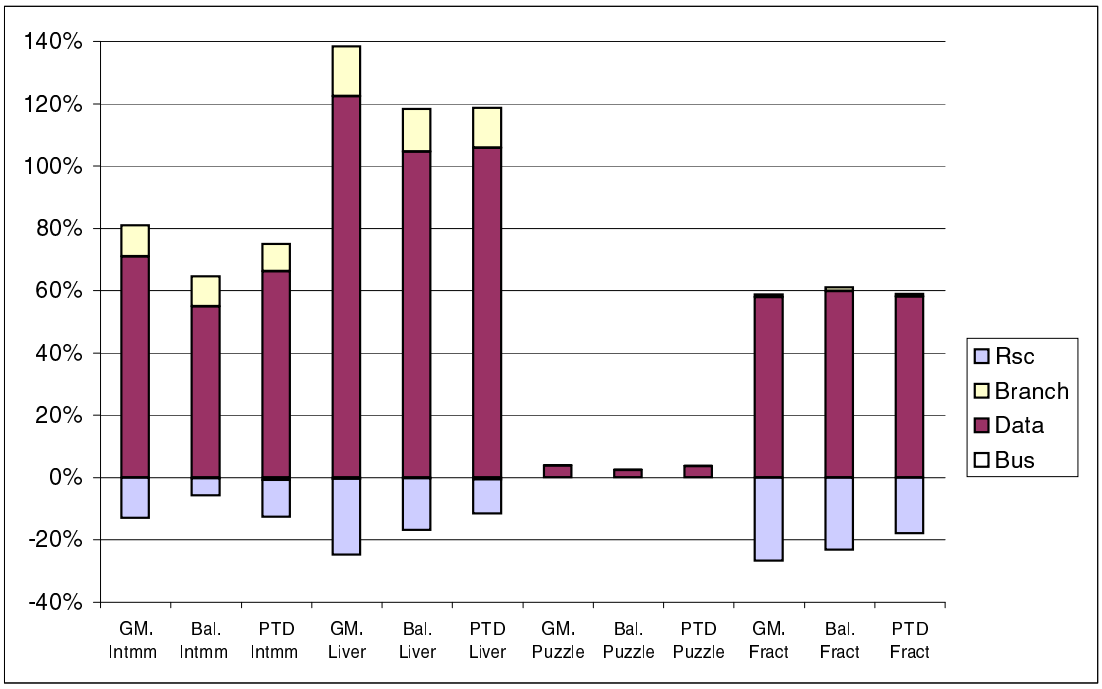
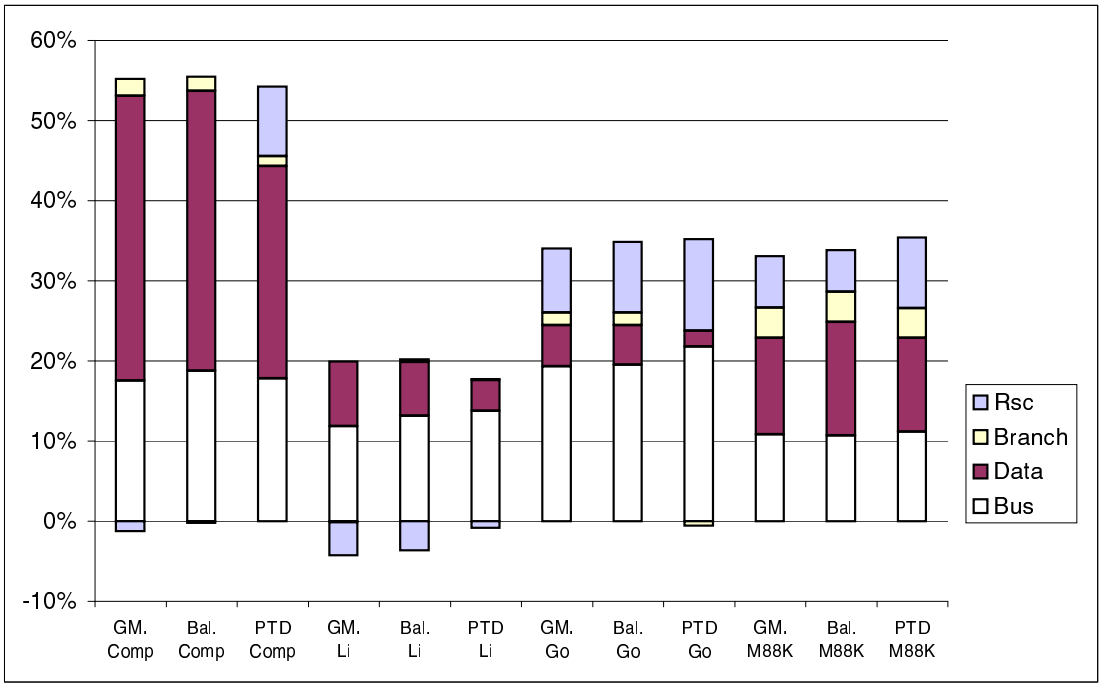


Figure 7.14: Percentage improvement in the issue stalls for 4 AU.



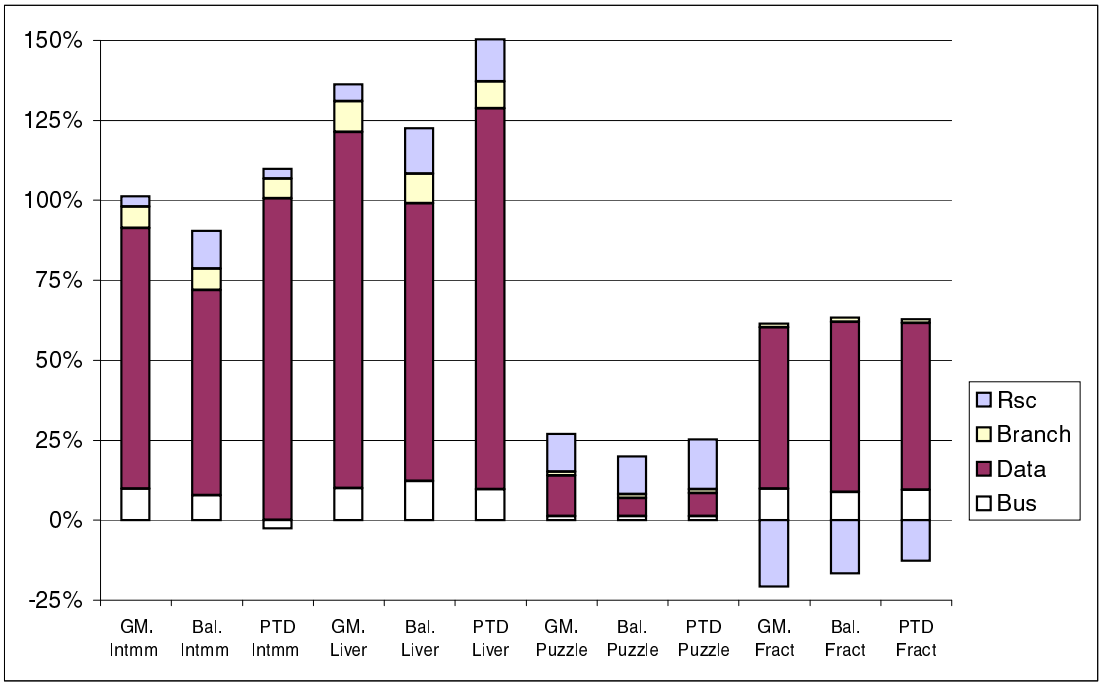
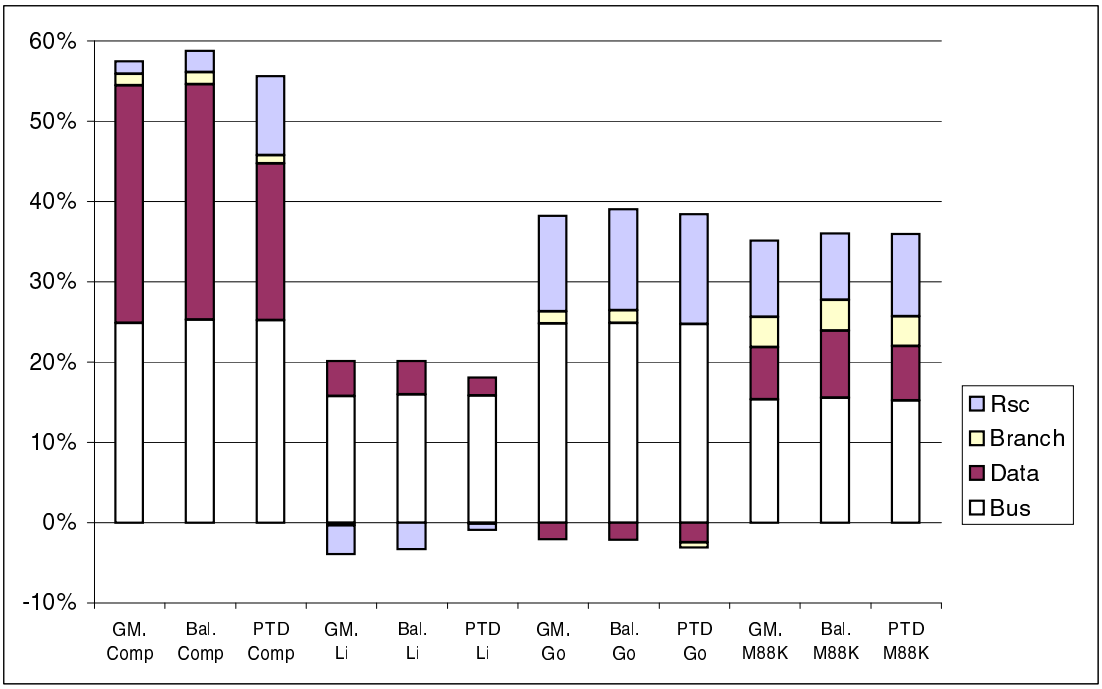


Figure 7.15: Normalised percentage improvement in the issue stalls for 2 AU.



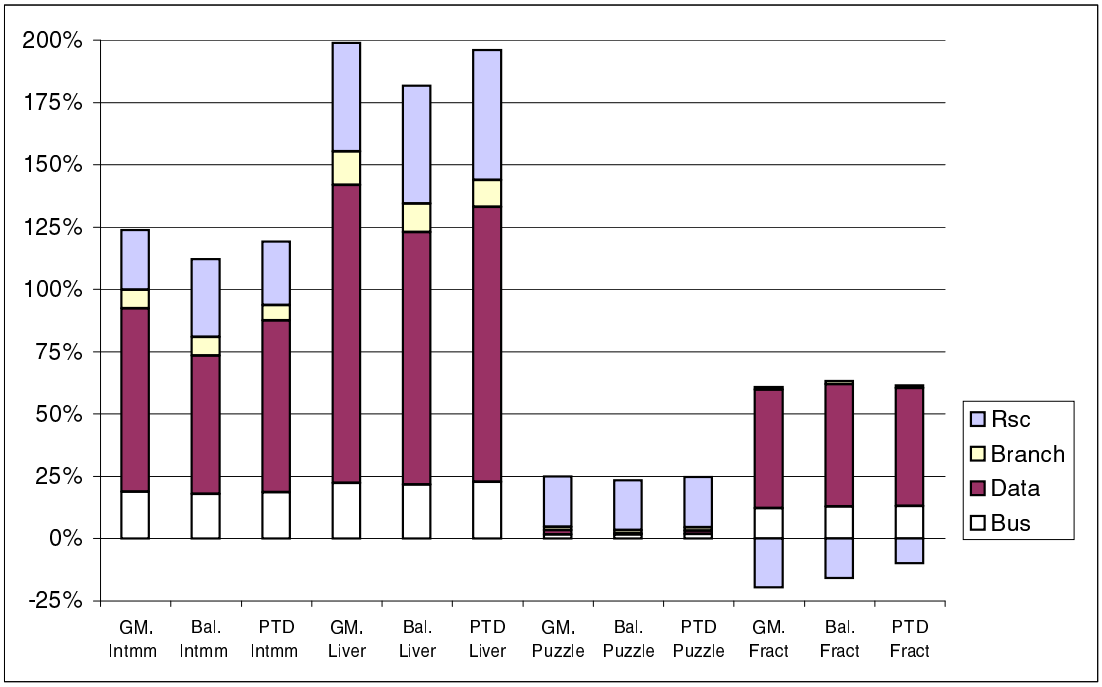
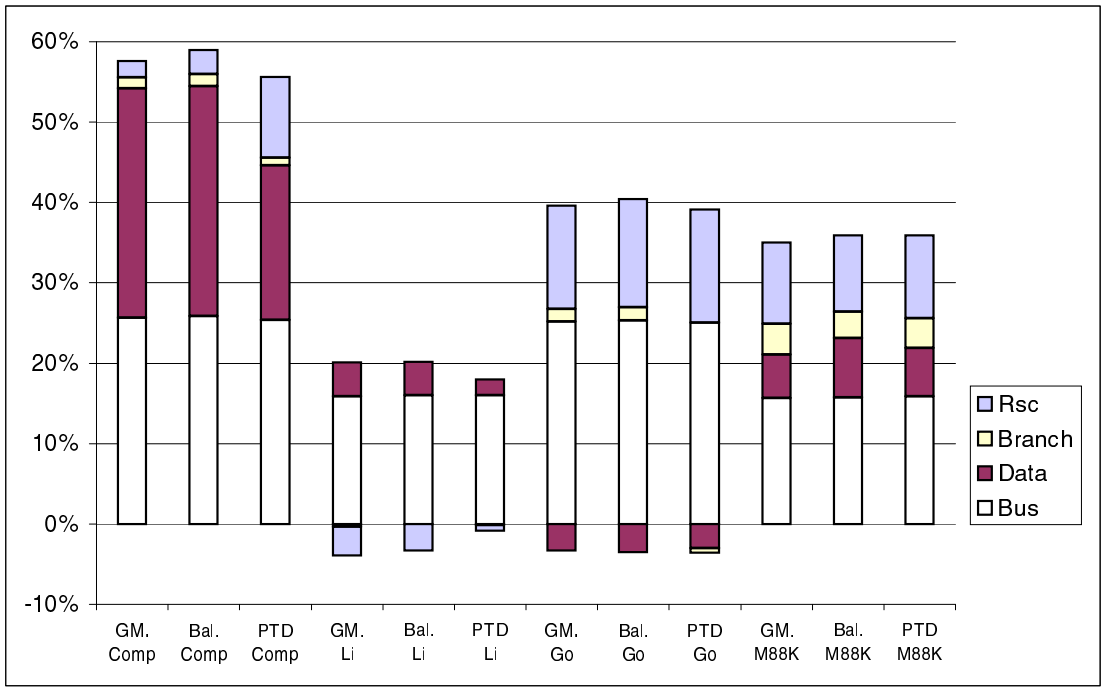


Figure 7.16: Normalised percentage improvement in the issue stalls for 3 AU.



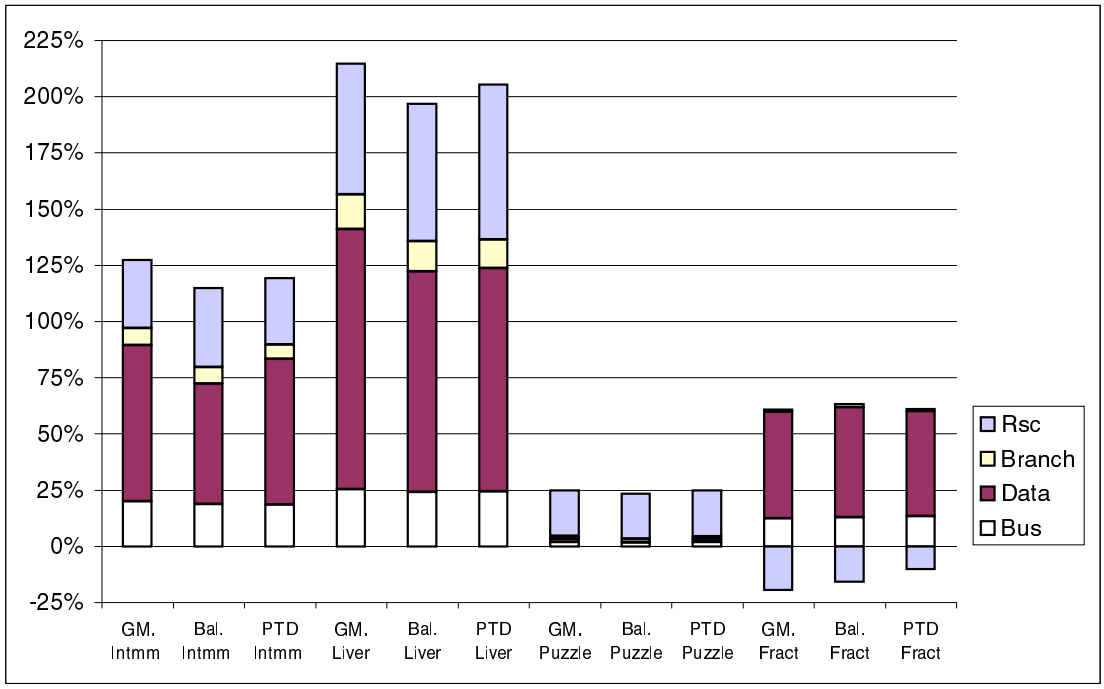


Figure 7.17: Normalised percentage improvement in the issue stalls for 4 AU.

Benchmark	No sch.	GM. sch.	Bal. sch.	PTD sch.
intmm	1131	392	386	408
livermore	1492	480	1055	1621
fract	284	364	426	638
li	1431	486	490	488
puzzle	1065	2169	3154	3222
compress	437	743	1003	1945
go	1038	1000	1179	1348
m88k	851	923	967	1326

Table 7.11: Number of out-of-order instructions (1 AU).

Benchmark	No sch.	GM. sch.	Bal. sch.	PTD sch.
intmm	1976	1565	1447	1360
livermore	4166	1980	3883	4068
fract	849	1249	1380	1077
li	4088	2983	2985	2987
puzzle	2411	4689	5517	5060
compress	1265	2107	3053	3264
go	2705	2709	3661	3029
m88k	2085	2192	2207	2536

Table 7.12: Number of out-of-order instructions (2 AU).

7.4.1.6 Out-of-order Instructions

One feature of the micronet-based processor is that although the instructions are issued in-order, independent instructions can overtake others and can be written back out-of-order. The numbers of instructions which are executed out-of-order are listed in Tables 7.11 to 7.14, show that the number of these instructions scales with the increase in arithmetic functional units.

Nevertheless, the out-or-order instructions are a small percentage of the total number of instructions which are executed per benchmark, mainly due to the data dependencies in the code. Even so, the PTD scheduler averages a higher number of out-of-order instructions. Although this does not contribute significantly towards the performance of the schedulers, they do serve to highlight the properties of the PTD scheduler.

Benchmark	No sch.	GM. sch.	Bal. sch.	PTD sch.
intmm	2125	2012	2426	2662
livermore	4200	2869	3805	4803
fract	886	1319	1343	1161
li	4214	3333	3334	3337
puzzle	2373	5274	5698	6383
compress	1504	2933	3728	3837
go	3022	3452	3618	3841
m88k	2221	2132	2146	2498

Table 7.13: Number of out-of-order instructions (3 AU).

Benchmark	No sch.	GM. sch.	Bal. sch.	PTD sch.
intmm	2220	2214	2919	2596
livermore	4251	3768	4601	5384
fract	860	1264	1326	1128
li	4226	5304	5365	5409
puzzle	2601	2169	3154	3267
compress	1438	3201	4210	4223
go	2950	3679	3998	4013
m88k	2236	2293	2273	2540

Table 7.14: Number of out-of-order instructions (4 AU).

7.4.1.7 Performance Execution

The performance comparisons of the execution times for the three local schedulers are displayed in Figures 7.18 to 7.21. They represent the percentage improvement against base cases with the 1 AU, 2 AU, 3 AU and 4 AU configurations. Memory disambiguation was applied to all the three schedules. The performance of the PTD scheduler shown in these figures is the same as those displayed in the last column of Figures 7.2 to 7.5. Subgraphs was only applied to the PTD scheduler since it is a particular heuristic designed for it.

For the 1 AU architecture, it can be seen that the PTD scheduler outperforms consistently the other two schedulers by an average of 4%. For the other configurations, the average improvement is reduced to within 2% as the architecture scales. This effect is the result of the overlapping penalties produced by the PTD schedulers as it optimises the code. The full listing of results of the local

schedulers is displayed in Appendix C.1.

The results from Figures 7.18 to 7.21 show the same pattern of the performance reduction due to issue stalls from Tables 7.6 to 7.9. As the architecture is scaled, there is a slight reduction of performance in the PTD scheduler when compared to the Balanced and GM schedulers. With the 2 AU, 3 AU and 4 AU configurations, the performance of the PTD scheduler is not as dominant as in the 1 AU architecture. The average for the 4 AU configuration shows that it performs as well as the others. In some of the cases, the PTD scheduler displays better improvements (`fract`, `li` and `m88k` benchmarks), although not as well in the case of `compress` and `go` benchmarks. This is the effect due to the overlapping penalties left in the code by the PTD scheduler.

The results of the local schedulers shown in Figures 7.18 to 7.21 are also shown in Table 7.15. The table shows the performance execution for the four configurations and their geometric means. It can be seen that the PTD scheduler's geometric means are better than the other schedulers in all the configurations.

7.4.1.8 Tolerance of the PTD scheduler

One important factor in the performance of the PTD scheduler is the ability to work for different range in the latencies, *i.e.* to ensure that the scheduler is not *sensitive* to high variations. Table 7.16 shows that even with the same latencies for all of the functional units, the PTD scheduler performs as well as in the case of different ranges in the latencies.

It can be observed that there is a general reduction in performance by all the schedulers when comparing these results against the performance improvements with different latencies as shown in Table 7.15, but only for the 1 AU configuration. This is because the non-scheduled code causes less stalls to the issue unit and the effect of scheduling consecutive dependent instructions can be masked to a certain degree since all the instructions have short and equal latencies. In turn, the results achieved by all three schedulers cannot reach the same levels as the ones with wider range in latencies. However, for the other three configurations, scaling the architecture enables higher improvements because the instructions tend to be completed sooner and thus, the issue unit does not need to stall as long as in configurations with different and longer latencies.

Another experiment to evaluate how the PTD scheduler handles different configurations is when the memory cache model is changed from a cache hit:miss ratio of 2:1 to a ratio of 9:1 and the cache hit:miss latency ratio from 1:2 to 1:10. Table 7.17 shows the results of this test for three of the benchmarks in all the configurations.

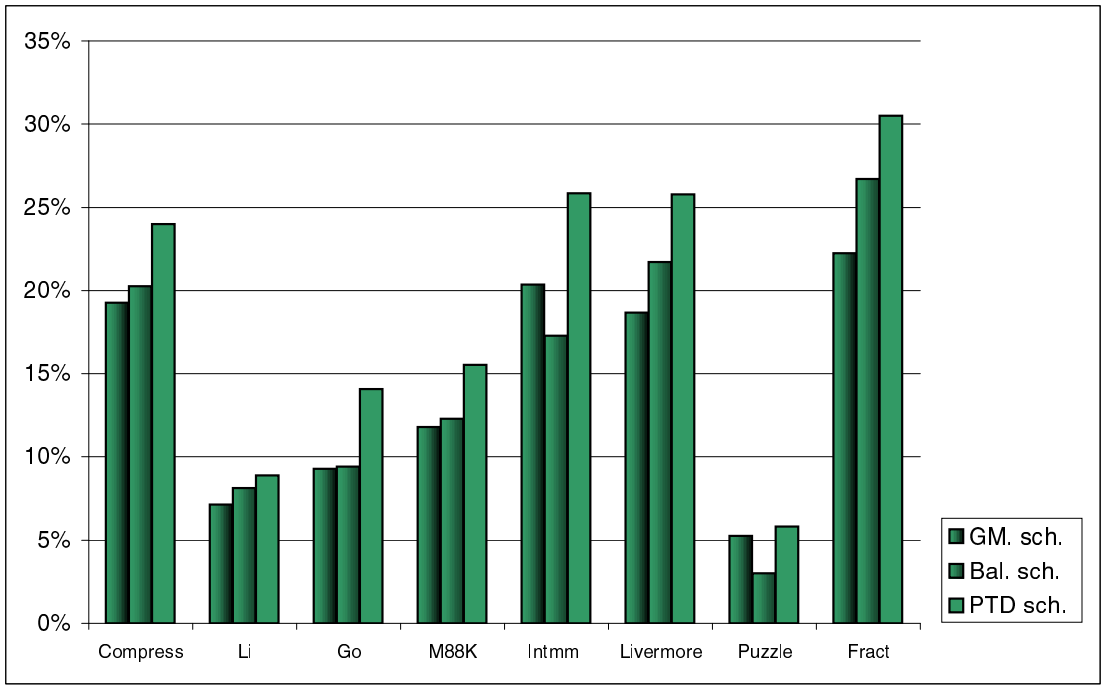


Figure 7.18: Local scheduler execution performance for the 1 AU configuration.

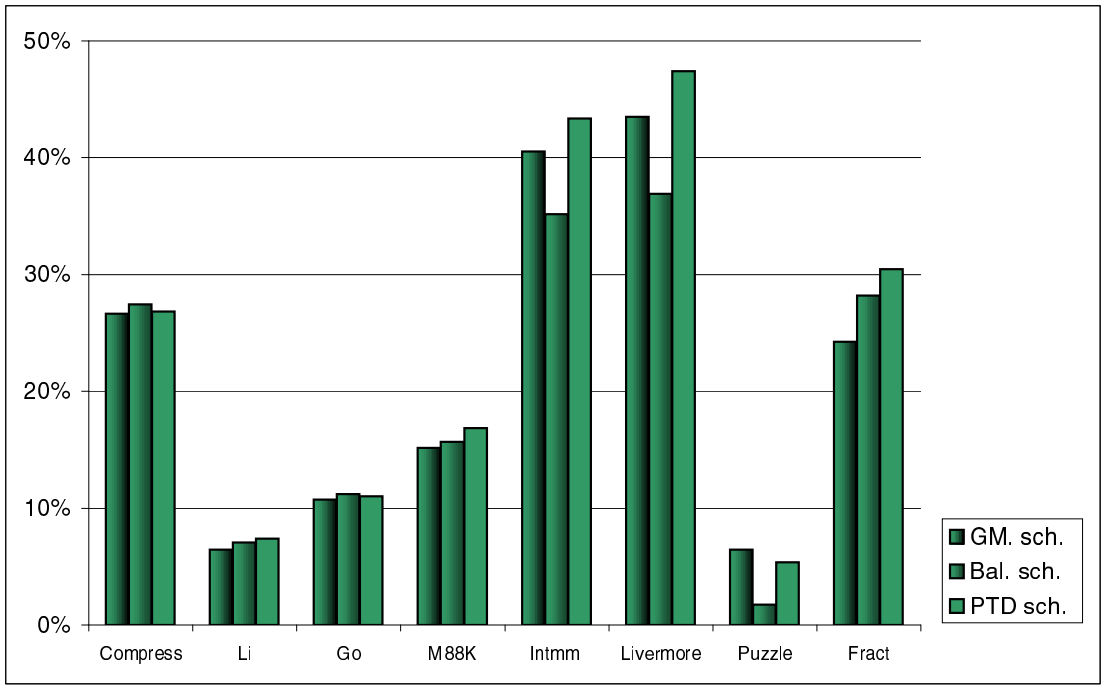


Figure 7.19: Local scheduler execution performance for the 2 AU configuration.

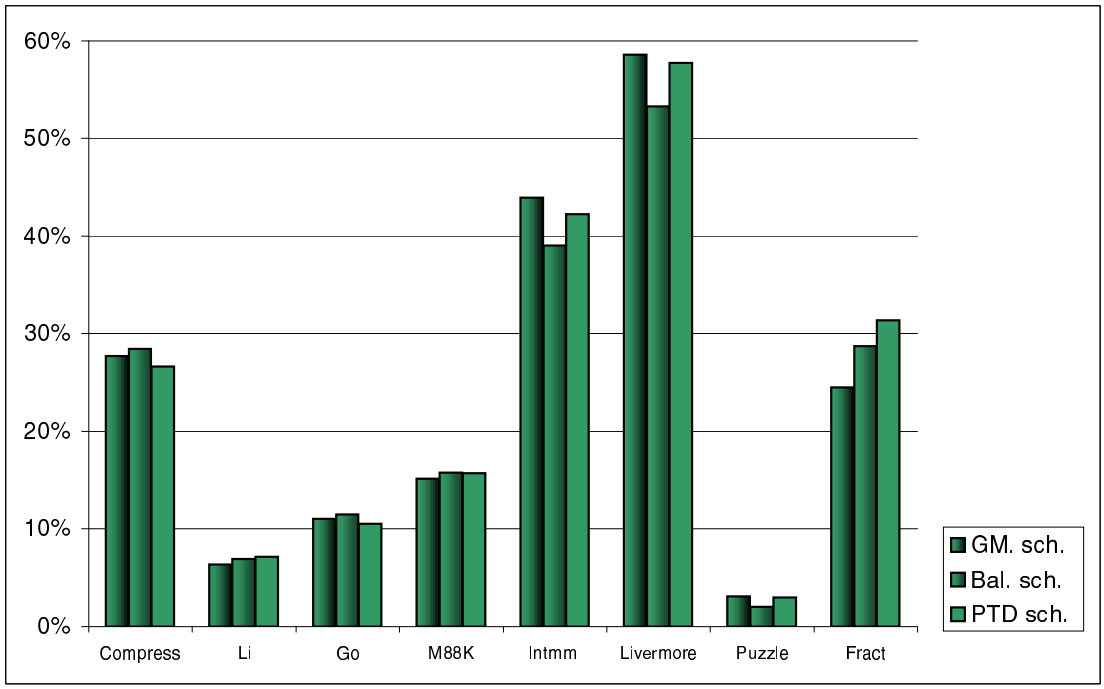


Figure 7.20: Local scheduler execution performance for the 3 AU configuration.

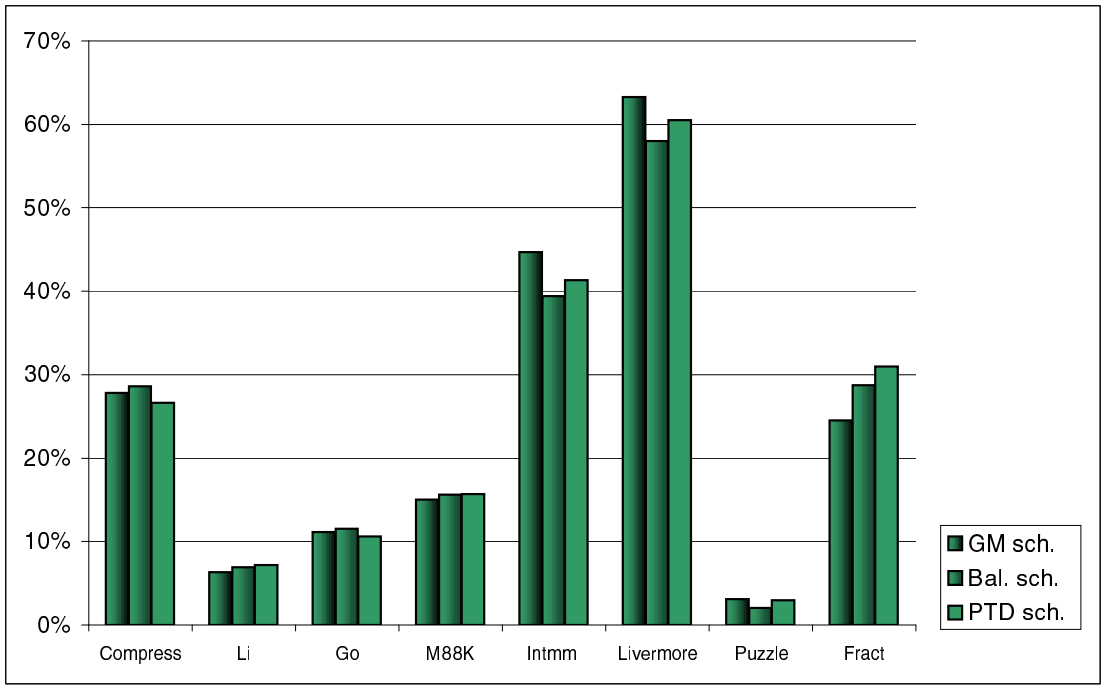


Figure 7.21: Local scheduler execution performance for the 4 AU configuration.

Conf.	Benchmark	GM. sch.	Bal. sch.	PTD sch.
1 AU	intmm	20.37 %	17.30 %	25.85 %
	livermore	18.66 %	21.72 %	25.76 %
	fract	22.26 %	26.72 %	30.50 %
	li	7.14 %	8.13 %	8.89 %
	puzzle	5.26 %	3.02 %	5.81 %
	compress	19.26 %	20.25 %	24.01 %
	go	9.28 %	9.41 %	14.09 %
	m88k	11.81 %	12.31 %	15.54 %
	Geometric Mean	12.68 %	12.45 %	16.51 %
2 AU	intmm	40.55 %	35.19 %	43.37 %
	livermore	43.52 %	36.90 %	47.40 %
	fract	24.25 %	28.21 %	30.48 %
	li	6.45 %	7.08 %	7.38 %
	puzzle	6.46 %	1.73 %	5.39 %
	compress	26.63 %	27.44 %	26.85 %
	go	10.71 %	11.20 %	11.02 %
	m88k	15.17 %	15.68 %	16.86 %
	Geometric Mean	17.22 %	14.68 %	18.27 %
3 AU	intmm	43.96 %	39.02 %	42.26 %
	livermore	58.61 %	53.27 %	57.76 %
	fract	24.53 %	28.74 %	31.38 %
	li	6.35 %	6.92 %	7.16 %
	puzzle	3.11 %	2.04 %	2.98 %
	compress	27.74 %	28.45 %	26.65 %
	go	11.07 %	11.51 %	10.52 %
	m88k	15.16 %	15.76 %	15.72 %
	Geometric Mean	16.62 %	16.02 %	17.06 %
4 AU	intmm	44.66 %	39.39 %	41.31 %
	livermore	63.27 %	57.98 %	60.50 %
	fract	24.53 %	28.74 %	30.95 %
	li	6.35 %	6.91 %	7.15 %
	puzzle	3.09 %	2.02 %	2.98 %
	compress	27.82 %	28.63 %	26.63 %
	go	11.16 %	11.53 %	10.64 %
	m88k	15.06 %	15.65 %	15.67 %
	Geometric Mean	16.80 %	16.20 %	17.10 %

Table 7.15: Performance execution improvement of the local schedulers for the four configurations, with functional units' latencies as defined in Table 4.1.

Conf.	Benchmark	GM. sch.	Bal. sch.	PTD sch.
1 AU	intmm	16.10 %	15.02 %	21.10 %
	livermore	18.01 %	17.97 %	18.59 %
	fract	18.27 %	19.98 %	21.98 %
	li	3.35 %	3.26 %	4.88 %
	puzzle	3.17 %	0.85 %	3.34 %
	compress	9.95 %	10.06 %	12.61 %
	go	5.13 %	3.88 %	7.43 %
	m88k	5.23 %	5.38 %	7.88 %
	Geometric Mean	7.89 %	6.49 %	10.04 %
2 AU	intmm	54.73 %	48.25 %	56.13 %
	livermore	58.93 %	52.68 %	53.44 %
	fract	33.33 %	35.10 %	34.81 %
	li	8.01 %	8.07 %	9.69 %
	puzzle	4.54 %	1.35 %	4.49 %
	compress	29.61 %	29.66 %	31.49 %
	go	13.10 %	11.72 %	14.53 %
	m88k	15.66 %	16.46 %	19.21 %
	Geometric Mean	19.81 %	16.53 %	21.14 %
3 AU	intmm	65.06 %	59.52 %	61.44 %
	livermore	83.98 %	70.07 %	64.35 %
	fract	35.91 %	39.04 %	37.55 %
	li	8.69 %	9.14 %	9.94 %
	puzzle	3.79 %	2.16 %	3.63 %
	compress	34.40 %	35.00 %	32.90 %
	go	14.88 %	14.02 %	14.74 %
	m88k	17.15 %	18.18 %	20.06 %
	Geometric Mean	22.09 %	20.29 %	21.85 %
4 AU	intmm	62.56 %	59.22 %	64.27 %
	livermore	88.90 %	73.43 %	64.75 %
	fract	35.94 %	39.13 %	36.85 %
	li	8.75 %	9.22 %	9.95 %
	puzzle	3.75 %	2.13 %	3.62 %
	compress	34.64 %	35.34 %	33.10 %
	go	14.79 %	13.91 %	14.69 %
	m88k	17.06 %	18.09 %	19.99 %
	Geometric Mean	22.12 %	20.38 %	21.94 %

Table 7.16: Performance execution improvement of the local schedulers for the four configurations, with latencies with equal range of values.

Conf.	Benchmark	GM. sch.	Bal. sch.	PTD sch.
1 AU	`li`	2.90 %	3.06 %	3.21 %
	`go`	2.62 %	2.94 %	5.87 %
	`m88k`	7.68 %	7.70 %	7.62 %
	Geometric Mean	3.88 %	4.11 %	5.24 %
4 AU	`li`	2.35 %	2.47 %	2.46 %
	`go`	3.48 %	3.79 %	3.37 %
	`m88k`	7.11 %	7.17 %	6.94 %
	Geometric Mean	3.88 %	4.07 %	3.86 %

Table 7.17: Performance execution improvement with a memory unit's cache hit:miss ratio of 9:1, and with cache penalty hit:miss ratio of 1:10.

The results show that all of the schedulers achieve very little improvements against the non-scheduled code, if we compare them against the ones with a cache hit:miss ratio of 2:1, displayed in Table 7.15. The main reasons for this behaviour is that there is only one memory unit and that given the delay penalty for a cache miss, there is not enough parallelism in the code to overcome such long delays. Even with such cache model, the PTD scheduler performs comparably well against the other schedulers.

7.4.2 Global Optimisations

The evaluation of the global scheduler is divided according to the performance due to code motion, tail duplication and the combined effect of both. The results from these global optimisations are also compared against the improvements due to local optimisations.

7.4.2.1 Code Motion and Code Duplication

As described previously in Chapter 6, code motion represents the first attempt at reducing the penalties after local scheduling. Only if code motion fails to remove a penalty is code duplication called which checks code expansion. Table 7.18 shows the statistics for code motion (first column of each scheduler) and code duplication (second and third columns). It can be observed that code motion is employed consistently more often than code duplication. The figures also reveal that the number of lines of code affected by the transformations is very low when compared to the total number of lines in the benchmarks (*c.f.* Table 7.3). The third column for each scheduler shows the percentage of code expansion due to

Benchmark	GM. scheduler			Bal. scheduler			PTD scheduler		
	Code motion	Code dup.		Code motion	Code dup.		Code motion	Code dup.	
		inst.	perc.		inst.	perc.		inst.	perc.
`intmm`	10	0	0.0%	9	0	0.0%	18	0	0.0%
`livermore`	42	0	0.0%	49	0	0.0%	73	0	0.0%
`fract`	73	19	0.3%	74	21	0.3%	122	38	0.7%
`li`	30	35	0.2%	50	50	0.3%	60	67	0.4%
`puzzle`	63	10	1.0%	55	12	1.3%	45	9	0.9%
`compress`	42	7	0.5%	50	7	0.5%	57	7	0.7%
`go`	2,388	322	0.3%	2,342	312	0.3%	2,608	423	0.5%
`m88k`	222	55	0.2%	599	138	0.4%	616	160	0.5%

Table 7.18: Code motion and code duplication statistics.

code duplication where applicable.

The table also shows a pattern in the number of movements which is greater in the case of the PTD scheduler. However, this cannot always be related to a performance improvement. This is because the effectiveness of the movements is subject to the run-time behaviour of the program. For instance, a single movement on a most commonly-executed path can be more effective from a performance point of view, than a number of movements in rarely-executed ones.

7.4.2.2 Tail Duplication and Block Merging

Tail duplication and block merging represent transformations which are independent of code motion. Table 7.19 displays the statistics regarding instances of tail duplication together with block merging (first column for each scheduler). The table also displays the total number of instructions duplicated as a result of those transformations (second column). The third column for each scheduler represents the percentage of code expansion due to tail duplication. It can be seen that the code expansion produced by tail duplication is much higher than in the case of code duplication. Even after merging non-empty blocks to increase ILP, the code expansion for some of the benchmarks is a cause for concern.

Another observation is that the number of instructions that are copied is the same for the three schedulers. The reason for this is that once a penalty is to be removed, the whole basic block is duplicated and merged, whereas in code duplication, the decision is made on the basis of individual instructions.

Benchmark	GM. scheduler			Bal. scheduler			PTD scheduler		
	Tail dup.	Inst. dup.		Tail dup.	Inst. dup.		Tail dup.	Inst. dup.	
		inst.	perc.		inst.	perc.		inst.	perc.
`intmm`	0	0	0%	0	0	0%	0	0	0%
`livermore`	14	306	15%	14	306	15%	14	306	15%
`fract`	69	1,036	19%	69	1,036	19%	69	1,036	19%
`li`	130	1,403	8%	130	1,403	8%	130	1,403	8%
`puzzle`	15	183	19%	15	183	19%	15	183	19%
`compress`	16	122	9%	16	122	9%	16	122	9%
`go`	1,557	13,993	16%	1,557	13,993	16%	1,557	13,993	16%
`m88k`	561	5,005	14%	561	5,005	14%	561	5,005	14%

Table 7.19: Tail duplication/block merging statistics.

7.4.2.3 Performance Benefits due to Global Optimisations

The comparisons of the performance execution achieved due to global optimisation techniques are displayed in Figures 7.22 to 7.25. The figures show the performance of the three schedulers divided in four categories (top right-hand corner key in each figure): local optimisations, code motion and code duplication, tail duplication and block merging, and the combined effect of both techniques. (The performance results of the local optimisations displayed in the first three bars for each benchmark correspond to the results previously shown in Figures 7.18 to 7.21. This clarifies the performance gains of global optimisations against the local ones).

The graphs show that, in general, a further improvement due to the global optimisations is achieved. This performance can be dissected into three parts: due to code motion, due to tail duplication and the combined effect of these two.

Code Motion/Code Duplication. The performance due to code motion and code duplication averages between 3.4% to 4.8% against local scheduling for the three benchmarks, in the four configurations. For the PTD scheduler, the improvements with respect to the local scheduler vary from 0.2% for the `li` benchmark to around 16% for the `puzzle` benchmark. For the 1 AU and 2 AU configurations, the PTD scheduler consistently outperforms the other two. For the 3 AU and 4 AU configurations, the performance of the schedulers is on par, with a slightly higher average for the PTD scheduler.

For benchmarks such as `compress` and `go`, local scheduling does not obtain the same performances as the balanced and GM schedulers. However, when code motion is applied, it can be seen that the PTD scheduler is able to outperform them.

It can also be observed that the performance patterns due to local scheduling results are preserved after both code motion and code duplication are applied. This implies that the PTD scheduler is not benefiting overly due to the global movements being triggered by the PTD measure.

The results also show that the performance suffers no degradation after global code motion with respect to the local scheduling results. This is an improvement on the work on code motion in [20], in which some of the benchmarks suffer from degradation. Appendix C.2 shows the exact percentage of improvement for the schedulers in the four AU configurations with respect to their base cases, *i.e.* unscheduled code.

Tail Duplication/Block Merging. The results from tail duplication and block merging show that they do not achieve the the same gain in performance as due to code motion. In fact, for many of the benchmarks there is very little improvement. The average improvements vary around 0.5% with respect to the local ones (The `intmm` benchmark does not show any tail duplication transformations). However, there are two benchmarks in which a greater improvement was obtained when compared to the code motion results. The `m88k` benchmark averages a 2.6% against local scheduling, which represents an additional 1.7% improvement over code motion for the four configurations. The benchmark `li` marginally outperforms the performance gain achieved by code motion, when applied to the PTD scheduler.

As with code motion, the pattern observed from the local optimisations results is maintained when tail duplication is performed. Furthermore, the optimisation does not cause performance degradation when compared to local scheduling. The actual figures for performance improvements from tail duplication and block merging are shown in Appendix C.2.

Combined Effect of Code Motion and Tail Duplication. The combined effect of code motion and tail duplication presents one particular case that does not provide the best performance. The `livermore` benchmark actually suffers a slight degradation of 0.8% over local scheduling for the 1 AU

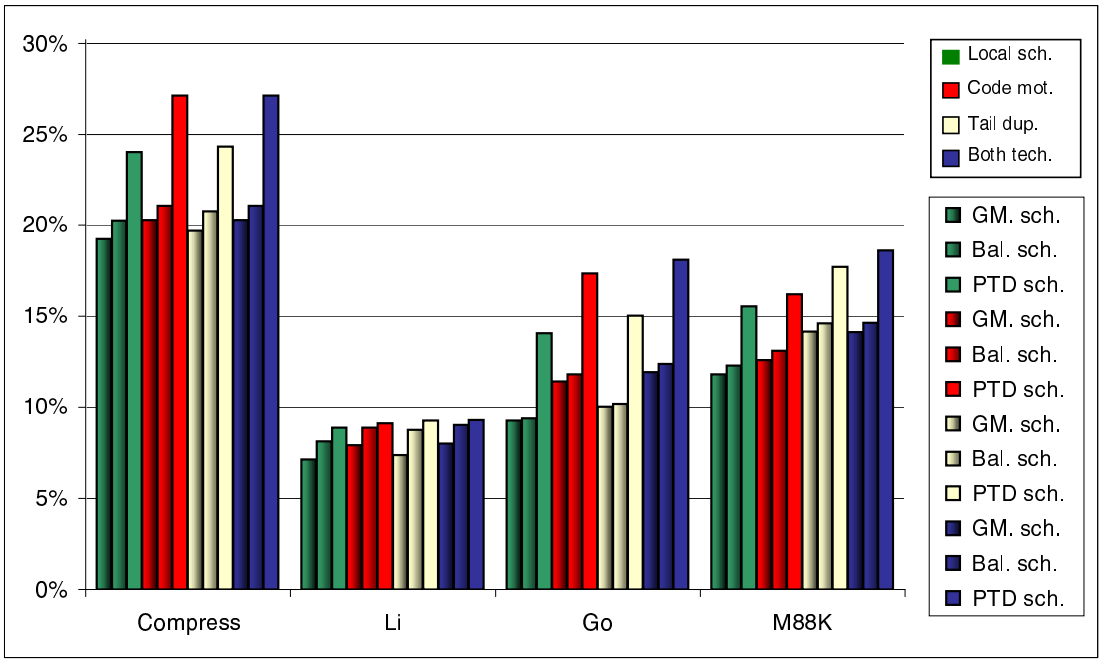
configuration. After code motion and tail duplication have both been applied, local scheduling also needs to be applied since it is more convenient to handle the complexity of the dependencies and penalties in the merged block in order to optimise it. However, the optimisation of the new block is subject to its new set of dependencies and initial schedule. The `livermore` benchmark shows that with the newer set of initial conditions, there may be cases in which the optimisations cannot necessarily yield the best performance.

As mentioned in Section 7.4.1.4, the `livermore` benchmark spends a considerable amount of time in the initialisation section. Any penalty that is left unreduced in the critical path of this section is subject to performance degradation. After analysing this example it was observed that if local scheduling had not been used after code motion and tail duplication, there would not have been any degradation.

Even with this degradation, the PTD scheduler performs better than the other schedulers for the 1 AU configuration. In general, although tail duplication does not yield a considerable improvement over local scheduling, the best performance levels are obtained when both code motion and tail duplication are utilised. The average improvements over local scheduling for the set of benchmarks range between 4.2% to 5.6%, for the four configurations — `puzzle` is the benchmark that achieves the highest performance of all, with an average improvement of 16.8%.

When compared to code motion, the technique with the greatest improvement over local scheduling, the combined effect of both code motion and tail duplication offers an additional average improvement of 0.7%. The `m88k` and `fract` benchmarks feature greater improvements with 2.9% and 1.2%, respectively.

Tables C.9 to C.12 in Appendix C.2 show the exact percentage of improvement of the combined effect of code motion and tail duplication for the four configurations.



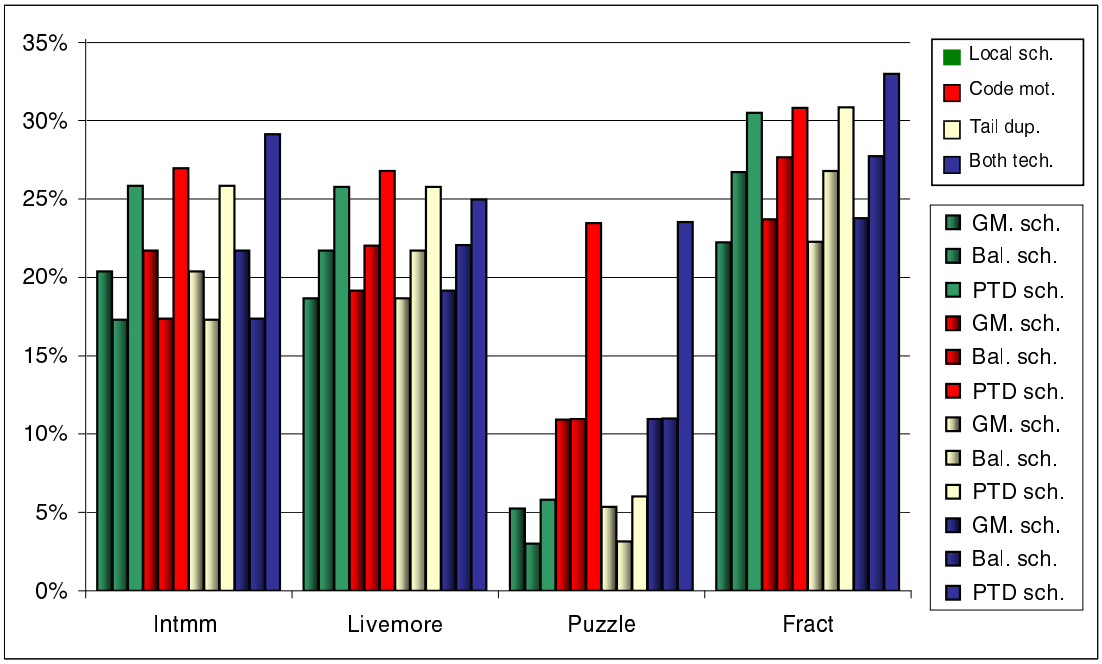
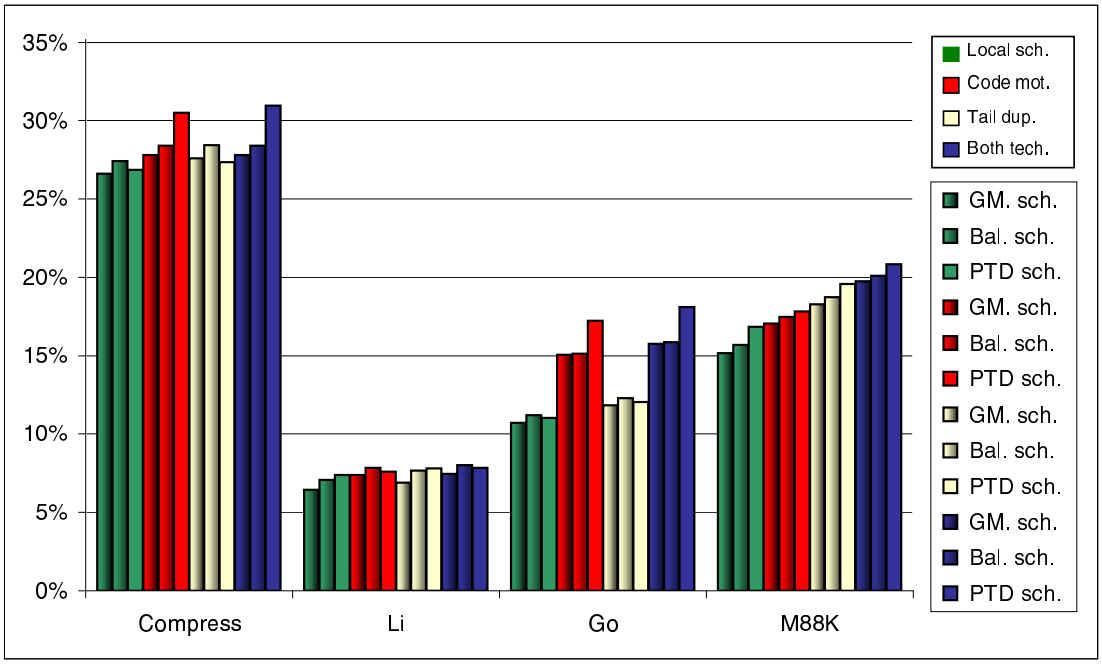


Figure 7.22: Simulation results for 1 AU.



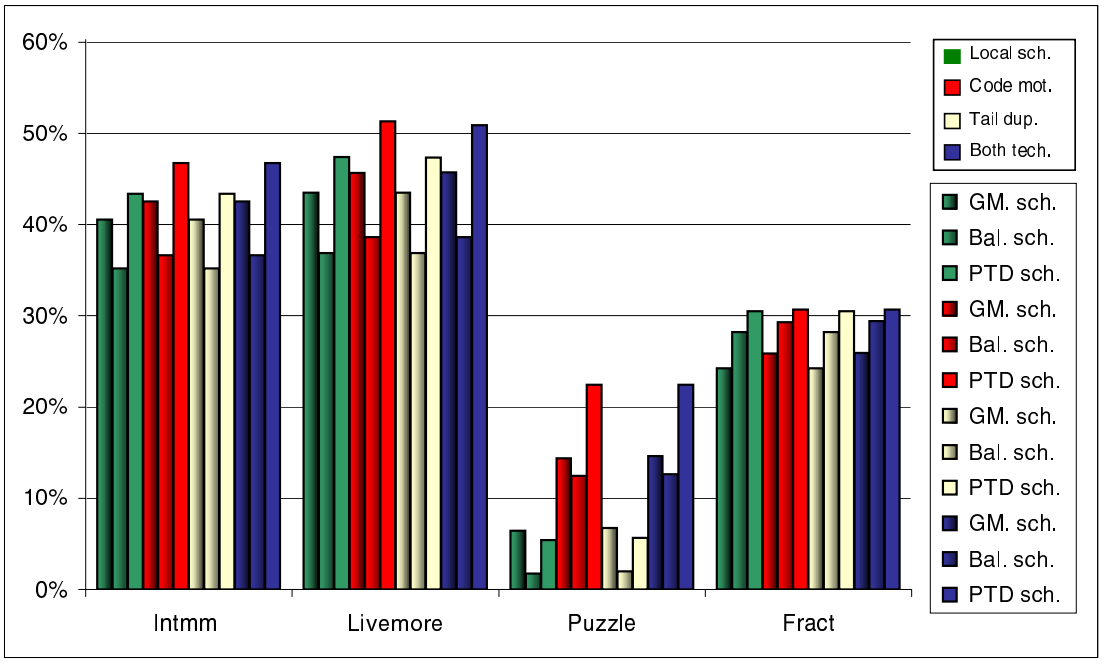
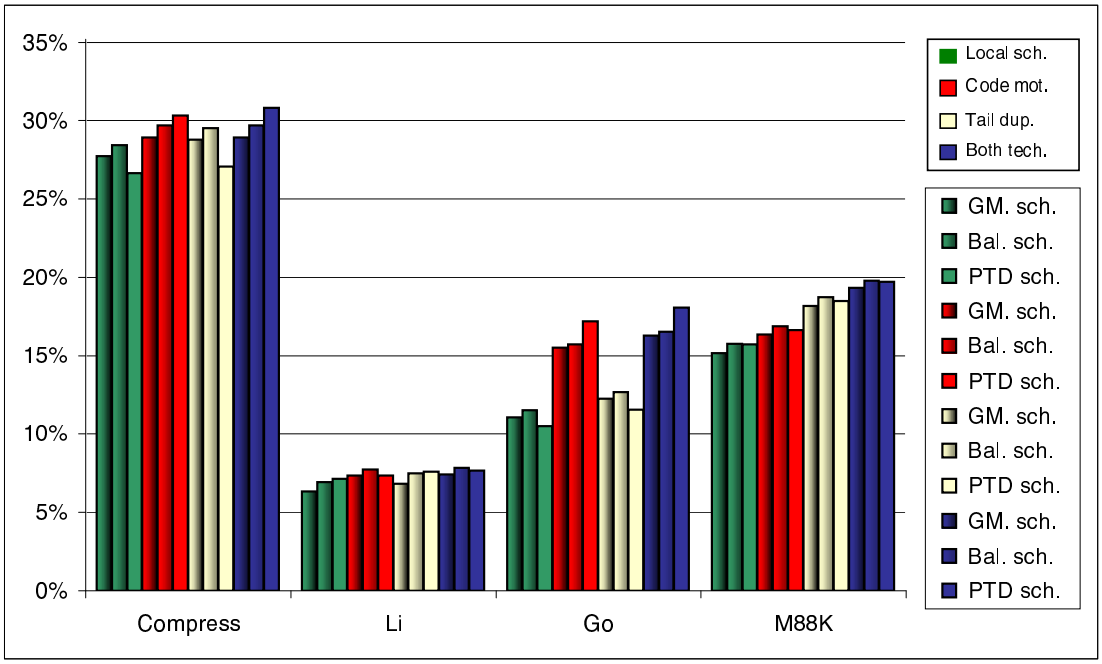


Figure 7.23: Simulation results for 2 AU.



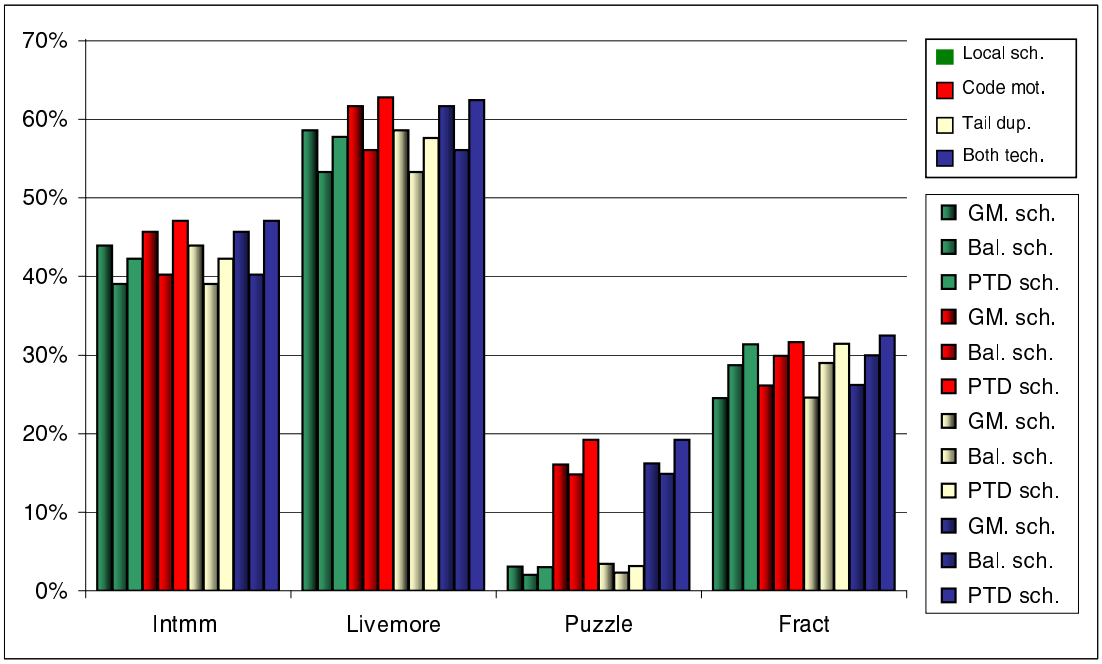
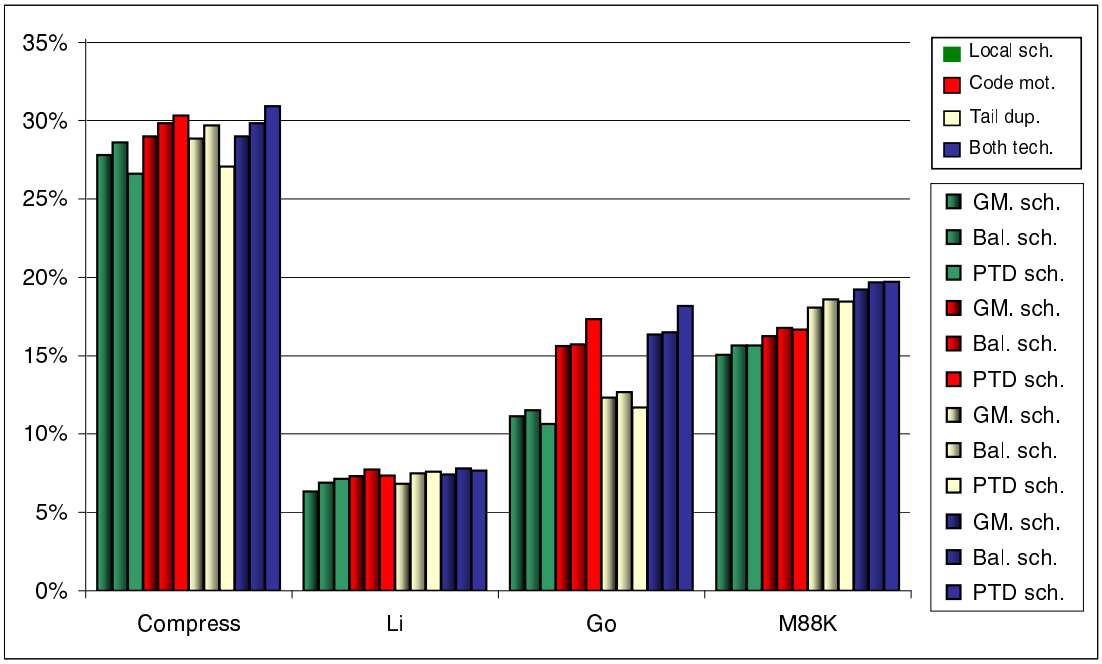


Figure 7.24: Simulation results for 3 AU.



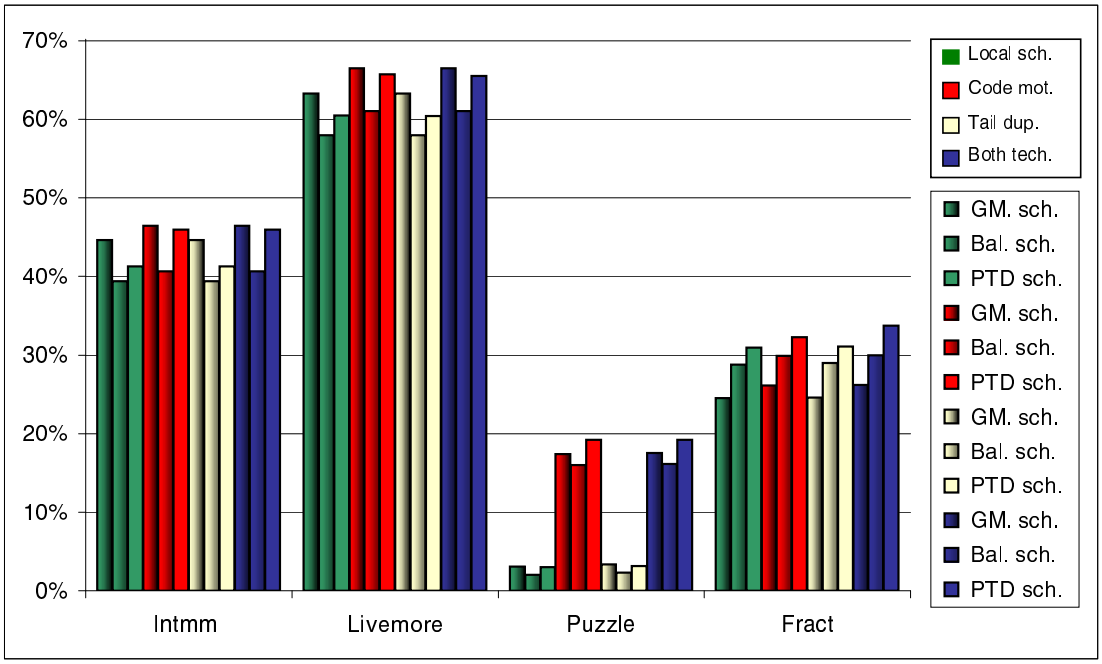


Figure 7.25: Simulation results for 4 AU.

Conf.	Benchmark	GM. sch.	Bal. sch.	PTD sch.
1 AU	intmm	21.72 %	17.37 %	26.98 %
	livermore	19.16 %	22.04 %	26.80 %
	fract	23.71 %	27.67 %	30.81 %
	li	7.94 %	8.89 %	9.12 %
	puzzle	10.92 %	10.96 %	23.47 %
	compress	20.27 %	21.06 %	27.13 %
	go	11.42 %	11.80 %	17.35 %
	m88k	12.60 %	13.11 %	16.22 %
	Geometric Mean	14.95 %	15.52 %	20.90 %
2 AU	intmm	42.53 %	36.64 %	46.77 %
	livermore	45.70 %	38.66 %	51.33 %
	fract	25.85 %	29.33 %	30.67 %
	li	7.40 %	7.85 %	7.58 %
	puzzle	14.41 %	12.47 %	22.42 %
	compress	27.80 %	28.41 %	30.52 %
	go	15.08 %	15.15 %	17.25 %
	m88k	17.05 %	17.47 %	17.84 %
	Geometric Mean	21.03 %	20.45 %	24.20 %
3 AU	intmm	45.71 %	40.26 %	47.10 %
	livermore	61.65 %	56.08 %	62.82 %
	fract	26.12 %	29.87 %	31.66 %
	li	7.34 %	7.74 %	7.37 %
	puzzle	16.05 %	14.78 %	19.21 %
	compress	28.93 %	29.68 %	30.34 %
	go	15.53 %	15.74 %	17.19 %
	m88k	16.37 %	16.89 %	16.65 %
	Geometric Mean	22.42 %	22.29 %	24.14 %
4 AU	intmm	46.46 %	40.66 %	46.00 %
	livermore	66.48 %	61.06 %	65.71 %
	fract	26.13 %	29.88 %	32.28 %
	li	7.34 %	7.73 %	7.37 %
	puzzle	17.40 %	16.02 %	19.21 %
	compress	28.99 %	29.83 %	30.32 %
	go	15.61 %	15.73 %	17.34 %
	m88k	16.27 %	16.77 %	16.68 %
	Geometric Mean	22.91 %	22.77 %	24.29 %

Table 7.20: Performance execution improvement of code motion for the four configurations.

Conf.	Benchmark	GM. sch.	Bal. sch.	PTD sch.
1 AU	intmm	20.37 %	17.30 %	25.85 %
	livermore	18.66 %	21.72 %	25.78 %
	fract	22.26 %	26.80 %	30.84 %
	li	7.40 %	8.76 %	9.28 %
	puzzle	5.35 %	3.14 %	6.04 %
	compress	19.71 %	20.78 %	24.32 %
	go	10.04 %	10.20 %	15.05 %
	m88k	14.16 %	14.61 %	17.71 %
	Geometric Mean	13.23 %	13.08 %	17.15 %
2 AU	intmm	40.55 %	35.19 %	43.37 %
	livermore	43.53 %	36.91 %	47.38 %
	fract	24.26 %	28.22 %	30.48 %
	li	6.90 %	7.67 %	7.83 %
	puzzle	6.71 %	1.98 %	5.64 %
	compress	27.59 %	28.43 %	27.35 %
	go	11.84 %	12.30 %	12.05 %
	m88k	18.29 %	18.74 %	19.56 %
	Geometric Mean	18.16 %	15.68 %	19.12 %
3 AU	intmm	43.96 %	39.02 %	42.26 %
	livermore	58.63 %	53.26 %	57.62 %
	fract	24.57 %	29.01 %	31.41 %
	li	6.82 %	7.49 %	7.62 %
	puzzle	3.39 %	2.31 %	3.17 %
	compress	28.79 %	29.54 %	27.09 %
	go	12.25 %	12.67 %	11.57 %
	m88k	18.19 %	18.72 %	18.50 %
	Geometric Mean	17.65 %	17.10 %	17.93 %
4 AU	intmm	44.66 %	39.39 %	41.31 %
	livermore	63.29 %	57.97 %	60.42 %
	fract	24.58 %	29.01 %	31.11 %
	li	6.82 %	7.49 %	7.61 %
	puzzle	3.36 %	2.28 %	3.16 %
	compress	28.88 %	29.72 %	27.07 %
	go	12.32 %	12.67 %	11.70 %
	m88k	18.09 %	18.61 %	18.45 %
	Geometric Mean	17.84 %	17.27 %	17.98 %

Table 7.21: Performance execution improvement of code duplication for the four configurations.

Conf.	Benchmark	GM. sch.	Bal. sch.	PTD sch.
1 AU	intmm	21.72 %	17.37 %	29.13 %
	livermore	19.16 %	22.05 %	24.96 %
	fract	23.78 %	27.75 %	32.99 %
	li	8.01 %	9.05 %	9.30 %
	puzzle	10.96 %	11.00 %	23.53 %
	compress	20.27 %	21.06 %	27.13 %
	go	11.93 %	12.39 %	18.10 %
	m88k	14.14 %	14.65 %	18.62 %
	Geometric Mean	15.28 %	15.88 %	21.64 %
2 AU	intmm	42.53 %	36.64 %	46.77 %
	livermore	45.72 %	38.66 %	50.92 %
	fract	25.95 %	29.44 %	30.67 %
	li	7.48 %	8.01 %	7.86 %
	puzzle	14.60 %	12.65 %	22.48 %
	compress	27.80 %	28.41 %	30.98 %
	go	15.76 %	15.88 %	18.10 %
	m88k	19.74 %	20.09 %	20.84 %
	Geometric Mean	21.61 %	21.03 %	24.96 %
3 AU	intmm	45.71 %	40.26 %	47.10 %
	livermore	61.65 %	56.06 %	62.42 %
	fract	26.21 %	29.97 %	32.47 %
	li	7.41 %	7.84 %	7.68 %
	puzzle	16.17 %	14.89 %	19.22 %
	compress	28.93 %	29.69 %	30.84 %
	go	16.28 %	16.52 %	18.08 %
	m88k	19.34 %	19.80 %	19.73 %
	Geometric Mean	23.09 %	22.94 %	25.05 %
4 AU	intmm	44.66 %	39.39 %	41.31 %
	livermore	63.29 %	57.97 %	60.42 %
	fract	24.58 %	29.01 %	31.11 %
	li	6.82 %	7.49 %	7.61 %
	puzzle	3.36 %	2.28 %	3.16 %
	compress	28.88 %	29.72 %	27.07 %
	go	12.32 %	12.67 %	11.70 %
	m88k	18.09 %	18.61 %	18.45 %
	Geometric Mean	17.84 %	17.27 %	17.98 %

Table 7.22: Performance execution improvement of both code motion and tail duplication for the four configurations.

Approach	GM. sch.	Bal. sch.	PTD sch.
Local scheduling	6.39 %	3.65 %	7.07 %
Code motion	13.43 %	13.48 %	28.79 %
Tail duplication	6.51 %	3.80 %	7.35 %
Both techniques	13.49 %	13.53 %	28.88 %

Table 7.23: Percentage reduction in the issue stall by the schedulers for the `puzzle` benchmark (1 AU).

Approach	GM. sch.	Bal. sch.	PTD sch.
Local scheduling	8.12 %	2.15 %	6.76 %
Code motion	18.48 %	15.91 %	29.33 %
Tail duplication	8.44 %	2.47 %	7.08 %
Both techniques	18.73 %	16.15 %	29.41 %

Table 7.24: Percentage reduction in the issue stall by the schedulers for the `puzzle` benchmark (2 AU).

7.5 Discussion

To explain the results shown in the previous section, the `puzzle` benchmark is analysed in detail. The `puzzle` benchmark offers the greatest improvements with code motion for any of the benchmarks, but it offers very little improvements when tail duplication is applied.

The benchmark has a function that executes for 60% of the total time. The control flow graph of this function is displayed in Figure 7.26 (c). Figures 7.26 (a) and (b) show the DAGs from two of the function's basic blocks, B_1 and B_5, respectively.

The function is characterised by basic blocks with many data dependencies that serialise the execution, and thus restrict local scheduling. The improvement achieved by code motion is based on the movement of instructions from block B_5 to block B_1 that reduces the consecutive penalties in both basic blocks. Even when there are non-consecutive penalties left in basic block B_1 (as a result of mixing instructions from B_1 and B_5), the increased ILP reduces the stalls of the issue unit. Of course, the effects of these movements are magnified by the fact that this function is executed considerably more often than the others.

Approach	GM. sch.	Bal. sch.	PTD sch.
Local scheduling	3.90 %	2.55 %	3.73 %
Code motion	20.78 %	19.07 %	25.08 %
Tail duplication	4.25 %	2.88 %	3.97 %
Both techniques	20.95 %	19.23 %	25.09 %

Table 7.25: Percentage reduction in the issue stall by the schedulers for the `puzzle` benchmark (3 AU).

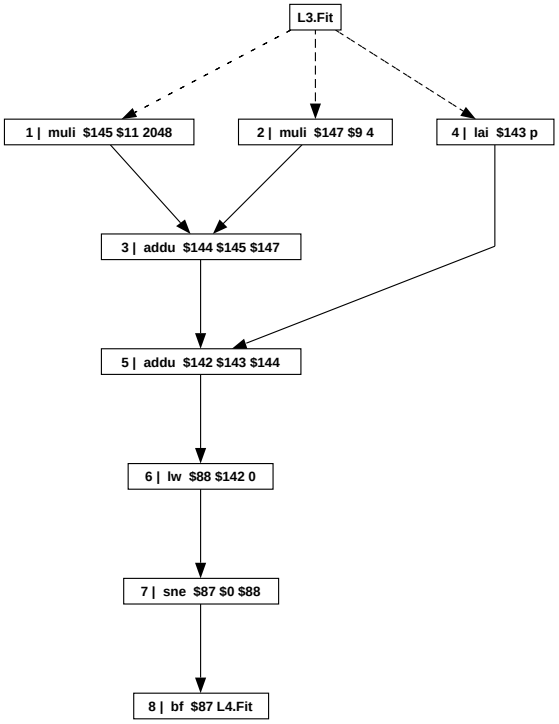
Approach	GM. sch.	Bal. sch.	PTD sch.
Local scheduling	3.87 %	2.53 %	3.73 %
Code motion	22.61 %	20.75 %	25.07 %
Tail duplication	4.21 %	2.85 %	3.97 %
Both techniques	22.79 %	20.91 %	25.09 %

Table 7.26: Percentage reduction in the issue stall by the schedulers for the `puzzle` benchmark (4 AU).

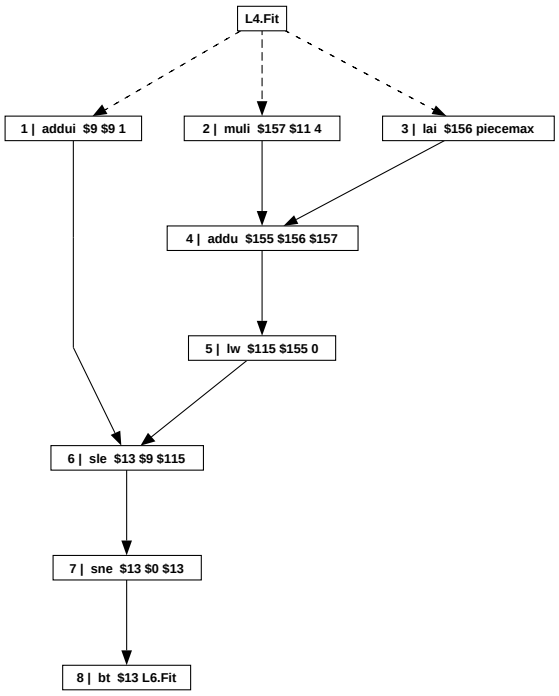
The PTD scheduler is not the only one to benefit from this scenario. Both GM and Balanced schedulers reduce the stalls of the issue unit after code motion. Tables 7.23 to 7.26 list the percentage improvement of the issue stalls due to different techniques when compared to unscheduled code. It can be observed that the three schedulers achieve substantial stall reductions to the issue unit with the help of code motion.

In contrast, when tail duplication is applied, even if basic blocks B_5 and B_8 in Figure 7.26(c) are duplicated and merged, the data dependencies between basic blocks B_6, B_7 and B_8 limit the scope for improvement, *i.e.* the instructions from basic blocks B_6 and B_7 cannot be mixed with the ones from basic block B_8. This is in line with the results for `puzzle` in Section 7.4.2.3. Tables 7.23 to 7.26 confirm the small improvement in stall reduction over local scheduling when tail duplication alone is enabled.

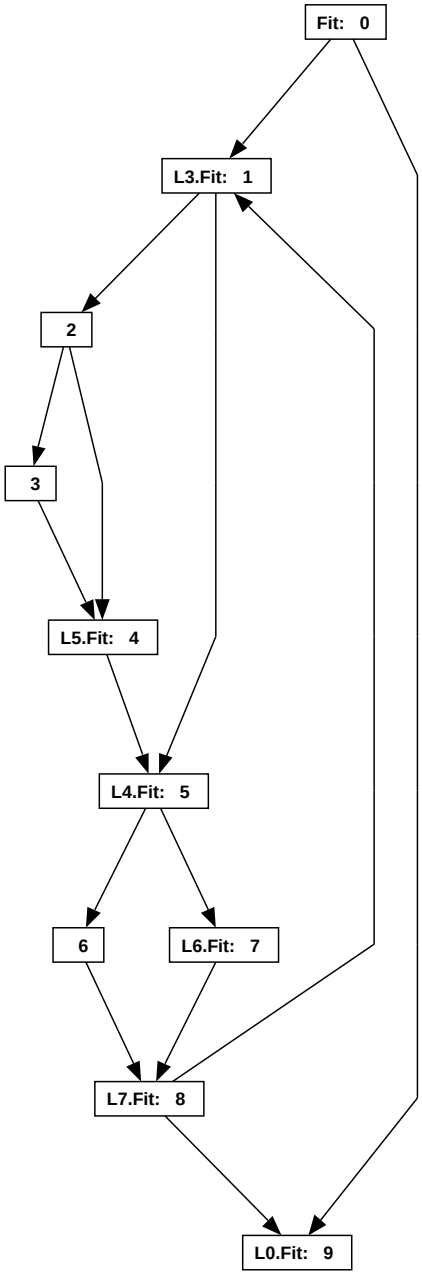
When both code motion and tail duplication are applied to the benchmark, then the improvement is dominated by code motion. It can be seen from the tables that there is a small percentage reduction in the issues stall when compared to code motion alone for the four configurations, and this reflects, in general, the best performance in terms of execution time.



(a) Basic block B_1 (`L3.Fit`).



(b) Basic block B_5 (`L4.Fit`).



(c) The CFG of the function.

Figure 7.26: The most frequently executed function from `puzzle`.

7.6 Summary

This chapter has presented the framework for evaluating the PTD scheduler. Within this framework, the SUIF compiler was selected to perform the back-end optimisations and to generate machine code for the micronet-based processor. The optimisations have been evaluated by compiling a set of C benchmarks and simulating their output schedules in the instruction-level simulator. The set of benchmarks covers a range of applications including loop-oriented and control-flow intensive programs.

The comparisons presented in this chapter were made against two well-known schedulers: the original list scheduler from Gibbons and Muchnick and the Balanced scheduler from Kerns and Eggers. The evaluation of the PTD scheduler has been separated between the use of local and global optimisations techniques.

The PTD scheduler attempts to maximise the rate of instruction issue of the micronet-based processor, by minimising the stalls incurred due to data dependencies and resource contentions. The results from the issue stalls to the issue unit reveal several characteristics of the PTD scheduler. Firstly, the stalls caused by resource contentions when instructions of the same type are scheduled, and there are not enough resources of that type, are reduced considerably. This reduction is, in general, greater than the reduction achieved by the other schedulers. Secondly, although the stalls caused by data dependencies are not reduced as much as the other schedulers, the total stall reduction of the issue unit is higher. This represents a higher overall reduction in relation to the base cases, when compared to the other well-known schedulers.

The performance of the local optimisations has shown that the PTD scheduler consistently produces better code than the other two. The scheduler achieves an average improvement of 18.81% for the 1 AU configuration, as opposed to 14.25% and 14.86% in the case of the GM and balanced schedulers, respectively. When the architecture is scaled, the average improvement of the PTD scheduler compares well against the other schedulers. For the 4 AU configuration, the average improvement is 24.48% which levels to 24.49% and 23.86% for the other two schedulers. These results show that the PTD scheduler produces, in general, better schedules, or at worst, comparable schedules for the micronet-based processor.

The complexity of the PTD scheduler described in Section 5.6 was evaluated by measuring the compilation times of the set of benchmarks. The algorithm complexity is governed by the number of penalties, and as the algorithm pro-

gresses the number of penalties is reduced. This represents an advantage over the algorithmic complexity of list-based schedulers which are of the order of n^2 (where n is the number of instructions). This advantage is confirmed by the results of the compilations times, in which the PTD scheduler averaged 46.55% faster compilation times than the other two schedulers.

The performance of the global optimisations applied to the local schedulers showed that the movement of instructions beyond basic blocks achieved further improvement. The results in this chapter have shown how the PTD measure represents an effective method to perform global optimisations. Although the improvement of the global movements is limited by data dependencies and influenced by the run-time behaviour of the programs, the results showed an overall improvements to the local scheduling. Code motion was responsible for the major percentage of improvement within global optimisations.

This chapter has shown that the PTD scheduler achieves better performances in terms of issue unit stall reductions and instruction execution times, when compared to two well-known schedulers. Furthermore, the PTD scheduler exhibits a better algorithm complexity.

Chapter 8

Conclusions and Future Work

The clock in a synchronous processor architecture provides precise timings for coordinating its operations. The trend towards increasing the clock speed will yield diminishing returns in the future for reasons outlined in Chapter 3. This thesis considered the problem of compiling for a micronet-based processor architecture which is composed of a network of functional units which operate concurrently and communicate asynchronously. Compiling for such a target poses unique problems due to the lack of precise timing models for the instructions.

Compiler optimisations which are dependent on the platform require a good understanding of the interactions between the back-end of the compiler and the target architecture. Optimising compilers for synchronous RISC architectures have been successful for the reason that precise timing models of datapath operations have been available. The datapath components have latencies defined in terms of clock cycles. The instruction scheduler in the back-end of the compiler uses this information to produce an efficient schedule (but not necessarily an optimal one) which minimises the makespan of the program. In the case of micronet-based asynchronous architectures by contrast, it is impossible to precisely model the instruction latencies, and hence the run-time behaviour of the programs. The latencies of the operations depend on a number of factors such as the input data, the type of components and the interaction between them. The order of instructions and their time of completion cannot be consistently predicted. This makes the task of optimising the instruction schedule for a micronet target a challenging one.

The micronet architecture has functional units connected as a network, which operate concurrently and supports fully out-of-order write-back. Instructions are issued at a rate which is limited by the dependencies between the instructions and the availability of functional units. The PTD scheduler aims to produce an instruction schedule so that they can be issued in quick succession without stalling

the issue unit either due to data dependencies or resource contention. Penalties are assigned to instructions with true dependencies and those which compete for the same functional unit. PTD first performs optimisations on the instructions within basic blocks to reduce their penalties. Next, global optimisations techniques which use code motion, code and tail duplication and block merging are applied to try and reduce the penalties remaining after local scheduling.

8.1 PTD Scheduler

8.1.1 Penalty Measure

The penalty measure is a useful metric for describing the goodness of a schedule at compilation time for programs targeted at micronet architectures. Although the measure is not strictly monotonic, however, in practice, schedules with lower measures display corresponding lower execution times. The measure has been demonstrated to be effective as a metric for the search heuristic when analysing schedules. PTD prioritises the order in which the penalties due to different types of instructions are reduced: those due to memory instructions are tackled first as they exhibit the longest delays, followed by the penalties due to true dependencies and the rest of the dependencies.

Such a priority scheme makes sense as long as the memory unit remains the slowest component in the datapath. For instance, if the latencies of the functional units are assumed to be even, then so would the priorities as the data dependencies due to either memory or arithmetic instructions would cause the same amount of stall and should therefore be penalised equally. Branch instructions preserve their priority as the cost of stalling the issue unit for every control flow change remains unaltered. Experiments using the same minimum and maximum latencies for all the architectural components are summarised in Table 7.16. They yielded the following results in the 1 AU case for the two largest benchmarks: PTD outperformed the other two competitors and was able to cope with the low priority penalties just as well.

8.1.2 Local Optimisations

The local PTD scheduler is different from other traditional techniques which are all based on a list-based scheduler. The penalty measure is simple yet an effective metric for statically evaluating the goodness of a schedule. When there is at least one penalty, the scheduler will traverse the basic block to find an independent instruction to reduce the penalty. Penalties due to resource contentions are re-

duced first; those due to data dependencies in consecutive and non-consecutive instructions are next tackled in that order. Safety conditions have been defined to restrict the movement of candidate instructions when the penalty measure cannot be strictly decremented. It terminates after two passes after the penalty measure cannot be reduced.

The complexity of the PTD scheduler is derived to be $\theta(\, et^2 + n - e\,)$, where e is the number of penalties in the basic block, n is the number of instructions and t is the distance in terms of number of instructions between the penalised and candidate instructions. It is observed that the complexity is governed by the number of penalties in a basic block, rather than the number of instructions. Also note that the number of penalties is reduced as the algorithm progresses. The figures for average compilation times for the benchmarks in Section 7.4.1.1 confirm the speed advantage over list-based schedulers. This is an useful attribute for just-in-time compilers which have fast scheduling requirements [37][160].

Memory disambiguation and subgraphs were introduced to further reduce penalties in the PTD scheduler. Without the former, memory instructions must be considered to be dependent as they might refer to the same location. Results in Section 7.4.1.2 have demonstrated that a considerable number of memory references can be disambiguated which in turn increases the scope for instruction-level parallelism in the program. Subgraphs were introduced to cope with the negative effects of patterns termed as overlapping penalties. These are the result of reducing penalties with neighbouring instructions under the safety conditions. Subgraphs mask parts of the DAG which constraints the search for candidate instructions, which reduced the effects of overlapping penalties. Although in some of the cases the introduction of constraints in a basic block with meagre parallelism resulted in under-optimised code. However, in general, the introduction of sub-graphs achieved better results in the case of the PTD scheduler.

8.1.3 Global Optimisations

The global optimisations presented in this thesis are an extension of the local scheduler when the parallelism found within basic blocks is limited as reflected in a number of penalties left by the local scheduler. Global movement of instructions after local scheduling offers the possibility of reducing the penalty measure further. The global extensions to the local PTD scheduler included code motion, code and tail duplication and code merging. These well-known techniques were implemented in the context of a micronet-based asynchronous architecture, and applied using the penalty-measure metric.

Dominator and post-dominator information from a region were used to determine which basic blocks were control-independent, so that instructions could be moved under the same control conditions. Code duplication is performed when code motion cannot be applied, mainly due to data dependencies. Selected instructions are moved to the parents' basic blocks, in a further attempt to reduce the penalties. Results in Chapter 7 have shown that with this order, the instances of code duplication can be kept to a minimum.

A generalisation of code duplication has also been considered as a global optimisation. Tail duplication copies and moves all the instructions of the basic block to its parents, as opposed to moving only the penalised instructions. These techniques were combined and applied to the PTD scheduler.

8.1.4 Performance of the PTD Scheduler

The main objective of the PTD scheduler was set to reduce as much as possible the stalls in the issue unit which are caused by data dependencies and resource contentions in a micronet-based processor. The results of the issue unit stalls presented in Section 7.4.1.5 showed that the local PTD scheduler achieved comparable levels of reduction against the other schedulers, and in some cases, the reductions were even higher. In general, the PTD scheduler reduced considerably more stalls due to resource contentions than the other two. Even though PTD fared less well in the case of stalls due to data dependencies for the four configurations, the overall performance of the PTD scheduler is better against the other schedulers. Therefore the quality of the code produced by the PTD scheduler is comparable to its two competitors, but with a much improved time complexity.

Benchmarks with extensive use of memory instructions are limited as there is only one memory unit considered. The `li` and `m88k` benchmarks have the greatest demand for the memory unit, and their results suffer most when the architecture is scaled. Their performance can be improved by increasing the number of memory functional units. This requires a dynamic memory disambiguator if parallelism between memory instructions is to be exploited.

8.2 Architectural Model

Although the issue unit of the micronet-based processor issues one instruction at a time, it operates at a faster speed than the rest of the components, which emulates multiple issue of instructions. However, even when the architecture is scaled and global optimisations are performed, the available parallelism is not sufficient

to maintain the issue unit without being stalled. Even when an independent instruction with its operands available is scheduled after an instruction that is stalled waiting for a result to be ready, the former instruction has to wait until the latter is issued.

A centralised issue unit represents a bottleneck, a characteristic of scalar processors. A possible solution is to have a multiple-instruction issue unit, but this would be considerably complex. The instruction set usually has to be modified as well, since independent instructions have to be made available by the compiler to the architecture explicitly.

Another source of bottleneck is the presence of only one memory unit in the architecture. It was demonstrated that benchmarks with large percentages of memory instructions do not scale well. Having more than one memory unit would enable more parallelism to be exploited, but a dynamic memory disambiguator must be included. An asynchronous design of a memory disambiguator represents a challenge, since the memory operations must be compared in a buffer, a process which must be synchronised which would damage average-case advantages.

The model described in Chapter 4 considers a datapath in which functional units do not have queues for holding more than one instruction when the functional unit is busy. The use of queues is characterised by a decoupling effect in which undesirable latencies are introduced. These latencies pose, in general, a problem for the data consistency scheme, since more instructions can be in-flight at the same time. In turn, the issue unit will be stalled by data dependencies most of the time, and less from resource contentions. The register locking mechanism will then have to be questioned.

Even with the use of queues, the PTD scheduler could still penalise the data dependencies, since the price of stalling the issue unit would hold. The penalties from memory instructions would not be as expensive, since a queue could compensate with their delay. If the processor changes towards an out-of-order issue unit, the behaviour will become more dynamic (dynamic scheduling), the complexity in hardware will be increased substantially, but more importantly, the model for the compiler will become more imprecise.

8.3 Future Work

The PTD scheduler has demonstrated that in the attempt to minimise the cost of data dependencies and resource contentions in an asynchronous processor, better schedules can be achieved relatively fast. However, when key penalties, *i.e.* ones

located in the critical path, could not be reduced, the schedules can end under-optimised.

Part of the future work would be to include a specific scheduling pass to reduce penalties due to the critical path. These would normally include, a first pass for consecutive penalties and a second pass for non-consecutive penalties. After these passes, the rest of the penalties would be dealt with as normal.

For global optimisations, an immediate work would be to include movement of instructions in the direction of the flow of control. This option may enable more movement, since it was observed that very little code is able to move. Another possible consideration is to change the method of moving a penalised instruction, by finding independent instructions to remove the penalties instead, as in the local scheduling approach.

8.3.1 Profile Information

The local and global optimisations described in this thesis have been developed without the use of profile information. The profile information could help tune both the global as well as local optimisations.

Using the statistics for frequency of execution, a heuristic for often-used basic blocks could be tailored for deciding whether or not to use subgraphs during local scheduling. In basic blocks with very limited parallelism, the option for applying subgraphs can be overturned to avoid the restrictions introduced for selecting candidate instructions. On the other hand, more scheduling passes can be spent on a basic block which is heavily executed, with the aim of reducing penalties remaining after the normal passes. For example, if a consecutive penalty could not be removed after the first and second scheduling passes, then the first pass could be invoked again, in a second attempt to reduce it. Any movement of the previously-executed second pass would change the starting order, so that it may be possible to reduce it after a rerun of the first scheduling pass.

For global optimisations, profile information could be used in frequently executed paths to put more effort in reducing penalties that could not normally be reduced. In fact, the use of speculative code motion could be applied if it can be statically evaluated that the cost of the increase of number of instructions executed can be outweighed by the gain obtained from penalty reductions.

Another consideration for global optimisation is to concentrate only on the penalty reductions in paths where the program execution spends most of the time. This would result in a more efficient method to improve the code with faster compilation times.

8.3.2 Other Optimisations

The global optimisations presented in this thesis are considered as acyclic optimisations. They represent an initial search space of code improvement around the PTD scheduler. Other global optimisations such as cyclic optimisations can be added to expose more parallelism. Loop unrolling is an example that will impact programs which contain loops.

8.4 Conclusions

Back-end compiler optimisations rely on an accurate timing model of the target architecture. This thesis has addressed the problem of optimisations for targets such as micronet-based asynchronous architectures which have uncertain latencies. The PTD measure was conceived as a way of statically determining the effect of stalls due to data dependencies and resource contentions in such architectures. Local and global schedulers based on the PTD measure were devised and their goodness over competing schedulers have been demonstrated for a set of benchmarks. PTD-based schedulers will find applications in future processor architectures in which uncertain communication latencies will dominate the cost of program execution.

Appendix A

Published Papers

A.1 Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures

Title:	Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures.
Authors:	Damal K. Arvind and Salvador Sotelo-Salazar.
Presented at:	The Third International Euro-Par Conference.
Place:	Passau, Germany.
Date:	August 1997.
Publisher:	Springer Verlag.

Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures

D. K. Arvind and S. Sotelo-Salazar

Department of Computer Science, The University of Edinburgh,
Mayfield Road, Edinburgh EH9 3JZ, Scotland.

Abstract. This paper addresses the problem of scheduling instructions in micronet-based asynchronous processors (MAP), in which the latencies of the instructions are not precisely known. A PTD scheduler is proposed which minimises true dependencies, and results are compared with two list schedulers - the Gibbons and Muchnick scheduler, and a variation of the Balanced scheduler. The PTD scheduler has a lower time complexity and produces better quality schedules than the other two when applied to twenty-three loop- and control-intensive benchmark programs.

1 Introduction

There has been a revival of interest in the use of asynchrony, albeit in a restricted form known as self-timing, in the design of processor architectures. Asynchronous circuits offer some distinct advantages. Their power consumption is generally much lower compared to their synchronous equivalent. This is because at any time only parts of the asynchronous system are active as required, with the rest remaining in a quiescent state. Self-timed systems allow a modular approach to processor design whereby parts can be added and deleted with little impact on the rest of the system. These systems are also robust to environmental changes.

The feature which is of most interest to our work and which was first recognised in the Micronet model [1] is that asynchrony offers scope for fine-grain concurrency in the processor architecture. The micronet model exposes this feature naturally, and asynchronous architectures based on this model are better able to exploit instruction-level parallelism.

A micronet-based architecture is viewed as a network of typed functional units. These units operate concurrently and communicate asynchronously with the rest of the architecture. The functional units themselves can be described at different levels of abstraction. In this paper the architecture is composed of the following functional unit types: one or more Arithmetic Unit (AU), a Logic Unit (LU), a Memory Unit (MU) and a Branch Unit (BU).

The issue and execution of an instruction consist of a sequence of micro-operations involving the Issue Unit (IU), the Register Bank, and the appropriate functional unit. An instruction is issued when both its operands are available. Once the instruction has been issued, it runs to completion unless it is stalled

due to contention for resources in the trajectory of the instruction at any one of these points: the read ports, the functional unit, the write-back port. The micronet model enables concurrent execution of the micro-operations of the different instructions in flight, and minimises the costs of instruction stalls due to resource contentions. The latency of the instruction depends on a number of factors: its type, the data on which it operates, and the contention for resources which depends on the mix of instructions.

This paper proposes a relatively inexpensive method for scheduling instructions within the basic block. The objective of the scheduler is to ensure the rapid issue of independent instructions, thereby minimising the number of stalls of the issue unit, and in reducing the contention for the functional units by enabling instructions of different types to be in flight at the same time. This is achieved by assigning penalties to data dependencies and successive instructions of the same type, and transforming the schedule by moving instructions to reduce the penalties. This results in a schedule in which dependent instructions are separated, and independent instructions of different types are issued in succession.

The next section describes the traditional list scheduling algorithms such as Gibbons and Muchnick and the Balanced schedulers.

2 Traditional scheduling heuristics

2.1 The Gibbons and Muchnick (GM) scheduler

This is a well-known example of a list scheduling algorithm proposed originally for scheduling instructions in pipelined architectures [2]. The algorithm selects the instructions to be scheduled from a directed acyclic graph, beginning at the roots. The instructions are selected for scheduling if all their immediate predecessors have been scheduled. These *ready* instructions are prioritised on the following basis: if possible, an instruction is scheduled that will not interlock with the one just scheduled; given a choice, an instruction will be scheduled which is most likely to cause interlocks with instructions after it. The complexity in the absence of any lookahead in the instructions is $\theta(n^2)$, where n is the number of instructions in a basic block.

2.2 The Balanced scheduler

The Balanced scheduler [3] was devised to take account of unpredictable memory access latencies. The idea is to compute weights for load instructions based on the number of available independent instructions. The instructions are scheduled as in a traditional list scheduler with independent instructions being distributed behind loads to buffer for unpredictable memory accesses. This idea is extended beyond the load instruction to all the instructions in the MAP architecture. The priority for ready instructions is based on a weighted sum of values derived from MAP tailored heuristics - whether the instruction uses the same resources as the previous scheduled one; the number of immediate successors of the instruction;

the length of the longest path from the instruction to the leaves of the DAG; and the number of source registers which are freed should the instruction be scheduled which effectively takes account of the register pressure.

3 The "Penalise True Dependencies" (PTD) scheduler

The essence of this heuristic is to identify true data and resource dependencies and re-order, where possible, the instructions such that their detrimental effect is reduced. The schedule is allocated a penalty measure based on the number and type of these dependencies. A true consecutive data dependency is penalised by one which is treated as the base case. If the dependency is with a branch or load instruction then it is penalised more severely. The actual value depends on the relative latencies of the functional units as shown in Table 1.

Instructions with resource dependencies are treated in a similar manner. If there are say p functional units of Type A, q units of Type B and r units of Type C, then a sequence containing more than p consecutive instructions of Type A, or q of Type B, or r of Type C will incur penalties. This assumes that the latencies of the three types of FUs are approximately the same; the run-length of the instructions can be suitably amended to take account of different latencies. The algorithm to derive this measure has a complexity of $\theta(n)$.

Cases of dependencies	Consecutive instructions	Separated by one inst.
True dependency with a load inst.	3	1
True dependency with a branch inst.	2	0
Resource dependency within mem. inst.	1	0
Normal true dependencies	1	0

Table 1. Table of penalties for true data dependencies.

We next demonstrate the correlation between the penalty measure considering only the true data dependencies and the makespans of the schedules for the program in Figure 1. The target asynchronous architecture has three types of functional units: an arithmetic unit (AU), logic unit (LU) and the memory unit (MU). The latency values for the units ranged over an interval, as shown in Table 2, with a Gaussian distribution. The results from a stochastic simulator which exhaustively simulated all the schedules (24,192) and averaged the results over 20 runs are shown in Figure 2. This result is representative of simulations of other programs with different spread of latencies. We can observe the trend that the penalty measure increases in step with the makespans of the schedules. This should ideally be a strict monotonic function, but the overlaps between the

schedules of neighbouring penalties are tolerable for the heuristic approach. A scheduler based on minimising the penalty measure is introduced in the next section.

```
                                    L4.main:
                                            muli    $13,$9,4
                                            la      $14,$29,0
main() {                                    addu    $15,$14,$13
                                            muli    $24,$9,4
                                            la      $25,$29,0
  int i, j, n = 10;                         addu    $11,$25,$24
    int x[10];                              lw      $12,$11,0
                                            muli    $13,$10,4
  for (i = 0; i < n; i++)                   la      $14,$29,0
    for (j = 0; j < n; j++)                 addu    $24,$14,$13
      x[i] = x[i] * x[j];                   lw      $25,$24,0
                                            mul     $11,$12,$25
                                            sw      $11,$15,0
}                                           addui   $10,$10,1
                                            slt     $12,$10,$8
                                            bt      $12,L4.main
```

Figure 1. C and MAP assembly code from our example.

Component type	Minimum latency	Maximum latency
Issue Unit (IU)	1.00 ns	2.00 ns
Input buses	2.00 ns	4.00 ns
Output buses	2.00 ns	4.00 ns
Arithmetic Unit (AU)	4.00 ns	8.50 ns
Logical Unit (LU)	2.00 ns	7.00 ns
Memory Unit (MU)	10.00 ns	20.00 ns

Table 2. Latencies values for the target architecture.

3.1 The PTD scheduler

The PTD scheduler works in two phases: in the first phase the contention for resources is minimised, and in the second phase consecutive data dependent instructions are separated.

In the first phase, the types of consecutive instructions are compared and instructions are moved, where possible, so that the overall penalty measure is reduced, such that the number of consecutive instructions of the same type is no greater than the number of functional units of that type.

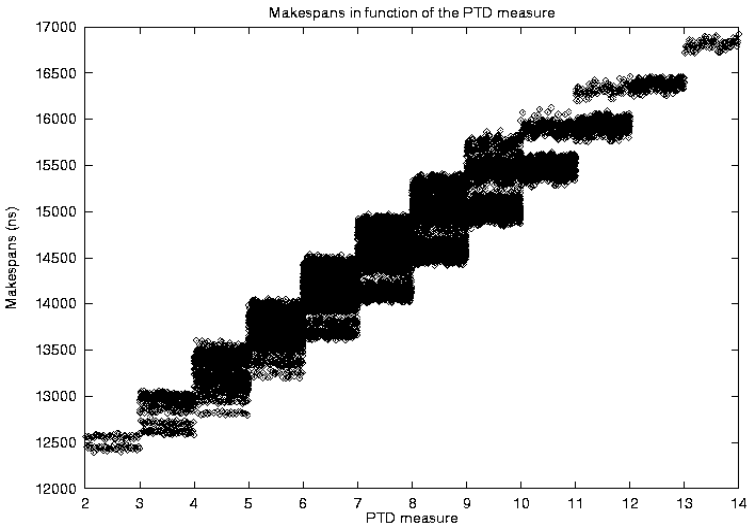


Figure 2. Execution distribution for the example.

In the second phase, the schedule is again scanned from start to finish, to identify consecutive data dependencies, and independent instructions are sandwiched in between them so that the overall penalty measure is reduced to zero or cannot be reduced any further due to the lack of suitable instructions. The details of the PTD scheduler are shown in Figure 3. The functions `PTD_arrange_left()` and `PTD_arrange_right()` traverse the schedule in both directions in search of independent instructions for insertion immediately after the penalised one. Two transformations are employed: a *swap* operation and a *move_ahead* operation and their use is illustrated in the following example.

Let l, m, n and o represent consecutive instructions in a schedule with a data dependency between n and o. This is represented by $n \rightarrow o$. The conditions for performing a $swap(m, n)$ transformation which eliminates (or reduces) the penalty to o, are the following:

- $m \parallel n$ (m is independent of n),
- $m \not\rightarrow o$ (not producing a penalty) and
- $l \not\rightarrow n$

If the penalties go beyond consecutive instructions then in order to ensure that the penalty measure will be reduced after the *swap*, the necessary condition is that the sum of penalties before the movement is greater than the measure after the transformation is made.

The conditions for performing a $move\_ahead(x, n)$ (moves x ahead of n) to eliminate (or reduce) the penalty to o, are the following:

- $x \parallel a, \ \ldots, \ x \parallel l, \ x \parallel m, \ x \parallel n$,
- $x \not\rightarrow o$ and
- $x_{-1} \not\rightarrow x_{+1}$ where x_{-1} and x_{+1} are the instructions previous and following x, respectively.

```
void PTD_second_phase(dagnodes *root) {
  measure = PTD_measure(root, second_phase);
  if (measure > 0)
    do {
      node = root;
      last_measure = measure;

      while (node != NULL) {
        if (node -> PTD.penalised > 0)
          PTD_arrange_left (node);
        if (node -> PTD.penalised > 0)
          PTD_arrange_right(node);
        node = node -> next;
      }
      measure = PTD_measure(root, second_phase);
    } while (measure < last_measure && measure > 0);
}
```

Figure 3. The PTD scheduling algorithm - Phase 2.

Again to generalise the rules to allow a *move_ahead*, the sum of penalties before the insertion must be greater than the total number of penalties after the instruction has moved.

The conditions just outlined apply for the `PTD_arrange_left()` function which examines the left-hand side of the penalised instruction. The analogous conditions apply for the `PTD_arrange_right()` function but have been omitted for the sake of brevity. These conditions are sufficient to preserve the semantics of the program and reduce the PTD measure.

There will be cases where the only way to decrease the PTD measure of a schedule would be to replace a high penalty, i.e. load from memory, with a less expensive one, such as a "move register" instruction. So in terms of the penalty, one of 3 is reduced to 2 by moving an offending instruction, but the goal of reducing the overall measure is still accomplished.

The complexity of the PTD scheduler is $\theta(n\,e)$ where e is the number of penalties in the schedule. The worst case is one in which the schedule has at most $n-1$ consecutive dependencies (a pure sequential code) giving a complexity of $\theta(n^2)$ and the best case is $\theta(n)$. The linear-time complexity for the PTD scheduler is better than the $\theta(n^2)$ for the list scheduler [2] and $\theta(n^2\,\alpha\,n)$ [1] for the balanced scheduler [3].

4 Results

We next compare the quality of schedules produced by the Balanced, Gibbons and Muchnick (GM) and the PTD schedulers for a range of benchmarks which

[1] α is the inverse of the Ackerman function.

represent both loop-intensive (Livermore loops) and control-intensive categories of programs. These were compiled on the SUIF Compiler for the MAP target, but without any MAP-specific optimisations, and provided the same base schedule for the three schedulers under comparison.

The schedules were simulated on a discrete-event model of the MAP architecture. An architecture file describes the functionality and interconnection, and the spread of latencies as shown in Table 2. The distribution of latencies were chosen to best reflect the behaviour of the functional unit. The bimodal distribution for the Memory Unit captures the behaviour due to cache hits and misses. The distribution of the latencies for the Arithmetic Unit is based on the graph in Figure 4 in [4], and the distribution is uniform for the Logic Unit.

The simulation results presented in Figure 4, represent the average of five simulation runs for each program. They represent the percentage improvement with respect to the base case, i.e. the SUIF compiler output. The PTD scheduler outperforms the other two schedulers on both the control-intensive and loop-intensive programs.

When the number of AUs is increased from one to two (Fig. 5), we see a marked improvement in the schedules, but this tapers off when the AUs are increased further. This could be improved upon by scheduling instructions beyond the basic blocks. The favourable run-time complexity of the PTD algorithm makes this a practical proposition.

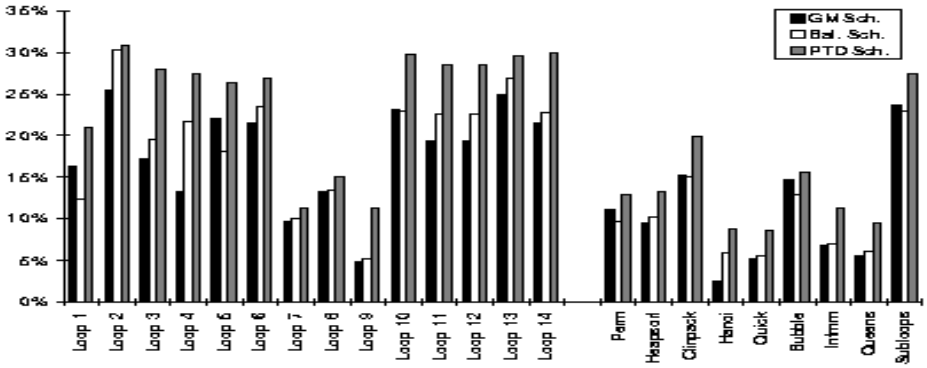


Figure 4. Average improvement for the whole set of benchmarks.

5 Conclusions

The PTD scheduler provides a simple yet effective method for scheduling instructions within basic blocks for programs running on MAP architectures. It has a better time complexity than the other two well-known list schedulers, and

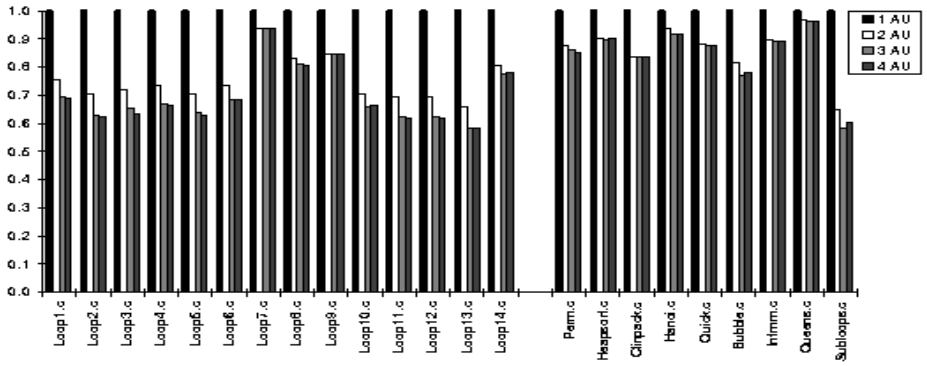


Figure 5. Ratio between the 1 AU and the other configurations.

the quality of the PTD schedules are better for a range of control- and loop-intensive benchmarks. The method reduces the stalls of the Issue Unit due to true data dependencies between instructions and enables better utilisation of the functional units by reducing the resource contention between instructions. The performance of the scheduler was investigated when the number of Arithmetic Units was scaled from 1 to 4. Future work will investigate the scheduling of instructions beyond the basic block boundaries for better utilisation as the functional units are scaled.

Acknowledgements

We would like to thank the members of the MAP group for useful discussions. S. Sotelo-Salazar was supported by a postgraduate studentship from the Science and Technology National Counsel in Mexico (CONACYT).

References

1. D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. *Proc. 3rd. International Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Belgium, August 1994, pp. 203-215.
2. P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proc. SIGPLAN 1986 Symposium on Compiler Construction*, SIGPLAN Notices, 21(7), July 1986, pp. 11-16.
3. D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *In ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 28(6), June 1993, pp. 278-289.
4. D. J. Kinniment. An evaluation of asynchronous addition. *IEEE Transactions on Very Large Scale Integration (VLSI) systems*, March 1996, pp. 137-140.

A.2 An Improved PTD Scheduler for MAP Architectures

Title: An Improved PTD Scheduler for MAP Architectures.

Authors: Damal K. Arvind and Salvador Sotelo-Salazar.

Presented at: The Forth UK Asynchronous Forum.

Place: Imperial College of Science, Technology and Medicine. London, United Kingdom.

Date: July 1998.

An Improved PTD Scheduler for MAP Architectures

D. K. Arvind and S. Sotelo-Salazar
Department of Computer Science, University of Edinburgh
Mayfield Road, Edinburgh EH9 3JZ, Scotland.

Abstract

The Penalise True Dependences (PTD) scheduler considered the effects of true dependences between successive instructions and contention for resources to better utilise the functional units in micronet-based asynchronous processors. This paper presents an improved version which considers dependences beyond successive instructions, and identifies clearly through subgraphing instructions which could be moved to reduce the effects. Performance results are presented where the improved PTD scheduler compares favourably against two well-known list schedulers - the Gibbons and Muchnick [4] and the balanced scheduler [5].

1 Introduction

Micronet-based asynchronous processor (MAP) architectures [1] consist of a network of functional units which operate concurrently and communicate asynchronously. The issue and execution of instructions consist of a sequence of micro-operations involving the Issue Unit, Operand Fetch Unit, the Register File and the appropriate Functional Units (Figure 1). An instruction is issued when its operands are available. It runs to completion unless it is stalled due to resource contention at any of the following points in the trajectory of the instruction: the read ports of the register files, the functional units, and the write-back port. The latencies of the instructions are not fixed (in contrast to clocked processors), but depend on a number of factors: instruction type, the data on which they operate, and the contention for resources which in turn depends on the mix of instructions. Data dependences between successive instructions introduce other delays in asynchronous architectures. In synchronous datapaths, for instance, we can predict exactly when the result of a previous instruction will be available in order to issue the following instruction. In the case of the micronet the issue unit will have to stall for a period of time until the result of the previous instruction is written back to the register file.

In scheduling instructions for a micronet-based target, we seek to minimise the effects of resource contention and data dependences in an environment where the latencies of the instructions are themselves not fixed but vary over a range. In a previous paper [2], we had proposed a method for scheduling instructions for MAP datapaths. The Penalise True Dependence (PTD) scheduler calculates a penalty measure which reflects the degree of resource contention and stalls due to data dependences. The scheduler addresses the problem by moving instructions around which would result in a legal schedule with a lower penalty measure. (Tables 1 and 2 give the range of latencies for the functional units and the penalty measures).

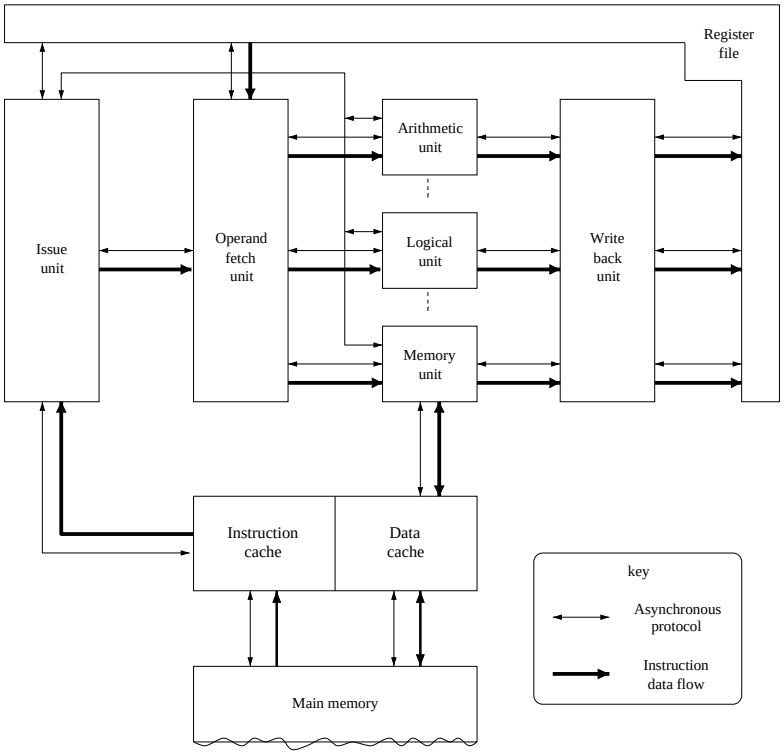


Figure 1: The MAP target architecture

This paper presents improvements to the PTD scheduler. Firstly, candidates for penalties are extended to include dependences beyond just successive instructions. Secondly, the basic blocks of instructions are subdivided into subgraphs to scope the candidates selected for moving instructions.

In the rest of this paper, the algorithm is described and its time complexity is derived, and performance results are presented where the improved PTD scheduler compares favourably against two well-known list schedulers - the Gibbons and Muchnick [4] and the balanced scheduler [5].

Component type	Minimum latency	Maximum latency
Issue Unit (IU)	1.00 ns	2.00 ns
Input buses	2.00 ns	4.00 ns
Output buses	2.00 ns	4.00 ns
Arithmetic Unit (AU)	4.00 ns	8.50 ns
Logical Unit (LU)	2.00 ns	7.00 ns
Memory Unit (MU)	10.00 ns	20.00 ns

Table 1. Latencies values for the target architecture.

Cases of dependences	Consecutive instructions	Separated by one inst.
True dependency with a load inst.	3	1
True dependency with a branch inst.	2	0
Resource dependency within mem. inst.	1	0
Normal true dependences	1	0

Table 2. Table of penalties for true data dependences.

2 An improved PTD scheduler

The objective of the scheduler is to minimise, where possible, the penalty measure for a given schedule of instructions within a basic block. The approach is a greedy one, whereby candidates for movement are chosen such that no new penalties are introduced. This guarantees that the penalty measure is always reduced after each movement. Consecutive instructions are assigned higher penalties as they can potentially result in larger stalls. The value of the penalty falls with the distance between the producer and consumer of the result. The maximum distance that we would need to consider is equal to the number of functional units which can potentially operate in parallel.

The value of the penalty also depends on the types of functional units involved. For example, the cost of a true dependency between a memory load instruction is higher than between a register one. Tables 1 and 2 illustrate the latencies of the different units and the respective penalties. The same idea is extended to penalising resource conflicts.

2.1 Complexity

The time complexity of the improved PTD scheduler is now $\theta(\, p\, nec + p\, nc + p\, n\,)$, where n is the number of instructions in the basic block, e is the number of penalties, p is the number of functional units in the architecture, and c is a small constant ($c = 2, 3, 4$). If we analyse the above expression, the complexity of the scheduler can be reduced to $\theta(\, ne\,)$. However, in general conditions, as the algorithm progresses the number of penalties is reduced and therefore n becomes bigger than e, which means that the complexity can be reduced to $\theta(\, n\,)$.

The upper bound, which is represented by a pure sequential code, is $\theta(\, n^2\,)$, with $e = n-1$ and $c = 2$. Conversely, the lower bound is represented by a pure independent code and is the order of $\theta(\, n\,)$, with $e = 0$.

2.2 Subgraphs

A basic block is composed of a group of instructions that are related in an ordered way which perform computation over data and which may be divided into subcomponents (subgraphs) that perform part of the overall computation. For example, two separate subgraphs would be the computation of an address and the data that would be loaded or stored in that address. An example of a directed acyclic graph DAG with two subgraphs is shown in Figure 2. The node numbering reflects the order in the schedule and the highlighted arcs denote penalties.

A reordering of instructions by moving instruction 5 in between instructions 2 and 3, and instruction 6 in between 3 and 4, resulting in the sequence "2 5 3 6 4", would improve the penalty measure. Any further improvement is restricted by the overlapping chains of dependences between 2 and 3, and, 5 and 6. Ideally, an unrelated instruction between 5 and 3 would further reduce the penalty measure. Dividing the basic block DAG into subgraphs identifies potential source of independent instructions which can be moved to a smaller search area. In the example, the subgraph on the right is a better prospect for independent instructions to move between 2 and 3, and, 3 and 4, and would not result in overlapping chains. In practice, there is a greater probability of finding independent instructions from other subgraphs.

The selection and size of the subgraphs deserve attention. If the size is too small, then there is a greater chance of producing overlapping chains. If the size is too large, then the advantages of the subgraphs are diluted.

The granularity concern for the scheduler can be exemplified in the DAG example in Figure 2. If we choose a subgraph formed from nodes 1, 2, 3 and 4, and another subgraph from nodes 5, 6, 7, 8 and 9, the scheduler would try to intermix the penalties marked and this will end with overlapping chains.

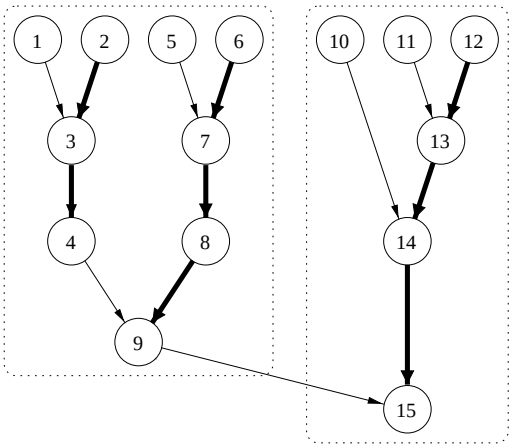


Figure 2: A basic block decomposed by two subgraphs.

The selection of subgraphs is based on the number of predecessors of each instruction and the ratio between the number predecessors and the height from its leaves. The number of predecessors is needed to indicate the size of the DAG at the point and the ratio between this parameter and the actual height is a rough measure of the potential parallelism in that part of the DAG.

3 Results

The improved PTD scheduler was compared against two other local schedulers, the *balanced* scheduler [5] and the original *list* scheduler from Gibbons and Muchnick (GM) [4]. In all the cases the scheduling was performed before register allocation and were tested over a set of benchmarks. An event-driven stochastic simulator was used to simulate them. The target architecture had one memory unit, one arithmetic unit, one logical unit and one branch unit.

The set of benchmarks chosen was the set of Livermore loops [3] (few basic blocks), and a set of control-intensive programs with a larger number of small basic blocks. Figure 3 depicts the comparison in performance for the three schedulers. The results presented are the average of five simulations for each benchmark.

The improved PTD scheduler consistently outperforms the other schedulers for the two set of benchmarks. In the case of `Loop7`, we see the detrimental effect of overlapping chains which are located in the critical path of the basic block in spite of subgraphing.

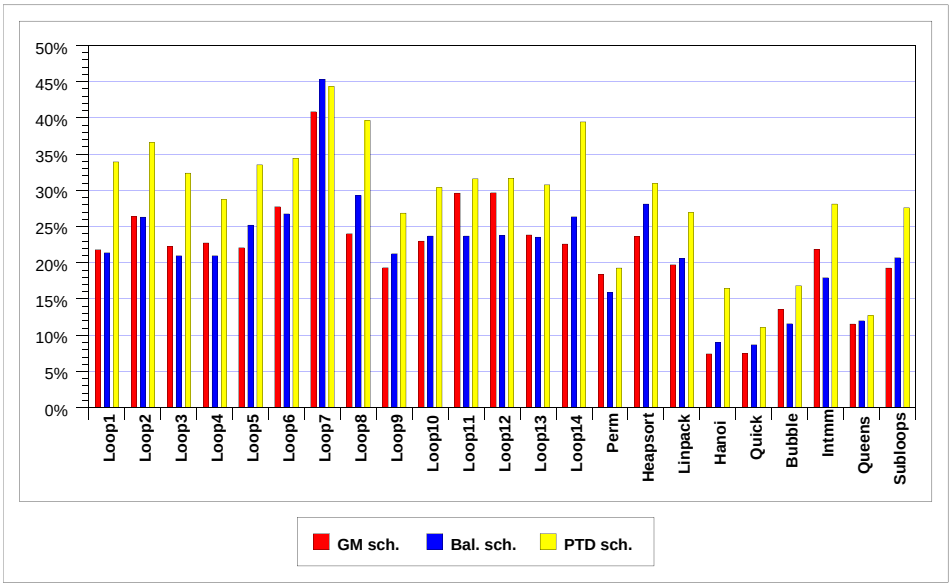


Figure 3: Performance comparison between the three schedulers.

4 Conclusions

The results from the simulations show that the PTD scheduler produces better quality schedules and has a lower time complexity than the list and balanced schedulers (The complexity of the list scheduler being $\theta(n^2)$, and $\theta(n^2\,\alpha(n))$ for the Balanced scheduler). The potential limitation is the introduction of overlapping chains, but in most cases this can be avoided by dividing the basic blocks into subgraphs.

Acknowledgements

We would like to thank the members of the MAP group for useful discussions. S. Sotelo-Salazar was supported by a postgraduate studentship from the Science and Technology National Counsel in Mexico (CONACYT).

References

[1] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. *Proc. 3rd. International Workshop on Algorithms and Parallel VLSI Architectures*, Leuven, Belgium, August 1994, pp. 203-215.

[2] D. K. Arvind and S. Sotelo-Salazar. Scheduling Instructions with Uncertain Latencies in Asynchronous Architectures. *Proc. 3rd. International Euro-Par Conference*. August 1997, pp. 771-778.

[3] J. Feo. An analysis of the computational and parallel complexity of the Livermore Loops. *Parallel Computing* Vol. 7, 1988, pp. 163-185.

[4] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proc. SIGPLAN 1986 Symposium on Compiler Construction*, SIGPLAN Notices, 21(7), July 1986, pp. 11-16.

[5] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. *In ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 28(6), June 1993, pp. 278-289.

Appendix B

Description File

```
#System

II_UNIT 0  1  3    1.00  1.55  2.00   0.00  0.50  1.00

XBUS    1  1  3    2.00  2.70  4.00   2.00  2.70  4.00
YBUS    1  1  3    2.00  3.15  4.00   2.00  3.15  4.00

ALUI    2  1  3    4.00  6.50  8.50   0.00  0.00  0.00
LUNIT   4  1  3    2.00  4.00  7.00   0.00  0.00  0.00
ZBUS    6  1  3    2.00  2.85  4.00   2.00  2.85  4.00

XFBUS   1  1  3    2.00  3.05  4.00   2.00  2.95  4.00
YFBUS   1  1  3    2.00  3.10  4.00   2.00  3.00  4.00
ALUF    3  1  3    6.00  6.95  8.00   0.00  0.00  0.00
ZFBUS   6  1  3    2.00  3.05  4.00   2.00  3.00  4.00

MUNIT   5  1  3   10.00 15.00 20.00   0.00  0.00  0.00
WBUS    1  1  3    2.00  3.25  4.00   2.00  3.25  4.00

#Enddef


#Group 0 Integer Alu Group
Number of Stages 4
II_UNIT
XBUS YBUS
ALUI
ZBUS

#Instruction
add    ALU_SUM
addi   ALU_SUM
addu   ALU_SUMU
addui  ALU_SUMU

sub    ALU_DIF
subi   ALU_DIF
subu   ALU_DIFU
subui  ALU_DIFU

neg    ALU_NEG
```

```
negu   ALU_NEGU

mov    ALU_CPY
la     ALU_LDA
li     ALU_LDC
lai    ALU_LDC
rem    ALU_REM
remi   ALU_REM
remui  ALU_REMU

div    ALU_DIV
divi   ALU_DIV
divu   ALU_DIVU
divui  ALU_DIVU

mul    ALU_MUL
muli   ALU_MUL
mulu   ALU_MULU
mului  ALU_MULU

mtc1      ALU_CPY
mtc1.d    ALU_CPY
ctc1      ALU_CPY
ctc1.d    ALU_CPY


#Group 1 Floating ALU Group
Number of Stages 4
II_UNIT
XFBUS YFBUS
ALUF
ZFBUS

#Instruction
add.s   FALU_SUM
sub.s   FALU_DIF
mul.s   FALU_MUL
div.s   FALU_DIV

add.d   FALU_SUM
sub.d   FALU_DIF
mul.d   FALU_MUL
div.d   FALU_DIV

add.di  FALU_SUM
sub.di  FALU_DIF
mul.di  FALU_MUL
div.di  FALU_DIV

neg.d   FALU_NEG

mov.d   FALU_CPY
mov.s   FALU_CPY

li.d    FALU_LDD
li.s    FALU_LDS
```

```
li.w    FALU_LDW

cfc1    FALU_CPY
cfc1.d  FALU_CPY
mfc1    FALU_CPY
mfc1.d  FALU_CPY

sqrt.d    FALU_SQRT
sqrt.s    FALU_SQRT
sqrt.w    FALU_SQRT

trunc.w.d FALU_TRUNC
trunc.w.s FALU_TRUNC
round.w.d FALU_ROUND
round.w.s FALU_ROUND

cvt.d.s   FALU_LDD
cvt.d.w   FALU_LDD
cvt.s.d   FALU_LDS
cvt.s.w   FALU_LDS
cvt.w.d   FALU_LDW
cvt.w.s   FALU_LDW

c.eq.d    FALU_EQ
c.eq.s    FALU_EQ
c.f.d     FALU_FALSE
c.f.s     FALU_FALSE
c.ge.d    FALU_GE
c.ge.s    FALU_GE
c.gl.d    FALU_GL
c.gl.s    FALU_GL
c.gle.d   FALU_GLE
c.gle.s   FALU_GLE
c.gt.d    FALU_GT
c.gt.s    FALU_GT
c.le.d    FALU_LE
c.le.s    FALU_LE
c.lt.d    FALU_LT
c.lt.s    FALU_LT
c.neq.d   FALU_NEQ
c.neq.s   FALU_NEQ
c.nge.d   FALU_NGE
c.nge.s   FALU_NGE
c.ngl.d   FALU_NGL
c.ngl.s   FALU_NGL
c.ngle.d  FALU_NGLE
c.ngle.s  FALU_NGLE
c.ngt.d   FALU_NGT
c.ngt.s   FALU_NGT
c.nle.d   FALU_NLE
c.nle.s   FALU_NLE
c.nlt.d   FALU_NLT
c.nlt.s   FALU_NLT
c.t.d     FALU_TRUE
c.t.s     FALU_TRUE
```

```
bc1t      BCXT
bc1f      BCXF


#Group 2 Memory load/store group
Number of Stages 4
II_UNIT
XBUS YBUS
MUNIT
ZBUS

#Instruction
lw     LD_W
lwi    LDI_W
sw     ST_W
swi    STI_W

lh     LD_B
lhi    LD_B
lhu    LD_BU
lhui   LDI_BU
sh     ST_B
shi    ST_B
shu    ST_BU
shui   STI_BU

lb     LD_B
lbi    LDI_B
lbu    LD_BU
lbui   LDI_BU
sb     ST_B
sbi    STI_B
sbu    ST_BU
sbui   STI_BU

sd     ST_W

l.d    FLD_W
l.di   FLDI_W
l.s    FLD_W
l.si   FLDI_W

s.d    FST_W
s.di   FSTI_W
s.s    FST_W
s.si   FSTI_W


#Group 3 Logical operation group
Number of stages 4
II_UNIT
XBUS YBUS
LUNIT
ZBUS

#Instruction
```

```
sll    LU_SL
slli   LU_SL

srl    LU_SR
srli   LU_SR

sra    LU_SRA
srai   LU_SRA

and    LU_AND
or     LU_OR
xor    LU_XOR
inv    LU_COM
not    LU_COM

andi   LU_AND
ori    LU_OR
xori   LU_XOR
invi   LU_COM

seq    LU_SEQ
sne    LU_SNE
slt    LU_SEL
sltu   LU_SEL
sle    LU_SLE
sleu   LU_SLE

nop    LU_NOP

jmp    JMP
bf     BRIF
bt     BRIT
call   CALL
ret    RET
reti   RETI
halt   HALT

#Enddef
```

Appendix C

Comparison of the schedulers

C.1 Local Scheduling

	Local Scheduling		
Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	20.37 %	17.30 %	25.85 %
`livermore`	18.66 %	21.72 %	25.76 %
`fract`	22.26 %	26.72 %	30.50 %
`li`	7.14 %	8.13 %	8.89 %
`puzzle`	5.26 %	3.02 %	5.81 %
`compress`	19.26 %	20.25 %	24.01 %
`go`	9.28 %	9.41 %	14.09 %
`m88k`	11.81 %	12.31 %	15.54 %
Average	14.25 %	14.86 %	18.81 %
Geo. Mean	12.68 %	12.45 %	16.51 %

Table C.1: Performance execution improvement for the 1 AU configuration.

	Local Scheduling		
Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	40.55 %	35.19 %	43.37 %
`livermore`	43.52 %	36.90 %	47.40 %
`fract`	24.25 %	28.21 %	30.48 %
`li`	6.45 %	7.08 %	7.38 %
`puzzle`	6.46 %	1.73 %	5.39 %
`compress`	26.63 %	27.44 %	26.85 %
`go`	10.71 %	11.20 %	11.02 %
`m88k`	15.17 %	15.68 %	16.86 %
Average	21.72 %	20.43 %	23.59 %
Geo. Mean	17.22 %	14.68 %	18.27 %

Table C.2: Performance execution improvement for the 2 AU configuration.

	Local Scheduling		
Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	43.96 %	39.02 %	42.26 %
`livermore`	58.61 %	53.27 %	57.76 %
`fract`	24.53 %	28.74 %	31.38 %
`li`	6.35 %	6.92 %	7.16 %
`puzzle`	3.11 %	2.04 %	2.98 %
`compress`	27.74 %	28.45 %	26.65 %
`go`	11.07 %	11.51 %	10.52 %
`m88k`	15.16 %	15.76 %	15.72 %
Average	23.82 %	23.21 %	24.30 %
Geo. Mean	16.62 %	16.02 %	17.06 %

Table C.3: Performance execution improvement for the 3 AU configuration.

	Local Scheduling		
Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	44.66 %	39.39 %	41.31 %
`livermore`	63.27 %	57.98 %	60.50 %
`fract`	24.53 %	28.74 %	30.95 %
`li`	6.35 %	6.91 %	7.15 %
`puzzle`	3.09 %	2.02 %	2.98 %
`compress`	27.82 %	28.63 %	26.63 %
`go`	11.16 %	11.53 %	10.64 %
`m88k`	15.06 %	15.65 %	15.67 %
Average	24.49 %	23.86 %	24.48 %
Geo. Mean	16.80 %	16.20 %	17.10 %

Table C.4: Performance execution improvement for the 4 AU configuration.

C.2 Global Scheduling

	Code Motion			Tail Duplication		
Benchmark	GM. sch.	Bal. sch.	PTD sch.	GM. sch.	Bal. sch.	PTD sch.
`intmm`	21.72 %	17.37 %	26.98 %	20.37 %	17.30 %	25.85 %
`livermore`	19.16 %	22.04 %	26.80 %	18.66 %	21.72 %	25.78 %
`fract`	23.71 %	27.67 %	30.81 %	22.26 %	26.80 %	30.84 %
`li`	7.94 %	8.89 %	9.12 %	7.40 %	8.76 %	9.28 %
`puzzle`	10.92 %	10.96 %	23.47 %	5.35 %	3.14 %	6.04 %
`compress`	20.27 %	21.06 %	27.13 %	19.71 %	20.78 %	24.32 %
`go`	11.42 %	11.80 %	17.35 %	10.04 %	10.20 %	15.05 %
`m88k`	12.60 %	13.11 %	16.22 %	14.16 %	14.61 %	17.71 %
Average	15.97 %	16.61 %	22.23 %	14.75 %	15.41 %	19.36 %
Geo. Mean	14.95 %	15.52 %	20.90 %	13.23 %	13.08 %	17.15 %

Table C.5: Performance execution improvement for the 1 AU configuration.

	Code Motion			Tail Duplication		
Benchmark	GM. sch.	Bal. sch.	PTD sch.	GM. sch.	Bal. sch.	PTD sch.
`intmm`	42.53 %	36.64 %	46.77 %	40.55 %	35.19 %	43.37 %
`livermore`	45.70 %	38.66 %	51.33 %	43.53 %	36.91 %	47.38 %
`fract`	25.85 %	29.33 %	30.67 %	24.26 %	28.22 %	30.48 %
`li`	7.40 %	7.85 %	7.58 %	6.90 %	7.67 %	7.83 %
`puzzle`	14.41 %	12.47 %	22.42 %	6.71 %	1.98 %	5.64 %
`compress`	27.80 %	28.41 %	30.52 %	27.59 %	28.43 %	27.35 %
`go`	15.08 %	15.15 %	17.25 %	11.84 %	12.30 %	12.05 %
`m88k`	17.05 %	17.47 %	17.84 %	18.29 %	18.74 %	19.56 %
Average	24.48 %	23.25 %	28.05 %	22.46 %	21.18 %	24.21 %
Geo. Mean	21.03 %	20.45 %	24.20 %	18.16 %	15.68 %	19.12 %

Table C.6: Performance execution improvement for the 2 AU configuration.

	Code Motion			Tail Duplication		
Benchmark	GM. sch.	Bal. sch.	PTD sch.	GM. sch.	Bal. sch.	PTD sch.
`intmm`	45.71 %	40.26 %	47.10 %	43.96 %	39.02 %	42.26 %
`livermore`	61.65 %	56.08 %	62.82 %	58.63 %	53.26 %	57.62 %
`fract`	26.12 %	29.87 %	31.66 %	24.57 %	29.01 %	31.41 %
`li`	7.34 %	7.74 %	7.37 %	6.82 %	7.49 %	7.62 %
`puzzle`	16.05 %	14.78 %	19.21 %	3.39 %	2.31 %	3.17 %
`compress`	28.93 %	29.68 %	30.34 %	28.79 %	29.54 %	27.09 %
`go`	15.53 %	15.74 %	17.19 %	12.25 %	12.67 %	11.57 %
`m88k`	16.37 %	16.89 %	16.65 %	18.19 %	18.72 %	18.50 %
Average	27.21 %	26.38 %	29.04 %	24.58 %	24.00 %	24.90 %
Geo. Mean	22.42 %	22.29 %	24.14 %	17.65 %	17.10 %	17.93 %

Table C.7: Performance execution improvement for the 3 AU configuration.

	Code Motion			Tail Duplication		
Benchmark	GM. sch.	Bal. sch.	PTD sch.	GM. sch.	Bal. sch.	PTD sch.
`intmm`	46.46 %	40.66 %	46.00 %	44.66 %	39.39 %	41.31 %
`livermore`	66.48 %	61.06 %	65.71 %	63.29 %	57.97 %	60.42 %
`fract`	26.13 %	29.88 %	32.28 %	24.58 %	29.01 %	31.11 %
`li`	7.34 %	7.73 %	7.37 %	6.82 %	7.49 %	7.61 %
`puzzle`	17.40 %	16.02 %	19.21 %	3.36 %	2.28 %	3.16 %
`compress`	28.99 %	29.83 %	30.32 %	28.88 %	29.72 %	27.07 %
`go`	15.61 %	15.73 %	17.34 %	12.32 %	12.67 %	11.70 %
`m88k`	16.27 %	16.77 %	16.68 %	18.09 %	18.61 %	18.45 %
Average	28.08 %	27.21 %	29.36 %	25.25 %	24.64 %	25.10 %
Geo. Mean	22.91 %	22.77 %	24.29 %	17.84 %	17.27 %	17.98 %

Table C.8: Performance execution improvement for the 4 AU configuration.

	Code Motion and Tail Duplication		
Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	21.72 %	17.37 %	29.13 %
`livermore`	19.16 %	22.05 %	24.96 %
`fract`	23.78 %	27.75 %	32.99 %
`li`	8.01 %	9.05 %	9.30 %
`puzzle`	10.96 %	11.00 %	23.53 %
`compress`	20.27 %	21.06 %	27.13 %
`go`	11.93 %	12.39 %	18.10 %
`m88k`	14.14 %	14.65 %	18.62 %
Average	16.25 %	16.92 %	22.97 %
Geo. Mean	15.28 %	15.88 %	21.64 %

Table C.9: Performance execution improvement for the 1 AU configuration.

	Code Motion and Tail Duplication		
Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	42.53 %	36.64 %	46.77 %
`livermore`	45.72 %	38.66 %	50.92 %
`fract`	25.95 %	29.44 %	30.67 %
`li`	7.48 %	8.01 %	7.86 %
`puzzle`	14.60 %	12.65 %	22.48 %
`compress`	27.80 %	28.41 %	30.98 %
`go`	15.76 %	15.88 %	18.10 %
`m88k`	19.74 %	20.09 %	20.84 %
Average	24.95 %	23.72 %	28.58 %
Geo. Mean	21.61 %	21.03 %	24.96 %

Table C.10: Performance execution improvement for the 2 AU configuration.

	Code Motion and Tail Duplication		
Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	45.71 %	40.26 %	47.10 %
`livermore`	61.65 %	56.06 %	62.42 %
`fract`	26.21 %	29.97 %	32.47 %
`li`	7.41 %	7.84 %	7.68 %
`puzzle`	16.17 %	14.89 %	19.22 %
`compress`	28.93 %	29.69 %	30.84 %
`go`	16.28 %	16.52 %	18.08 %
`m88k`	19.34 %	19.80 %	19.73 %
Average	27.71 %	26.88 %	29.69 %
Geo. Mean	23.09 %	22.94 %	25.05 %

Table C.11: Performance execution improvement for the 3 AU configuration.

	Code Motion and Tail Duplication		
Benchmark	GM. sch.	Bal. sch.	PTD sch.
`intmm`	46.46 %	40.66 %	46.00 %
`livermore`	66.48 %	61.06 %	65.52 %
`fract`	26.22 %	29.98 %	33.71 %
`li`	7.41 %	7.83 %	7.67 %
`puzzle`	17.53 %	16.14 %	19.22 %
`compress`	28.99 %	29.83 %	30.93 %
`go`	16.35 %	16.51 %	18.19 %
`m88k`	19.22 %	19.68 %	19.73 %
Average	28.58 %	27.71 %	30.12 %
Geo. Mean	23.59 %	23.44 %	25.27 %

Table C.12: Performance execution improvement for the 4 AU configuration.

Bibliography

[1] V. Agarwal, M. S. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 248–259, June 2000.

[2] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 1986.

[3] V. H. Allan, J. Janardhan, R. M. Lee, and M. Srinivas. Enhanced region scheduling on a program dependence graph. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 72–80, December 1992.

[4] D. K. Arvind, R. D. Mullins, and V. E. F. Rebello. Micronets: A model for decentralising control in asynchronous processor architectures. In *Asynchronous Design Methodologies*, pages 190–199. IEEE Computer Society Press, May 1995.

[5] D. K. Arvind and V. E. F. Rebello. Instruction-level parallelism in asynchronous processor architectures. In *Asynchronous Design Methodologies*, pages 203–215. Elsevier Science Publishers, August 1994.

[6] D. K. Arvind and V. E. F. Rebello. On the performance evaluation of asynchronous processor architectures. In *3rd. International Workshop on Modelling Analysis and Simulation of Computer and Telecommunication Systems MASCOTS'95*, pages 100–105. IEEE Computer Society Press, January 1995.

[7] D. K. Arvind and V. E. F. Rebello. Static scheduling of instructions on micronet-based asynchronous processors. In *2nd International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'96)*, pages 80–91. IEEE Computer Society Press, March 1996.

[8] D. K. Arvind and S. Sotelo-Salazar. Scheduling instructions with uncertain latencies in asynchronous architectures. In *3rd. International Euro-Par Conference*, pages 771–778. Springer, August 1997.

[9] D. I. August et al. Integrated predicated and speculative execution in the IMPACT EPIC architecture. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 227–237, June 1998.

[10] H. G. Baker. Precise instruction scheduling without a precise machine model. *ACM Computer Architecture News*, 19(6):4–8, December 1991.

[11] K. R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley & Sons, 1974.

[12] V. Bala and N. Rubin. Efficient instruction scheduling using finite state automata. In *Proceedings of the 28th Annual International Symposium on Microarchitecture (MICRO-28)*, pages 46–56, December 1995.

[13] S. J. Beaty, S. Colcord, and P. H. Sweany. Using genetic algorithms to fine-tune instruction-scheduling heuristics. In *Proceedings of the Second International Conference on Massively Parallel Computing Systems (MPCS'96).* EuroMicro, May 1996.

[14] M. E. Benitez. Register allocation and phase interactions in retargetable optimizing compilers. Technical Report CS-94-13, University of Virginia, Department of Computer Science, April 1994.

[15] C. H. K. van Berkel. *Handshake Circuits.* Cambridge University Press, 1993.

[16] C. H. K. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F. Schalij. A fully-asynchronous low-power error corrector for the DCC player. *IEEE Journal of Solid-State Circuits*, 29(12):1429–1439, December 1994.

[17] C. H. K. van Berkel, M. B. Josephs, and S. M. Nowick. Scanning the technology. *Proceedings of the IEEE. Special Issue in Asynchronous Circuits*, 87(2):223–233, February 1999.

[18] D. Bernstein, D. Cohen, and H. Krawczyk. Code duplication: An assist for global instructions scheduling. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, pages 103–113, November 1991.

[19] D. Bernstein, D. Cohen, Y. Lavon, and V. Rainish. Performance evaluation of instruction scheduling on the IBM RISC System/6000. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 226–235, November 1992.

[20] D. Bernstein and M. Rodeh. Global instruction scheduling for superscalar machines. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 241–255, June 1991.

[21] D. Berson, R. Gupta, and M. L. Soffa. An evaluation of integrated scheduling and register allocation techniques. In *11th International Workshop on Languages and Compilers for Parallel Computing*, pages 247–262. LNCS Springer Verlag, August 1998.

[22] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, July/August 1999.

[23] D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–131, April 1991.

[24] P. Briggs. Register allocation via graph coloring. Technical Report TR92-183, Rice University, Department of Computer Science, April 1992.

[25] J. Bruno, J. W. Jones, and K. So. Deterministic scheduling with pipelined processors. *IEEE Transactions On Computers*, 29(4):308–316, April 1980.

[26] J. A. Brzozowski and C-J. H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1995.

[27] W. P. Burleson, M. Ciesielski, F. Klass, and W. Liu. Wave-pipelining: A tutorial and research survey. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 6(3):464–474, September 1998.

[28] V. Cerny. A thermodynamical approach to the travelling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Application*, 45(1):41–45, January 1985.

[29] P. P. Chang, S. A. Mahlke, and W. W. Hwu. Using profile information to assist classic code optimisations. *Software Practice and Experience*, 21(12):1301–1321, December 1991.

[30] D. M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems.* PhD thesis, Stanford University, October 1984.

[31] B. Cheng, D. A. Connors, and W. W. Hwu. Optimizing memory accesses using advanced compile-time memory disambiguation techniques. Technical Report IMPACT-99-03, Computer and Systems Research Lab, University of Illinois, June 1999.

[32] H. Chou and C. Chung. An optimal instruction scheduler for superscalar processor. *IEEE Transactions on Parallel and Distributed Systems*, 6(3):303–313, March 1995.

[33] C. Clic. Global code motion/Global value numbering. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 246–257, June 1995.

[34] E. G. Coffman Jr. et al. *Computer and Job-shop Scheduling Theory.* John Wiley & Sons, 1976.

[35] H. Corporaal. *Transport Triggered Architectures. Design and Evaluation.* PhD thesis, Electrical Engineering Department, Delft University of Techonology, January 1995.

[36] H. Corporaal and H. J. M. Mulder. MOVE: A framework for high performance processor design. In *Proceedings of the 1991 International Conference on Supercomputing*, pages 692–701. IEEE Computer Society Press, November 1991.

[37] T. Cramer, R. Friedman, T. Miller, D. Seberger, R. Wilson, and M. Wolczko. Compiling Java just in time: Using runtime compilation to improve Java program performance. *IEEE Micro*, 17(3):36–43, May/June 1997.

[38] B. Davari, R. H. Dennard, and G. G. Shahidi. CMOS scaling for high performance and low power — The next ten years. *Proceedings of the IEEE*, 83(4):595–606, April 1995.

[39] J. W. Davidson and A. M. Holler. Subprogram inlining: A study of its effects on program execution time. *IEEE Transactions on Software Engineering*, 18(2):89–101, February 1992.

[40] J. W. Davidson and S. Jinturkar. An aggressive approach to loop unrolling. Technical Report CS-95-26, University of Virginia, Department of Computer Science, June 1995.

[41] J. W. Davidson and S. Jinturkar. Improving instruction-level parallelism by loop unrolling and dynamic memory disambiguation. Technical Report CS-95-11, University of Virginia, Department of Computer Science, February 1995.

[42] J. B. Dennis. Modular asynchronous control structures for a high performance processor. In *Project MAC Conference on Concurrent Systems and Parallel Computation*, pages 55–80, 1970.

[43] D. J. Eaglesham. $0.18\,\mu m$ CMOS and beyond. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 703–708, June 1999.

[44] J. R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, Cambridge, Mass, 1986.

[45] C. J. Elston, D. B. Christianson, P. A. Findlay, and G. B. Steven. Hades — Towards the design of an asynchronous superscalar processor. In *Asynchronous Design Methodologies*, pages 200–209. IEEE Computer Society Press, May 1995.

[46] T. Emden-Weinert and M. Proksch. Best practice simulated annealing for the airline crew scheduling problem. *Journal of Heuristics*, 5(4):419–436, December 1999.

[47] P. B. Endecott. *SCALP: A Superscalar Asynchronous Low-Power Processor.* PhD thesis, Department of Computer Science, University of Manchester, 1996.

[48] P. B. Endecott. Superscalar instruction issue in an asynchronous microprocessor. *IEE Proceedings on Computers and Digital Techniques*, 143(5):266–272, September 1996.

[49] J. T. Feo. An analysis of the computational and parallel complexity of the Livermore loops. *Parallel Computing*, 7(2):163–185, June 1988.

[50] J. Ferrante, K. J. Ottenstein, and J. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[51] J. A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30(7):478–490, July 1981.

[52] M. J. Flynn, P. Hung, and K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4):11–22, July/August 1999.

[53] M. Franklin and G. S. Sohi. ARB: A hardware mechanism for dynamic reordering of memory references. *IEEE Transactions On Computers*, 45(5):552–571, May 1996.

[54] S. M. Freudenberger, T. R. Gross, and P. G. Lowney. Avoidance and suppression of compensation code in a trace scheduling compiler. *ACM Transactions on Programming Languages and Systems*, 16(4):1156–1214, July 1994.

[55] S. B. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions On VLSI Systems*, 4(2):247–253, June 1996.

[56] S. B. Furber, P. Day, N. C. Paver, and J. V. Woods. AMULET1: A micropipelined ARM. In *Proceedings of the IEEE Computer Conference (CompCon'94)*, pages 476–485, March 1994.

[57] S. B. Furber, J. D. Garside, S. Temple, P. Day, and N. C. Paver. AMULET2e. In *Embedded Microprocessor Systems EMSYS'96 - OMI Sixth Annual Conference*, September 1996.

[58] S. B. Furber, J. D. Garside, S. Temple, J. Liu, P. Day, and N. C. Paver. AMULET2e: An asynchronous embedded controller. In *3rd. International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'97)*, pages 290–299. IEEE Computer Society Press, April 1997.

[59] H. van Gageldonk, D. Baumann, C. H. K. van Berkel, D. Gloor, A. Peeters, and G. Stegmann. An asynchronous low-power 80C51 microcontroller. In *4th International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'98)*, pages 96–107. IEEE Computer Society Press, April 1998.

[60] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu. Dynamic memory disambiguation using the memory conflict buffer. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 183–195, October 1994.

[61] M. R. Garey and D. R. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, San Francisco, 1979.

[62] J. D. Garside. A CMOS VLSI implementation of an asynchronous ALU. In *Asynchronous Design Methodologies*, pages 181–207. Elsevier Science Publishers, 1993.

[63] J. D. Garside, S. B. Furber, and S.-H. Chung. AMULET3 revealed. In *5th International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'99)*, pages 51–59. IEEE Computer Society Press, April 1999.

[64] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, 21(7):11–16, July 1986.

[65] N. Giffin. *Fractal Formula File From Noel Giffin.* `http://spanky.fractint.org/pub/fractals/formulas/noel1.frm`.

[66] N. Giffin. *Hydra Set.* `http://spanky.fractint.org/pub/fractals/images/noel/full_sets/hydra.png`.

[67] M. J. González Jr. Deterministic processor scheduling. *ACM Computing Surveys*, 9(3):173–204, September 1977.

[68] M. R. Greenstreet and B. de Alwis. How to achieve worst-case performance. In *7th International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'01)*, pages 206–217. IEEE Computer Society Press, March 2001.

[69] J. W. Grossman and R. S. Zeitman. An inherently iterative computation of Ackermann's function. *Theoretical Computer Science*, 57(2-3):327–330, May 1988.

[70] R. Gupta and M. L. Soffa. Region scheduling: An approach for detecting and redistributing parallelism. *IEEE Transactions on Software Engineering*, 16(4):421–431, April 1990.

[71] O. Hauck and S. A. Huss. Asynchronous wave pipelines for high throughput datapaths. In *IEEE International Conference on Electronics, Circuits and Systems*, pages 283–286, September 1998.

[72] S. Hauck. Asynchronous design methodologies: An overview. *Proceedings of the IEEE*, 83(1):69–93, January 1995.

[73] A. Hemani, T. Meincke, S. Kumar, A. Postula, T. Olsson, P. Nilsson, J. Öberg, P. Ellervee, and D. Lundqvist. Lowering power consumption in clock by using globally asynchronous, locally synchronous design style. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 873–878, June 1999.

[74] J. L. Hennessy and T. Gross. Postpass code optimisation of pipeline constraints. *ACM Transactions on Programming Languages and Systems*, 5(3):422–448, July 1983.

[75] J. L. Hennessy, N. P. Jouppi, F. Baskett, T. Gross, and J. Gill. Hardware/software tradeoffs for increased performance. In *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 2–11. ACM Press, March 1982.

[76] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, 2nd. edition, 1996.

[77] J. Hoogerbrugge. *Code Generation for Transport Triggered Architectures.* PhD thesis, Department of Electrical Engineering, Delft University of Technology, February 1996.

[78] J. Hoogerbrugge and H. Corporaal. Register file port requirements of transport triggered architectures. In *Proceedings of the 27th Annual International Symposium on Microarchitecture (MICRO-27)*, pages 191–195, December 1994.

[79] J. Hoogerbrugge and H. Corporaal. Transport-triggering versus operation-triggering. In *5th International Conference on Compiler Construction (CC'94)*, pages 435–449, April 1994.

[80] J. Hoogerbrugge and H. Corporaal. Resource assignment in a compiler for transport triggered architectures. In *Proceedings of the 2nd Annual Conference of the Advanced School for Computing and Imaging*, June 1996.

[81] M. S. Hrishikesh, N. P. Jouppi, K. I. Farkas, D. Burger, S. W. Keckler, and P. Shivakumar. The optimal logic depth per pipeline stage is 6 to 8 FO4 inverter delays. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 14–24, May 2002.

[82] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9(6):841–848, 1961.

[83] H. Hulgaard, S. M. Burns, and G. Borriello. Testing asynchronous circuits: A survey. Technical Report 94-03-06, Department of Computer Science and Engineering, University of Washington, March 1994.

[84] K. Hwang. *Advanced Computer Architecture: Parallelism, Scalability and Programmability.* McGraw-Hill, 1st. edition, 1993.

[85] W. W. Hwu et al. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, 7(1–2):229–248, May 1993.

[86] W. W. Hwu et al. Compiler technology for future microprocessors. *Proceedings of the IEEE*, 83(12):1623–1640, December 1995.

[87] M. J. Bourke III, P. H. Sweany, and S. J. Beaty. Extending list scheduling to consider execution frequency. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, pages 122–131, January 1996.

[88] H. Iwai. CMOS technology — Year 2010 and beyond. *IEEE Journal of Solid-State Circuits*, 34(3):357–366, March 1999.

[89] J. Jansen. *Compiler Strategies for Transport Triggered Architectures.* PhD thesis, Electrical Engineering Department, Delft University of Techonology, September 2001.

[90] R. Johnson, D. Pearson, and K. Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI)*, pages 171–185, June 1994.

[91] G. Kane and J. Heinrich. *MIPS RISC Architecture.* Prentice Hall, 1992.

[92] D. R. Kerns and S. J. Eggers. Balanced scheduling: Instruction scheduling when memory latency is uncertain. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, pages 278–289, June 1993.

[93] J. Kessels and P. Marston. Designing asynchronous standby circuits for a low-power pager. In *3rd. International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'97)*, pages 268–278. IEEE Computer Society Press, April 1997.

[94] D. J. Kinniment. An evaluation of asynchronous addition. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):137–140, March 1996.

[95] S. Kirkpatrick, C. D. Gelatt Jr., and M.P. Vecchi. Optimisation by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[96] J. Knoop, O. Rüthing, and B. Steffen. The power of assignment motion. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 233–245, June 1995.

[97] M. Ko. Instruction scheduling for Micronet-based asynchronous processors. Master's thesis, Department of Computer Science, University of Edinburgh, 1995.

[98] G. Koren and D. Shasha. An optimal scheduling algorithm with a competitive factor for real-time systems. Technical Report TR1991-572, New York University, July 1991.

[99] G. Kostadinidis et al. Implementation of a third-generation 1.1GHz 64b microprocessor. In *Proceedings of the 2002 International IEEE Solid-State Circuits Conference ISSCC'2002*, pages 726–731, February 2002.

[100] S. M. Kurlander, T. A. Proebsting, and C. N. Fischer. Efficient instruction scheduling for delayed-load architectures. *ACM Transactions on Programming Languages and Systems*, 17(5):740–776, September 1995.

[101] A. Leung, K. V. Palem, and A. Pnueli. A fast algorithm for scheduling time-constrained instructions on processors with ILP. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 158–166, October 1998.

[102] A. Leung, K. V. Palem, and C. Ungureanu. Run-time versus compile-time instruction scheduling in superscalar (RISC) processors: Performance and tradeoffs. Technical Report 699, New York University, Computer Science, July 1995.

[103] D. J. Lilja. Exploiting the parallelism available in loops. *Computer*, 27(2):13–27, February 1994.

[104] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification.* Addison Wesley, 1996.

[105] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, R. P. Nix, J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow Trace Scheduling compiler. *The Journal of Supercomputing*, 7(1–2):51–142, May 1993.

[106] Luqi and M. Shing. Real-time scheduling for software prototyping. *Journal of Systems Integration. Special Issue on Computer-Aided Prototyping*, 6(1):41–72, May 1996.

[107] U. Mahadevan and S. Ramakrishnan. Instruction scheduling over regions: A framework for scheduling across basic blocks. In *5th International Conference on Compiler Construction (CC'94)*, pages 419–434, April 1994.

[108] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann. Effective compiler support for predicated execution using the Hyperblock. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 45–54, December 1992.

[109] R. T. Maniwa. *Global Distribution: Clocks and Power.* `http://www.eedesign.com/editorial/1995/coverstory9508.html`.

[110] A. J. Martin. Programming in VLSI: From communication processes to delay-insensitive circuits. In *Developments in Concurrency and Communication*, pages 1–64. Addison-Wesley, 1990.

[111] A. J. Martin, S. M. Burns, T. K. Lee, D. Borković, and P. J. Hazewindus. The design of an asynchronous microprocessor. In *Decennial Caltech Conference on VLSI*, pages 351–273. C. L. Seitz, MIT Press, March 1989.

[112] A. J. Martin, S. M. Burns, T. K. Lee, D. Borković, and P. J. Hazewindus. The first asynchronous microprocessor: The test results. *ACM Computer Architecture News*, 17(4):95–110, April 1989.

[113] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Penzes, R. Southworth, and U. Cummings. The design of an asynchronous MIPS R3000 microprocessor. In *Advanced Research in VLSI*, pages 164–181, September 1997.

[114] H. Massalin. Superoptimiser — A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 122–126, October 1987.

[115] D. E. Maydan, J. L. Hennessy, and M. S. Lam. Efficient and exact data dependence analysis. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, June 1991.

[116] A. Moitra. Analysis of hard real-time systems. Technical Report TR85-693, Department of Computer Science, Cornell University, July 1985.

[117] J. Moreno et al. Architecture compiler and simulation of a tree-based VLIW processor. Technical Report RC20495, IBM Research Division, Computer Sciences/Mathematics, July 1996.

[118] R. Motwani, K. V. Palem, V. Sarkar, and S. Reyen. Combining register allocation and instruction scheduling. Technical Report CS-TN-95-22, Stanford University, Department of Computer Science, August 1995.

[119] F. Mueller and D. B. Whalley. Avoiding unconditional jumps by code replication. In *Proceedings of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pages 322–330, June 1992.

[120] F. Mueller and D. B. Whalley. Avoiding conditional branches by code replication. In *Proceedings of the ACM SIGPLAN'95 Conference on Programming Language Design and Implementation (PLDI)*, pages 56–66, June 1995.

[121] T. Nanya, Y. Ueno, H. Kagotani, M. Kuwako, and A. Takamura. TITAC: Design of a quasi-delay-insensitive microprocessor. *IEEE Design & Test of Computers*, 11(2):50–63, Summer 1994.

[122] C. Norris and L. L. Pollock. Register allocation sensitive region scheduling. In *Proceedings of the 1995 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 1–10, June 1995.

[123] S. Onder and R. Gupta. Dynamic memory disambiguation in the presence of out-of-order store issuing. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture (MICRO-32)*, pages 170–176, November 1999.

[124] K. V. Palem and B. B. Simons. Scheduling time-critical instructions on RISC machines. *ACM Transactions on Programming Languages and Systems*, 15(4):632–658, September 1993.

[125] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, January 1985.

[126] N. C. Paver, P. Day, C. Farnsworth, D. L. Jackson, W. A. Lien, and J. Liu. A low-power, low noise, configurable self-timed DSP. In *4th International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'98)*, pages 32–42. IEEE Computer Society Press, April 1998.

[127] N. C. Paver, P. Day, S. B. Furber, J. D. Garside, and J. V. Woods. Register locking in an asynchronous microprocessor. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 351–355. IEEE Computer Society Press, October 1992.

[128] M. Pedram. Power minimization in IC design: Principles and applications. *ACM Transactions on Design Automation of Electronic Systems*, 1(1):3–56, January 1996.

[129] O. A. Petlin and S. B. Furber. Built-in testing of micropipelines. In *3rd. International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'97)*, pages 22–29. IEEE Computer Society Press, April 1997.

[130] M. Pinedo. *Scheduling: Theory Algorithm, and Systems.* Prentice Hall, 1995.

[131] S. S. Pinter. Register allocation with instruction scheduling: A new approach. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI)*, pages 248–257, June 1993.

[132] T. A. Proebsting and C. N. Fischer. Linear-time, optimal code scheduling for delayed-load architectures. In *Proceedings of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation (PLDI)*, pages 256–267, June 1991.

[133] W. Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 Conference on Supercomputing*, pages 4–13, November 1991.

[134] J. M. Rabaey and M. Pedram. *Low Power Design Methodologies.* Kluwer Academic Publishers, 1996.

[135] B. Rahardjo. *Asynchronous Tools, available on the Internet.* `http://www.cs.man.ac.uk/async/tools/index.html`.

[136] B. R. Rau and J. A. Fisher. Instruction-level parallelism processing: History, overview and perspective. *The Journal of Supercomputing*, 7(1–2):9–50, May 1993.

[137] C. A. Rey and J. Vaucher. Self-synchronized asynchronous sequential machines. *IEEE Transactions On Computers*, 23(12):1306–1311, December 1974.

[138] W. F. Richardson and E. L. Brunvand. The NSR processor prototype. Technical Report UUCS-92-029, Department of Computer Science, University of Utah, December 1992.

[139] W. F. Richardson and E. L. Brunvand. Fred: An architecture for a self-timed decoupled computer. Technical Report UUCS-95-008, Department of Computer Science, University of Utah, May 1995.

[140] P. A. Riocreux, L. E. M. Brackenbury, M. Cumpstey, and S. B. Furber. A low-power self-timed Viterbi decoder. In *7th International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'01)*, pages 15–24. IEEE Computer Society Press, March 2001.

[141] S. Rotem et al. RAPPID: An asynchronous instruction length decoder. In *5th International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'99)*, pages 60–70. IEEE Computer Society Press, April 1999.

[142] M. S. Schlansker et al. Achieving high levels of instruction-level parallelism with reduced hardware complexity. Technical Report HPL-96-120, HP Laboratories, November 1994.

[143] M. S. Schlansker and B. R. Rau. EPIC: An architecture for instruction-level parallel processors. Technical Report HPL-1999-111, HP Laboratories, February 2000.

[144] M. A. Schuette and J. P. Shen. An instruction-level performance analysis of the Multiflow TRACE 14/300. In *Proceedings of the 24th Annual International Symposium on Microarchitecture (MICRO-24)*, pages 2–11. IEEE Computer Society Press, November 1991.

[145] C. L. Seitz. Self-timed VLSI systems. In *Proceedings of the 1st Caltech Conference on Very Large Scale Integration*, pages 345–355, January 1979.

[146] Semiconductor Industry Association. *The International Technology Roadmap for Semiconductors*, San Jose, Calif. 1994, 1997, 2000 and 2001.

[147] SGI Compiler Group. *SGI Pro64*™. `http://oss.sgi.com/projects/Pro64/`.

[148] H. Sharangpani and K. Arora. Itanium processor microarchitecture. *IEEE Micro*, 20(5):24–43, September/October 2000.

[149] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark. Branch prediction, instruction window size, and cache size: Performance trade-offs and simulation techniques. *IEEE Transactions On Computers*, 48(11):1260–1281, November 1999.

[150] M. J. Sebastian Smith. *Application-Specific Integrated Circuits.* Addison-Wesley, 1997.

[151] C. Sotiriou. *Design of an Asynchronous Processor.* PhD thesis, Division of Informatics, University of Edinburgh, December 2000.

[152] SPEC95. *SPEC CINT95 Benchmarks.* `http://www.spec.org/osg/cpu95/CINT95/`.

[153] R. F. Sproull, I. E. Sutherland, and C. E. Molnar. Counterflow pipeline processor architecture. Technical Report SMLI TR-94-25, Sun Microsystems Laboratories Inc., April 1994.

[154] Stanford SUIF Compiler Group. *The SUIF Compiler — Man Pages, Porky.* `http://suif.stanford.edu/suif/suif1/docs/man_porky.1.html`.

[155] Stanford SUIF Compiler Group. *The SUIF Compiler System.* `http://suif.stanford.edu/`.

[156] Stanford SUIF Compiler Group. *The SUIF Library (Version 1.0).* `http://suif.stanford.edu/suif/suif1/docs/suif_toc.html`.

[157] J. A. Stankovic. Real-time and embedded systems. *ACM Computing Surveys*, 28(1):205–208, March 1996.

[158] K. Stevens et al. CAD directions for high performance asynchronous circuits. In *Proceedings of the 36th ACM/IEEE Design Automation Conference*, pages 116–121, June 1999.

[159] K. Stevens, R. Ginosar, and S. Rotem. Relative timing. In *5th International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'99)*, pages 208–218. IEEE Computer Society Press, April 1999.

[160] T. Suganuma et al. Overview of the IBM Java just-in-time compiler. *IBM Systems Journal – Java Performance*, 39(1):175–193, 2000.

[161] I. E. Sutherland. Micropipelines. *Communications of the ACM*, 32(6):720–738, June 1989.

[162] M. N. S. Swamy and K. Thulasiraman. *Graphs, Networks, and Algorithms.* John Wiley & Sons, 1981.

[163] P. H. Sweany and S. J. Beaty. Dominator path scheduling — A global scheduling method. In *Proceedings of the 25th Annual International Symposium on Microarchitecture (MICRO-25)*, pages 260–263, December 1992.

[164] A. Takamura et al. TITAC-2: An asynchronous 32-bit microprocessor based on scalable-delay-insensitive model. In *Proceedings of the International Conference on Computer Design (ICCD)*, pages 288–294. IEEE Computer Society Press, October 1997.

[165] J. Teifel and R. Manohar. A high-speed clockless serial link transciver. In *9th International Symposium on Advanced Research on Asynchronous Circuits and Systems (ASYNC'03)*, pages 151–161. IEEE Computer Society Press, May 2003.

[166] H. Terada, S. Miyata, and M. Iwata. DDMP's: Self-timed super-pipelined data-driven multimedia processors. *Proceedings of the IEEE*, 87(2):282–296, February 1999.

[167] J. A. Tierno, A. J. Martin, and D. Borković. An asynchronous microprocessor in gallium arsenide. Technical Report CS-TR-93-38, Department of Computer Science, California Institute of Technology, November 1993.

[168] V. Tiwari, D. Singh, S. Rajgopal, G. Mehta, R. Patel, and F. Baez. Reducing power in high-performance microprocessors. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*, pages 732–737, June 1998.

[169] Trimaran Consortium. *An Infrastructure for Research in Instruction-Level Parallelism.* `http://www.trimaran.org/`.

[170] J. D. Ullman. NP-complete scheduling problems. *Journal of Computer and Systems Sciences*, 10(3):384–393, June 1975.

[171] M. G. Valluri and R. Govindarajan. Evaluating register allocation and instruction scheduling techniques in out-of-order issue processors. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 78–83, October 1999.

[172] T. Verhoeff. Delay-insensitive codes — An overview. *Distributed Computing*, 3(1):1–8, 1988.

[173] T. Verhoeff. *A Theory of Delay-Insensitive Systems.* PhD thesis, Eindhoven University of Technology, Faculty of Mathematics and Computing Science, 1994.

[174] H. S. Warren Jr. Instruction scheduling for the IBM RISC System/6000 processor. *IBM Journal on Research and Development*, 34(1):85–91, January 1990.

[175] N. H. E. Weste and K. Eshraghian. *Principles of CMOS VLSI Design.* Addison-Wesley, 2nd. edition, 1993.

[176] R. P. Wilson et al. *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers.* `http://suif.stanford.edu/suif/suif1/suif-overview/suif.html`.

[177] Q. Wu, M. Pedram, and X. Wu. Clock-gating and its application to low power design of sequential circuits. *IEEE Transactions on Circuits and Systems*, 47(103):415–420, March 2000.

[178] Y. Zorian. System-chip test strategies. In *Proceedings of the 35th ACM/IEEE Design Automation Conference*, pages 752–757, June 1998.