ADDING LIMITED RECONFIGURABILITY TO SUPERSCALAR PROCESSORS

THÈSE N° 3124 (2004)
PRÉSENTÉE À LA FACULTÉ SCIENCES ET TECHNIQUES DE L'INGÉNIEUR
Institut de traitement des signaux
SECTION D'ÉLECTRICITÉ
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE
POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES TECHNIQUES

PAR
Marc EPALZA
ingénieur électricien diplômé EPF
de nationalité suisse et originaire du Grand-Saconnex (GE)

acceptée sur proposition du jury:
Prof. D. Mlynek, directeur de thèse
Dr B. Hochet, rapporteur
Prof. M. Kunt, rapporteur
Dr D. Nicoulaz, rapporteur

Lausanne, EPFL
2004
Abstract

For the last thirty years, electronics, at first built with discrete components, and then as *Integrated Circuits* (IC), have brought diverse and lasting improvements to our quality of life. Examples might include digital calculators, automotive and airplane control assistance, almost all electrical household appliances, and the almost ubiquitous *Personal Computer*. *Application-Specific Integrated Circuits* (ASICs) were traditionally used for their high performance and low manufacturing cost, and were designed specifically for a single application with large volumes. But as lower product lifetimes and the pressures of fast marketing increased, ASICs’ high design cost pushed for their replacement by *Microprocessors*. These processors, capable of implementing any functionality through a change in software, are thus often called *General Purpose Processors*.

General purpose processors are used for everyday computing tasks, and found in all personal computers. They are also often used as building blocks for scientific supercomputers. Superscalar processors such as these require ever more processing power to run complex simulations, video games or versatile telecoms services. In the case of embedded applications, e.g. for portable devices, both performance and power consumption must be taken into account.

In a bid to adapt a processor to some extent to select applications, fully reconfigurable logic can greatly improve the performance of a processor, since it is shaped for the best possible execution with the available resources. However, as reconfigurable logic is far slower than custom logic, this gain is possible only for some specific applications with large parallelism, after a detailed study of the algorithm. Even though this process can be automated, it still requires large computing resources, and cannot be performed at run time.

To reduce the loss in speed compared to custom logic, it is possible to limit the reconfigurability to increase the breadth of applications where performance can be improved. However, as the application space increases, a careful analysis and design of the reconfigurability is required to minimize the speed loss, notably when dynamic reconfiguration is considered.
As a case study, we analyze the feasibility of adding limited reconfigurability to the Floating Point Units (FPUs) of a general purpose processor. These rather large units execute all floating point operations, and may also be used for integer multiplication. If an application contains few or infrequent instructions that must be executed by the FPU, this idle hardware only increases power consumption without enhancing performance. This is often the case in non-scientific applications and even many recent and detailed video games which make heavy use of hardware display accelerators for 3D graphics.

In a fast multiplier such as can be found in the FPU of a high performance processor, the logic to perform multiplication is a large tree of compressors to add all the partial products together. It is possible to add logic to allow the reconfiguration of part of this tree as several extra Arithmetic and Logic Units (ALU). This requires a detailed timing analysis for both the reconfigurable FPU and the extra ALUs, taking into account effects such as added wires and longer critical paths. Finally, the algorithm to decide when and how to reconfigure must be studied, in terms of efficiency and complexity.

The results of adding this limited reconfigurability to a mainstream superscalar processor over a large set of compute intensive benchmarks show gains of up to 56% in the best case, with an average gain of 11%. The application to an idealized huge top processor still shows slightly positive average gains, as the limits of available parallelism are reached, bounded by both the application and many of the characteristics of the processor. In all cases, binary compatibility is maintained, allowing the re-use of all existing software.

We show that adding limited reconfigurability to a general purpose superscalar processor can produce interesting gains over a wide range of applications while maintaining binary compatibility, and without large modifications to the original design. Limited reconfigurability is worthwhile as it increases the design space, allowing gains to apply to a larger set of applications. These gains are achieved through careful study and optimization of the reconfigurable logic and the decision algorithm.
Version Abrégée

Durant les trente dernières années, l’électronique, d’abord construite avec des composants discrets, puis sous forme de Circuits Intégrés (IC)), a apporté de nombreuses améliorations durables à notre qualité de vie. Les calculatrices, l’aide au pilotage de voitures ou d’avions, presque tous les appareils ménagers et le presque omniprésent Ordinateur Personnel (PC) en sont des exemples. Les Circuits Intégrés Dédies à une Application (ASIC) ont traditionnellement été utilisés pour leurs hautes performances et leur bas prix de fabrication, et étaient conçus spécialement pour une seule application et produits en très grandes quantités. Mais la diminution de la durée de vie des produits et les pressions pour les mettre le plus rapidement possible sur le marché ont poussé au remplacement des ASICs aux coûts de développement élevés par des Microprocesseurs. Ces processeurs, capables d’implémenter n’importe quelle fonctionnalité par une modification du logiciel, sont donc souvent appelés Processeurs Génériques.

Les processeurs génériques sont utilisés pour les tâches informatiques de tous les jours et présents dans tous les ordinateurs personnels. Ils sont aussi souvent utilisés comme blocs de base pour la construction de superordinateurs. Ces processeurs superscalaires ont besoin d’encore plus de puissance de calcul pour exécuter des jeux vidéo ou des simulations complexes. Dans le cas d’applications embarquées, par exemple pour des appareils portables, la puissance de calcul et la consommation d’énergie doivent être pris en compte.

Dans le but de s’adapter dans une certaine mesure à quelques applications précises, un circuit entièrement reconfigurable peut améliorer de façon notable les performances d’un processeur, dans la mesure où ce circuit prend alors la forme la plus efficace possible vu les ressources disponibles. Cependant, comme les circuits reconfigurables sont beaucoup plus lents que les circuits dédiés, ce gain de performances n’est possible que pour certaines applications bénéficiant d’un large parallélisme, et suite à une étude détaillée de l’algorithme concerné. Même en automatisant ce processus, une grande puissance de calcul est nécessaire et la reconfiguration ne peut donc pas être effectuée pendant l’utilisation du circuit.

Pour minimiser la perte de vitesse face à un circuit dédié, il est possible
de limiter la reconfigurabilité pour accroître l’éventail des applications dont la performance peut être améliorée. Cependant, une conception minutieuse de la reconfigurabilité est nécessaire pour réduire au maximum la perte de vitesse qui accompagne l’élargissement de l’espace des applications, en particulier lorsque la reconfiguration dynamique entre en jeu.

Comme exemple, la faisabilité d’ajouter une reconfiguration limitée aux unités de calcul en virgule flottante (FPU) d’un processeur générique sera analysée. Ces unités plutôt volumineuses exécutent toutes les instructions à virgule flottante, et peuvent aussi être utilisées pour les multiplications entières. Si une application contient des instructions nécessitant la FPU qui sont peu nombreuses ou intermittentes, ces transistors ne font que consommer de l’énergie. Ceci est souvent le cas dans des applications non scientifiques, ainsi que dans beaucoup de jeux vidéo récents qui font grand usage d’accélérateurs graphiques pour les images en 3 dimensions.

Dans un multiplieur rapide tel que celui présent dans la FPU d’un processeur à haute performance, le circuit effectuant la multiplication est un grand arbre de compression qui sert à additionner tous les produits partiels. Il est possible d’ajouter des transistors pour permettre la reconfiguration d’une partie de cet arbre sous la forme de plusieurs unités arithmétiques supplémentaires (ALUs). Cette opération requiert une analyse temporelle détaillée pour la FPU reconfigurable et les ALUs supplémentaires, en tenant compte d’effets dus à des fils supplémentaires et un chemin critique plus long, par exemple. Enfin, l’algorithme pour décider de la configuration à prendre à chaque instant doit aussi être analysé en termes d’efficacité et de complexité.

Les résultats de l’addition de cette reconfigurabilité limitée à un processeur superscalaire de moyenne gamme montrent des gains jusqu’à 56% dans les meilleurs cas, avec un gain moyen de 11% sur une vaste palette de programmes utilisant intensivement le processeur. L’application à un processeur haut de gamme idéalisé montre encore des gains très légèrement positifs, alors que les limites du parallélisme disponible sont atteintes, celui-ci étant borné par les applications et les nombreuses caractéristiques du processeur. Dans tous les cas, la compatibilité binaire est préservée, permettant l’utilisation de tous les logiciels existants sur ce processeur reconfigurable.

Cette thèse montre que l’ajout d’une reconfigurabilité limitée à un processeur superscalaire générique donne des gains de performance intéressants sur une vaste palette d’applications et ce en maintenant la compatibilité binaire et avec peu de changements à apporter au processeur original. La reconfigurabilité limitée est intéressante car elle augmente l’espace des choix du concepteur en permettant des gains dans un plus grand éventail d’applications. Ces gains sont obtenus au prix d’une étude rigoureuse et d’une optimisation des circuits reconfigurables et de l’algorithme de décision.
Acknowledgments

As with all works of this size, although my name is on the cover, this thesis would not exist without the many contributions to this work, large or small, that many others have made.

First, I would like to thank Ana, Séverine, and all my very good friends, for not laughing too loud when I announced I was planning on pursuing a PhD, and for their warm supppport whenever problems arose, academic or otherwise. Sergei in particular was a great help in all things mathematic.

Second, I must thank my thesis adviser, prof. Daniel Mlynek, for giving me the opportunity to do this PhD, and also for the glimpse at corporate life through the two years at Transwitch. Most of all, he believed in me, and allowed me to follow the subject I was interested in, even if he never believed it would actually work!

I would like to thank all the folks (and ex-folks) from Transwitch for their assistance and advice during my years there, but especially Stéphane Devise, for his patience as my direct supervisor, Marc Morgan, for introducing me to the wonders of Unix and its command line, and most of all vi, and Yves Pizzuto, for his help in all things software, and for answering my numerous annoying questions about C++.

Prof. Paolo Ienne, director of the Processor Architecture Lab (LAP) has my warmest thanks for his interest in my work ("When I applied for the post as professor at EPFL, I explicitly said I would NOT do this kind of stuff!"), and for helping me turn a wild idea into a couple of published papers and something vaguely resembling a thesis. Thank you for your time, even at incredible hours, and your attention to detail.

I should next offer my thanks to Sylvain Aguirre, first for his help in my work on SystemC at TranSwitch, and then as my office companion-in-labor, for his assistance in VHDL and the many fruitful discussions regarding more topics than I can remember. I wish you all the best for your own PhD work!

My thanks go to the Spanish clan, Francesc Font, Eduardo Juarez and vii
David Tarongi, for their varied help, support, ideas, and general companionship during my years here.

I would like to thank Ljubisa Miskovic for taking the time to listen to my vexing problem about the reconfiguration decision mechanism, and for astutely referring me to Michel Bierlaire, who took this problem I had spent several months on and off fighting and said: "Oh, this is easy. We do it all the time!", and then proceeded to solve it in about one hour. Thank you both for your help!

My thanks go to Alain Vachoux and Paul Debevec for their help in setting up accounts and making the various VLSI tools work, as obscure an art as any!

Finally, general thanks to everyone in the lab at LTS3, past and present, for your help and companionship during these long years, notably Aline Gruaz for her assistance in all those little annoying administrative tasks that distract us from the real (to me) work.
# Contents

Abstract iii
Version Abrégée v
Acknowledgments vii
Contents ix
List of Figures xiii
List of Tables xv

1 Introduction 1

2 Processor State of the Art 5
    2.1 Why use a Microprocessor? 5
    2.2 Importance of Dynamic Scheduling 7
        2.2.1 Static Scheduling 7
        2.2.2 Out-of-Order Execution Problems 8
        2.2.3 Out-of-Order and Superscalar 9
        2.2.4 Other Technologies 11
    2.3 Limitations 12
        2.3.1 The Memory Wall 12
        2.3.2 Limits of Instruction Level Parallelism 13
        2.3.3 Power Consumption 14
    2.4 Conclusions 14

3 Reconfigurability 15
    3.1 Technological Possibilities 15
        3.1.1 PLDs and Complex PLDs 16
        3.1.2 Field Programmable Gate Arrays 16
        3.1.3 Configurability-Speed Dilemma 17
    3.2 Dynamic Reconfiguration 17
    3.3 Architectural Possibilities 17
## CONTENTS

5.4 Detailed Design ................................................. 65
5.4.1 Internal Routing ........................................... 65
5.4.2 Threshold Decision Algorithm ......................... 67
5.4.3 Multiplier Tree Design .................................. 67
5.4.4 Timing Parameters for Simulation ..................... 77
5.5 Conclusions ...................................................... 77

6 Results ................................................................ 79
6.1 Methodology ....................................................... 79
6.1.1 SimpleScalar .................................................. 79
6.1.2 SPEC CPU2000 Benchmarks .............................. 82
6.1.3 Processor Models ............................................. 85
6.2 Integer Benchmarks ............................................. 88
6.2.1 ALU Benchmarks ............................................ 90
6.2.2 MUL Benchmarks ............................................ 90
6.3 Floating Point Benchmarks ................................. 91
6.3.1 Light FP Benchmarks ....................................... 91
6.3.2 Heavy FP Benchmarks ..................................... 91
6.4 Dynamic Analysis .............................................. 105
6.5 Conclusions ...................................................... 110
6.6 Sensitivity Analysis ............................................ 111
6.6.1 Methodology .................................................. 112
6.6.2 Parameters Considered ..................................... 112
6.6.3 Differences between sensitivity and Simpoints results 113
6.6.4 Results ......................................................... 113
6.6.5 Conclusions ................................................... 118
6.7 Problems and Limitations .................................... 133
6.7.1 Issue and Commit Widths ................................. 133
6.7.2 Complexity of Good Decision Algorithm ............ 133
6.7.3 FP-Intensive Code .......................................... 135
6.8 Conclusions ...................................................... 135

7 Conclusion .............................................................. 137
7.1 Conclusions ....................................................... 137
7.2 Contributions ..................................................... 138
7.3 Perspectives ....................................................... 139

Bibliography .......................................................... 141

A VHDL Schematics and Reports ............................... 153
A.1 Multiplier Tree ................................................... 153
A.2 Decision Algorithm ............................................. 168
B Complete Decision Algorithm Example  
  B.1 Example Description .......................... 173
  B.2 Dynamic Solution ............................. 174
  B.3 Optimal Integer Linear Programming Solution  
    B.3.1 Integer Linear Program .................... 174
    B.3.2 Optimal Result ............................ 174

C Simplescalar Configurations  

D List of Acronyms  

Curriculum Vitae
List of Figures

2.1 CPU vs Memory speed gap ................................. 12
4.1 Checksum Operations ................................. 26
4.2 Checksum Hardware ................................. 27
4.3 L2TP DSLAM Application ................................. 27
4.4 Checksum custom instructions results .................... 31
4.5 Speed-Reconfigurability Graph ................................. 32
5.1 Average FPU usage in SPEC ................................. 36
5.2 FPU Reconfiguration ................................. 39
5.3 Superscalar processor pipeline ...................... 41
5.4 Carry Save Adder (CSA) ................................. 43
5.5 Iterative multiplier .................................. 44
5.6 Structure of a fast multiplier ................................. 46
5.7 4-to-2 Compressor .................................. 47
5.8 High-level FPU structure ................................. 47
5.9 Reconfiguration problem description .................... 50
5.10 Reconfiguration Decision Problem State Model .................. 53
5.11 Implementation of the threshold algorithm .................. 58
5.12 Instruction scheduling for instruction arrivals (baseline) .................. 59
5.13 Instruction scheduling for instruction arrivals (threshold) ................. 60
5.14 Instruction scheduling for instruction arrivals (optimal) .................. 60
5.15 Instruction scheduling for instruction dependencies (baseline) ................. 61
5.16 Instruction scheduling for instruction dependencies (threshold) ................. 61
5.17 Instruction scheduling for instruction dependencies (optimal) ................. 62
5.18 Speedups by balance vs. threshold .................... 64
5.19 Unified and distributed Reservation Stations .................. 67
5.20 Balanced multiplier tree with 4 CPAs .................... 68
5.21 4-to-2 Compressor built from two CPAs .................. 69
5.22 Unbalanced tree with 3 CPAs .................... 70
5.23 Full 64-input compressor trees (1) .................... 73
5.24 Full 64-input compressor trees (2) .................... 74
5.25 Full 64-input compressor trees (3) .................... 75
LIST OF FIGURES

5.26 Full 64-input compressor trees (4) ........................................ 76

6.1 Speedups of baseline mainstream vs original mainstream .......... 89
6.2 Results for the mainstream models ........................................ 93
6.3 Speedups for the mainstream models .................................... 94
6.4 Results for the optimal mainstream models ............................ 95
6.5 Speedups for the optimal mainstream models ......................... 96
6.6 Average number of xALUs active/cycle (mainstream) ............... 97
6.7 Results for the compact dynamic mainstream models ............... 98
6.8 Speedups for the compact dynamic mainstream models ............. 99
6.9 Results for the top models .................................................. 100
6.10 Speedups for the top models .................................................. 101
6.11 Results for the optimal top models ..................................... 102
6.12 Speedups for the optimal top models ................................... 103
6.13 Average number of xALUs active/cycle (top) .......................... 104
6.14 Instruction types for galgel .................................................. 105
6.15 Instruction types and state for mcf .................................. 106
6.16 Instruction types and state for sixtrack .............................. 107
6.17 Structural stalls for vortex (mainstream) ............................... 108
6.18 Structural stalls for vortex (top) ........................................ 108
6.19 Structural stalls for lucas (mainstream) ............................... 109
6.20 Structural stalls for sixtrack (mainstream) ............................ 109
6.21 Comparison of Simpoints and fast ..................................... 114
6.22 Reconfiguration frequency .................................................. 116
6.23 Results for variations of the FPU Multiplier latency ............... 120
6.24 Speedups for variations of the FPU Multiplier latency ............. 121
6.25 Results for variations of the xALU latency ............................ 122
6.26 Speedups for variations of the xALU latency .......................... 123
6.27 Results for variations of the reconfiguration latency .............. 124
6.28 Speedups for variations of the reconfiguration latency .......... 125
6.29 Results for variations of the reconfiguration factor ............... 126
6.30 Results for variations of the pipeline width .......................... 127
6.31 Results for variations of the number of ALUs ........................ 128
6.32 Results for variations of the number of FPUs ........................ 129
6.33 Results for variations of the memory latency .......................... 130
6.34 Speedups for variations of the memory latency ........................ 131
6.35 Results for variations of the number of LSUs ........................ 132
6.36 Percentage of load and store instructions ........................... 134

A.1 Schematic of a 64-bit multiplier ........................................ 155
A.2 Schematic of an unbalanced 64-bit multiplier ........................ 156
A.3 Schematic of the threshold implementation .......................... 169

B.1 Timing chart of the Decision Algorithm Example ................... 174
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Full software Checksum</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>HW/SW Checksum</td>
<td>29</td>
</tr>
<tr>
<td>6.1</td>
<td>Instruction type distributions in SPEC CPU 2000</td>
<td>86</td>
</tr>
<tr>
<td>6.2</td>
<td>Processor model resources</td>
<td>88</td>
</tr>
<tr>
<td>6.3</td>
<td>Summary of results for the <em>mainstream</em> models</td>
<td>110</td>
</tr>
<tr>
<td>6.4</td>
<td>Summary of results for the <em>top</em> models</td>
<td>110</td>
</tr>
<tr>
<td>6.5</td>
<td>Summary of Sensitivity Analysis Results</td>
<td>119</td>
</tr>
<tr>
<td>A.1</td>
<td>Timing report for a <em>balanced</em> tree (1)</td>
<td>157</td>
</tr>
<tr>
<td>A.2</td>
<td>Timing report for a <em>balanced</em> tree (2)</td>
<td>158</td>
</tr>
<tr>
<td>A.3</td>
<td>Timing report for a <em>fully unbalanced</em> tree (1)</td>
<td>159</td>
</tr>
<tr>
<td>A.4</td>
<td>Timing report for a <em>fully unbalanced</em> tree (2)</td>
<td>160</td>
</tr>
<tr>
<td>A.5</td>
<td>Timing report for an <em>optimally unbalanced</em> tree (1)</td>
<td>161</td>
</tr>
<tr>
<td>A.6</td>
<td>Timing report for an <em>optimally unbalanced</em> tree (2)</td>
<td>162</td>
</tr>
<tr>
<td>A.7</td>
<td>Area report for a <em>balanced</em> tree</td>
<td>163</td>
</tr>
<tr>
<td>A.8</td>
<td>Area report for a <em>fully unbalanced</em> tree</td>
<td>163</td>
</tr>
<tr>
<td>A.9</td>
<td>Area report for an <em>optimally unbalanced</em> tree</td>
<td>164</td>
</tr>
<tr>
<td>A.10</td>
<td>Power report for a <em>balanced</em> tree</td>
<td>165</td>
</tr>
<tr>
<td>A.11</td>
<td>Power report for a <em>fully unbalanced</em> tree</td>
<td>166</td>
</tr>
<tr>
<td>A.12</td>
<td>Power report for an <em>optimally unbalanced</em> tree</td>
<td>167</td>
</tr>
<tr>
<td>A.13</td>
<td>Timing report for the <em>threshold</em> implementation</td>
<td>168</td>
</tr>
<tr>
<td>A.14</td>
<td>Area report for the <em>threshold</em> implementation</td>
<td>170</td>
</tr>
<tr>
<td>A.15</td>
<td>Power report for the <em>threshold</em> implementation</td>
<td>171</td>
</tr>
<tr>
<td>B.1</td>
<td>Dynamic trace</td>
<td>175</td>
</tr>
<tr>
<td>B.2</td>
<td>Integer Linear Program (1)</td>
<td>176</td>
</tr>
<tr>
<td>B.3</td>
<td>Integer Linear Program (2)</td>
<td>177</td>
</tr>
<tr>
<td>B.4</td>
<td>Integer Linear Program (3)</td>
<td>178</td>
</tr>
<tr>
<td>B.5</td>
<td>Integer Linear Program (4)</td>
<td>179</td>
</tr>
<tr>
<td>B.6</td>
<td>Integer Linear Program (5)</td>
<td>179</td>
</tr>
<tr>
<td>B.7</td>
<td>Integer Linear Program (6)</td>
<td>180</td>
</tr>
<tr>
<td>B.8</td>
<td>Integer Linear Program (7)</td>
<td>181</td>
</tr>
<tr>
<td>B.9</td>
<td>Integer Linear Program (8)</td>
<td>181</td>
</tr>
</tbody>
</table>
B.10 Integer Linear Program (9) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 182
B.11 Integer Linear Program Search Result . . . . . . . . . . . . . . . . . . . . . . . 182
B.12 Integer Linear Program Solution. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
C.1 Single standard simulation points . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 186
C.2 Baseline mainstream configuration file for Simplescalar. (1) . 187
C.3 Baseline mainstream configuration file for Simplescalar. (2) . 188
C.4 Baseline mainstream configuration file for Simplescalar. (3) . 189
C.5 Baseline mainstream configuration file for Simplescalar. (4) . 190
C.6 Differences in the dynamic mainstream configuration . . . . . . . . 190
C.7 Differences in the optimal dynamic mainstream configuration 191
Chapter 1

Introduction

By permitting the integration of vast and complex logic functions in a tiny space, *Integrated Circuits (IC)* are a very important, yet hidden, component of modern life. These chips lie at the heart of many if not most of all the modern electrical items we take for granted today.

Initially, processors and *Application Specific Integrated Circuits (ASIC)*, hardwired logic dedicated to an application, had clearly defined and separate roles: processors handled complex tasks that would be difficult to put in hardware and tasks that were not executed often, and ASICs would be designed to handle everything else. However, several different changes increased the pressure and benefits of using processors over ASICs, leading to a situation where fewer and fewer ASICs are being designed, unless their function is to assist a processor in a given task.

The main advantages of an ASIC are high performance and, usually, a lower cost not only for manufacturing, but also power consumption. These lead to very powerful and efficient systems. On the other hand, the design costs, especially in newer, smaller technologies, are very high, thus requiring large volumes to be cost-effective. The increasing number of bugs—i.e., errors of design or programming—due to the increased complexity and reduced lifetimes of products and the great costs of any modification to a chip [4] conspire to lower the benefits of this approach.

Processors have always been designed with generality in mind. The first ever commercial general purpose microprocessor, built by Intel [65, 23], was actually intended to be placed in a calculator, where ASICs usually dominate. Its success led Intel to focus on microprocessors, eventually leading to the x86 family of processors spawned by the decision taken by IBM to place it at the heart of the original *Personal Computer* (PC). Thirty years of advances in silicon technology and processor design have enabled what is now called the *Information Age*, with millions of PCs being used daily around the
world for all manner of tasks, such as office work, scientific calculations or video games. General purpose processors are also increasingly being used in the world of embedded systems, where ever higher performance and a thirst for diverse and rapidly changing applications make their use very attractive.

All the advantages of a processor revolve around its capability to execute any stream of instructions it receives. As such, applications or functionality, implemented as software, are more quickly created than the equivalent hardware blocks would be. Through the use of compilers capable of translating and optimizing a high-level representation of an algorithm understandable by humans into code understandable by a specific processor, a great ease of programming is achieved. As these compilers can have different target processors, the application can usually be compiled for any general-purpose processor with little extra work. All these advantages lead to a wide variety of applications being available, which further reinforces the trend favoring processors, due to code re-use.

In many applications, such as personal computers or high-performance computing, the main focus of processor design is to increase the performance of the processor when running a wide set of applications, with little regard to power. On the other hand, embedded processors such as those found in portable devices must balance sufficient performance to run the desired applications with keeping power consumption to a minimum. However, as a processor will generally provide less performance than an ASIC in a given technology, it may not be able to fulfill these requirements. While this is only an inconvenience in a PC running a word processor, it becomes far more critical in a power plant’s control system.

Two broad options are available to increase the performance of a processor, in addition to improvements in the underlying silicon technology: these are adding custom logic—essentially the joining of a pure processor and an ASIC—or through the use of some form of reconfigurable logic. In both cases, the added hardware can be more or less tightly coupled to the processor core, depending on the need for interaction between the two.

Custom logic to assist a processor can take many forms, depending on the amount and frequency of interaction between the processor’s program and the functionality implemented in the custom logic. At one extreme, a coprocessor in another chip, or even on another board, can execute functions requiring relatively little interaction with the processor while the latter executes other functionality, usually related to the control plane. As the distance is reduced, this coprocessor can be placed on the same chip, forming the basis of a System-on-Chip (SoC). In both these cases, the functionalities of the custom logic and the processor are defined separately, and great care must be taken with their interactions. The closest binding possible is obtained by adding a custom functional unit to the processor, which will then simply be accessible as a set of new instructions available to the pro-
grammer. The custom logic is always dedicated to an application or an application domain defined beforehand, and must be designed in a way similar to an ASIC, although some tools simplify the insertion into the processor [112, 118, 150, 152].

Fully reconfigurable logic is a way to remove the requirement of foreknowledge about the application whose performance must be improved. It allows the downloading of new functions during run-time, simply by writing configuration data into a memory controlling the configuration of the reconfigurable logic. However, as the process of mapping an algorithm to reconfigurable logic is fairly complex, these mappings must be done in advance and then downloaded from a configuration memory. Reconfigurable logic is also rather slow compared to custom logic, with about an order of magnitude difference between the two. This contributes to the limitation in the applicability of reconfigurable logic, as the gains obtained by having hardware adapted to an algorithm must also compensate this slower speed for an overall gain to emerge.

In order to alleviate these issues, it is possible to limit the reconfigurability allowed. This will greatly reduce the loss in speed incurred by reconfigurable logic, thus allowing smaller increases in the performance of an algorithm to produce overall gains. This in turn will increase the number of applications that can benefit from reconfigurability.

A detailed case study will focus on adding limited reconfigurability to a superscalar processor’s Floating Point Unit (FPU) to re-use part of this large functional unit for other instructions. In this context, attention will focus on the modifications and delays in the out-of-order execution core and the FPU, but also on the algorithm to make decisions about reconfiguration. To show the broad range of improved applications, standard industry benchmarks for processor performance will be used to compare results.

Chapter 2 will detail the interest of using microprocessors, continuing with the current state of the art and its limitations. Chapter 3 will explain the methods to achieve reconfigurable circuits and some of the problems involved. Limiting reconfigurability will be explored in chapter 4, including the role of custom instructions. Chapter 5 contains the detailed analysis of the applications of limited reconfigurability to superscalar processors, covering both theoretical aspects and parts of the design implementation. The results of this study will be presented in chapter 6, including sensitivity analysis. Finally, chapter 7 will conclude this dissertation and offer an outlook on future research.
Chapter 2

Processor State of the Art

The importance of the general-purpose processors at the heart of all computers in everyday life has greatly increased in the last decade and a half: at first reserved for complex scientific and finance applications in the form of large mainframes, processors are now nearly ubiquitous in a modern society. They are at the heart of the personal computer on every office worker’s desk, but also powering mobile phones, video games, most household appliances, and even cars, trains and airplanes. But since processors have existed for more than 20 years, why weren’t they used before?

2.1 Why use a Microprocessor?

Most of the advantages of processors are independent of the particular technology used and the environment and alternatives available, as they are closely linked to the very concept of a processor:

**Versatility** A processor’s hardware contains all the basic blocks needed to build any logic of mathematical function imaginable. Thus, the processor need be designed only once, and then any change in functionality is obtained merely by changing the software used to control the processor. This capability, always desirable, is even more important as the pace of technological advances grows.

**Ease of programming and tools** allow us to take advantage of the versatility of a processor: the only language a processor understands is a long sequence of bits called *machine language*. However, as the number of instructions understood are relatively small and fairly basic, abstraction levels can be built to allow the writing of algorithms in languages closer to the human manner of thinking. The many tools to assist in the programming of a processor and the advances in automation and expressivity of programming languages make designing
applications for a processor ever easier and allow ever greater complexity to be handled. The cost of writing and debugging software is also far lower than the cost of designing an equivalent piece of hardware.

**Porting** The ability to port, or target an application to a different processor has several aspects: first, if the *Instruction Set Architecture (ISA)*, which is the vocabulary the processor understands, is kept identical, several different processor models may execute the same software without any changes. This *Binary compatibility* preserves the ever greater investments in software. Some computers even tried having multiple ISAs and changing the current ISA based on the program that was about to be executed [103]. The second aspect is the possibility of recompiling a program for a completely different processor. The use of high-level languages and a certain uniformity in ISAs greatly help in this regard. However, the tools to compile these programs used to be very expensive and system or vendor specific.

**Free software** has become an increasingly important enabler for processors: through the dedicated actions of a few individuals such as Linux Torvalds [137] and Richard Stallman [149], the concept of free software was born. In the world of mainframes, all software was written for a specific machine, and sold with the machine, usually for a high price. This is still the general model in the PC world using Microsoft software, which made its fortune selling its well-known operating system, called *Windows* [138]. Free software advocates contend that software should be free, both in the sense that it can be modified and that it can be obtained without payment, and have garnered enough support to make this a reality [120, 121]. Thus, it is now possible to find both an operating system and a compiler tool chain for almost any existing processor, free of charge, and with the source code available. This means that anyone can write an application for any processor at no cost, which has lead to an explosion in the number of applications, thus increasing the demand for processors and the domains where they can be used, as a consequence of the lower barriers to entry.

**Extendable** Finally, through their integration into computer systems, processors are easily extendable: it is a relatively simple affair to design a board for a specific need and connect it to a processor. Thus, while the special board does all the very specific data work, all the control and algorithms are kept in the processor, where they can be easily modified.

Some advantages of processors owe their existence to the rapid advance of silicon technology in recent years, notably following the famous *Moore’s Law*, named after Gordon Moore [139], former CEO of Intel Corporation,
the largest processor maker in the world. Thanks to these advances, many applications that could only be performed with dedicated hardware, such as ASICs, may now be easily performed by a processor, with a great reduction in cost and the ability to fix problems far more quickly and simply. This is compounded by the exponential increase in the mask costs for an integrated circuit, making dedicated ICs far less interesting than buying a standard processor and writing software for the desired function. The increased complexity of ASICs also means a faster time to market for products based on processors, always a relevant point but which is gaining importance in our rapidly-changing world. Finally, the ever increasing number of processors has led to very high volumes which, coupled with competition, has made most processors a fairly low-cost part, replacing very expensive custom supercomputers with a large number of cheap commercial processors and some high-speed interconnect.

2.2 Importance of Dynamic Scheduling

Although lately much growth and attention has been focused on exploiting coarse-grained parallelism in embedded processors, where there commonly are many tasks to be performed at the same time, single-threaded performance is still very important in many domains, such as scientific and engineering computing, and video games. The most common options for increasing processor performance in single-threaded applications will be presented in the following sections.

2.2.1 Static Scheduling

Static scheduling was the first method used to schedule instructions in a processor. In its most simple form, the processor takes the instructions as they arrive and executes them, writing the results back to the register file immediately. To avoid stalls due to long memory latency or, in the case of pipelined execution, dependent operations, a number of techniques, such as data forwarding or delay slots, may be used.

The main advantage of static scheduling is its simplicity, both in terms of the number of logic gates needed to implement it and in the terms of debugging the design. This low transistor count also translates into a rather low-power design. As such, almost all embedded processors are single-issue statically scheduled processors, although some companies are beginning to design superscalar processors for the high-end of this market [12, 114].

However, no matter what techniques are used to reduce delays and stalls in the pipeline, the processor can never execute more than one instruction per cycle, thus limiting processor performance to the evolution of technology. This is even more visible in the case of Reduced Instruction Set Computer
(RISC) architectures, where each instruction performs only very simple operations. While the parallelism available in most embedded applications allows several such processors to be used in parallel, the limitation on the performance of a single processor makes it unsuitable for applications using a single thread and containing larger amounts of parallelism requiring fast execution.

From this observation, the question of whether it is possible to dynamically reorder instructions arises, and with it, the possibility of processing more than one instruction per cycle.

### 2.2.2 Out-of-Order Execution Problems

When executing a program in order, the only special case that must be handled is a stall when the data needed for an instruction is not available, because of dependencies between instructions that take more than one cycle to execute or memory latencies. However, if we attempt to reorder the instructions, great care must be taken to avoid changing the behavior of the program: to execute a load instruction before a store, the processor must make sure that they do not access the same location in memory, for example [102].

Another problem that arises when considering out-of-order execution is how to handle branch instructions. These instructions are tests, and, as their name implies, they cause branches in the control flow graph. If the processor wants to reorder beyond a branch, it must ensure that the reordering will not alter the behavior of the program regardless of the outcome of the branch instruction.

Finally, the last major problem to consider is the problem of exceptions or interrupts. Despite their name, exceptions happen quite frequently in a modern processor, as they are often used to avoid polling in the case of access to devices slower than the CPU—i.e., almost everything except the cache. They are also used to indicate that something has gone wrong and must be corrected. This latter case is problematic: if an instruction that the processor executed out-of-order causes an exception, it must be certain that this exception would also have occurred had the instructions been executed in program order. The processor should then give control to the exception handler so it can perform the necessary corrections to allow the program to continue execution; however, the exception handler expects the processor to have executed the instructions in program order, and may thus fail in unforeseen ways.

One method to avoid some of the problems above is speculation. It allows the processor to execute instructions without being sure that they should be executed, and the processor waits until the moment the instruction would be executed in the normal program order before writing the results of the instruction to the memory or register file. It will be discussed in greater
2.2. IMPORTANCE OF DYNAMIC SCHEDULING

To extract the greatest amount of parallelism from a program requires a mix of concepts: out-of-order execution and multiple issue provide the capacity to execute several instructions per cycle. Speculation makes it work in practice. All these concepts aim to execute the greatest amount of instructions every cycle, thus decreasing the total program execution time. As with all methods that try to reach a limit (the amount of parallelism available in the program), this is a clear case of diminishing returns, where the gains of subsequent refinements must always be weighed against the added cost [24].

A superscalar processor is a processor that can execute more than one instruction per cycle. This is achieved by having a higher number of functional units and a large pipeline that can handle several instructions in parallel. To increase the possibilities for executing instructions in parallel, we allow the processor to reorder instructions, giving us an out-of-order superscalar processor. However, due to the large number of branches in most programs [28], it is difficult to fill all the execution slots—i.e., use all the parallelism the processor provides. Speculation reduces this problem by allowing the execution of instructions before the processor is sure that the instruction should be executed. This allows the processor’s pipelines to be filled with more instructions. However, there is no guarantee that these instructions will actually be useful: in the case where all the speculation turned out to be incorrect, the net result would be the same as that without speculation, but with higher power consumption.

The main component of speculative schemes is called branch prediction. This means that for each branch instruction encountered, we predict, following some more or less complex method, what the outcome will be, and continue fetching and executing instructions as if the prediction was true, but without writing the results to the main registers. Then, when the branch instruction’s condition is resolved, the results of the subsequent instructions are either written to the registers and memory, or discarded, depending on the real outcome of the branch instruction. This highlights the importance of a good branch prediction algorithm. Many such algorithms have been proposed to improve the percentage of correct predictions, e.g. taking the history of branches into account [5], or even by storing a stretch of instructions encompassing several branches [35, 76]. The best values attained today are in the order of 90% of correctly predicted branches [34].

Another approach in speculative schemes is to predict the values that will result from an operation, usually a load (e.g. [2]). This allows execution to continue without waiting for the result of the instruction, at the cost of more changes to roll back in the case of a misprediction.

Dependencies may take many forms, but some of them, called name
dependencies, are caused by a lack of architectural registers: most ISAs specify a rather small number of registers, and compilers thus usually try to fit the code in the smallest number of registers possible, thus causing several instructions to refer to the same register at different times, but not to the same data! Similarly, some older ISAs, such as the x86 ISA understood by the processors running most PCs, have very few registers. In this case, register renaming allows the processor to use extra physical registers, which are not part of the ISA, to store results of independent instructions using the same architectural—i.e., visible to the software—register.

Finally, exceptions are one of the most complex problems. The concept of exceptions, described in section 2.2.2, is linked to in-order execution. They are a way to simplify error detection and recurrent tasks for the software, but require support from the processor hardware. In the case of a complex, out-of-order superscalar processor, exceptions are split into several groups [28]:

**Interrupts** are caused by devices operating asynchronously with the processor. As the interrupt service code is independent from the code the processor was executing just before the interrupt, out-of-order execution has no impact other than to complicate the storage and retrieval of the processor state to service the interrupt.

**Precise Exceptions** need to be handled exactly as they would have been handled in an in-order processor. This usually requires squashing many instructions—i.e., discarding their results and any changes they may have made—around the one causing the exception that were executed out-of-order and executing them in order one by one [80]. Then, if the exception does occur, the exception handler will find the processor in the state it expected. When the handler is done, execution can resume normally. This method can be costly in terms of performance in applications with a large number of exceptions.

**Imprecise Exceptions** are a way of limiting the impact of exceptions on performance. In this case, instead of restoring the processor to the state it would have been in had the instructions been executed in order, we just call the exception handler when the exception occurs, without changing the state of the processor. This results in a gain in performance, but requires more care in the writing of the exception handler code.

In addition to the limitations faced by all processors, the performance of superscalar, out-of-order processors is limited by a number of factors, the most important of which are the number of instructions it can consider for execution, called the execution window; the complexity of the scheduler and the pipeline limiting the clock rate; and the increased pressure on the
memory hierarchy to provide both several instructions and several pieces of data every cycle [33].

An interesting development in the domain of superscalar processors is the ability to execute instructions from more than one thread at a time, generally called Simultaneous MultiThreading (SMT) [30, 92]. The gains in well chosen applications can be significant, and it opens possibilities for further optimizations [56].

2.2.4 Other Technologies

There are some alternatives to superscalar processors in the race toward single-threaded high performance. These are mainly Very Long Instruction Word (VLIW) processors; using many small processors and a mesh to connect them; or automatically adding custom instructions to a processor to make a processor dedicated to an application or application domain.

VLIW processors were born from the observation that, in a large superscalar processor, there is almost as much hardware trying to figure out what to execute and when as there is hardware actually executing instructions. Thus, the VLIW philosophy\(^1\) is to push all the scheduling, dependency analysis, . . . into software, to be performed by the compiler. The compiler then writes ‘super-instructions’ whose width is equal to the width of the VLIW pipeline, adding \texttt{NOP}\(^2\) instructions in the slots it cannot fill—e.g. [32]. Pushing the complexity to the compiler increases the time and resources available to optimize the code, but it also makes the processor’s performance very dependent on the quality of the compiler. Likewise, VLIW code is not automatically backward compatible, as changing latencies and other scheduler constraints mean that software must generally be recompiled for a precise processor model.

One way of circumventing some of the limitations of VLIW processors is to use dynamic binary translation, in effect fooling the software into believing it is running on one architecture when the real hardware is quite different. This allows freedom of actual processor design, at the cost of a complex runtime translation [3, 41], although some work on the translation overhead is possible [82].

The second direction, derived from highly parallel supercomputers, is to define a small in-order processor with a little local memory, and replicate this a large number of times, linking each processor through a fast interconnect network [87]. The operating system or compiler must then partition the application into as many parallel pieces as possible, trying to make use of the enormous amount of parallel processing resources available. Performance can be very good for specific applications with large parallelism [88], but in the worst case, only a single processor can be used. In addition, the

---

\(^1\)It is also called Explicitly Parallel Instruction Computing (EPIC) [78].

\(^2\)No OPeration.
CHAPTER 2. PROCESSOR STATE OF THE ART

Figure 2.1: Graph of the difference in speed between CPU and memory over time. Note the logarithmic scale of the y-axis. Memory shows a 7% decrease in latency per year, whereas CPUs show an increase of 35% per year until 1986, and 55% thereafter (from [28]).

partitioning of an application onto a number of processors and the scheduling of the communications between them is not a trivial task [52, 58].

Finally, a direction most often taken in the embedded world, especially with the apparition of automatic generation tools, is to customize a simple in-order processor with special instructions to accelerate the performance over a specific task or domain, often by quite a large margin. An example of such custom instructions will be described in section 4.1.

2.3 Limitations

The limitations of processors are mostly due to the way a processor splits applications into a long sequence of very simple operations, unlike dedicated hardware such as an ASIC. As it cannot adapt to the application, a processor basically tries hard to execute the program as fast as possible, but in some cases cannot do so with good performance. This low performance and a high cost were the main reasons for the limited initial use of processors, as they were used only for applications with no viable alternative. The limitations of the processor approach are mainly due to memory latencies, limits in the parallelism available in a program, and power consumption.

2.3.1 The Memory Wall

What has been called the memory wall [64, 108] is the rapid increase in the ratio of speeds of processors compared to memory, as shown in figure 2.1. This effect is mitigated by the use of caches, small, fast memories that serve
as buffers for the processor, but even increasing the number of levels of cache in recent processors does not avoid this problem due to growing workload sizes. For the fastest processors today, an access to main memory can take thousands of cycles, making useless small increases in the parallelism that can be processed.

In addition to caches and the many algorithms used to allocate them, research into new types of memory, such as those based on magnetism [9, 60] or quantum mechanics [6], provide interesting directions and prototypes that might one day alleviate this issue. On the algorithmic side, methods aiming to load data far in advance, called *prefetching* and using distributed memory and processing, as discussed briefly in section 2.2.4, also aim to reduce the distance between the storage and processing parts of a computation [57].

### 2.3.2 Limits of Instruction Level Parallelism

Parallelism in computer architecture can take several forms, depending on what is being executed in parallel: instructions, threads, transactions, or even programs. The smaller the parts, the more difficult it is to execute them in parallel. As we focus on single-threaded performance, only *Instruction Level Parallelism* (ILP) is considered here. The main reason why most code is still written as a single-thread is that it is natural to think about a process as a sequence of steps to be executed in order. While some programming or design languages allow explicit parallelism in expressions, this comes with an greatly increased complexity in debugging and compilation (or hardware synthesis). However, in hardware, everything is naturally parallel, and it is the design of the hardware that forces the ordering of operations. Thus, hardware carefully designed for a specific application can attain levels of performance far higher than a processor. However, this is always achieved at the cost of design or programming complexity.

One aspect of processor performance that has been very thoroughly studied, and yet still has room for improvement, is the amount of parallelism offered by an application (e.g., [48, 98]). While it is possible, through simulation or mathematical analysis, to estimate the parallelism available in a given *implementation* of an application as a program, the inherent parallelism of the application is far more complex to measure [28, 53, 90]. It is thus possible to re-write the code for a Fast Fourier Transform (FFT) from a naive implementation to an optimized one to gain a factor of about 100 on a general purpose processor—e.g., [38, 43]. This has led to the development of special compilers, called *vector compilers*, that attempt to parallelize code written in a sequential programming language [17, 50, 152].

For most applications, due to any of the many parameters limiting the possibilities of the processor, the parallelism visible to a processor is a fraction of the total parallelism inherent even in the specific implementation considered. In some cases, the program itself has so little parallelism that
nothing short of removing the particular obstacle to parallelism can improve performance [21]. An example for this is the mcf benchmark used in chapter 6, which is completely blocked by memory latency.

### 2.3.3 Power Consumption

The last main drawback of processors is their power consumption. As the hardware is generic to handle any application, most of the power consumption of a processor is spent moving data from one place to another and controlling the state of the processor. Although power consumption can be reduced with circuit design techniques such as clock gating [26, 91] and reductions in the clock speed under light loads [136], it is difficult to reduce power consumption by a large margin, as the caches, critical to performance for most applications, require a lot of power [40, 95]. As the use of processors in embedded and mobile applications increases, this is becoming the greatest issue for processor designers for these markets. This is also an issue for large data processing centers, where, in the summer, more power is expended to keep the environment cool than to run the computers!

### 2.4 Conclusions

The major advantages of processors, and the current directions in processor design, have been described. The drawbacks of using processors have also been highlighted, and these drawbacks are becoming more important with increasingly smaller technology. Recently, the pace of increases in the single-threaded performance of processors has slowed due to these limitations.

In order to keep the many advantages of processors while minimizing the impact of their drawbacks, many options for adding some form of adaptability, or reconfigurability, have been studied. This will be addressed in detail in chapter 3.
Chapter 3

Reconfigurability

Reconfiguration, the ability to adapt usually fixed hardware to the functionality it should perform, is a very elegant approach to many problems. Indeed, instead of taking a fixed architecture and fitting the algorithm as well as possible, the hardware morphs into the best possible architecture to fit the current algorithm, and will change for the next algorithm.

This is made possible by two related technologies, Programmable Logic Devices (PLDs) and Field Programmable Gate Arrays (FPGAs), that allow changing the logic functions of a chip. These will be detailed below. The technology behind reconfigurable logic, notably FPGAs, has improved greatly in recent years, leading to an increased interest and a wide range of applications being implemented using this logic.

3.1 Technological Possibilities

There are two main possibilities available for reconfigurable logic. Programmable Logic Devices have existed for many years, while the more recent FPGAs have revolutionized the world of reconfigurable logic, and nowadays, reconfigurable almost always means some sort of FPGA.

The two major types of programmable logic devices today are field programmable gate arrays (FPGAs) and complex programmable logic devices (CPLDs). Of the two, FPGAs offer the highest amount of logic density, the most features, and the highest performance. CPLDs, by contrast, offer much smaller amounts of logic. But CPLDs offer very predictable timing characteristics and are therefore ideal for critical control applications [97].

The circuits are always based on silicon transistor technology, in the same way as hardwired logic circuits, so special tricks are needed to provide reconfigurability.
CHAPTER 3. RECONFIGURABILITY

3.1.1 PLDs and Complex PLDs

The first programmable logic devices were Programmable Logic Arrays, capable of performing sum-of-products logic expressions. The connections were formed by burning 'fuses' on the chip, which could only be configured once.

With the shift to Complementary Metal Oxide Semiconductor (CMOS) technology, re-programmable, or erasable PLDs appeared, using transistors to control the connections. These use a technology called floating-gate MOS [61]. These transistors have 2 gates, a normal gate and a floating gate. When the floating gate is not charged, it works like any other transistor. When a high voltage is applied to this floating gate, it acquires a charge, which will then prevent the transistor from turning 'on'. This charge remains for at least 10 years, so the programming can be considered permanent for most applications. It is possible to erase this programming through the application of a voltage of the opposite polarity.

Complex PLDs (CPLD) are simply a way of obtaining larger PLDs while maintaining the same logic speed. Indeed, it is not possible to scale the size of PLDs, as the number of inputs increases dramatically, and along with it, capacitive effects and leakage currents. A CPLD is thus a collection of regular PLDs surrounded by a programmable interconnect on a single chip. PLDs and CPLDs are fairly slow to program, as this is usually done through a small serial port called Joint Test Action Group (JTAG) port.

CPLDs require extremely low amounts of power and are very inexpensive, making them ideal for cost-sensitive, battery-operated, portable applications such as mobile phones and digital handheld assistants.

3.1.2 Field Programmable Gate Arrays

A field programmable Gate Array is like a CPLD turned inside out: the logic is broken into a large number of programmable logic blocks that are individually smaller than a PLD. They are distributed across the chip in a sea of programmable interconnections, while the array is surrounded by programmable I/O blocks. An FPGA chip contains many more programmable logic blocks than a CPLD contains PLDs, leading to richer capabilities.

Each programmable logic block is capable of performing any 4-input logic function. This is achieved by considering the truth table of such a function, with 16 entries. Using a 16 word by 1 bit memory, applying the inputs as the address to the memory produces the result of the function as data output. The logic blocks can thus also be used a small memories.

The programmable interconnect provides communication between cells, through an incomplete mesh that encourages the use of locality in programming. These devices are not usually programmed by hand; the configurations are generated from high-level code or netlists by automatic tools, using algorithms similar to the place and route of traditional ICs.
3.2. DYNAMIC RECONFIGURATION

As opposed to (C)PLDs, an FPGA’s programming is not retained when the power is cut, and thus a Read-Only Memory (ROM) is often used to initialize the FPGA configuration upon power up.

FPGAs are used in a wide variety of applications ranging from data processing and storage to instrumentation, telecommunications, and digital signal processing. These applications benefit from a fast time to market, as there is no need to wait for the foundry to produce the chips and send them for testing. Likewise, the correction of bugs is achieved through the simple downloading of a new configuration, as opposed to a full re-spin, or at least a metal patch, for a hardwired chip.

3.1.3 Configurability-Speed Dilemma

The extra logic to allow programmability, in the form of special transistors or small look-up tables, slows down reconfigurable logic considerably compared to hardwired logic. This leads to a configurability-speed dilemma, with either fast, hardwired logic on one hand and slow, fully reconfigurable logic on the other. In order to add choices to these two extremes, it is possible to apply the techniques described above at a larger grain of configurability, affecting words or even functions instead of individual bits.

3.2 Dynamic Reconfiguration

Dynamic reconfiguration is the capability of reconfiguring the reconfigurable hardware while it is being used. There can be different levels of dynamic reconfiguration, depending on where the decision to reconfigure is made, and how frequently it is made. Indeed, updating the configuration of devices in the field every several weeks has far different constraints than a system that must adapt quickly and autonomously to changes in its workload. This thesis will focus on the latter aspect of dynamic reconfiguration.

Automatic dynamic reconfiguration requires a lower latency than offline reconfiguration, as too large a delay would make obsolete the data upon which the decision was based. The decision itself may range from simply choosing which part of an already configured FPGA should be used to actually fetching and downloading configurations from storage based on the current needs. This reconfiguration can produce better results that more infrequent reconfigurations, at the cost of an increased complexity.

3.3 Architectural Possibilities

Irrespective of the exact type used, there are several ways to use reconfigurable logic in an application, each bringing the reconfigurable logic closer to the processor. The first approach is simply to use only reconfigurable
logic for the entire application. The entire logic is thus slow, but suited for highly parallel applications. The second approach is to add reconfigurable logic as a coprocessor to a standard processor, leading to few changes but slow data transfer between the two. With increasing chip densities, the two can be combined on a single chip for greater efficiency. Finally, the reconfigurable logic can be added as reconfigurable functional unit in the processor. As almost all the uses of reconfigurable logic are FPGAs, these will be used for the examples in most of this section.

3.3.1 Stand Alone

FPGAs are complex chips with somewhat different manufacturing requirements than normal ICs. As such, it is often interesting to design an entire application on an FPGA, notably for prototyping, and reconfigure the logic as needed, usually on a relatively infrequent basis. The appearance of FPGAs with embedded hardwired logic, such as large multipliers and even simple processors [154], has made this option attractive as data can be processed in the FPGA while the control resides in the embedded processor. In any case, the FPGA must be designed as an IC, following a traditional design flow, which requires specific tools and expertise [155].

3.3.2 Coprocessors

The use of an FPGA as a coprocessor is only slightly different from the stand alone approach described above, the main difference being that the FPGA is no longer in control of the input/output of the chip. The FPGA must still be designed with hardware design tools, including a method to interact with the processor.

Off-Chip

Until relatively recently, it was impossible to combine enough reconfigurable logic and hardwired logic in a single chip to make the result worthwhile. This led to reconfigurable systems with several chips, at least one for the FPGA and one for the processor. The interaction between the two was usually managed in ways similar to that of multiprocessor systems [25, 14]. This implies that exchanges of data between the processor and FPGA must occur as seldom as possible, and be mostly insensitive to latency, making this approach interesting only for applications with large amounts of parallelism and little control.

On-Chip

With the advances of technology, it is now possible to integrate a large FPGA with a few tens of millions of hardwired transistors on a single chip, resulting
in reconfigurable Systems-On-Chip. The main advantage of such an architecture, in addition to the reduced cost for the packaging and board, is the faster communication between the processor and FPGA. Thus, applications requiring more control and with somewhat less parallelism may benefit from reconfiguration [105]. Dynamic reconfiguration may also become possible in some cases.

3.3.3 Functional Units

Finally, the closest integration possible is to place the FPGA inside the processor, acting as one or more functional units of the processor, with varying degrees of reconfigurability [10, 27, 75]. It can thus read and write the general purpose registers of the processor, and interact with the other functional units. This architecture provides the closest interaction between the processor and reconfigurable logic, essentially giving the processor almost single-cycle access to the reconfigurable logic. To increase the performance of the FPGA, it may be interesting to give it its own dedicated wide access to memory, usually bypassing the processor’s cache hierarchy. This results in a sort of hybrid load/store and execution unit with some large gains in suitable applications. Depending on the architecture, the scheduling of instructions and forwarding around the FPGA may be complex or require stalls, and the best gains from this architecture are generally modest compared to the coprocessor solutions above, although more applications may benefit from the reconfigurability.

A case where such a fully reconfigurable functional unit shines, however, is for bit-manipulation operations, a domain where general purpose processors are rather weak. Although most newer processors have extra vector instructions for the most common of these operations [89, 93], FPGAs are far superior when varied and unpredictable operations are needed. Due to the relatively small size of the reconfigurable unit, dynamic reconfiguration is also possible in this case, and a larger FPGA can be used to mask the reconfiguration latency by having several configurations loaded at the same time and switching between them when needed.

The major weakness of this approach is the necessity of a supporting compiler toolchain, as it is no longer possible to use standard multiprocessing synchronization mechanisms to coordinate the processor and FPGA. Ideally, this toolchain should also handle the generation of the FPGA configurations and download them at run time when needed. Some research has been done on this topic—e.g., [45, 83, 109]. In some cases, the new reconfigurable architecture requires a rethinking of the compilation process [69, 94].
3.4 Weaknesses

This section will present the issues that arise from the use of reconfigurable logic. Technology is the greatest factor, affecting both the speed and cost of FPGAs, and imposing some conditions on the downloading of configurations. Access to data, whether from another chip or block or from external memory is also often a bottleneck. Most of these issues are linked to the (re)configurability itself, and thus decrease as the grain of reconfigurability increases.

3.4.1 Slow Speed

In any given technology, fully reconfigurable logic such as an FPGA is about 5 to 10 times slower than hardwired logic [97]. This performance gap, which is caused by the fundamental nature of reconfigurable logic, is not likely to lessen with improvements in technology. One of the operations where FPGAs are notoriously slow are multipliers [74], which require either many slow cycles or a large amount of look-up tables. The slow clock speed also means that large gains must be made clock for clock to obtain a positive speedup once the differing clock speeds are taken into account. This has traditionally kept FPGAs in the domain of signal processing, where huge amounts of parallelism give them a significant advantage over processors, as the former can function as a Digital Signal Processor (DSP), treating many small samples at a time. In this domain, the parallel architecture of the FPGA can deliver far more than a 10x increase in performance, resulting in very interesting overall gains. For general purpose processing, however, such a speed hit is crippling, as will be shown in section 6.6.

The difference in speed also raises the issues of synchronization and interaction with the rest of the system, which might cause delays through buffering, and always increases complexity. Finally, the power consumption of an FPGA is higher than for an ASIC [31, 54], although the large speedups usually result in a smaller energy draw than a processor through reduced execution times.

3.4.2 Configuration

Before an FPGA can be used, it must be configured or reconfigured, and this configuration phase, dependent on the technology and the size of the reconfigurable logic, can take thousands of cycles of the fast hardwired logic surrounding it. This makes reconfigurable logic uninteresting for applications requiring frequent changes to the FPGA configuration. Although this problem is somewhat alleviated by the use of FPGAs which can be reconfigured by blocks, this "caching" of configurations increases the area of the reconfigurable logic, and thus its power consumption. The issue of access
bandwidth to the configuration database must also be taken into account, as no real-time compilation of logic to an FPGA exists today.

In addition, the correction of programming must also be verified in (C)PLDs through the application of inputs with known outputs, as there is no way to check that the programming has been done correctly. For FPGAs, it is also possible to read all the configuration memories, but this takes considerable time.

3.4.3 Data and Memory Access

As in all non-streaming applications, the access to data and memory must be carefully studied. In the case of slow reconfigurable logic requiring some parallelism to be cost effective, this problem is more acute. The extra bandwidth needed to download configurations to the FPGA only makes things worse. If, as is often the case, a processor is used for the control parts of the application, some bandwidth must be reserved for this communication too. All these factors place some strain on the I/O for the reconfigurable logic.

In the general purpose processor model, the access to memory is slow, with the huge latencies hidden by a hierarchy of caches of increasing size and latency. This model performs relatively well for many applications typically implemented in software on a processor, but is not at all suitable for the large parallelism required by the FPGA.

This problem can be solved in part by giving the FPGA a dedicated access to memory, but this greatly increases the cost and power consumption and thus might not be feasible, depending on the application. It can also become difficult to manage if the processor and reconfigurable logic each handle part of the processing, as the data must be passed from one to the other, and this link will likely become a bottleneck.

3.4.4 Cost

While not a technical problem per se, the cost of FPGAs should also be taken into account. Indeed, current high-performance FPGAs cost several thousand dollars, far more than almost all hardwired ICs. FPGAs are thus interesting for prototypes, where the speed and ease of testing are paramount; and for small volumes with high added value, where the cost of the FPGA is only a small part of the overall cost. The costs of a set of masks for a hardwired implementation, which are above 1 million dollars in 0.13\(\mu\)m technology, are prohibitive. The integration of a block of reconfigurable logic on a hardwired logic chip is also expensive and has not been done in great volume, which will tend to keep costs high.
3.5 Solutions

This section will describe some of the methods that can be used to alleviate or avoid most of the problems associated with reconfigurable logic. Many of the solutions presented here modify a traditional FPGA by adding hardwired blocks. These blocks give good performance for applications that cannot be efficiently implemented with an FPGA.

There has been a lot of focus on FPGA and its technology in recent years, which has produced vast improvements in both the technology and the tools needed to use it. In parallel, shrinking technology sizes and increasing transistor counts in hardwired logic are straining both the control of the silicon processes and the design methodologies, making FPGAs more attractive by comparison.

Higher transistor densities allow ever larger FPGAs and the addition of hardwired blocks for commonly-used functions, giving the best of both worlds. Embedding ASIC parts to handle control is also a possibility, but expensive, as a custom FPGA is needed, and thus limited to designs that will be produced in very large quantities. Finally, adding full blown hardwired processors in an FPGA turns the notion of adding reconfigurability to a processor on its head, with the FPGA handling the I/O for the processor. These processors are small RISC processors dedicated to simple control tasks, but it should be possible to integrate more complex processors in the near future.

3.6 Conclusions

Fully reconfigurable logic, such as that provided by FPGA technology, has many very useful applications, especially for tasks with large inherent parallelism. The speed with which an application can be designed and implemented make such an approach very appealing for prototypes and small product runs, and the applicability is increasing with the rapid advances in technology and tools.

The clock speed gap compared to hardwired logic, and the far greater cost per unit, however, make fully reconfigurable logic unsuited to several application domains, and these differences are not likely to lessen in the near future. In order to add reconfigurability to a broad range of applications, the grain of reconfigurability must be increased, or the possibilities of reconfiguration limited. This is explored in the next chapter.
Chapter 4

Limited Reconfigurability

This chapter presents different approaches available for increasing the performance of a general purpose processor as an alternative to the fully reconfigurable logic presented in chapter 3. Limiting the amount of reconfigurability reduces the difference in speed between configurable logic and fixed logic. The clock rate that can be attained with this limited reconfigurable logic is thus closer to that of hardwired logic than to that of an FPGA. Hence, whereas an FPGA needs to show an improvement of at least an order of magnitude in the number of cycles to offset its slow speed, limited reconfigurability showing a smaller gain in the number of cycles will still lead to an overall performance gain. Limited reconfigurability can thus be applied to a larger set of domains.

First, the possibility of customizing the processor itself will be presented. This can be either by modifying the Instruction Set Architecture (ISA) or other elements of the processor, such as memory management or branch prediction. Next, a few ways of considering limited reconfigurability will be shown.

4.1 Customizing Processors for an Application

Although a general purpose processor is easy to program and can perform any functionality, the performance compared to an ASIC varies greatly, depending on the application and the amount of hardware involved. Many years of research into ISA design have produced a generally accepted set of instructions that form the basis of most current ISAs. This set, with some variations, can be found in all current general purpose processors.

For most of the applications commonly performed by processors, this basic set allows a fairly efficient execution with ease of programming in high-level languages through advanced compiler techniques. However, as more and more applications, mostly in the embedded space, move from ASICs to processors, spurred on by shorter lifetimes and the pressure of time-to-
market, the reduction in performance becomes a problem. Since many of these embedded applications, such as automotive control, have some form of real-time requirements, as opposed to word processing, a reduction in performance may not be tolerable in terms of safety or functionality. As an example, imagine having to brake hard to avoid an accident, and being asked to wait because the processor is busy! In an attempt to have almost the same performance as an ASIC while maintaining the ease of programming, a small number of special instructions may be added to the ISA.

4.1.1 General Idea

A processor’s ISA is based on the functions it must perform, which are then mapped onto a set of hardware resources. As any algorithm or function can be reduced to a set of arithmetic operations and tests, the minimum hardware a processor needs is an Arithmetic and Logic Unit (ALU). With this hardware, multiplication and division take a very large amount of time, since they must be performed iteratively. An improvement is thus to add a hardware multiplier or divider. Likewise, a square root function evaluator might be added in some cases, etc. Finally, when the application is taken into account, higher-level functions, such as Fast Fourier Transform (FFT) might be implemented in hardware.

The complexity of hardware extensions can vary greatly, going from a simple instruction executed in a few cycles, such a the MultiMedia eXtensions (MMX) present in Intel’s Pentium Processors [8, 72], to complex co-processors, such as the sound or graphic processors present in many personal computers, which can be even more complex than the CPU itself. Nvidia’s [141] newest chip contains 220 million transistors [142], compared to ‘only’ 125 million for the latest Pentium 4 [44]. As the complexity of the function increases, so does its performance gain, but the interactions between the processor and the extra hardware take longer, thus limiting the functions that can be implemented in this way. This specificity is necessary to make significant gains that will compensate for the loss in speed, resulting in an overall gain. We propose to reduce the configurability to constrain the speed penalty. This will allow us to increase the generality of the applications where reconfiguration is interesting and worthwhile.

4.1.2 Importance of Applications

The application or application domain considered is a very important aspect of performance. Indeed, some applications have a very specific structure that is too far from traditional architectures for any extra instructions to be practical. Some applications typically mapped on Digital Signal Processors (DSP) require massive parallel resources for efficient computation. On the other hand, some applications exhibit little or no parallelism, and their
performance can only be improved by performing each operation faster—i.e., by increasing the clock rate. A few of the benchmarks shown in section 6.1.2 show this behavior to a certain degree, notably *swim*.

In general, the more parallelism available in an application, the greater the design options to balance performance, cost and power. Similarly, high parallelism allows tighter coupling between the processor’s core and the application-specific extra logic. This in turn implies a greater re-use of the resources in the processor core, leading to a more economic design.

Some applications can show great gains obtained by adding custom instructions [13, 104] or tightly coupled coprocessors (e.g. [86]) in domains ranging from telecoms to highly realistic graphics for recent video games. The best gains are obtained by hand-tuning of the custom hardware, the interaction with the processor core and the application, all of which require a large investment in time and considerable expertise. This is similar to the programming of DSPs, where gains in performance of an order of magnitude can be made by hand tuning beyond the work of compilers.

However, as the code for applications becomes longer due to extra functionality and options, precise hand-tuning becomes ever more costly and may soon be impractical.

### 4.1.3 Automatic Methods

When the performance of many different applications must be optimized, in the case of very rich functionality or tight design time requirements, hand tuning might not be feasible, and some sort of automation is possible. The work of identifying useful parts in an application and tuning them is rather systematic and uninteresting which makes it suitable for automation. While automation almost always produces results inferior to analysis and tuning by hand, it allows the optimization of far larger applications with acceptable performance results.

Automatic methods for processor customization generally work in two steps: first, profile or analyze the application code to find the parts of the code that are executed most of the time, called *kernels*. Second, reduce the time or power necessary to execute each kernel via analysis of the functionality and inclusion of custom logic. Several solutions exist, both in the academic world [1], and recently, as commercial products [150, 152]. It is also desirable to reduce the added hardware by combining the hardware for the special instructions with the existing processor hardware, and the special instructions together, especially when their use is exclusive.

These methods take considerable time, and cannot be performed on the fly. They are thus not suited for dynamic reconfiguration. The results of automatic methods often show large gains in a relatively short exploration time.
CHAPTER 4. LIMITED RECONFIGURABILITY

26

CHAPTER 4. LIMITED RECONFIGURABILITY

4.1.4 Checksum Instructions Example

As an example of a hand-tuned application, several custom instructions to help calculate the Internet Checksum were designed. This checksum is needed by the Internet Protocol (IP) [129], the Transmission Control Protocol (TCP) [127], and the User Datagram Protocol (UDP) [128], notably. These protocols are the foundations of the current Internet. The custom instructions are used to accelerate one of the tasks in a network processor, for example in a Digital Subscriber Line (DSL) Access Multiplexer (DSLAM) application, providing DSL access

1This is a method for broadband Internet access.
4.1. CUSTOMIZING PROCESSORS FOR AN APPLICATION

Figure 4.2: Hardware structure implementing the checksum instructions. The data alternates paths from the memory to the two registers to hide the load-use latency of 2 cycles in the processor pipeline. The dark path is already present in the processor core, with \( ar[s] \) being the general purpose register holding the address of the current data.

Figure 4.3: L2TP DSLAM Application. The traffic from the clients in encapsulated with Layer 2 Tunneling Protocol (L2TP), forming a Virtual Private Network (VPN), at the Broadband Access Server (BAS) to go through the carrier’s network to the L2TP Network Server (LNS). It then joins the data in the Internet Service Provider’s (ISP) network linked to the Internet (from [140]).
CHAPTER 4. LIMITED RECONFIGURABILITY

unsigned short SWcksum(char * data, long len)
{
    long sum=0; // assume 32 bit long, 16 bit short

    while (len>1)
    {
        sum+=*((unsigned short *) data)++;;
        if (sum & 0x80000000) // if high-order bit set, fold
            sum = (sum & 0xFFFF) + (sum >> 16);
        len -=2;
    }

    if (len) // take care of leftover byte
        sum += (unsigned short) *(unsigned char *) data;

    while (sum >> 16)
        sum = (sum & 0xFFFF) + (sum >> 16);

    return ~sum;
}

Table 4.1: C Source code for the full software approach (from [107]). As a 32-bit register is available, overflow need only be checked for bit 31. The while loop contains 5 instructions when no overflow occurs and 8 instructions with overflow, not counting the loop control.

way to the destination, and finally verified once at the destination. In this case, the routers have no need to calculate the checksum. This is the original design for internet communication, with complexity kept at the edges of the network.

However, when using L2TP encapsulation, the Broadband Access Server (BAS) must encapsulate packets from the user before sending them through the carrier’s Asynchronous Transfer Mode (ATM) network to the Internet Service Provider’s (ISP) servers in a Virtual Private Network (VPN), where they will be decapsulated. This encapsulation allows detailed authentication, through the Remote Authentication Dial In User Service (RADIUS) server, and administration, since the servers know exactly to which subscriber each packet belongs, which is needed for billing and security, among others. In this model, both the DSLAM/BAS and the L2TP Network Server (LNS) in the ISP’s premises must perform a checksum calculation for packets in both directions.

L2TP encapsulation [131] is a means of emulating any Open Systems Interface (OSI) layer 2 protocol over a standard TCP/IP network. This
4.1. CUSTOMIZING PROCESSORS FOR AN APPLICATION

unsigned short Checksum (char* dataptr, int length)
{
    int baseptr = dataptr & 0xFFFFFFFF; // clear low 3 bits
    int StartOffset = dataptr - baseptr;

    CS_INIT( basePtr, StartOffset );
    for (int i=0; i<(length - 8 + StartOffset); i+=8)
    { // main calculation loop
        CS_EXEC( basePtr );
    }
    CS_END( StartOffset, Length );

    CS_NORM();
    CS_NORM();
    CS_NORM();
    CS_NORM();

    return CS_STORE();
}

Table 4.2: C Source code using custom instructions. The for loop contains a single instruction, and can be implemented as a zero overhead loop as the number of iterations is known in advance.

requires the addition of an L2TP header and an UDP header, with the latter containing the checksum. This checksum is the two's complement [68] of the 16-bit sum performed over all the data an a pseudoheader containing most of the UDP header plus some information from the IP header. The packets in L2TP applications are usually large, as the emulated layer 2 protocol is ATM or Ethernet [126], which can have large maximum frame sizes.

The reference software code for the calculation of the checksum is found in the BSD² network code [107, 115], and shown in table 4.1. This code runs rather slowly, mostly due to overflow detection, related to the register width, which limits the size of the words that can be processed at each iteration. With single cycle memory accesses, the performance on a 32 bit processor tends toward 4 cycles per byte of data to be checksummed in a 32-bit processor, while a 64-bit processor should need about 2 cycles per byte, plus some clean-up to reduce the result to a 16 bit value which becomes negligible as packet sizes grow.

A full hardware implementation is of course possible, where any number of bytes can be processed each cycle. However, the construction of the pseudoheader increases the complexity of the hardware, as does the header

²Berkeley Software Distribution
parsing—i.e., structured analysis—that may be necessary depending on the configuration of the L2TP tunnel and the underlying network.

A third solution is to accelerate the checksum calculation through the use of some relatively simple custom instructions, with software modified to make use of them. The aim is to process as many bytes as possible in parallel, using some custom hardware to take care of the eventual overflows that cripple the full software solution, while keeping the parsing and pseudoheader construction in software. The structure of the calculation is shown in figure 4.1.

This algorithm has been mapped onto the 5 following custom instructions:

- $CS_{\text{INIT}}$ sets-up the parallel adder, taking the memory alignment of the beginning of the packet into account.
- $CS_{\text{EXEC}}$ performs the main loop of the checksum calculation, $n$ bytes at a time, with overflow management.
- $CS_{\text{END}}$ performs the last addition, taking the memory alignment of the end of the packet into account.
- $CS_{\text{NORM}}$ completes the calculation by reducing the $n$-byte result to 16 bits. It must be called a number of times to complete the calculation. This is due to some limitations on multicycle instructions in the version of the Xtensa tools used, and could be implemented as a multicycle instruction called only once.
- $CS_{\text{STORE}}$ reads the final value, complements it and stores it into a general purpose register in the processor core’s register file.

Except for the problematic overflow management, all the control is kept in software, keeping the extra hardware to a minimum while greatly increasing the performance of the checksum calculation.

These instructions were implemented on a Tensilica Xtensa 4 processor [152], using the tools provided for a quick design of the special instructions. The memory bus chosen was 64 bits wide, and thus our checksum instructions process 8 bytes per cycle. The hardware structure implementing all 5 instructions is shown in figure 4.2, while the source code using these instructions is shown in table 4.2.

The performance gain of these instructions as compared to the software implementation depends on the size of the packet, as shown in figure 4.4, and the overhead due to setting up the custom instructions might make their gain of a factor of 4 uninteresting for very small packets, such as standard 20 byte IP header checksum calculation. In L2TP applications however, with large packets that must be summed very often, the custom instructions approach’s performance tends toward 0.125 cycles per byte, or 8
4.2. LIMITED RECONFIGURABILITY

Figure 4.4: Cycle counts for the full software approach compared to using custom instructions for varying packet sizes. The overhead of the custom instructions make the approach costly for very small packets, but the gain becomes significant as packet sizes grow.

bytes per cycle. This is a factor of 32 compared to a 32-bit processor, and 16 compared to a 64-bit processor, at the cost of four 16-bit adders, two 64-bit registers, a few multiplexors and some steering logic. The coming transition to Internet Protocol, version 6 IPv6 [130] will also greatly increase the size of packet headers, which might make this approach interesting even for simple unencapsulated packets.

The checksum calculation is only part of the entire TCP/IP stack, and thus the overall speedup will be less than the one shown above. However, other parts of specific network processor applications might also see their performance improved by custom instructions.

4.2 Limited Reconfigurability

Following the approach described above for many applications leads to a large number of custom instructions being added to a processor. In this case, it might be interesting to try to merge some of the added hardware together
CHAPTER 4. LIMITED RECONFIGURABILITY

Figure 4.5: Speed-Reconfigurability curve: limited reconfigurability attempts to stay close to hardwired logic in terms of switching speed while providing enough reconfigurability to allow performance gains.

to reduce the cost, while maintaining the functionality. A generalization of such custom instructions is to add one or more functional units that have some reconfigurability, thus allowing them to perform several varied tasks that do not need to be executed at the same time, and yet may share some hardware. The Intel MMX instructions [72] follow a similar approach, reusing some of the functional units’ hardware for vector instructions. We propose to extend this idea to more than one functional unit at a time. This limited reconfigurability can be seen as a middle point between many custom instructions and a block of fully reconfigurable logic, as shown schematically in figure 4.5.

The reconfiguration possibilities are limited to maintain the speed of the circuit as close to custom logic as possible, as fully reconfigurable logic is rather slow. In this vein, it is similar to coarse-grained reconfiguration approaches. It is possible to apply this directly at the instruction set level [55].

4.2.1 Increase the Solution Space

The detail and amount of reconfiguration allows many design options instead of the two extremes embodied by fast custom logic and slow fully reconfigurable FPGAs. The granularity of the reconfiguration also increases
the design possibilities, making the search of an optimum in the speed-configurability space a possibility [49]. This would require a specific domain of application, and assigning weights to the performance and cost of each solution.

The possibility of having multicycle instructions or reconfigurable logic is very important, as many of the interesting algorithms cannot be split into single-cycle instructions. Another factor in favor of multicycle operations is the cost of reconfiguration, which always has some impact on timing, and must be taken into account. Different configurations of the same hardware need not have the same latencies either.

4.2.2 Coarse-Grain Reconfiguration

One way to limit reconfigurability is to apply it more broadly than is done in an FPGA: instead of having single bit memories, larger tables or functions may be reconfigured. This can lead to better results for some applications [106]. It is also possible to switch between mappings of different functional blocks, such as adders; this shows similarity to reconfigurable systolic arrays [46]. However, the use of large memories may also slow down the logic.

4.2.3 Block Reconfigurability

In an attempt to limit the loss in speed due to reconfigurable hardware, it is interesting to group reconfigurable blocks and allow reconfiguration only over this entire group. This is also necessary when a larger unit can be reconfigured as several smaller units, as for the case study presented in chapter 5. This non-uniform resource space, where the number of resources varies in addition to the types of units according to the configuration, greatly increases the complexity of any partitioning and allocation mechanism. The complexity of the reconfiguration itself is reduced by the lower amount of configuration information necessary and the smaller number of switching elements.

Block reconfiguration is similar to coarse-grained reconfiguration in the sense that a single configuration change affects more than a single bit, but with the added difficulty from the non-uniform resource space.

4.3 Dynamic Reconfiguration

No matter the type of reconfiguration, in most cases, reconfiguring is seen as a relatively infrequent event. Indeed, even with partial reconfiguration, downloading a configuration to an FPGA still takes several tens of milliseconds, equivalent to 30 million instructions in a fast processor. When reconfiguration delays decrease, the possibility of doing dynamic reconfiguration
becomes both interesting and feasible. The interest of dynamic reconfiguration is a greater autonomy and the possibility to adapt very quickly to changing application demands.

4.4 Conclusions

Adding custom instructions to a processor to enhance its performance for some specific tasks can produce large gains in performance, as an example applied to the network domain has shown. However, as the number of applications increases, this approach is no longer feasible. In its place, limited reconfigurability, either in the classical functional units or in the hardware implementing custom instructions, can produce interesting gains in a wide variety of applications. The low speed overhead of the limited reconfigurability makes this possible, notably when the reconfiguration is performed dynamically. Chapter 5 will detail an application of limited dynamic reconfiguration.
Chapter 5
Case Study

To give more substance to the concepts presented in chapter 4, a case study of adding limited reconfigurability will be detailed. This case study will show the feasibility of the approach and provide some quantitative results. It is also useful in bringing attention to the many details of a design with limited reconfigurability.

This case study details the application of limited reconfigurability to an out-of-order, superscalar processor (discussed in chapter 2). Specifically, the Floating Point Unit (FPU) will be given some reconfigurability to improve overall performance. All aspects of the design will be covered, including the hardware design of the reconfigurable multiplier, the timings of the parts that are modified, and the reconfiguration decision algorithm.

This is an example of limiting reconfigurability to broaden the range of improved applications, as compared to adding custom instructions or fully reconfigurable logic.

5.1 Basic Idea and Context

The source of this case study is the observation that many of the functional units in a superscalar processor can be idle for long periods of time, depending on the applications running. However, this hardware must be present because the processor would run the applications that actually need this functional unit very slowly if it were absent. For example, most integer code\(^1\) uses the FPU—for multiplication—less than 1% of the time, and even several floating point benchmarks use it less than 20% of the time, as can be seen in table 6.1 on page 86. To corroborate this, figure 5.1 shows the average usage of the FPU for the same benchmarks. Integer benchmarks make almost no use of the FPU, and many floating point benchmarks use less than one on average. Hence the idea to try to have the processor adapt somewhat

\(^1\)Code using mostly integer operations.
Almost all the integer benchmarks (left up to twolf) make little or no usage of the FPU, and only sixtrack comes close to full utilization.
to the application currently being executed by adding reconfigurability to the FPU.

The focus is on a superscalar processor capable of executing more than one instruction per cycle as it would make no sense to add parallel resources to a processor that cannot use them. The approach is to make the hardware already present in the processor more versatile instead of adding more hardware. This also has the positive effect of keeping power consumption under control. Though not the main focus of this study, power is becoming an important factor in general-purpose processors, as the thermal densities approach those of a nuclear power plant. Any modification that avoids adding hardware or reduces the execution time will reduce the power requirements of the processor or the overall energy expenditure, respectively.

A general purpose processor must be able to execute a very diverse set of applications, many of which might not even exist at the time the processor is designed. This case study follows in the same direction, with the aim of improving performance over the broadest selection of applications possible. As such, the gains will by necessity be more modest than those obtainable by focusing on a single algorithm than can be highly parallelized. To present a fair comparison of a superscalar with limited reconfigurability with a non-reconfigurable case, all aspects will be considered, including timing, extra wires and any modifications to the structure of the functional units and the reconfiguration decision. Another important objective, again contrasting with fully reconfigurable logic or custom instructions, is to maintain binary compatibility (section 2.1). This aspect is important to preserve the large investments in software, which has become one of the greatest cost in most current information systems.

5.1.1 General-purpose Processor Definition

A definition of a general purpose processor could be the following:

“A machine that can execute any functionality or algorithm by reading a properly formatted sequence of instructions.”

This functionality is obtained by dividing the desired algorithm into a sequence of very simple operations which the processor’s functional units can execute. This operation is usually performed by a compiler, allowing the input to be written in a high-level language, far simpler to understand than the machine language accepted by the processor.

This implies that a choice about the set of simple operations to provide must be made when designing a processor. The set of basic operations a processor can perform called the Instruction Set Architecture (ISA), generally contains 3 groups of instructions: arithmetic, control and input/output. Due to the cost of adding too many instructions, notably in terms of instruction
sizes, most ISAs are very similar in the number and functionality of instructions they contain, which is a set capable of expressing most operations succinctly. There are actually two main approaches: Complex Instruction Set Computers (CISC) try to provide instructions for almost all possible cases, resulting in instructions of variable sizes and compact code. On the other hand, Reduced Instruction Set Computers (RISC) try to limit the ISA to a minimal set of instructions capable of implementing any functionality, trading a greater simplicity in instruction decoding for a larger code. Recent CISC processors, however, actually break longer CISC instructions into smaller RISC-like \( \mu \)operations to avoid the issues raised by the processing of long, complex instructions, rendering the distinction somewhat obsolete [28]. In order to achieve high performance, both out-of-order processing and a superscalar architecture will be used, to make as much parallelism as possible available to the processor.

5.1.2 Performance Evaluation

To be able to evaluate the impact of our modifications, some way of quantifying the performance of different alternatives is required. The generally accepted way of comparing the performance of processors is through the use of benchmarks, i.e., a well-chosen set of specific programs that represent typical uses of a processor or examples that stress the processor performance in some way. These are then run with predefined inputs, the execution time then giving a reasonably precise estimate of performance. A benchmark suite contains several programs to test different parts of a processor, with the overall result giving an estimate of all-round performance. As our simulator, described in detail in section 6.1.1, allows constraints on the number of instructions executed, we will use as comparison metric for our designs the Instructions Per Cycle (IPC), a standard measure of superscalar performance. This is simply defined as the ratio of the number of instructions executed by the number of cycles necessary to execute them [28].

\[
IPC = \frac{\text{Number of Instructions}}{\text{Number of Cycles}}
\]  

(5.1)

5.1.3 Applying Limited Block Reconfiguration

Limited reconfiguration will be added to the processor’s Floating Point Unit, as described in [19]. The FPU is the unit responsible for executing all floating point instructions, and several such units may be present in a processor to increase performance. In some cases, the FPU may be split into a floating point add/subtract unit and a floating point multiply/divide unit, although we will consider a full FPU capable of all floating point operations. This large unit also must manage conversions between integer and floating point
numbers, and can often be idle when the processor is not executing floating point code. In addition, a processor may use the FPU’s multiplier for integer multiplications [133, 134], since, as shown in section 5.2.5, the internal structure is very similar. The FPU was chosen due to its large size and infrequent use in many applications. The FPU(s) cannot simply be removed, as the performance for applications that make use of floating point operations would drop by several orders of magnitude without dedicated hardware. In this case, each floating point operation would have to be cut into many integer instructions. For many years, this is how processors operated, and applications requiring many floating point instructions would require a floating point coprocessor to assist the main processor. The FPU is thus a prime candidate to receive limited reconfigurability.

Figure 5.2: Reconfiguration of the FPUs. Each FPU can be configured either as a FPU (a) or as several xALUs with a higher latency than a normal ALU (b). The extra paths to bring integer instructions to the xALUs are shown in blue.
CHAPTER 5. CASE STUDY

The FPU will have the possibility of being reconfigured as several extra Arithmetic and Logic Units (xALUs), by re-using some of the hardware in the multiplier tree. These xALUs are capable of executing most of the instructions normal ALUs can, excluding the operations requiring a lot of extra logic above the adder/subtractor obtained from the FPU, such as a variable shifter (e.g., [97]). The reconfiguration will be in blocks to keep the reconfiguration logic as small as possible, and to minimize extra wires and multiplexors. This means that we are trading a single FPU for several xALUs at once. The number of xALUs per FPU can vary, and the consequences of this design choice will be detailed in section 5.4.3. As the sensitivity analysis in section 6.6 will show, the range of values of interest is actually quite short, due to the limits on the amount of parallelism available. Whether switching from the FPU to the xALUs or the reverse, the entire block implementing the FPU must be idle, i.e. either the FPU must be idle, or all the xALUs obtained by reconfiguration of this FPU must be idle. While this condition could be somewhat relaxed, the result is an important simplification of the wiring and control of the reconfiguration, leading to a faster design.

5.2 Hypotheses

This section will define the context of our case study among the many different processor architectures available and detail existing blocks that will be impacted by the addition of some reconfigurability. The choices and basic assumptions guiding the rest of the study will also be outlined. We consider a 64-bit processor, which is becoming mainstream for personal computers [85, 110, 111, 135]. All high performance general purpose processors today are out-of-order and superscalar, and our base model will reflect this. The overall processor configuration chosen will be shown, including the internals of the execution engine and the detailed structure of an FPU and a multiplier. The various configuration parameters for the processor will also be discussed.

5.2.1 Superscalar, Out-of-Order Processor

A superscalar processor can execute more than one instruction per cycle; however, the peak rate of instructions is almost never reached. The average number of instructions per cycle is limited by the parallelism available in the application, and usually follows a law of diminishing returns. In addition, an out-of-order processor may change the order in which it executes the instructions of the application to increase the parallelism it can exploit. This requires a complex scheduler to manage all the instructions that are in flight—being considered for execution—at a given moment. A superscalar processor uses a combination of methods to achieve high performance. The list below and figure 5.3 detail the most important:
5.2. HYPOTHESES

Figure 5.3: A full superscalar processor pipeline. Each of the 5 stages shown can take any number of cycles, depending on the clock speed and the complexity of the ISA. The instruction flow is roughly: pre-processing, read register file, execute, write to register file. The forwarding paths may bypass this schema to gain speed.
Reservation Stations hold all the instructions whose dependencies have not yet been fulfilled. They are usually placed in front of the functional units.

Precise Exceptions are required for a superscalar to match the behavior of an in-order processor when errors or interruptions occur. One way of achieving this is by having a complex in-order commit engine at the end of the out-of-order pipeline that serializes the instructions that have been successfully executed.

The Register File must have a fair number of read and write ports, as several instructions may be executed in a single cycle, requiring many words to be read each cycle.

Shadow Registers are extra registers that might hold copies of the Register File or even different versions of the same register, and help reduce the pressure on the Register File.

Forwarding paths between the functional units also help reduce the pressure on the register file by providing the result of an instruction directly to whichever functional unit might need it without going through the register file.

The Execution Window bounds the number of instructions that can be considered for execution at any given time, and thus the parallelism that can be extracted from a given code. It also places bounds on the complexity of the scheduler.

Dispatch, Issue and Commit widths represent the number of instructions per cycle that are read from the program into the out-of-order engine; the number of instructions per cycle whose dependencies are fulfilled and are sent to the functional units for execution; and the number of instructions per cycle that have completed execution and can be retired, respectively.

5.2.2 Internal Processor Structure

A superscalar and out of order processor contains a number of functional units, generally splittable into two groups, integer and floating point units. A complex out-of-order scheduler is present, including reservation stations to hold instructions whose dependencies have not yet been fulfilled and a commit unit to force the writing of the results in the original program order. Forwarding paths between functional units to reduce the latency of dependent instructions are also present wherever possible.

The general structure of a processor pipeline is as shown in figure 5.3. Each stage shown can actually be partitioned into several cycles to increase
5.2. HYPOTHESES

5.2.3 Adders

There are two types of adders used in computer arithmetic: the first set is what is generally referred to as an adder, taking 2 \( n \) bit inputs, with a possible carry in bit, and producing a single \( n \) bit output, with a carry out bit. These are called Carry Propagate Adders (CPA). The fastest adder in this group is the Carry-Lookahead Adder \([51, 71]\) and its many variants. The other group is also referred to as compressors, as these adders do not produce a sum, but reduce the number of inputs to a number \( \geq 2 \). They are denoted as \( x \)-to-\( y \) compressor, where \( x \) inputs are compressed into \( y \) outputs, \( y \geq 2 \). The simplest of these compressors is the Carry Save Adder (CSA), which is a 3-to-2 compressor. This compressor avoids any sort of propagation in the adder, and is thus much faster than a CPA. The structure of an \( n \)-bit CSA, shown in figure 5.4, is simply \( n \) full adders in parallel. The delay is thus independent of the number of bits. For 64-bit adders, a CPA is about 5 times slower than a CSA, independently of technology \([68]\).

5.2.4 MUL/DIV Functional Unit Implementation

Hardware multiplication is obtained by a method directly copied from the way humans perform multiplication, i.e. multiplying each digit of the multiplier by the multiplicand and then adding the resulting partial products. Simple multipliers follow this iterative approach, using only an adder and some registers, as shown in figure 5.5 \([70]\). There are two steps involved, namely calculating the partial product and then adding it to the temporary result. This approach requires one step for each bit of the multiplicand, although high-radix number representations can reduce this number at the...
Figure 5.5: Iterative multiplier using a single adder. At each cycle, the logical AND of a single bit from the multiplier and the multiplicand is added to the partial product, which is also shifted by one bit. The total number of cycles is thus equal to the width of the multiplier.

cost of conversion hardware. This iterative process can be managed entirely in software, or can be simplified by the use of a multiply-step instruction performing the shifts and the addition for a single iteration. High-radix approaches and Booth recoding [96] combine several bits or use redundant coding to process more than one bit in a single iteration, and can use a small tree of CSAs to form the partial products. They may increase the speed of the multiplier in some cases, but not all [84]. However, redundant coding implies the use of signed numbers, thus adding sign-extension hardware to the partial products generation. For simplicity, our multiplier will not use Booth recoding. For a high performance processor, a fast multiplier is a necessity, even in algorithms that don’t explicitly use multiplication—e.g., for the generation of the addresses of the elements of an array when the element size is not a power of two.

Fast multipliers use the same principle as iterative ones, but calculate all the partial products in parallel, and then add them together with a complex reduction tree [15, 99], as shown in figure 5.6. It is possible to use a very similar tree to perform division, following a convergence algorithm [22]. In this case, a multiply/divide unit is formed, requiring less hardware than two
5.2. HYPOTHESES

separate units at the cost of some speed and complexity. Some commercial
processors, such as the Intel Itanium 2, use this kind of functional unit.

In binary representation, the partial products are simply the logical AND
of each bit of the multiplier with the multiplicand. The multiplier tree is
then composed of many CSAs, with a final CPA producing the multiplication
result. In some cases, to achieve better regularity for integration, a 4-to-2
compressor [101] might be built from 2 CSAs as in figure 5.7 and used for the
compressor tree. However, this would slightly increase the number of levels
in the tree, although the speed of these compressors can be optimized [62].
Higher order compressors may be built [66], such as a 9-to-2 compressor
described in [81]: the best solution to reduce the partial products is to
optimize the entire compressor tree [67, 84]. The final adder can be any
fast adder for the target technology. However, it can best be optimized by
taking into account that the bits from the sum of the partial products do not
arrive at the same time, as they travel through different numbers of layers
of compressors [7]. 4-to-2 compressors ‘flatten’ the delay profile of the tree,
making the design of the final adder somewhat simpler.

The delay of the 64-bit multiplier tree can be estimated with technology
independent calculations, to be refined in section 5.4 with the complete
design. Without using any high-radix representation, the number of levels
in the tree can be calculated as follows: there are initially \( n \) partial products.
As we can reduce the number of partial products by a factor of at most \( 2/3 \)
at each level, the total number of levels to reduce the \( n \) partial products into
a sum and a carry is given by the recursion in equation (5.2).

\[
H(n) = 1 + H(\lceil 2n/3 \rceil) \quad (5.2)
\]

These will then be added by the final CPA, producing the result of the
multiplication. Using equation (5.2), we find that a 64-bit tree requires 10
levels. Following the structure of figure 5.6, we can calculate the delay for
the entire multiplier as \( 10 \cdot 1\tau + 5\tau = 15\tau \). For comparison, a 64-input tree
built using 4-to-2 compressors would also need \( \lceil \log_2(64) \rceil - 1 = 5 \) levels
of 4-to-2 compressors, or 10 levels of CSAs built following figure 5.7 (left).
However, a 63-input tree would only require 9 levels of CSAs, but still 5
levels of 4-to-2 compressors.

5.2.5 Floating Point Unit

As many scientific calculations must handle large variations in the values
they calculate, any form of fixed point representation would introduce too
many rounding errors. To avoid this problem, a floating point representation
can be used. This splits a word into two parts, the mantissa and exponent
[125]. It is possible to use an integer ALU to perform floating-point calcu-
lations, by separating the exponents and mantissas, and calculating each of
Figure 5.6: Structure of a fast multiplier. The 64-bit sources $A$ and $B$ are multiplied to produce the 127-bit product $P$. A CSA has a delay of $1\tau$, a CPA has a delay of about $5\tau$, with $\tau$ being the delay of a full adder. The total delay of the tree is $15\tau$, while the partial products are simply the logical AND of $A$ and $B$, suitably shifted.
Figure 5.7: A 4-to-2 Compressor based on two CSAs (3-to-2 compressors, left). This can be used for better regularity in integration, as shown on the right.

Figure 5.8: High-level structure of an FPU: The multiplier (in gray) is framed by the unpack/pack and normalize logic. A sample latency for the entire FPU being 4 cycles, the unpack is estimated to take 1 cycle, the multiplication 2 cycles, and the normalize and pack logic the last cycle.
them separately, while reassembling the result into floating point representation. However, this takes a large amount of time, taking several tens of cycles in addition to performing the calculation itself. A simple FPU might contain only FP addition and subtraction, in a way similar to an integer arithmetic unit. A more complex FPU, like the one considered here, contains also multiplication, division and function evaluation, such as square root. Finally, it must always be able to perform conversions to and from integer representation. A floating point unit capable of multiplication and division can be built by adding floating point specific logic to an integer multiplier/divider, as the operands must be internally converted to fixed point before performing the calculation. The final adder in the multiplier can be used to perform floating point addition and subtraction, resulting in a complete FPU. Other functions can be implemented through iterations of the multiplier or divider.

A Floating Point unit, capable of performing the basic operations of addition/subtraction and multiplication/division, is composed of 4 stages, as shown in figure 5.8. From top to bottom, these are

**Floating Point Unpack** Floating Point values, stored in packed format, must be separated to calculate the exponent of the result and the alignment of the mantissas for the calculation stage.

**Multiplier Tree** The multiplier is very similar to the one shown in figure 5.6, capable of integer multiplication, with eventually extra logic and wiring to handle division, and addition/subtraction using the final adder.

**Normalize** The result of the multiplication must then be normalized, i.e., shifted to conform to IEEE Floating Point specifications\(^2\), and rounded if necessary. Some checks, such as division by zero and overflow, might also be performed here.

**Floating Point Pack** Finally, the fixed point result must be converted to floating point, taking into account the sign and exponent of the result and packing them back into a single word.

A fast implementation of a complete FPU like the one described above can be found in the Intel Itanium2 processor. The datasheet for this processor indicates a latency of 4 cycles for all FPU operations, which we estimate\(^3\) are split as in figure 5.8, with the unpack taking 1 cycle, the multiplier taking 2 cycles, and the normalize and pack logic taking the last cycle. As detailed in section 5.4.3, should the normalize and pack take 2 cycles, leaving a single cycle for the multiplier, the results would be affected in a positive way, since we would have even more *slack* for our modifications.

---

\(^2\)Notably, ensure that the *Most Significant Bit* (MSB) is a ‘1’

\(^3\)The actual values are a closely guarded secret.
5.3 Scheduling of Reconfiguration

As for any reconfiguration, a decision must be made about when and how to reconfigure the hardware to extract the best performance possible. The importance of this choice grows with the duration of the reconfiguration and its impact on the performance. In the case of the FPU, a wrong decision might leave no functional unit capable of executing floating point instructions, making this decision critical. The reconfiguration is dynamic, thus requiring robustness and relatively quick decisions.

The decision algorithm can range from a relatively simple finite state machine to a complex sequence of calculations. Although a theoretical approach can take as much time as necessary, the complexity of the algorithm must be kept low for the implementation in hardware, lest the gains from reconfigurability be cancelled by the size and delay of the decision algorithm.

The problem to be solved will be detailed, leading to a theoretical model. This model will then serve as a basis to propose several decision algorithms of varying complexity.

5.3.1 Problem Definition

The inputs of the problem are the instructions being issued, with their dependencies resolved, and ready to be executed by a functional unit—i.e., a subset of the instructions in the reservation stations plus any independent instructions dispatched.

There are several types of instructions, generally splittable into two groups: those executable by an ALU, and those executable by an FPU. As the xALUs can only execute a subset of normal ALU operations, there are three groups to consider: instructions that can be executed by the FPU, instructions that can be executed by the xALU or the ALU, and instructions that can be executed by the ALU. In a similar way, we have three different types of resources, or functional units, each with their own parameters. The first significant parameter, latency, is the delay between when an instruction enters a functional unit and when the result is available. The second parameter is the issue rate, representing the number of cycles to wait between the beginning of execution of two consecutive instructions on a functional unit. As all our units are fully pipelined, the issue rate is always one instruction per cycle.

The problem can now be expressed as follows:

\[ \text{Given the set of instruction arrivals of the three different types at each cycle, and given the set of resources available for execution of these three types, find the best mapping between the two sets—i.e., the mapping that executes all the instructions in the smallest number of cycles.} \]
Figure 5.9: Problem Description. Integer (A) and Floating Point (F) instructions arrive, and must be dispatched to the functional units in the most effective way. The black and blue paths show the static case, while the red path shows the extra options gained by reconfiguration.

The problem is shown in figure 5.9, with the black and blue paths representing the case of fixed resources, where the problem is rather trivial: each cycle, simply select a free resource capable of executing each instruction until there are no idle resources or no instructions left, the number of total instructions sent to functional units being bound by the issue width. This is the job done by the scheduler in a superscalar processor [28, 63]. The reservation stations simplify this work by maintaining queues in front of each functional unit or type of functional unit.

In the case of a dynamic set of resources due to reconfigurability, shown in red in figure 5.9, the problem is no longer trivial, as there is a trade-off between the use of different resources with different latencies that cannot be used at the same time. When combined with the fact that the future arrivals of instructions are not known, no closed form solution exists. This leads to the use of the discrete state model detailed below.

5.3.2 Non-Linear State Equation Model

In order to determine a good algorithm for the decision about reconfigurability, an analytical model is needed. To simplify the model, the following reductions are made:

- Even though the xALUs cannot execute all ALU instructions, the most frequent instructions can be executed by both functional units, and the distinction will not be made for the decision algorithm. Instructions that can only be executed by the ALUs will not be taken into account
by the decision algorithm and will be sent directly to an ALU, never to a \textit{xALU}.

- Computing the tally of the instructions of each type whose dependencies are fulfilled would take too much time, and so all the instructions in the execution window will be counted, regardless of dependencies. The impact of this simplification can vary fairly widely, especially in the case of applications with many dependent instructions, but this simplification cannot be avoided as the alternative would make the counting logic very complex and slow.

- When the algorithm can take advantage of it, the future arrivals of instructions will be considered known. This is useful to find an upper bound to the performance available.

The model of the decision problem we must solve requires the following definitions.

Let

- \( m \) be the number of ALUs present.
- \( n \) be the number of reconfigurable FPUs present.
- \( \alpha \) be the number of \textit{xALUs} obtained by reconfiguring an FPU.
- \( L_a \) be the latency of an ALU.
- \( L_f \) be the latency of an FPU.
- \( L_{xa} \) be the latency of an \textit{xALU}.
- \( k \) be the current cycle, a discrete time with \( k \geq 0 \).
- \( u_a(k) \) be the number of \textit{integer} instructions arriving at time \( k \), with \( u_a(k) \geq 0 \ \forall k \).
- \( u_f(k) \) be the number of \textit{floating point} instructions arriving at time \( k \), with \( u_f(k) \geq 0 \ \forall k \).
- \( x_a(k) \) be the number of \textit{integer} instructions waiting for execution at time \( k \), with \( x_a(k) \geq 0 \ \forall k \).
- \( x_f(k) \) be the number of \textit{floating point} instructions waiting for execution at time \( k \), with \( x_f(k) \geq 0 \ \forall k \).
- \( y_a(k) \) be the number of \textit{integer} instructions that commit at time \( k \), with \( y_a(k) \geq 0 \ \forall k \).
- \( y_f(k) \) be the number of \textit{floating point} instructions that commit at time \( k \), with \( y_f(k) \geq 0 \ \forall k \).
$r(k)$ be the number of FPUs reconfigured as $xALUs$ at time $k$, \( \{0 \leq r \leq n ~ \forall k\} \).

The problem is now to find the set of optimal numbers of reconfigured FPUs at each cycle $r_{opt}(k)$ that allows the system to execute all instructions in the shortest time—i.e., that minimizes the finishing time $k_{end}$.

As the problem is intrinsically discrete, no discretization effects occur. The state equations for this model, derived directly from analysis of the execution of the processor pipeline, can then be written as:

\[
x_a(k+1) = x_a(k) - (m + \alpha \cdot r(k)) + u_a(k) \quad (5.3)
\]

\[
x_f(k+1) = x_f(k) - (n - r(k)) + u_f(k) \quad (5.4)
\]

\[
y_a(k) = \min(x_a(k), m) + \\
\min\left(\max\left(x_a(k-(L_{xa}-1)), m\right), \alpha \cdot r(k-(L_{xa}-1))\right) \quad (5.5)
\]

\[
y_f(k) = \min(x_f(k-(L_f-1)), n - r(k-(L_f-1))) \quad (5.6)
\]

These two sets of non-linear equations, also shown in figure 5.10, model the flow and delay of instructions as they travel through the processor’s functional units. They define the maximum number of instructions of each type that can issue or commit, and then limit this value with the actual number of instructions present, taking into account the functional unit latencies. From this model, several decision algorithms can be developed. In all cases, any decision will be limited by the need to wait for a functional unit to be completely idle before reconfiguring it.

### 5.3.3 Integer Linear Programming Model

Following an integer linear programming approach allows the linearization of the problem, at the cost of an increase in complexity [100]. This complete model takes all effects into account and is detailed below. This is an oracle model, since it considers the arrival times of all the instructions as known when performing the optimization. Let:

- $u_j$ be the logical functional units (FU). Thus, several $u_j$ represent the same physical functional unit, but may have different latencies as they represent the different configurations. This removes the non-linearity at the expense of several extra variables. The $u_j$ can be grouped into 4 overlapping sets, $ALU$, $xALU$, $FPU$ and $RFPUs$, being the ALUs, $xALUs$, FPUs and Reconfigurable FPUs, respectively, with $xALU \cup FPU = RFPUs$. An example, with 3 ALUs, 2 FPUs and a reconfiguration factor $\alpha$ of 3, could be:
Figure 5.10: Decision Problem State Model. The inputs $u_a(k)$, $u_f(k)$ and $r(k)$ are highlighted in dark, and the outputs $y_a(k)$ and $y_f(k)$ are highlighted in light gray. $m$ and $n$ are the number of ALUs, resp. FPUs. $\alpha$ is the number of xALUs per FPU, $A_1$ and $A_2$ are the left and right parts of the sum in equation (5.5), and $F$ is the right part of equation (5.6). $L_{xa}$ and $L_f$ are the latencies of an xALU and an FPU, resp., and $\delta(t)$ represents a delay of $t$ cycles.
with \( \{u_0, u_1, u_2\} \in ALU, \{u_3, \ldots, u_{10}\} \in RFPU, \{u_3, u_7\} \in FPU, \) and \( \{u_4, \ldots, u_6, u_8, \ldots, u_{10}\} \in xALU. \)

- \( l_u \) be the latency of logical FU \( u. \)
- \( I_w \) be the issue width of the processor (considered equal to the commit width).
- \( x_{i,u,t} \) be the boolean variable indicating whether instruction \( i \) started execution on FU \( u \) at time \( t. \)
  \[ x_{i,u,t} = \begin{cases} 1 & \text{if instruction } i \text{ issued to FU } u \text{ at time } t \\ 0 & \text{otherwise} \end{cases} \quad \forall i, \forall u, \forall t \]
- \( \xi_{u,t} \) be the availability of functional unit \( u \) at time \( t. \) This variable encompasses the reconfiguration decision, which can be simply derived from it; reconfigurable FPU \( u_n \) is configured as an FPU at time \( t \) if \( \xi_{u_n,t} = 1, \quad u_n \in FPU. \)
  \[ \xi_{u,t} = \begin{cases} 1 & \text{if FU } u \text{ is available at time } t \\ 0 & \text{otherwise} \end{cases} \quad \forall u, \forall t \]
- \( t_i \) be the time at which instruction \( i \) has finished execution— i.e., is ready to commit. We have
  \[ i : t_i = \sum_t \sum_u x_{i,u,t} \cdot (t + l_u) \quad \forall i \]
- \( T \) be the finishing time of the last instruction
- \( \rho_{u,t} \) be the occupation indicator of functional unit \( u \) at time \( t \)—i.e., whether an instruction started execution at time \( t. \) As the issue rate is 1 instruction per cycle, the instructions that started execution in previous cycles do not block the functional unit:
  \[ u, t : \rho_{u,t} := \sum_i x_{i,u,t} \quad \forall u, \forall t \]

The problem can now be defined as:

\[ \min f = T \]

Under the following constraints:
5.3. SCHEDULING OF RECONFIGURATION

1. $T$ is equal to the greatest $t_i$—i.e., the greatest finishing time must be minimized:

   $$i : T \geq t_i \quad \forall i$$

2. Each functional unit executes at most one instruction each cycle:

   $$u, t : \rho_{u,t} \leq 1 \quad \forall u, \forall t$$

3. Each instruction must be executed exactly once:

   $$i : \sum_{u,t} x_{i,u,t} = 1 \quad \forall i$$

4. At most $I_w$ instructions can be issued every cycle (limit on the issue width):

   $$t : \sum_{i,u} x_{i,u,t} \leq I_w \quad \forall t$$

5. The reconfiguration possibilities are limited by functional unit occupation. In essence, if a functional unit is still executing an instruction—i.e., an instruction started execution less than $l_u$ cycles before, this functional unit cannot be reconfigured at this time. This must be defined for each $u \in RFPU$. We have

   $$\rho_{u,t} = 1 \Rightarrow \zeta_{u,s} = 1 \quad s = t, ..., t + l_u$$

   which is equivalent to,

   $$t : \rho_{u,t} \leq \zeta_{u,s} \quad s = \{t, ..., t + l_u\} \quad \forall u \in RFPU, \forall t$$

6. A single physical FU cannot execute an integer and a floating point instruction at the same time—i.e., it can only execute an instruction if it was available:

   $$u, t : \rho_{u,t} \leq \xi_{u,t} \quad \forall u, t$$

   Likewise, a FPU cannot be used at the same time as any of its xALUs. Let $\{u_n, ..., u_{n+\alpha}\} \in RFPU, u_n \in FPU, \{u_{n+1}, ..., u_{n+\alpha}\} \in xALU$:

   $$t : \alpha \cdot \xi_{u_n,t} + \sum_{v=u_{n+1}}^{u_{n+\alpha}} \xi_{v,t} = \alpha \quad \forall u_n \in RFPU, \forall t$$

7. ALUs cannot be reconfigured:

   $$\xi_{u,t} = 1 \quad \forall u \in ALU, \forall t$$

8. Some $x_{i,u,t}$ are always 0:
• Integer instructions cannot be executed on FPUs.

\[ x_{i,u,t} = 0 \quad \forall i \in \text{Int}, \ \forall u \in \text{FPU}, \ \forall t \]

Likewise, floating-point instructions cannot be executed on ALUs or xALUs.

\[ x_{j,v,t} = 0 \quad \forall j \in \text{FP}, \ \forall v \in \{\text{ALU} \cup \text{xALU}\}, \ \forall t \]

• Instruction arrivals imply \( x_{i,u,t} = 0 \ \forall u, \forall t < t_{\text{arrival}}(i) \)

• Instruction dependencies. If instruction \( j \) of type \( J \), executable on the set of FUs \( V \), is dependent on instruction \( i \) of type \( I \), executable on the set of FUs \( U \), it cannot be executed until instruction \( i \) has completed execution on one of the FUs that can execute it:

\[
\sum_{\tau=0}^{t-l_u} x_{i,u,\tau} = 0 \Rightarrow x_{j,v,t} = 0 \quad \forall i \in I, \forall j \in J, \forall u \in U, \forall v \in V, \forall t
\]

which is equivalent to,

\[
x_{j,v,t} \leq \sum_{u \in U} \sum_{\tau=0}^{t-l_u} x_{i,u,\tau} \quad \forall i \in I, \forall j \in J, \forall v \in V, \forall t
\]

Thus, by setting the arrival and dependency constraints according to traces of the desired benchmark, an optimal solution can be obtained via linear programming tools, such as Ilog CPLEX [116].

The complexity of Integer Linear Programs is usually expressed in terms of the number of variables and constraints. This algorithm has a number of variables proportional to \( i \cdot u^2 \cdot t^2 \), and a number of constraints on the order of \( i^5 \cdot u^4 \cdot t^{10} \), although some of these constraints are redundant.

### 5.3.4 Theoretical Analysis

The linear programming model shown above is very complex and makes the assumption that all the instruction arrival times are known. An optimal solution can only be obtained for about 300 instructions over 75 cycles, with 3 ALUs, 2 RFPUs and 3 xALUs per RPU, for a total of 11 logical functional units, when no dependencies are included. In this case, there are about 400000 variables and 2000000 constraints. With dependencies, problems with about 60 to 70 instructions can be solved, as there are many more constraints. This takes a large amount of processing power and memory, and is completely impossible in real-time. It serves, however, to quantify the upper bound of the performance that can be achieved by the limited reconfiguration of the FPU, and thus give an estimate of the quality of the suboptimal algorithms presented below.
The linear program was solved using CPLEX in mixed integer mode, with a program defined following the equations in section 5.3.3 and the processor parameters in table 6.2 on page 88. The result of this solution is compared to the sub-optimal algorithms below in section 5.3.5.

Based on the non-linear state model, several algorithms can be defined. A naive solution using a local optimum to maximize the number of instructions issued at every cycle produced poor results, on the order of 15% worse than the other solutions presented here, and was immediately discarded. However, it gives a glimpse of the inherent complexity of a seemingly relatively simple problem.

Another solution, called balanced, considers the two equations \((5.3)\) and \((5.4)\) as linear functions of \(r(k)\), and then attempts to balance the number of instructions of each type with the number of appropriate functional units. This follows a wider view of having the processor’s resource repartition be roughly proportional to the instructions’ repartition. Adding a weighing factor, we pose \(x_a(k) = \lambda \cdot x_f(k)\) to get:

\[
x_a(k) - m - \alpha \cdot r = \lambda \left( x_f(k) - n + r \right) \tag{5.7}
\]

\[
r_{opt}(k) = \frac{x_a(k) - m + \lambda \left(n - x_f(k)\right)}{\lambda + \alpha} \tag{5.8}
\]

The optimal \(r_{opt}(k)\) in this equation is seldom integer, and must thus be rounded to the nearest integer\(^4\) to get the final number of FPUs to reconfigure as \(xALUs\). While this method produces good results, the calculations in floating point, which might be approximated with integers or fixed point, make any implementation costly.

Finally, an experimental approach called threshold, derived from control theory, uses simple thresholds with hysteresis to control the changes to the FPU [20]: We first determine the thresholds \(T_a(k)\) and \(T_f(k)\), and normalize the number of instructions \(A(k)\) and \(F(k)\) to get \(N_a(k)\) and \(N_f(k)\). \(S_{RS}\) is the size of the reservation stations. As above, \(m\) and \(n\) are the number of ALUs and FPUs, respectively, \(\alpha\) is the number of \(xALUs\) per FPU and \(r(k)\) is the number of FPUs reconfigured as \(xALUs\) at time \(k\).

\[
T_a(k) = m + \alpha \cdot r(k - 1) \tag{5.9}
\]

\[
T_f(k) = n \quad \forall k \tag{5.10}
\]

\[
A_n(k) = \left\lfloor 2^3 \cdot \frac{A(k)}{(S_{RS})} \right\rfloor \tag{5.11}
\]

\[
F_n(k) = \left\lfloor 2^4 \cdot \frac{F(k)}{(S_{RS})} \right\rfloor \tag{5.12}
\]

with \(\{0 \leq r(k) \leq n \quad \forall k\}\)

\(^4\)anything up to .499 is rounded down, 0.500 and above is rounded up.
CHAPTER 5. CASE STUDY

It is then possible to perform comparisons between the normalized values and the thresholds for both integer and floating point instructions to make a decision.

\[ r(k) = \begin{cases} 
  r(k-1) & \text{if } N_a(k) \leq T_a(k) \text{ and } N_f(k) \leq T_f(k) \\
  r(k-1) + 1 & \text{if } N_a(k) > T_a(k) \text{ and } N_f(k) \leq T_f(k) \\
  r(k-1) - 1 & \text{if } N_a(k) \leq T_a(k) \text{ and } N_f(k) > T_f(k) \\
  n - 1 & \text{if } N_f(k) > 0 \text{ and } n = r(k-1) 
\end{cases} \] (5.13)

The last option is necessary to handle cases where many integer instructions are dependent on very few floating point instructions. Without this check, the threshold to switch an FPU back to floating-point operation would never be reached, and the FP instructions would stall the processor forever. This will be called forced reconfiguration. As all the sums can only take a very limited number of values, and the two normalized values are obtained by constant shifts, a very simple and fast implementation can be designed (figure 5.11).
5.3. SCHEDULING OF RECONFIGURATION

The algorithms balance and threshold have very different complexities, and thus only the latter will be considered for hardware implementation. The difference in the performance of these algorithms is detailed in section 5.3.6.

5.3.5 Trace Results

Due to the very large mathematical complexity of the optimal solution and the necessity of knowing all the arrival times in advance, comparisons with this solution can only be performed on very short traces. This section will present the differences between the Integer Linear Programming solution and both threshold and balance algorithms to quantify the reduction in performance due to sub-optimal reconfiguration. The balance algorithm is included as it will be used to qualify the performance of the threshold algorithm over the full benchmark suite in section 5.3.6. In these traces, threshold and balance produce the exact same results, so only the results for threshold are shown. The scheduling of the baseline case will also be shown as reference.

For the threshold algorithm, the traces are simulated in an implementation of the dynamic non-linear problem described by equations (5.3) to (5.6) and figure 5.10 implemented in C++. In the case of the optimal integer programming solution, the solution and end times are directly given by the CPLEX solution. In this case, the trace must first be converted to
CHAPTER 5. CASE STUDY

A A A F xA xA xA F xA xA xA Finish Time

<table>
<thead>
<tr>
<th>t</th>
<th>u0</th>
<th>u1</th>
<th>u2</th>
<th>u3</th>
<th>u4</th>
<th>u5</th>
<th>u6</th>
<th>u7</th>
<th>u8</th>
<th>u9</th>
<th>u10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>12</td>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>14</td>
<td>15</td>
<td>16</td>
<td>17</td>
<td>18</td>
<td>19</td>
<td>20</td>
<td>21</td>
<td>22</td>
<td>t0</td>
<td>t1, t11, t12, t13</td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>24</td>
<td>25</td>
<td>26</td>
<td>27</td>
<td>28</td>
<td>29</td>
<td>30</td>
<td></td>
<td>t14, t15, t16</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finish Time

Figure 5.13: Detailed instruction scheduling for instruction arrivals in the dynamic model using the threshold algorithm. As the instructions 5 to 30 arrive at time 1, the reconfiguration decision is to execute one FP instruction on each FPU (instructions 0 and 1 on u3 and u7). The many integer instructions must then wait until the FPUs are idle before being executed by the xALUs, needing a total of 8 cycles to execute the 31 instructions, giving an IPC of 3.88. The arrows show the latency of the functional units.

A A A F xA xA xA F xA xA xA Finish Time

<table>
<thead>
<tr>
<th>t</th>
<th>u0</th>
<th>u1</th>
<th>u2</th>
<th>u3</th>
<th>u4</th>
<th>u5</th>
<th>u6</th>
<th>u7</th>
<th>u8</th>
<th>u9</th>
<th>u10</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15</td>
<td>6</td>
<td>30</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>28</td>
<td>13</td>
<td>21</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>11</td>
<td>18</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>29</td>
<td>19</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>23</td>
<td>26</td>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Finish Time

Figure 5.14: Detailed instruction scheduling for instruction arrivals in the dynamic model using the optimal algorithm. The FP instructions are scheduled one after the other on the same FPU (u3), and the other FPU is reconfigured to help execute the many integer instructions arriving at time 1. The 31 instructions take 7 cycles to execute, for an IPC of 4.43. The arrows show the latency of the functional units.
5.3. SCHEDULING OF RECONFIGURATION

Figure 5.15: Instruction scheduling for instruction dependencies in the baseline model. There is no reconfiguration, and instructions are executed in their order of arrival. The dependency between instructions 3 and 8 has no effect on the total time needed, 4 cycles, resulting in an IPC of 2.25. The black arrow shows the latency of the functional units, while the gray dotted arrow shows the dependency.

Figure 5.16: Instruction scheduling for instruction dependencies in the dynamic model with the threshold algorithm. As instructions are allocated to the functional units in their order of arrival starting with the ALUs, instruction 3 is assigned to an xALU (u4). Thus, dependent instruction 8 must wait for 1 cycle before being executed, for a total time of 4 cycles. The resulting IPC is 2.25. The black arrows show the latency of the functional units, while the gray dotted arrow shows the dependency.
A linear program in a format acceptable to CPLEX. A complete short example with a few instructions, including the linear program and output, is presented in appendix B. It shows the output of all the variables in the dynamic simulation, and the linear program and resulting output and optimal solution for the integer linear programming approach.

To emphasize the difference in performance between the results of the threshold algorithm and the optimal solution, two cases have been considered:

- The first example deals with instruction arrivals that induce the threshold algorithm to make an unwise decision about reconfiguration and scheduling. At time 0, 3 integer and 2 FP instructions arrive. Then, at time 1, another 26 integer instructions arrive, for a total of 31 instructions. The scheduling of the baseline processor, shown in figure 5.12, is straightforward: all the integer instructions are executed by the ALUs, and the 2 FP instructions are executed at cycle 0 by the FPs. 11 cycles are needed to execute all the instructions, giving an overall IPC of 2.82.

When the threshold algorithm is used in the dynamic model, the initial scheduling will be as in the baseline case, with the FPs each executing one of the FP instructions (figure 5.13). This decision impedes the reconfiguration of the FPs when the other integer instructions arrive, and they can only be reconfigured at cycle 4, where the xALUs start executing the remaining integer instructions, finishing in 8 cycles, resulting in an IPC of 3.88, an increase of 38% over the baseline case.

Finally, applying the optimal solution to the dynamic model gives the schedule shown in figure 5.14, where one FPU executes both FP instructions in 2 consecutive cycles, while the other is reconfigured to start executing the many integer instructions. With this method, the
5.3. SCHEDULING OF RECONFIGURATION

The second example concerns the dependencies between instructions: 8 independent integer instructions arrive at time 0, with instruction 8, dependent on instruction 3, arriving at time 1, for a total of 9 instructions. The baseline model will again schedule these instructions in order, without encountering any problems due to the dependency, as shown in figure 5.15. All 9 instructions will be executed in 4 cycles, resulting in an IPC of 2.25.

The dynamic model using the threshold algorithm will reconfigure both FPUs as xALUs and issue all the instructions available at time 0, as shown in figure 5.16. However, as instruction 3 has been issued to an xALU, instruction 8 cannot issue at time 1, since instruction 3 has not finished execution due to the xALU’s latency of 2 cycles. Instruction 8 executes at time 2, so 4 cycles are needed, giving the same IPC as the baseline model, 2.25.

Using the optimal scheduling for the dynamic model, instruction 3 will be issued to a normal ALU, as shown in figure 5.17. Instruction 3 will thus be completed at time 1, allowing instruction 8 to finish at time 2, for a total of 3 cycles. This gives an IPC of 3.0, an increase of 33% over both the baseline and the threshold results.

There are differences of up to 33% between the implemented threshold algorithm and the optimal solution given by the integer linear program in these hand-crafted cases. The difference in longer benchmarks should be far less than this value, as this special case will not occur every cycle. Indeed, somewhat longer cases with up to 100 instructions showed differences of less than 10%.

5.3.6 Simulation Results

This section presents the results of simulations for complete benchmarks with the balance and threshold algorithms. Descriptions of the benchmarks can be found in section 6.1.2. The comparisons show the small difference in performance between the two algorithms, thus validating the threshold decision algorithm for all further simulations due to its simplicity. Figure 5.18 shows very small differences in performance for the benchmarks where the results for balance were better than for threshold. These differences are of less than 1%, with an average gain of only 0.14%. Note that not all benchmarks are represented: indeed, although the balance algorithm produces better results in many cases, it is slow to adapt to rapidly changing patterns in instruction type usage, and thus will give worse results than...
threshold for some benchmarks that exhibit these quick changes. In such cases, the comparison is meaningless, as we seek to quantify the upper bound in performance.

The difference in complexity between the balance and threshold algorithms shows the variety of approaches that may be designed to control the reconfigurability. Of course, any increase in reconfigurability would probably be accompanied by a corresponding increase in the complexity of the decision algorithm. It is worth noting that the different approaches followed produce results that depend on the switching activity of the applications, and thus a better algorithm might combine the threshold and balance algorithms to attain greater performance. A greater number of functional units should favor the balance algorithm, due to less rounding errors.

5.3.7 Complexity-Performance Trade-off

The results of the three reconfiguration decision algorithms exposed in the previous sections show that a complexity-performance trade-off appears. However, a highly complex algorithm would increase the area and power consumption of a chip, making this limited reconfigurability less interesting.
5.4. DETAILED DESIGN

Luckily, the performance varies slowly for differing complexities, moving the optimal trade-off toward simpler algorithms. This also expresses the well-known law of diminishing returns in computer architecture. In addition, as section 6.6.4 will show, there is some margin for making decisions about reconfiguration, thus giving some freedom in the choice of the algorithm.

It is impossible to implement an optimal algorithm, since the future arrival times are not known. However, the sub-optimal algorithms give good results close to the optimum. A complex approach is able to produce the optimal solution for a very small search space, but this model is completely problem specific.

5.3.8 Conclusion

The decision algorithm has an important impact on the feasibility of the application of limited reconfigurability in a superscalar processor. The problem of choosing the optimal configuration at run-time is a very complex one, due to the many dependencies and limits that must be considered. Complexity and performance must also be balanced to obtain an overall gain. This decision might also be enhanced through the use of compiler 'hints' inserted in the code, although this would break binary compatibility, an important feature of this application of limited reconfigurability. The threshold algorithm will be used for all the results presented in chapter 6.

5.4 Detailed Design

This section will detail the modifications that must be made to a superscalar processor to enable the limited reconfiguration presented in section 5.1. The multiplier tree and the threshold decision algorithm have been implemented in Very large scale integration Hardware Description Language (VHDL) and synthesized in UMC 0.18µm technology [153]. The aim is to measure all the delays that might be introduced and take all details into account. The wiring that must be added will also be discussed.

5.4.1 Internal Routing

Several modifications to the paths in the execution core are necessary to make the limited reconfiguration work:

- Extra paths to and from the Reservation Stations and the Register File are necessary to feed the $z$ALUs and get the results in return.

- Extra forwarding paths must be added to connect the $z$ALUs to the standard ALUs. The paths in between the $z$ALUs and to the FPUs are very short and should have no impact.
• Finally, several multiplexors to select the correct operands for all these new forwarding paths are needed. The multiplexors added to the multiplier will be discussed in section 5.4.3.

Due to the large complexity in designing a complete superscalar core, the delays associated with the extra wires above cannot be accurately measured. However, considering that the latency of a complete ALU is a single cycle, the latency of our xALUs will be lower than this as they are less complex. In deep sub-micron technology, such as 0.13\(\mu\)m, wires account for about 2/3 of the delays, and the differences between 0.13\(\mu\)m and 0.09\(\mu\)m are not so important in this regard. The increase in wiring to reach the xALUs is estimated at about double that needed for normal ALUs. Thus, if a normal ALU has a latency of 1 cycle, split as 1/3 gates and 2/3 wires, doubling the wires gives a xALU latency of 5/3. Taking the multiplexers to select the adders in the FPU into account, a conservative estimate for the latency of all extra ALU units is to double the latency of normal ALUs, for a latency of 2. As confirming this timing would require designing the entire functional core of a superscalar processor, a complex task beyond our means, simulations with a very conservative latency of 3, where about 89% of the xALU delay is in the wires, have also been performed. Additionally, some of the bypass paths necessary to keep the pipeline as full as possible, and counted in the above calculations, are likely to already be present in the multiplier’s tree linking the xALUs together. This also means that the overhead is less than that of simply adding ALUs to the processor.

As will be shown in section 6.6.4, the estimate of this latency is important, but not critical to the gains of this example of limited reconfiguration. On the other hand, the latency of the FPU is not affected by any extra wiring.

Reservation Stations, Reorder Buffer and Scheduler

In the case where there is only one reservation station for integer instructions and one reservation station for FP instructions, no extra complexity, except for that necessary to support a possibly larger pipeline width, arises from the addition of reconfigurability (figure 5.19 (left)). However, should distributed reservation stations be used to increase the clock rate by decreasing the distances over which the data must travel, as shown in figure 5.19 (right), the issue of deciding to which reservation station to send integer instructions must be handled.

If the FPU is almost always reconfigured as xALUs, then a simple round-robin scheduling is fine, as the throughput of the ALUs and xALUs is the same. In the case of frequent reconfiguration or long-term FP usage, however, we must limit the amount of instructions sent to the xALU reservation stations, and also prevent reconfiguration while these reservation stations
Figure 5.19: Unified reservation stations (left) and distributed reservation stations (right). The unified case is simpler, but the data must travel a greater length before being executed (path in bold). In the distributed case, the larger distance is travelled when going to the reservation station, which is only critical if the reservation station is empty.

are not empty. As almost all benchmarks show either a clear tendency toward one type of instructions or separate phases using one or the other, this limitation has little impact on the results.

5.4.2 Threshold Decision Algorithm

The proposed implementation for the threshold algorithm is quite simple, as shown in figure 5.11, since the additions in the equations can be reduced to multiplexors. The implementation in UMC 0.18µm technology reports a delay of 0.3ns and an area of 500µm$^2$, and the corresponding schematics and reports can be found in appendix A. The measured delay gives a clock rate of 3.3 GHz, meaning it would take at most 2 cycles at the clock rate of the fastest superscalar processors available, which are implemented in 0.13µm and 90nm technology, and a single cycle in any slower processor. This decision algorithm will be used for all the simulations presented in chapter 6. The results of a quantitative comparison of the different decision algorithms were shown in section 5.3.6.

5.4.3 Multiplier Tree Design

The objective is to reconfigure some of the FPU’s multiplier logic as arithmetic units. The multiplier in figure 5.6 shows a tree of CSAs followed by a single CPA to produce the result of the multiplication. However, the CSAs cannot be used to perform a single addition, since they do not propagate
Figure 5.20: Balanced Tree where 3 CSAs have been replaced by CPAs. The extra delay of the CPAs increases the total delay by 4 cycles—i.e., the difference in delay between a CSA and a CPA.
the carry bits to gain speed. Thus, to be able to reconfigure some of the
adders in the multiplier tree as individual adders, CPAs must be used. As
stated in section 5.2.4, a 64-bit CPA is about five times slower than a $n$-bit
CSA. Thus, simply replacing a number of CSAs in the tree by CPAs and
adapting the wiring would greatly increase the delay of the multiplier, from
$15\tau$ to $19\tau$, or 27%, based on the assumptions presented in section 5.2.4, as
shown in figure 5.20.

**Fully Unbalanced Tree**

To reduce the added delay as much as possible, the compressor tree can be
unbalanced to give the CPAs more time to finish their calculations. Such
a tree, shown in figure 5.22, shows a delay of only $17\tau$. The CPAs and
multiplexors are no longer in the critical path of the multiplier, as confirmed
by the measures on the synthesized tree shown in appendix A. The final
adder at the bottom of the tree could also be used for this purpose, but as
this would add a multiplexor to the critical path, the final CPA will not be
used for the xALUs. Should a tree of 4-to-2 compressors be desired, they
can be built from CPAs, as shown in figure 5.21, and eventually further
optimized (see section 5.2.4).

Using estimated timing methods, the delay of the unbalanced tree with
CPAs is only an increase of 13% compared to the original design. To validate
this result, the VHDL code for both the balanced and unbalanced trees was
written and synthesized in UMC 0.18$\mu$m technology. Both trees have been
obtained with an algorithm similar to the Three-Greedy Approach [84]. The
experimental value for the ratio $\text{Delay}_{CPA}/\text{Delay}_{CSA}$ is 5.4, quite close to
the theoretical value of 5. The reported delay increases by only 4.6%, and
is due to the two extra levels of CSAs needed to compress the CPA’s results
into the final multiplication result. An area increase for the tree of 11% is
due to the far larger size of fast CPAs as compared to CSAs. As the latency
of any functional unit must be an integral number of cycles, and we make
the hypothesis that there is no slack in the critical path of the multiplier,
the latency of the FPU must thus be increased by one cycle with this design.
Figure 5.22: Unbalanced tree with 3 CPAs and the corresponding multiplexors inserted. Estimated delay is $17\tau$, with the critical path going through the CSA tree.
Although power consumption is not a primary constraint of this application, it is still interesting to avoid increasing power consumption. The factors contributing to power consumption are the extra wires to and from the reservation stations, the decision algorithm, and the modifications to the multiplier, including multiplexors. The power of the extra wires is difficult to measure accurately, but should not be significant compared to the size of the entire processor. The power consumption of the implementation in 0.18 µm of the threshold decision algorithm shown in figure 5.11 is only 850 µW as reported by the synthesis tools. Likewise, the power consumption of the multiplier tree increases by 32 mW, or about 1%. Overall, the power consumption can thus be considered to be unaffected by the reconfigurability introduced.

**Optimally Unbalanced Tree**

Contrasting with the approach described in the previous section, it is possible, within certain constraints, to build an unbalanced tree with a number of CPAs and no increase in the total depth, and thus delay, of this tree. Figure 5.23 shows the original balanced tree and the fully unbalanced tree, used as starting points for the construction of an optimal tree. The main parameter controlling the shape and size of the resulting tree is the ratio of the latencies of a CPA and a CSA, \( \frac{L_{CPA}}{L_{CSA}} \), experimentally seen to be equal to about 5.4 in the technology considered. The original tree has 15 stages, whereas the fully unbalanced tree has 17 stages, as previously shown in figures 5.6 and 5.22, respectively.

Figure 5.24 shows the case of an unbalanced tree with 3 CPAs of latencies of 5 and 6 cycles (top and bottom resp.). In both cases, the total number of stages in the tree is the same as the original balanced version, 15 stages. Increasing the latency of the CPAs to 7 and 8 cycles produces the trees in figure 5.25, where the total delay is increased by one stage, for a total of 16.

Note that our fully unbalanced tree is equivalent to assuming a \( \frac{L_{CPA}}{L_{CSA}} \) ratio of 9, which is a gross overestimate. Thus, the extra delay in the tree due to adding some CPAs can be reduced to zero, resulting in an FPU that suffers no ill effects from the possibility of reconfiguration even if an application is unable to take advantage of it.

To formalize this, we can derive, based on the recursive formulas for the height of a compressor tree with \( n \) entries and the maximum number of entries for a tree of height \( h \) from [70], shown in section 5.2.4 and repeated in equations (5.14) and (5.15).

\[
H(n) = 1 + H(\lceil 2n/3 \rceil) \tag{5.14}
\]

\[
N(h) = \lfloor 3N(h-1)/2 \rfloor \tag{5.15}
\]

First, the number of Stages in a tree with \( m \) CPAs of latency \( L_{CPA} \) is
given by equation (5.16):

\[ H_{CPAs}(n) = L_{CPA} + H(n' + m/2) \]  

(5.16)

where \( n' \) is equal to the number of inputs left after compressing \( n \) inputs through \( L_{CPA} \) stages, i.e.:

\[ n' = n \]

\[ for(i = 0; \ i < L_{CPA}; \ i + +) \]

\[ n' = \left\lceil 2n'/3 \right\rceil \]  

(5.17)

The limit on the number of CPAs of latency \( L_{CPA} \) can now be derived. Let

- \( n_b \) be the number of inputs for the original balanced tree.
- \( n_u \) be the number of inputs for the unbalanced tree.
- \( 2m \) be the number of inputs that go to the CPAs of latency \( L_{CPA} \). We thus have \( m \) CPAs.

Thus, we have that \( n_b = n_u + 2m \). Define \( n'_u \) as the number of inputs after \( L_{CPA} \) stages, using equation (5.17). The number of inputs that can be added at this stage without increasing the total depth of the tree is:

\[ Slack = N(H(n_u) - L_{CPA}) - n'_u \]  

(5.18)

Finally, we assert that the number of extra inputs, the results of the \( m \) CPAs, must be less than or equal to the slack available:

\[ m \leq N(H(n_u) - L_{CPA}) - n'_u \]  

(5.19)

Applying equation (5.16) to a 64-bit tree where 3 CPAs of latency 6 have been added produces a total number of stages equal to 10, identical to that of the balanced tree. A 64-input tree has maximum slack, as a 63-input tree would have only 9 levels of CSAs. Equation (5.18) further states that up to 4 CPAs of latency 5 can be added to the tree without increasing its delay, but only 3 CPAs of latency 6 and 2 CPAs of latency 7 can be added without increasing the tree depth. These results are compatible with the constructions of figures 5.24, 5.25 and 5.26.

The results of the synthesis of this optimally balanced tree, shown in detail in section A.1 of appendix A, confirm that the delay of the multiplier need not increase when equation (5.19) holds. The measured increase in delay is 0.01ns, equal to 0.24%, and well within the error margins of the tools used. The area increases by 7.5%, while the power consumption increases by 3.8%.
5.4. DETAILED DESIGN

Using value from a previous stage
Unused at this stage
Using value from a previous stage

Total of (10 + 5) = 15 cycles
Total of (12 + 5) = 17 cycles

Figure 5.23: Full 64-input compressor trees for the unmodified balanced case (top), and the fully unbalanced case using 3 CPAs described at the beginning of section 5.4.3, equivalent to having \( L_{CPA} \leq 9 \) (bottom). The difference in delay is due to the stages needed to assimilate the results of the CPAs.
Total of (10 + 5) = 15 cycles

Figure 5.24: Full 64-input compressor trees for the CSA and CPA, using value from a previous stage or unused at this stage, with 15 cycles.

In both cases, the total delay of the tree is the same as for the balanced tree in Figure 5.23 (top).
Figure 5.25: Full 64-input compressor trees for the partially unbalanced case using 3 CPAs when $6 \leq \frac{L_{CSA}}{2} \leq 7$ (top), and when $7 \leq \frac{L_{CSA}}{2} \leq 8$ (bottom). In these cases, the total delay of the tree is one stage more than for the balanced case in figure 5.23 (top), but still better than the fully unbalanced tree in figure 5.23 (bottom).
5.23 It shows that the total delay of 6 cycles means the tree requires an extra level of CSAs (bottom).

However, increasing the latency to 6 cycles means the tree requires an extra level of CSAs (bottom). With a latency of 6 cycles, the total delay of the tree is the same as for the balanced tree in Figure 5.26: Full 64-input compressor trees for the balanced tree in figure.

5.23 It shows that the total delay of 6 cycles means the tree requires an extra level of CSAs (bottom). With a latency of 6 cycles, the total delay of the tree is the same as for the balanced tree in Figure 5.26: Full 64-input compressor trees for the balanced tree in figure.

Figure 5.26: Full 64-input compressor trees for the balanced tree in figure.

Using value from a previous stage
Unused at this stage
Unused at this stage
Unused at this stage

Figure 5.26: Full 64-input compressor trees for the balanced tree in figure.
5.4.4 Timing Parameters for Simulation

The synthesis results of the previous sections provide a set of timing parameters for the architectural simulations that provide quantitative results about this case study.

Concerning the reconfigurable FPU, two designs will be considered:

- An FPU which can be reconfigured as 4 xALUs, but with a higher latency than an unmodified FPU. This design shows the best gains that can be attained by limited reconfiguration, although it will produce some losses in benchmarks which make heavy use of the multiplier. This is the design shown in figure 5.26 (bottom).

- An FPU which can only be reconfigured as 3xALUs, but whose latency does not increase compared to the non-reconfigurable FPU. In this case, the gains will be slightly more modest, but no losses should appear. This design is also a bit more realistic from the point of view of wire complexity, and is shown in figure 5.24 (bottom).

The xALUs will always have a latency of 2, due to the longer wires needed to reach them and get the results back. However, the case of a processor with no normal ALUs, where the integer reservation stations and reorder buffer are placed close to the FPU, thus providing a latency of 1 cycle for xALUs, will also be considered.

The high-level schematics for the designs and the full area and timing reports are in appendix A.

5.5 Conclusions

This chapter has shown that adding limited reconfigurability to a superscalar processor's FPU is a feasible task with little cost, although it requires a careful study of several parts of the design. This careful study has also produced several timing results for different designs that will be used for all the simulations to be presented in chapter 6. The complexity of the decision algorithm and the different possibilities to solve it have been presented.

As with all processor architecture research, adding limited reconfigurability might make other architectural options more or less interesting. As we are greatly increasing the integer processing resources available, predicate execution might prove more interesting than, or at least complementary to speculative execution. Likewise, a multithreaded (SMT) processor might make more parallelism visible to the processor, and thus make better use of the extra functional units.

Finally, it should be noted that the design for the reconfigurable FPU detailed in this chapter is not the most aggressive design possible: it would be possible to reconfigure the FPU in floating-point mode while we are
still executing the second stage of the $xALUs$, since the first cycle of the FPU does not use the multiplier. In the same vein, a reconfiguration into $xALUs$ should be possible while the FPU is in the 3rd and 4th stages, as the CPAs are no longer in use in these stages. While this would reduce the reconfiguration delay and thus increase the performance slightly, it would require far more control logic than the case presented, which would probably eliminate any gains obtained. It would also be possible to use the FPU for floating point add/subtract while in $xALU$ configuration, as the final CPA in the multiplier tree, used for these operations, is not reconfigured as an $xALU$. However, this would greatly increase the complexity of the scheduler and the forwarding paths.
Chapter 6

Results

This chapter will present the tools, methodology and results of the limited reconfigurability application detailed in chapter 5. Several processor models have been considered, with varying parameters to show the broad applicability of the reconfiguration.

The results are obtained through the use of a detailed superscalar processor simulator for the hardware side, with a broad set of benchmarks to measure the performance of the modifications.

The results will be analyzed through the different benchmarks in the suite, and a sensitivity analysis will also be performed. Finally, the limitations of adding limited reconfigurability to a superscalar processor’s FPU will be discussed.

6.1 Methodology

This section will present the procedures followed to obtain quantitative results for the addition of limited reconfigurability to the FPU of a superscalar processor. Both the hardware and software aspects will be covered.

The simulation of the superscalar, out-of-order processor is based on the Simplescalar Toolset [11], which contains a detailed timing simulator of a pipelined, superscalar processor. The performance of the reconfigurability was obtained by using standard benchmarks for measuring a processors’ performance for a variety of tasks.

6.1.1 Simplescalar

On the hardware side, the Simplescalar tool set will be used. This tool set simulates a detailed out-of-order superscalar processor and is widely used for research in processor architecture, providing a wealth of configuration parameters and statistics for the simulations. As it is distributed in source
code form, it is possible to modify the simulator to validate new architectural ideas.

The Simplescalar Toolset is a collection of programs that simulate the different parts of a processor to varying degrees of detail and precision. It is widely used in processor architecture research, although its results show some quantifiable difference with hardware comparable to that simulated, due to some modelling errors and omissions, and the fact that it does not directly model a specific processor [16].

The detailed out-of-order engine, sim-outorder, was modified to implement the functions detailed in the following sections, and add many statistics on functional unit usage.

Sim-outorder implements a full superscalar processor pipeline, including cache and memory accesses. I/O operations are emulated through system calls, allowing almost any program to be executed. It also contains a fast simulation engine to allow the speedy execution of a number of instructions without simulating a detailed pipeline, before starting the detailed simulation. Sim-outorder processes the logical stages of a processor pipeline in reverse order, thus needing only a single pass for each cycle. It is written in the C programming language [39], and compiles on a variety of architectures, in both little and big endian\(^1\) modes. Cross-endian support is included, but was not used in this research, as both our host and the simulated target were little-endian.

The tool set contains well-written options and statistics management packages, allowing additions with very little work. To support our research, the following options were added:

- **-res:ialu** controls the number of ALUs present in the processor.
- **-res:memport** sets the number of memory ports (Load/Store Units).
- **-res:gpfpu** controls the number of General Purpose Floating Point Units. Each of these performs all FP operations in addition to integer multiplication.
- **-res:xialu** determines the presence of static xALUs, which are identical to ALUs except for the latency, which can be defined below.
- **-use_dyn_fu** toggles the activation of dynamic reconfiguration.
- **-res:dyfu_factor** sets the number of xALUs obtained for every FPU reconfigured.
- **-xialu:lat** controls the operation and issue latencies for the xALUs.

\(^1\) Endian-ness refers to the order in which bits or bytes are stored and transmitted. Little endian means the least significant bit is stored or transmitted first.
-**gpfpu:** controls the operation and issue latencies for the FPU for instructions that do not use the parallel multiplier.

-**gpfpu:** controls the operation and issue latencies for the FPU for instructions that make use of the reconfigurable multiplier hardware.

-**dyn_fu:** adds a reconfiguration latency in addition to the inherent delay until the FU is idle.

-**do_instr_dump** directs the simulator to trace the number and type of instructions executed.

-**instr_dump:** controls the number of cycles between each trace dump.

Additionally, a number of statistics to guide the research were added:

- The total number of instructions not issued due to busy functional units. This provides a rough measure of the upper limit to the gain that might be achieved under specific simulation constraints.

- The maximum number of instructions not issued due to busy functional units in a single cycle.

- The number of instructions of each type not issued due to busy units.

- The number of instructions of each type committed.

Finally, some statistics about the dynamic reconfiguration itself are collected:

- The number of reconfiguration decisions made by the algorithm.

- The number of decisions actually enforced, after considering functional unit occupation.

- The number of *forced reconfigurations* due to instructions that could not be executed in the current configuration (see equation (5.13)).

- The average number of cycles between reconfigurations.

- The number of stalls for each FU type that could have been avoided with a better reconfiguration algorithm. This measure ignores past decisions, so this number cannot always be reduced to zero, even with a perfect decision algorithm.

- The maximum and average number of FUs of each type in use in a single cycle.
In addition to the options and statistics added above, the following changes and additions were performed to implement the dynamic reconfiguration and decision algorithm functionality:

**sim-outorder.c** The definition of the resources and calls to the dynamic reconfiguration simulation code were added.

**sim-outorder.h** This file contains some definitions that were previously in `sim-outorder.c` that are also needed by the reconfiguration code.

**resource.c/h** These files contain the functions related to the resources—i.e., the functional units. Functions to *free* a resource set and *copy the status of a set* of functional units were added. A check on *unexecutable instructions* was removed, since this case may now appear for a few cycles in very asymmetric workloads until a *forced reconfiguration* is triggered.

**dyn_fu.c/h** contains all the code related to the dynamic reconfiguration, including the decision algorithms.

The Simplescalar Tool Set can be targeted at several different ISAs, the two main ones being PISA\(^2\), a research-oriented ISA invented by the author of Simplescalar, and the DEC\(^3\) Alpha [18] processor ISA. The latter was selected for all tests. It has been implemented in several different models and held the processor performance crown for many years [146]. Ports of Simplescalar for the PowerPC [119, 124] and ARM [113] processors are also available from third-party sources.

### 6.1.2 SPEC CPU2000 Benchmarks

In order to compare different processors, a number of metrics may be considered, the most common being performance, area—roughly equal to cost, and power consumption. The latter two are fixed values for a given processor, although power consumption can vary with the workload. However, the performance of a processor is not so simple to compare. The clock speed can give an idea of performance, but can be very misleading: a very fast single issue processor might be outdone on many tasks by a far slower, but wider, superscalar or VLIW processor. The essence of the problem lies in the choice of tasks considered: a processor does not have a performance on its own, it has a measured performance for a specific application.

The choice of the application or, most often, applications used to measure performance can greatly impact the results; running a simple single-threaded application on a supercomputer will give results no better, and

\(^2\)Portable ISA

6.1. METHODOLOGY

probably worse, than running this same application on a desktop computer. However, the same comparison using a complex scientific calculation will show orders of magnitude of difference. To solve this problem, a standard set of applications, called benchmarks, have been defined. These benchmarks attempt to represent a significant portion of the real-world applications which the processor might be required to execute, thus providing meaningful comparisons.

As we are aiming to improve a very large range of applications, we will use the Standard Performance Evaluation Corporation (SPEC) CPU 2000 benchmark suite [29, 144]. This suite contains 26 benchmarks, representative of real-world tasks requiring a high performance processor. SPEC are the most widely used benchmarks to measure the performance of a single thread in a single processor for compute-intensive applications, which are the focus of this case study. The pre-compiled SPEC benchmarks were obtained from the Simplescalar WWW site [147], and were compiled with the peak configuration. This means that the compiler used the best options for each specific benchmark when compiling [148].

There are other benchmark suites that are widely used, even for general purpose processors, although they have an orientation to a particular application domain, such as EEMBC [117] for embedded processors and OLTP [143] for online and database applications.

SPEC CPU2000 contains 14 integer benchmarks and 16 floating-point benchmarks, each with different characteristics in terms of functional unit, memory, cache or scheduling requirements, stressing as many of the parts of a processor as possible. Results for each benchmark will be presented, with the overall average result being our main comparison metric, although variations in individual benchmarks will also be discussed when appropriate. The integer and floating point designations indicate the general tendency of a benchmark; indeed, a few integer benchmarks use floating point instructions, and the percentage of FP instructions in the floating point benchmarks can vary from 16% to 64% of all instructions.

The benchmarks in SPEC CPU are designed to be run on real hardware, where the entire suite takes less than 24 hours on a recent processor. However, as the fastest detailed software simulations are between 16000 and 1.5 million times slower than the fastest real hardware, depending on the application, a single complete simulation run would take several months, even when distributed on a large number of processors. Several methods to reduce this time exist, such as using smaller data sets [42], or simulating only the most representative parts of each benchmark, either by skipping a number of cycles before starting the detailed simulation (which will be called skip) or through an analysis tool called SimPoint. The two latter methods have been used to obtain the results starting in section 6.2.

Simpoint is a tool that analyzes the behavior of a specific program on a specific architecture. The analysis then provides information about the most
CHAPTER 6. RESULTS

significant parts of the program. This allows the compilation of statistically valid simulation points [79], that can be used to estimate the behavior of an entire benchmark without having to simulate it from beginning to end. At the cost of some precision, an ever shorter, but still representative, simulation run can be obtained by using the first significant simulation point within an error margin. A variant, making for slightly longer simulations but with less error, uses the single most significant simulation point, called single standard simulation points by the authors [73] for all the benchmarks, and referenced as Simpoints hereafter. This approach is used for the main results in this thesis. Each complete run of the 26 SPEC benchmarks takes about 3 weeks on a 2.8GHz Pentium 4 processor.

The difference in results between the two approaches, the faster skip and the more accurate Simpoints, will be examined for the cases where the Simpoints approach was used to assess the validity of the sensitivity analysis.

The benchmarks in the SPEC CPU 2000 suite are briefly presented below, beginning with the integer benchmarks:

- **gzip** is a popular data compression program which uses Lempel-Ziv coding (LZ77) as its compression algorithm [122].
- **vpr** performs placement and routing in Field-Programmable Gate Arrays.
- **gcc** is a C language compiler based on gcc Version 2.7.2.2. It generates code for a Motorola 88100 processor.
- **mcf** is derived from a program used for single-depot vehicle scheduling in public mass transportation.
- **crafty** is a high-performance computer chess program.
- **parser** is a syntactic parser of English language text.
- **eon** is a probabilistic ray tracer based on Kajiya’s 1986 ACM SIGGRAPH conference paper [37].
- **perlbmk** is a cut-down version of Perl v5.005.03, a scripting language.
- **gap** implements a language and library designed mostly for computing in groups.
- **vortex** is a single-user object-oriented database transaction benchmark.
- **bzip2** is based on bzip2 version 0.1.
- **twolf** is a placement and global routing package for standard cells.

The floating point benchmarks are:
6.1. METHODOLOGY

wupwise is an acronym for "Wuppertal Wilson Fermion Solver", a program in the area of lattice gauge theory (quantum chromodynamics).

swim is a weather prediction program, originally intended for use on supercomputers only [77].

mgrid is a very simple multigrid solver computing a three dimensional potential field.

applu performs calculations related to the Computational Fluid Dynamics and Computational Physics fields.

mesa is a free OpenGL work-alike library.

galgel is devoted to numerical analysis of oscillatory instability of convection in fluids.

art is a neural network used to recognize objects in a thermal image.

equate simulates the propagation of elastic waves in large, highly heterogeneous valleys.

facerec is an implementation of a face recognition system [47].

ammp runs molecular dynamics on a protein-inhibitor complex.

lucas performs the Lucas-Lehmer test to check primality of Mersenne numbers.

fma3d is a finite element method to simulate the response of 3-dimensional structures subjected to sudden loads.

sixtrack simulates particles in a model of a particle accelerator to check the long term stability of the beam.

apsi performs complex weather prediction.

The instruction type distribution of these benchmarks for the Simpoints, and thus representative of the entire benchmarks, are summarized in table 6.1. Note that the percentage of floating point instructions varies greatly from one benchmark to another.

6.1.3 Processor Models

Given the huge variety of existing general-purpose processors today, a number of models, representative of the different sizes of processors, are needed. Our main reference, called the mainstream model and based on a current desktop processor, will be described below. To this, we will add a top model, describing a non-existent processor with more resources than are currently
### Table 6.1: Instruction Type Distributions in Percent of All Instructions excluding Memory Accesses for the 26 Benchmarks of the SPEC CPU 2000 Suite from the Simpoints.

All numbers are in percent of total instructions. The top 12 are classified as integer benchmarks, with the lower 14 being floating-point. From left to right, the instruction types are ALU, integer divide, integer multiply, FP add/subtract, FP compare, FP convert, FP multiply, FP divide, and FP square root. Only values above 0.1% are shown.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>ALU</th>
<th>Idiv</th>
<th>Imul</th>
<th>Fadd</th>
<th>Fcmp</th>
<th>Fcvt</th>
<th>Fmul</th>
<th>Fdiv</th>
<th>Fsqrt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apss</td>
<td>68.7</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sixtrack</td>
<td>35.8</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gzip</td>
<td>100</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Vpr</td>
<td>89.3</td>
<td>9</td>
<td>1.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gcc</td>
<td>99</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mcf</td>
<td>99.9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Crafty</td>
<td>99.6</td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parser</td>
<td>99.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eon</td>
<td>85</td>
<td>10</td>
<td>3.6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mgrid</td>
<td>42</td>
<td>50</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Applu</td>
<td>43.7</td>
<td>29.3</td>
<td>25.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mesa</td>
<td>75</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fma3d</td>
<td>68.4</td>
<td>0.2</td>
<td>15.1</td>
<td>15.4</td>
<td>0.4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sixtrack</td>
<td>35.8</td>
<td>28.5</td>
<td>35.8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apsi</td>
<td>68.7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wupwise</td>
<td>72</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From left to right, the instruction types are ALU, integer divide, integer multiply, FP add/subtract, FP compare, FP convert, FP multiply, FP divide, and FP square root.
implemented in any superscalar processor. This model will be used to show the limits of the approach as the ILP available becomes the limit to performance improvement. Some variations around these models will be discussed in addition to the sensitivity analysis.

The mainstream model is loosely based on the Power4 processor (single core), made by IBM [36, 123]. Although the Power4 is a server processor, a smaller version, under the name PowerPC970 or G5, is to be found in many personal computers built by Apple. Each core is a 4-way superscalar processor, and has 2 ALUs, 2 load/store units, one branch unit and 2 FPUs. The original mainstream model will thus have 3 ALUs, 2 FPUs, 2 load/store units, and a frontend 4 instructions wide.

The top model is inspired by the Intel Itanium 2 processor [59]. This is clearly a server processor, costing several thousands of dollars. Although it is a VLIW processor, it is currently one of the fastest and largest processors available, and thus serves as a reference of the greatest number of resources in a processor today [132]. It has 2 ALUs, 4 load/store units capable of performing ALU operations, 3 branch units, and 2 floating-point units that can also perform integer multiplication. Our derived original top will have 6 ALUs, 2 FPUs, 4 load/store units and a frontend 8 instructions wide. As this processor does not exist, and probably never will due to lack of available parallelism in applications, it is considered the limit of a ’fat’ processor design [28]. Note that the addition of SMT technology and an increase in multithreaded code might make larger superscalars an interesting option.

A fair comparison between different architectures is difficult, as it is always possible to argue that the methodology is biased against one of the options. In our case, the limited reconfigurability increases the number of parallel resources available to the processor’s scheduler. As the statistics of simulations show (section 6.4), most benchmarks that make use of the xALUs find themselves limited by the number of load/store operations. This is in addition to any limitations due to the memory bandwidth and latency. Thus, to overcome this limitation, the number of load/store units (LSU), which perform only address generation and send a request to the cache/memory manager, has been increased. In a similar vein, the issue, dispatch and commit widths should also be increased somewhat.

In both cases, to keep the comparison as fair as possible, the same increases to the number of LSUs and the pipeline width were made on the original models, giving us our baseline models, to which all our dynamic models are compared. Thus, the difference between two models is only the activation of the dynamic reconfiguration, with its eventual increase on the latency of the FPUs depending on the timing parameters chosen. The effect of these modifications for the baseline model are shown in figure 6.1. The baseline mainstream model shows an increase in performance of 7.8% in integer benchmarks, and an increase of 9.9% in floating point benchmarks over the original mainstream. All the results shown below are compared to
the baseline case, thus the speedups are in addition to those gained from these modifications, and only due to the dynamic reconfiguration.

In addition to these models, the option of simply adding a number of ALUs to the top configuration exists. To show that the difference in performance is very small compared to the difference in cost, such a configuration, called supertop, has also been defined. In the opposite direction, a model comparable to the baseline mainstream, but with zero ALUs, and the xALUs placed close to the register file, giving them a latency of 1 cycle, has also been simulated. This will be called the compact dynamic mainstream model.

All these parameters are summarized in table 6.2, while the list of all the other parameters used for the Simplescalar simulations are to be found in appendix C.

<table>
<thead>
<tr>
<th>Model</th>
<th>#ALUs (latency)</th>
<th>#FPUs (latency)</th>
<th>#Load/Issue-units</th>
<th>xALUs per FPU (latency)</th>
<th>Issue-commit widths</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Mainstream</td>
<td>3(1)</td>
<td>2(4)</td>
<td>2</td>
<td>-</td>
<td>4 – 4 – 4</td>
</tr>
<tr>
<td>Original Top</td>
<td>6(1)</td>
<td>2(4)</td>
<td>4</td>
<td>-</td>
<td>8 – 8 – 8</td>
</tr>
<tr>
<td>Baseline Mainstream</td>
<td>3(1)</td>
<td>2(4)</td>
<td>4</td>
<td>-</td>
<td>8 – 8 – 8</td>
</tr>
<tr>
<td>Baseline Top</td>
<td>6(1)</td>
<td>2(4)</td>
<td>5</td>
<td>-</td>
<td>12 – 12 – 8</td>
</tr>
<tr>
<td>Dynamic Mainstream</td>
<td>3(1)</td>
<td>2(5)</td>
<td>4</td>
<td>4(2)</td>
<td>8 – 8 – 8</td>
</tr>
<tr>
<td>Dynamic Top</td>
<td>6(1)</td>
<td>2(5)</td>
<td>5</td>
<td>4(2)</td>
<td>12 – 12 – 8</td>
</tr>
<tr>
<td>Optimal Dynamic Mainstream</td>
<td>3(1)</td>
<td>2(4)</td>
<td>4</td>
<td>3(2)</td>
<td>8 – 8 – 8</td>
</tr>
<tr>
<td>Optimal Dynamic Top</td>
<td>6(1)</td>
<td>2(4)</td>
<td>5</td>
<td>3(2)</td>
<td>12 – 12 – 8</td>
</tr>
<tr>
<td>SuperTop</td>
<td>10(1)</td>
<td>2(5)</td>
<td>5</td>
<td>-</td>
<td>12 – 12 – 8</td>
</tr>
<tr>
<td>Compact Dynamic Mainstream</td>
<td>0</td>
<td>2(4)</td>
<td>4</td>
<td>3(1)</td>
<td>8 – 8 – 8</td>
</tr>
</tbody>
</table>

Table 6.2: Processor model resources. The baseline mainstream and baseline top processors were compared to their dynamic and optimal dynamic counterparts in all simulations. The original mainstream and original top models are only shown as references. Supertop is equivalent to dynamic top with 4 additional ALUs and no reconfiguration. The compact dynamic mainstream model has no static ALUs.

6.2 Integer Benchmarks

The integer benchmarks generally make little use of the floating point unit, and are thus able to make gains through the use of the xALUs. Many use a few integer multiplications or divisions, and some actually make a relatively significant use of FP instructions.

The compact dynamic mainstream model, shown in figures 6.7 and 6.8, which has no normal ALUs, shows strong gains in integer benchmarks, almost equal to that of the optimal dynamic mainstream, with an average gain
Figure 6.1: Speedups of the baseline mainstream model compared to the original mainstream model. There is an average increase in performance due to the higher number of LSUs and the larger pipeline width of about 9%, with small variations.
of 19% and a small loss of 1.4% in \textit{vpr}. The greatest gain is \textit{vortex}, with almost 54%.

### 6.2.1 ALU Benchmarks

These benchmarks, consisting of \textit{gzip}, \textit{gcc}, \textit{mcf}, \textit{crafty}, \textit{parser}, \textit{perlbmk}, \textit{gap}, \textit{vortex} and \textit{bzip2}, make very small use of the FPU, less than 0.1%. Most are thus able to make gains in excess of 10% thanks to dynamic reconfiguration in the baseline configurations. The only exception, \textit{mcf}, is completely limited by memory accesses and cache misses, with a tiny gain of 0.2%. There is little difference between the \textit{dynamic mainstream} and \textit{optimal dynamic mainstream} processors. The best gain is obtained by \textit{vortex}, with 56%, and the worst result is a loss of 3.8% in \textit{sixtrack} in the \textit{dynamic mainstream} model, which turns into a gain of 0.24% in the \textit{optimal dynamic mainstream} model, which has no losses at all. The good gains are due to the fact that in this case, we are adding resources in the form of the \textit{xALUs}, and removing almost nothing, as the FPUs are almost never needed.

For the \textit{top} models, the gains are strongly reduced, with only one benchmark, \textit{vortex}, showing gains of more than 10%. The \textit{dynamic top} model shows losses in a few benchmarks, with the greatest loss, of 0.3%, in \textit{gzip}. The \textit{optimal dynamic top} model, however, shows slightly higher gains and very few losses that are due to the greater latency of the \textit{xALUs} in benchmarks with many dependent ALU instructions. In this case, the worst benchmark is \textit{vpr}, with a small loss of 0.3%.

### 6.2.2 MUL Benchmarks

These benchmarks, though considered ‘integer’ benchmarks, actually make some use of the FPU, though not really for integer multiplication: \textit{vpr} has 10% of FP add operations, while \textit{eon} has 10% of FP add and 5% of FP multiply operations. Finally, \textit{twolf} has a few FP add and FP convert operations, for a total of about 5% of FPU use. These results are less impressive, especially in the case of \textit{vpr}, limited by lack of parallelism, but still show reasonable gains. The greatest gain in the \textit{dynamic mainstream} model is \textit{eon}, with 14%, and the worst is \textit{vpr}, with a gain of 2.7%. Going to the \textit{optimal dynamic} model, \textit{eon} and \textit{twolf} see a small increase to their gains. The alternance of integer and FP instructions usually follows a pattern, allowing the reconfiguration algorithm to adapt to the instructions.

The \textit{dynamic top} model shows a loss of up to 1% in \textit{eon}. This loss is replaced with a gain of 1% in the \textit{optimal dynamic top} model, benefiting from the improved FPU latency. The gains of reconfiguration for the \textit{top} models are uninteresting for these benchmarks.
6.3 Floating Point Benchmarks

The floating point benchmarks in the SPEC suite can be divided into two groups, based on their use of FP instructions: the distinction below is made on whether a benchmark is composed of more or less than 50% of FP instructions. Many of the FP benchmarks make relatively little use of the FPU, although this does not always translate into higher gains for dynamic reconfiguration.

Except for ammp, which shows a tiny gain of 0.4%, all floating point benchmarks have losses in the compact dynamic mainstream model compared to the baseline mainstream. These losses appear because the cost in FP performance of having an FPU reconfigured to execute the integer instructions is greater than the small advantage in integer execution width obtained when both FPUs are reconfigured. The worst result is a loss of almost 16% in sixtrack, with an average loss of 5.7%.

6.3.1 Light FP Benchmarks

This set of benchmarks, composed of wupwise, mesa, galgel, art, equake, facerec, ammp, fma3d and apsi, almost always shows some gain, with mesa, having only 16% of FP instructions, showing the best gain of almost 17% in the dynamic mainstream model. Conversely, art and equake suffer from little available parallelism, and thus cannot benefit from the extra resources even though the FPU is not heavily used, with equake showing a tiny loss of 0.4%. facerec and apsi show the ideal case for FP applications: they have a fairly large percentage of FP instructions (around 30%), but have good parallelism and many independent FP instructions, allowing the reconfiguration to change often to adapt to the arriving instructions. The optimal dynamic mainstream model shows improved gains on all the benchmarks, and forms the greatest impact on the overall gain of the optimised design. The gain in mesa increases to 18%, while the small loss in equake turns to an equally small gain of 0.3%.

In the case of the dynamic top model, all these benchmarks show losses, up to almost 6% for ammp. The reduction in FPU latency provided by the optimal dynamic top model, however, reverses this, with only galgel still showing a loss of 1.7%, probably due to the slower xALUs and the difficulty of finding good reconfiguration possibilities, as will be shown in section 6.4 below.

6.3.2 Heavy FP Benchmarks

The benchmarks in this category make heavy use of FP instructions, up to 64% in some cases. They are swim, mggrid, applu, lucas and sixtrack.

Except for lucas, which shows a small gain of 1.6%, all these benchmarks
get slightly negative speedups from dynamic reconfiguration in the *dynamic mainstream* case, although by a very small margin, 0.5% or less. Finally, the worst-case example is posed by *sixtrack*, which has 64% of FP instructions, of which almost 36% are FP multiply instructions; thus, increasing the latency of the FPU creates a loss of 3.8%, the worst result in the entire suite with this model. This is caused by a sequence of dependent floating-point multiplies and adds, where each iteration suffers the extra FP multiplier latency penalty. Eliminating the FPU latency penalty with the *optimal dynamic mainstream* model cancels the losses, with the best gains provided by *mgrid*, with 0.3%, and *sixtrack*, with 0.2%, and the others showing no effect at all.

The *dynamic top* model shows small losses of up to 3.8% for *sixtrack*, as these benchmarks all suffer from the extra latency of the FPU. As expected, the *optimal dynamic top* model reduces most losses to 0.

Many of these benchmarks can almost never benefit from the reconfiguration, due to heavy use of the FPU, which means it can never be reconfigured. Figures 6.6 and 6.13 show the average usage per cycle of the xALUs on the *mainstream* and *top* models. All the benchmarks showing poor results make almost no use of the extra resources. This is often, but not always, joined by a lack of parallelism in the application.
Figure 6.2: Simulation results of the SPEC benchmarks on the baseline mainstream (light) and dynamic mainstream (dark) models. There are large variations in both IPC and gains, with some significant gains for the dynamic model.
Figure 6.3: Speedups between the baseline mainstream and dynamic mainstream models. The integer benchmarks show universal gains, up to 56% for vortex, whereas the FP results are more varied. Except for sixtrack, all negative speedups are very small, less than 1% slower than the baseline model.
Figure 6.4: Simulation results of the SPEC benchmarks on the baseline mainstream (light) and optimal dynamic mainstream (dark) models. There are large variations in both IPC and gains, and no losses compared to the baseline model.
Figure 6.5: Speedups between the baseline mainstream and optimal dynamic mainstream models. The integer benchmarks show universal gains, up to 56% for vortex. There are no longer any losses in any benchmark, as the FPU latency is the same in both models.
Figure 6.6: Average number of xALUs active per cycle in the *mainstream* model. This measure gives an idea of the usefulness of dynamic reconfiguration. *Swim* never uses the xALUs at all.
Figure 6.7: Simulation results of the SPEC benchmarks on the baseline mainstream (light) and compact dynamic mainstream (dark) models. The integer benchmarks show almost universal gains, whereas the FP benchmarks show almost universal losses.
Figure 6.8: Speedups between the baseline mainstream and compact dynamic mainstream models. There is a clear separation between integer benchmarks that do not need the FPU and make gains of 19% on average, and the FP benchmarks which, with the exception of ammp, cannot make good enough use of the zALUs to make a gain. The FP benchmarks thus have a loss of about 5%, leading to an average gain of about 5%.
Figure 6.9: Simulation results of the SPEC benchmarks on the baseline top (light) and dynamic top (dark) models. Some integer benchmarks now show small losses (of up to 1% for eon), while the FP benchmarks generally suffer from the increased FP latency without being able to use the xALUs. The best gain is still vortex, with almost 11%. 

Benchmark

gzip
vpr
gcc
mcf
crafty
parser
eon
perlbmk
gap
vortex
bzip2
twolf
wupwise
swim
mgrid
applu
mesa
galgel
art
equake
facerec
ammp
lucas
fma3d
sixtrack
apsi

IPC
Figure 6.10: Speedups between the baseline top and dynamic top models. The speedups have been dramatically reduced from the mainstream results, showing the limits of parallelism available in most of these benchmarks.
The only benchmark with a noticeable loss is galgel with a loss of 1.7% due to many closely dependent ALU operations. The best gain is still obtained by vortex, with almost 11%.
Figure 6.12: Speedups between the baseline top and optimal dynamic top models. The speedups have been dramatically reduced from the mainstream results, showing the limits of parallelism available in most of these benchmarks. Some small losses appear, due to the difference in latency between the ALUs and xALUs in this very large processor model.
Figure 6.13: Average number of xALUs active per cycle in the top model. Swim, applu, and lucas never use the xALUs, and the number used is very low for most benchmarks, showing that having more than 1 or 2 xALUs per FPU is completely useless with this model.
6.4 Dynamic Analysis

This section will provide examples of the activity of the dynamic reconfiguration, drawn from several of the benchmarks. The benchmarks used are *mcf*, *galgel*, *wupwise* and *sixtrack*. These were chosen because they exhibit clear examples of the different possible dynamics during their startup phases. However, they generally do not represent the long-term behavior of a particular benchmark.

Figure 6.14 shows the instruction types for *galgel*. As the number of instructions of each type are closely linked, there is no opportunity for reconfiguration here, and thus the configuration, not shown, is to never use the *xALUs*. The opposite case, drawn from *mcf*, is displayed in figure 6.15; almost no FP instructions are present, and thus the configuration is to use the *xALUs* all the time, switching back to the FPU configuration when a rare instruction requiring the FPU arrives.

Finally, the dynamic case from the benchmark *sixtrack*, displays an alternance of ALU and FP instructions present, and is shown in figure 6.16. The pattern shown is one of the startup loops in the application, and repeats regularly around the instruction count shown. At around 200 cycles, there are more FPU instructions than ALU ones, and the switching mechanism does not allocate any *xALUs*. However, at 300 cycles, the situation reverses, and one FPU is converted into 4 *xALUs*. A sharp spike in ALU instructions coupled with a sharp drop in FP instructions at 450 cycles will cause both FPU's to be reallocated as 8 *xALUs* for a brief moment, before resuming FP functions. A long period of relative stability, between 650 and 850 cycles
Figure 6.15: Instruction types and dynamic reconfiguration state for *mcf*. With very few instructions needing the FPU, the configuration uses the xALUs almost all the time, switching back only to service the occasional FP instruction.
Figure 6.16: Instruction types and dynamic reconfiguration state for six-track. This case shows a good example of the variations in the relative number of integer and FP instructions, with the dynamic reconfiguration quickly adapting to each case.
leads to a unchanging configuration.

The structural stalls—i.e., the number of instructions of each type that could not be executed due to a lack of available functional unit will also be discussed. These are taken from the Simpoints over a period of 10^8 instructions, and are thus representative of the entire benchmark. They are drawn from vortex, lucas and sixtrack.

Figure 6.17 shows the case most favorable to our dynamic reconfiguration: vortex only has ALU structural stalls in the baseline model (left), and thus can greatly benefit from any increase in the number of such resources. The dynamic case (right) shows that ALU stalls have almost completely disappeared, and stalls are now caused by a lack of load/store units to feed the ALUs (note the difference in vertical scale). The case of the top model running the vortex benchmark, shown in figure 6.18, still shows a significant
Figure 6.19: Structural stalls for *lucas* in the *baseline mainstream* (left) and *dynamic mainstream* (right) models, for a simulation of $10^8$ instructions. The number of ALU stalls diminishes slightly, providing a small gain, but performance is quickly limited by the large number of FP stalls.

Figure 6.20: Structural stalls for *sixtrack* in the *baseline mainstream* (left) and *dynamic mainstream* (right) models, for a simulation of $10^8$ instructions. There are very few ALU stalls, so no gains possible by reconfiguration. The large number of FP multiply stalls indicates a larger number of FPUs could be useful in this benchmark.
CHAPTER 6. RESULTS

<table>
<thead>
<tr>
<th>Model</th>
<th>Integer Gain/Loss</th>
<th>Floating Point Gain/Loss</th>
<th>Overall Gain/Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Original Mainstream</td>
<td>-7.2%</td>
<td>-8.8%</td>
<td>-8.1%</td>
</tr>
<tr>
<td>Baseline Mainstream</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic Mainstream</td>
<td>19.3%</td>
<td>3.5%</td>
<td>10.4%</td>
</tr>
<tr>
<td>Optimal Dynamic Mainstream</td>
<td><strong>19.5%</strong></td>
<td><strong>5.0%</strong></td>
<td><strong>11.4%</strong></td>
</tr>
<tr>
<td>Compact Dynamic Mainstream</td>
<td>19.0%</td>
<td>-5.7%</td>
<td>5.2%</td>
</tr>
</tbody>
</table>

Table 6.3: Summary of results for the *mainstream* models. The *baseline mainstream* model is used as reference, with all speedups, positive and negative, expressed in percent relative to this model. The *original mainstream* model is used to show the impact of the modifications described in section 6.1.3.

<table>
<thead>
<tr>
<th>Model</th>
<th>Integer Gain/Loss</th>
<th>Floating Point Gain/Loss</th>
<th>Overall Gain/Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline Top</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Dynamic Top</td>
<td>2.53%</td>
<td>-1.56%</td>
<td>0.24%</td>
</tr>
<tr>
<td>Optimal Dynamic Top</td>
<td><strong>2.79%</strong></td>
<td><strong>0.13%</strong></td>
<td><strong>1.30%</strong></td>
</tr>
<tr>
<td>SuperTop</td>
<td>2.89%</td>
<td>0.13%</td>
<td>1.35%</td>
</tr>
</tbody>
</table>

Table 6.4: Summary of results for the *top* models. The *baseline top* model is used as reference, with all speedups, positive and negative, expressed in percent relative to this model. *Supertop* shows the limits of performance achievable with these benchmarks on superscalar architectures.

reduction in the number of ALU stalls, but these are far less numerous to begin with, and the gain is thus reduced from 56% to about 11%.

The case of *lucas*, shown in figure 6.19, is a mixed case where there are some ALU stalls that can be reduced, but the large number of stalls due to FP instructions increases by a similar amount, leading to little or no gains. The worst case, taken from *sixtrack* and displayed in figure 6.20, has almost no stalls due to ALU instructions, which barely get reduced, and thus cannot benefit from reconfiguration. A greater number of FPUs would probably be of use for this benchmark.

6.5 Conclusions

The results for the various *mainstream* models, summarized in table 6.3, show an average gain of 11.4% for the *optimal dynamic mainstream* model compared to the *baseline mainstream* model. The *original mainstream* model is about 8% slower than the *baseline mainstream*, due to the increased number of load/store units and larger pipeline width. Although the differ-
ence between the dynamic mainstream and the optimal dynamic mainstream models in integer benchmarks is very small, the faster multiplier in the optimal model eliminates the losses due to an increased FPU latency, leading to a speedup of 5% in floating point benchmarks, and adding 1% to the average gain at no extra cost. The greatest gain, by the vortex benchmark, is of 56% over the baseline mainstream model. There are large differences in the performance and gains of each benchmark, showing the broad applicability of the proposed dynamic limited reconfiguration. Benchmarks with mostly integer operations benefit the most, but many floating point benchmarks also show interesting gains in the 10% range.

On the other hand, the compact dynamic mainstream model removes all the normal ALUs compared to the baseline mainstream model. While the performance in integer benchmarks is impressive with a gain of 19%, almost equal to that of the optimal dynamic mainstream model, over the baseline at a lower cost, the floating point performance is somewhat weak. The loss of 5.7% in these benchmarks is caused by the need for at least one FPU to be reconfigured as xALUs to handle integer instructions, thus severely limiting the FP instruction issue rate. It might be an interesting tradeoff for some application domains however, as we are exchanging a gain of 19% in integer benchmarks and a reduction in complexity and cost for a loss of 5.7% in floating point benchmarks.

The results for the top models are summarized in table 6.4. The dynamic top model is clearly uninteresting, as the increased complexity of the dynamic reconfiguration brings a negligible average gain of 0.24%, with losses in many floating point benchmarks. The optimal dynamic top model eliminates most of these losses, for an average gain of 1.3%. This small gain shows the limits of parallelism extraction in current and near-future superscalar processors on these CPU intensive benchmarks, as the supertop model, with all static functional units, only has an average gain of 1.35% compared to the baseline top. The slight difference with the optimal dynamic top model is due to the faster and larger ALUs replacing the xALUs in this model, with the corresponding increase in complexity and cost.

6.6 Sensitivity Analysis

Due to the many constraints and parameters that have an impact on the performance of a superscalar processor on one hand, and the many estimates for some of the simulation parameters discussed in section 5.4 on the other hand, an analysis of the sensitivity of this application of dynamic reconfiguration should be performed. This will also provide interesting information about the parameters that are important with respect to performance, and insight into the characteristics of the individual benchmarks of the suite.
6.6.1 Methodology

To avoid contraining the results, a set of rather generous, and thus somewhat unrealistic, parameters are defined as the reference for all simulations. These are listed in appendix C. Sensitivity analysis is then performed by varying only a single parameter or a set of linked parameters (such as issue, dispatch and commit widths).

As the sensitivity analysis would be impossibly long using even the Simpoints, it has been performed by skipping a smaller number of instructions before simulating in detail. This method was used for all sensitivity analysis simulations, where each configuration took about 30 hours on the same processor as the one that was used for the main simulations. While the performance of the individual benchmarks is not representative of the behavior for the entire length of the program, these simulations are long enough to show the trends we are outlining with the sensitivity analysis.

All the simulations in this section were performed by fastforwarding the benchmarks for $10^9$ instructions, and then running them for $5 \cdot 10^7$ instructions.

6.6.2 Parameters Considered

The parameters that were considered in this sensitivity analysis are:

- **FPU Multiplier Latency** This is the latency of the multiplier in the FPU. The latency of an unmodified multiplier is 4 cycles.

- **$xALU$ Latency** This is the latency of the extra ALUs obtained by reconfiguration. The estimated value is 2 cycles.

- **Reconfiguration Latency** This is the number of cycles needed to reconfigure in addition to waiting for the FPU hardware to be idle (either the FPU or all the corresponding $xALUs$).

- **Reconfiguration Factor** This is the number of $xALUs$ obtained from the reconfiguration of an FPU.

- **Pipeline Width** This is the number of instructions that can be issued, dispatched and committed in a single cycle.

- **Number of ALUs** The number of normal ALUs in the processor.

- **Number of FPUs** The number of reconfigurable FPUs in the processor.

- **Memory Latency** The latency of main memory. The cache configurations are unchanged.

- **Number of Load/Store Units** The number of LSUs—i.e., Address Generation Units, in the processor. The total memory bandwidth is unchanged.
6.6.3 Differences between sensitivity and Simpoints results

Figure 6.21 shows the instructions per cycle for all 26 benchmarks using both the Simpoints and the faster method, skip, described in 6.6.1. There are very large variations for the individual benchmarks, but the overall speedups for the mainstream model are about 10% for the Simpoints method and about 15% for the skip method. Thus, while no conclusions about the overall behaviour of the benchmarks may be drawn, the results nonetheless show the performance trends when the various parameters are altered.

6.6.4 Results

The sensitivity analysis results are for the fully unbalanced tree, with a FPU latency of 5 cycles in the dynamic models, and 4 cycles in the baseline models. Thus, all losses in FP benchmarks (and the integer benchmark eon) are somewhat overestimated for the optimal dynamic mainstream model.

Multiplier Latency

The latency of the FPU’s multiplier was discussed in section 5.4.3, with the baseline case being a latency of 4 cycles. Simulations with latencies for the dynamic model ranging from 4 (no increase) to 10 (an increase of 150%) cycles have been performed. The performance comparison in figure 6.23 shows the impact of this increasing latency. Confirming the data from table 6.1, the integer benchmarks, except for eon and gap, make almost no use of the multiplier, some benchmarks being completely unaffected by it. On the other hand, the sensitivity to FPU latency on the part of the FP benchmarks is varied, with about half being very sensitive to the multiplier latency, notably galgel, ammp and sixtrack. This sensitivity is also emphasized by figure 6.24, which shows that almost all benchmarks showing a gain over the baseline model still do so when the latency increases to 6 or 7 cycles. It would seem intuitive, but not necessarily correct, to assume that, with an unchanged latency, there is no loss incurred by adding reconfiguration, as the decision algorithm makes good choices about reconfiguration. This supposition is borne, as the simulations with a multiplier latency of 4 cycles show gains that are either positive or zero (for swim and lucas).

xALU Latency

The most important factor impacting the performance of our dynamic reconfiguration is the latency of the xALUs. Figure 6.25 shows that all benchmarks with a high IPC that make good use of the xALUs see their gains dramatically reduced when this latency increases. Similarly to the analysis of the multiplier latency, the benchmarks unable to use the xALUs, such as swim, mgrid, applu and art are completely unaffected by the increase. As
Figure 6.21: Results for the dynamic mainstream model, using the Simpoints (light) and skip (dark) method for sensitivity analysis. There are large differences in some benchmarks (vpr, mcf, wupwise, equake), but the overall gain compared to the baseline mainstream (not shown in the figure) is similar in each case; 10% for Simpoints versus 15% for skip.
figure 6.26 shows, most benchmarks gaining from reconfiguration can accept latencies of up to 4 cycles—i.e., 4 times the latency of a normal ALU, with 94% of the delay in wires, before seeing their gains turn to losses. Wupwise shows a large decrease in performance when the latency goes from 4 to 5 cycles, indicating the probable presence of one or more loops with dependent instructions 4 cycles apart.

The counterpart of the sensitivity of performance on the latency of the xALUs is that a design making an effort to place the elements of the processor core in such a way that this latency is kept at one cycle, same as any other ALU, would give an appreciable increase in performance.

Reconfiguration Latency

Except for the benchmarks most benefiting from dynamic reconfiguration, the effects of higher reconfiguration latencies, shown in figure 6.27, are not very important (note the non-linear increase in latencies). The speedups, shown in figure 6.28, do not diminish overmuch with increasing latencies, and thus it would be possible and perhaps interesting to take more time to perform the reconfiguration in order to reduce the latency of either the FPU or the xALUs. Figure 6.22 shows that the average number of cycles between reconfigurations has no clear link to performance. However, a program with frequent reconfigurations clearly suffers more from increased reconfiguration latency, showing the need for a dynamic reconfiguration.

Reconfiguration Factor

The number of xALUs per FPU, called reconfiguration factor in figure 6.29, is certainly the parameter that most greatly affects cost, due to the large number of extra wiring needed for each additional xALU. Except for very special cases, such as vortex and apsi, most benchmarks can make use of only 2 or 3 xALUs. In some cases (mcf, sixtrack), a very large number of xALUs actually hurts performance, as instructions with very close dependencies that would have been executed in the normal ALUs with latency 1 are executed in xALUs with latency 2. Taking cost into account, the most interesting option is probably a reconfiguration factor of one or two, which produces gains only slightly lower than those with a factor of 4, but with a far lower increase in complexity.

Pipeline Width

Figure 6.30 shows the variation of performance with the pipeline width—i.e., the number of instructions issued, dispatched and committed in a single cycle. There are substantial gains obtained by an increase from 2 to 4, with small gains, and in fewer benchmarks, for an increase to a pipeline width of 8. There are almost no gains from using higher pipeline widths. Simplescalar
Figure 6.22: Reconfiguration frequency. Swim is shown to use dynamic reconfiguration very seldom. The lack of correlation with the performance gains in figure 6.3 shows the need for a dynamic reconfiguration.
imposes pipeline widths that are a power of 2, which explains the absence of results for the interesting case of a width of 6 instructions.

**Number of ALUs**

The number of normal ALUs present in the processor has a clear impact on the interest of reconfiguration, as shown in figure 6.31. Indeed, models with a small number of ALUs can make impressive gains through reconfiguration, while processors starting with many ALUs usually cannot find enough parallelism to make the extra ALUs worthwhile. Benchmarks with very high parallelism, such as this part of mcf, are able to gain from all these integer resources even with 8 ALUs, but show strongly diminishing returns.

**Number of FPUs**

Figure 6.32 shows the effects of varying the number of reconfigurable FPUs in the processor. Similarly to the case with the reconfiguration factor, most integer benchmarks do not use the FPUs, and cannot take advantage of the many xALUs obtained by reconfiguring the FPUs. Several FP benchmarks are also unable to use many FPUs. On the other hand, some FP benchmarks, notably mesa, facerec and apsi, see their performance increase more with dynamic FPUs than with normal FPUs, as they take better advantage of reconfiguration to adapt the functional unit resources to their varying, often cyclic, instruction type distributions.

**Memory Latency**

The effect of increasing memory latencies, shown in figure 6.33, is obviously to reduce the parallelism available as the processor must wait for the data to arrive. This is the case for all benchmarks, with varying degrees of sensitivity—e.g. notice the difference between eon and swim. The speedups however, as shown in figure 6.34, are far less affected by variations in memory latency. Altough obviously small latencies increase the parallelism available to the processor, who can thus profit more from the extra resources, all benchmarks showing a gain with a memory latency of 1 cycle still gain, by far smaller margins, with a memory latency of 1000 cycles. As expected, the loss-making benchmarks see their losses reduced by increasing memory latencies, as these latencies hide the increase in the latency of the multiplier.

**Number of Load/Store Units**

As explained in sections 6.1.3 and 6.7.1, the dynamic reconfiguration greatly increases the pressure on the Load/Store units, as there is an imbalance between the instruction and functional unit distributions. This is clearly shown in figure 6.35, where the performance of the baseline model hardly
increases with more than 2 LSUs, whereas the performance of the *dynamic*
model can often make use of at least 4 LSUs before the performance levels
off. Note that the performance of some floating point benchmarks are almost
completely independent of the number of LSUs, being bound by processing
power alone.

6.6.5 Conclusions

Table 6.5 shows a summary of the results of the sensitivity analysis. For
each benchmark and parameter, the unsigned relative difference between the
four first values of the parameter were calculated. These were then averaged
to produce the values in each cell. The columns are comparable since they
represent the average over the same number of samples of variations of
parameters, with variations relevant to each parameter considered. These
sensitivity values were then averaged over both benchmarks and parameters
to produce the values on the last line and column, respectively.

Clearly, the most important parameter in the proposed reconfiguration
is the pipeline width: as we are increasing the number of parallel functional
units, any extra issue slots can often be used. This progression follows a
law of diminishing returns, as shown in figure 6.30. The next most sensitive
parameter, by some margin, is the number of LSUs, for similar reasons. The
extra parallelism available means that instructions will be processed in fewer
cycles, and thus, that the number of *load* and *store* instructions that must
be executed every cycle will increase, as they represent about 20% of all
instructions [28]. These two results confirm the necessity of increasing the
pipeline width and the number of LSUs in our *baseline* simulation models
compared to the *original* models.

The third most important parameter, though with far less impact, is
the memory latency. This is because the benchmarks with the lowest IPC
are very sensitive to memory latency, irrespective of whether dynamic re-
configuration is used or not. The latency of the *xALUs* is not an important
parameter, with only the few benchmarks that make great use of them seeing
a difference of more than 10% when it increases. Somewhat surprisingly, the
parameter with the least effect on performance is the latency of the multiplier.
This result is caused by the indifference of almost all integer benchmarks
to this parameter, which lowers the average. Although the effect is modest,
this parameter is interesting as it can be optimized at little cost.

The sensitivity of particular benchmarks to variations is closely linked
to the available parallelism in this benchmark: benchmarks with high IPC
in the portion analyzed, such as *mcf*, have relatively high sensitivity, while
benchmarks with very little parallelism, such as *art*, all but ignore the dy-
namic reconfiguration regardless of the parameters used.
### Table 6.5: Summary of Sensitivity Analysis Results

For each benchmark and parameter considered, the average sensitivity is shown, calculated as the relative difference for the 4 first values of the parameter. The average sensitivity for each benchmark and each parameter are also shown.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Multiplier Latency</th>
<th>xALU Latency</th>
<th>Reconfig. Latency</th>
<th>Reconfig. Factor</th>
<th>Pipeline Width</th>
<th>Number of ALUs</th>
<th>Number of FPUs</th>
<th>Memory Latency</th>
<th>Number of LSUs</th>
<th>Average per benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>gzip</td>
<td>0.00%</td>
<td>5.66%</td>
<td>1.03%</td>
<td>2.11%</td>
<td>23.31%</td>
<td>4.83%</td>
<td>0.56%</td>
<td>1.00%</td>
<td>9.97%</td>
<td>5.39%</td>
</tr>
<tr>
<td>vpr</td>
<td>0.03%</td>
<td>4.61%</td>
<td>5.61%</td>
<td>1.72%</td>
<td>21.78%</td>
<td>3.86%</td>
<td>0.74%</td>
<td>8.35%</td>
<td>9.62%</td>
<td>6.26%</td>
</tr>
<tr>
<td>gcc</td>
<td>0.12%</td>
<td>5.65%</td>
<td>3.35%</td>
<td>1.79%</td>
<td>24.95%</td>
<td>3.83%</td>
<td>0.29%</td>
<td>1.06%</td>
<td>17.38%</td>
<td>6.49%</td>
</tr>
<tr>
<td>mcf</td>
<td>1.13%</td>
<td>10.47%</td>
<td>5.95%</td>
<td>4.80%</td>
<td>43.30%</td>
<td>6.12%</td>
<td>1.09%</td>
<td>1.45%</td>
<td>20.47%</td>
<td>10.53%</td>
</tr>
<tr>
<td>crafty</td>
<td>0.63%</td>
<td>6.47%</td>
<td>5.35%</td>
<td>4.01%</td>
<td>31.04%</td>
<td>4.04%</td>
<td>1.28%</td>
<td>2.08%</td>
<td>18.38%</td>
<td>8.14%</td>
</tr>
<tr>
<td>parser</td>
<td>0.07%</td>
<td>4.11%</td>
<td>1.87%</td>
<td>1.15%</td>
<td>15.96%</td>
<td>2.92%</td>
<td>0.32%</td>
<td>10.18%</td>
<td>8.60%</td>
<td>5.02%</td>
</tr>
<tr>
<td>eon</td>
<td>2.17%</td>
<td>4.07%</td>
<td>11.12%</td>
<td>3.21%</td>
<td>23.81%</td>
<td>3.74%</td>
<td>4.52%</td>
<td>0.11%</td>
<td>11.62%</td>
<td>7.15%</td>
</tr>
<tr>
<td>perl-bmk</td>
<td>0.00%</td>
<td>6.58%</td>
<td>4.18%</td>
<td>1.55%</td>
<td>21.01%</td>
<td>2.67%</td>
<td>0.69%</td>
<td>4.59%</td>
<td>12.40%</td>
<td>5.96%</td>
</tr>
<tr>
<td>gap</td>
<td>1.18%</td>
<td>2.88%</td>
<td>6.06%</td>
<td>1.50%</td>
<td>23.13%</td>
<td>2.71%</td>
<td>1.01%</td>
<td>4.28%</td>
<td>6.50%</td>
<td>5.47%</td>
</tr>
<tr>
<td>vortex</td>
<td>0.24%</td>
<td>7.49%</td>
<td>4.95%</td>
<td>4.57%</td>
<td>37.18%</td>
<td>4.13%</td>
<td>1.54%</td>
<td>2.77%</td>
<td>20.79%</td>
<td>9.30%</td>
</tr>
<tr>
<td>bzip2</td>
<td>0.00%</td>
<td>15.57%</td>
<td>1.92%</td>
<td>6.66%</td>
<td>33.79%</td>
<td>3.78%</td>
<td>5.98%</td>
<td>4.63%</td>
<td>14.35%</td>
<td>9.63%</td>
</tr>
<tr>
<td>twolf</td>
<td>0.84%</td>
<td>5.03%</td>
<td>8.23%</td>
<td>1.73%</td>
<td>16.73%</td>
<td>3.80%</td>
<td>1.65%</td>
<td>2.12%</td>
<td>5.45%</td>
<td>5.06%</td>
</tr>
<tr>
<td>wupwise</td>
<td>1.94%</td>
<td>1.32%</td>
<td>1.61%</td>
<td>0.33%</td>
<td>11.38%</td>
<td>1.60%</td>
<td>6.75%</td>
<td>11.03%</td>
<td>1.07%</td>
<td>4.11%</td>
</tr>
<tr>
<td>swim</td>
<td>0.43%</td>
<td>0.00%</td>
<td>0.02%</td>
<td>0.01%</td>
<td>13.26%</td>
<td>2.42%</td>
<td>7.46%</td>
<td>19.61%</td>
<td>3.11%</td>
<td>5.15%</td>
</tr>
<tr>
<td>mgrid</td>
<td>0.56%</td>
<td>0.23%</td>
<td>1.24%</td>
<td>0.19%</td>
<td>13.33%</td>
<td>1.33%</td>
<td>9.30%</td>
<td>14.47%</td>
<td>4.07%</td>
<td>4.97%</td>
</tr>
<tr>
<td>applu</td>
<td>0.73%</td>
<td>0.07%</td>
<td>0.76%</td>
<td>0.09%</td>
<td>8.21%</td>
<td>1.18%</td>
<td>4.41%</td>
<td>21.49%</td>
<td>2.57%</td>
<td>4.39%</td>
</tr>
<tr>
<td>mesa</td>
<td>2.14%</td>
<td>3.37%</td>
<td>11.60%</td>
<td>4.91%</td>
<td>27.24%</td>
<td>3.58%</td>
<td>9.43%</td>
<td>9.30%</td>
<td>8.03%</td>
<td>8.03%</td>
</tr>
<tr>
<td>galgel</td>
<td>2.85%</td>
<td>7.07%</td>
<td>3.76%</td>
<td>2.07%</td>
<td>36.50%</td>
<td>6.05%</td>
<td>8.04%</td>
<td>4.35%</td>
<td>12.96%</td>
<td>9.29%</td>
</tr>
<tr>
<td>art</td>
<td>0.49%</td>
<td>1.14%</td>
<td>1.09%</td>
<td>0.11%</td>
<td>5.63%</td>
<td>0.56%</td>
<td>1.14%</td>
<td>19.35%</td>
<td>1.85%</td>
<td>3.48%</td>
</tr>
<tr>
<td>equake</td>
<td>1.03%</td>
<td>10.58%</td>
<td>6.73%</td>
<td>2.04%</td>
<td>21.36%</td>
<td>3.52%</td>
<td>4.34%</td>
<td>6.34%</td>
<td>6.84%</td>
<td>6.98%</td>
</tr>
<tr>
<td>facerec</td>
<td>1.76%</td>
<td>1.18%</td>
<td>4.99%</td>
<td>1.43%</td>
<td>27.32%</td>
<td>3.39%</td>
<td>14.41%</td>
<td>7.76%</td>
<td>9.27%</td>
<td>7.94%</td>
</tr>
<tr>
<td>ammp</td>
<td>6.70%</td>
<td>5.53%</td>
<td>0.37%</td>
<td>1.66%</td>
<td>20.10%</td>
<td>2.13%</td>
<td>7.47%</td>
<td>2.62%</td>
<td>8.95%</td>
<td>6.17%</td>
</tr>
<tr>
<td>lucas</td>
<td>1.40%</td>
<td>3.19%</td>
<td>0.77%</td>
<td>0.00%</td>
<td>6.40%</td>
<td>1.11%</td>
<td>6.38%</td>
<td>18.98%</td>
<td>0.23%</td>
<td>4.27%</td>
</tr>
<tr>
<td>fma3d</td>
<td>4.22%</td>
<td>10.75%</td>
<td>2.15%</td>
<td>1.69%</td>
<td>22.58%</td>
<td>2.22%</td>
<td>11.39%</td>
<td>0.69%</td>
<td>8.88%</td>
<td>7.17%</td>
</tr>
<tr>
<td>sixtrack</td>
<td>4.25%</td>
<td>1.97%</td>
<td>4.06%</td>
<td>1.49%</td>
<td>22.87%</td>
<td>2.86%</td>
<td>11.02%</td>
<td>1.58%</td>
<td>6.69%</td>
<td>6.31%</td>
</tr>
<tr>
<td>apsi</td>
<td>2.71%</td>
<td>2.10%</td>
<td>9.04%</td>
<td>3.62%</td>
<td>34.06%</td>
<td>5.35%</td>
<td>14.18%</td>
<td>2.40%</td>
<td>13.74%</td>
<td>9.69%</td>
</tr>
<tr>
<td>Average per parameter</td>
<td>1.45%</td>
<td>4.89%</td>
<td>4.15%</td>
<td>2.09%</td>
<td>22.55%</td>
<td>3.22%</td>
<td>5.21%</td>
<td>6.70%</td>
<td>9.42%</td>
<td>6.63%</td>
</tr>
</tbody>
</table>

The average sensitivity for each benchmark and each parameter are also shown.
Figure 6.23: Results for variations of the FPU Multiplier latency. A non-reconfigurable FPU multiplier has a latency of 4 cycles. The benchmarks with flat or almost flat profiles make little use of the multiplier. Under certain conditions, the same applies to a reconfigurable multiplier. A conservative value would be a latency of 5 cycles. The benchmarks with flat or almost flat profiles make little use of the multiplier.
Figure 6.24: Speedups for variations of the FPU Multiplier latency. Benchmarks are either very sensitive to increases in the FPU latency (ammp, fma3d, sixtrack) or rather unaffected by it, with few benchmarks showing a light slope. A non-reconfigurable FPU multiplier has a latency of 4, and, under certain conditions, the same applies to a reconfigurable multiplier. A conservative value would be a latency of 5 cycles.
Figure 6.25: Results for variations of the latency of the xALUs. Benchmarks that cannot use them are clearly visible (swim, mgrid, applu), while most other benchmarks make some use of the extra functional units. A latency of 2 cycles is a typical value, with a latency of 3 being a very conservative estimate.
Figure 6.26: Speedups for variations of the latency of the xALUs. Most benchmarks that profit from reconfiguration show a gain with latencies of up to about 4 cycles, above which almost all benchmarks have losses due to stalls waiting for the results from one of the xALUs. A latency of 2 cycles is a typical value, with a latency of 3 being a very conservative estimate.
The reconfiguration studied takes a single cycle, and multiple cycles up to 3 cycles in a conservative estimate. However, most of the benchmarks are not strongly affected by the delay in reconfiguration, indicating that taking some time to reconfigure for a very fast execution later is worthwhile. The results for variations of the reconfiguration latency (in addition to waiting for all reconfigurable units to be idle) are shown in Figure 6.27.
Figure 6.28: Speedups for variations of the reconfiguration latency. Most benchmarks show a gain for latencies up to 10 cycles, with some producing good results even with a latency of 50 cycles. The reconfiguration decision algorithm does not take this latency into account for its decisions. The reconfiguration studied takes a single 1 cycle, and might take up to 3 cycles in a conservative estimate.
Figure 6.29: Results for variations of the reconfiguration factor—i.e., the number of xALUs obtained by reconfiguration of a single FPU. For almost all benchmarks, the gains level out with a factor of 4. A conservative value for this parameter would be 2 xALUs per FPU, while the studied cases use a factor of 3.
Figure 6.30: Results for variations of the pipeline width—i.e., the number of instructions issued, dispatched and committed in a single cycle. The gains obtained from going from a width of two to four are visible in almost all benchmarks, with applu showing the smallest gain. Likewise, increasing the width to eight is generally interesting, but with more exceptions (swim, applu, art, lucas). Almost no benchmarks benefit from a higher pipeline width. Both the typical and conservative values for this benchmark are around a width of 4 to 6 instructions per cycle.
Figure 6.31: Results for variations of the number of ALUs in the processor. A very small number makes reconfiguration very interesting, as there are always sections of code with many ALU operations. Higher values are only interesting for benchmarks with high IPC. Most superscalar processors have 3 ALUs.
Figure 6.32: Results for variations of the number of FPUs in the processor. Unsurprisingly, most integer benchmarks, except for *eon*, ignore this parameter. On the FP side, gains are visible with up to three FPUs, with the *baseline* model showing smaller gains than the *dynamic* model as the number of FPUs increases. Most designs have 2 FPUs, with some processors having only a single one.
FIGURE 6.33: Results for variations of the main memory latency. The cache parameters are kept constant. Most benchmarks have a visible dependence on memory latency, with some, such as `eon`, impressively insensitive to memory delays, perhaps due to efficient caching. Typical memory latencies are on the order of 100 cycles, and a conservative estimate would be around 200 cycles.
Figure 6.34: Speedups for variations of the main memory latency, again with the cache parameters kept constant. Although the gains do decline, especially for large values due to diminishing available parallelism, all benchmarks that produce a gain still do so with a large memory latency. Also note the diminishing loss in swim, applu and lucas as the memory latency hides the delay of the reconfigurable FPU. Typical memory latencies are on the order of 100 cycles, and a conservative estimate would be around 200 cycles.
Figure 6.35: Results for variations of the number of LSUs in the processor. Many benchmarks show a need for at least 3 LSUs in the dynamic case, with some floating point benchmarks having few I/O needs and seeing no gains above 2 LSUs. A conservative design has 2 LSUs, while some aggressive processors might have 6 such units.

A conservative design has 2 LSUs, while some aggressive processors might have 6 such units.

Benchmark

Instructions Per Cycle

Figure 6.35: Results for variations of the number of LSUs in the processor. Many benchmarks show a need for at least 3 LSUs in the dynamic case, with some floating point benchmarks having few I/O needs and seeing no gains above 2 LSUs. A conservative design has 2 LSUs, while some aggressive processors might have 6 such units.
6.7 Problems and Limitations

This section discusses some of the issues that must be taken into account, such as the pipeline width and the complexity of the reconfiguration decision algorithm. The limits of this example of dynamic limited reconfiguration, mostly due to intensive use of FP multiply instructions, are also discussed.

6.7.1 Issue and Commit Widths

As this application of limited reconfiguration is based on increasing the number of parallel resources in a processor, the gains obtained are highly dependent on the parallelism available to the out-of-order core. In addition to the parallelism inherent in the application, which is a parameter that cannot be altered, the pipeline width is also important in this regard: as the number of possible functional units is increased by reconfiguration, so should the number of instructions that are processed each cycle, both before (issue and dispatch) and after (commit) the execution engine. For fairness in comparison, the original models were given the same pipeline width.

One could argue that, given the original models, which are relatively balanced, the baseline models, obtained by giving the same amount of pipeline width and load/store units, are unbalanced. These increases, which are needed to avoid including the effects of a greater pipeline width or the additional load/store units in the results, do indeed slightly unbalance the baseline models. However, the difference in performance between the original and baseline models is not very great, as the fewer parallel resources limit the usage of the increased pipeline width, as shown in figure 6.1.

The alternative option, which would be to keep both the pipeline width and the number of LSUs identical in the original and dynamic models, would cripple the latter, as, in the case of the dynamic mainstream, we would have a possible maximum of 13 functional units (3 ALUs, 8 xALUs and 2 LSUs) completely strangled by a pipeline width of 4 instructions. Similarly, as the ratio of load and store to other instructions varies between 23% (lucas) and 50% (eon), with an average of 37%, as shown in figure 6.36, the ratio of LSUs to all functional units, a mere 15%, is clearly insufficient. For comparison, the original mainstream has a ratio of 29%, and the original top a ratio of 33%.

6.7.2 Complexity of Good Decision Algorithm

As examined in sections 5.3.2 and 5.3.3, the problem of deciding when and how to reconfigure a dynamic system is a very complex one. The search for an optimal solution, while possible, is very complex, and requires prior knowledge of all instruction arrival times, thus making it unfeasible. Heuristics or approaches based on control theory are possible, but the complexity
Figure 6.36: Percentage of load and store instructions using the Simpoints. On average, 37% of instructions are memory references.
of these methods means they are often not cost-effective. This is mostly
due to hardware complexity more than delay, as section 6.6.4 showed that
a fairly large number of cycles may be used for the decision and reconfig-
uration. However, using the processor’s functional units to implement the
reconfiguration decision algorithm poses many problems, especially in the
case of interrupts, such as storage, code separation and scheduling.
Considering only hardcoded decision algorithms, even relatively simple
approaches, such as the balance algorithm discussed in section 5.3.4, would
have a very high cost, mainly due to multiplication and division by values
other than powers of two\(^4\).

6.7.3 FP-Intensive Code

In the case of code using the FPU intensively, there are few possibilities for
reconfiguration, as there are always FP instructions waiting to be executed.
Thus, no gains from reconfiguration can be obtained. If these instructions
are mostly multiply instructions, such as in sixtrack, the increase in the
multiplier’s latency in the fully unbalanced tree leads to a loss in perfor-
mance, roughly proportional to the percentage of multiply instructions and
the increase in multiplier latency.

These benchmarks shown the importance of a careful design of the multi-
plier tree to avoid an increase in latency used for the optimal dynamic model,
which eliminates the loss of almost 4% in the sixtrack benchmark running on
the dynamic baseline model, and means dynamic reconfiguration produces
only gains in performance instead of having to make a trade-off between
integer and floating point performance.

6.8 Conclusions

The many results presented in this chapter have shown the viability and
advantages of this application of limited dynamic reconfiguration for a very
large variety of benchmarks. The gains of up to 56%, with an average of
11% obtained with mainstream processor models are very interesting, while
the gains of up to 11%, with an average of only 1.3%, make the case for top
processors less appealing, as the limits of single-thread parallelism available
in the SPEC CPU benchmarks are reached. The compact dynamic baseline
model, using less resources than the baseline mainstream model, still shows
a gain of about 5% over the latter. This overall gain is due to a good gain in
integer benchmarks and a loss of 5% in floating point benchmarks, leading
to a tradeoff between performance of different benchmarks and cost.

\(^4\)Although it is possible to choose values of \(\lambda\) and \(\alpha\) such that the multiplication and
division in equation (5.8) can be replaced by shifts, it severely constricts the algorithm
and design.
As with all research on processor and computer architecture, there are many parameters to consider, implying many choices in the design, that must be adapted to the constraints in terms of performance, cost and power consumption. Similarly, the gains to be expected by applying a new method or idea are very difficult to estimate in advance, as there is no simple way of exposing the bottlenecks in the system. When measured over a large set of predefined applications and data sets, the gains in performance are always modest, since only part of the system is being modified.

The effect of the multiplier latency, while quite light overall, is quite visible in the floating point benchmarks of the SPEC suite. The detailed design and timing optimization of the multiplier tree is thus an important aspect of the dynamic reconfiguration proposed, especially since the optimally unbalanced tree is not only faster, but also less costly in terms of size and power consumption than the fully unbalanced tree.

Due to the very general nature of the benchmarks used, and the unknown applications a general purpose processor must be able to execute reasonably efficiently, a dynamic reconfiguration is needed, with low reconfiguration delays. Furthermore, this dynamism requires a fast and effective decision algorithm for any gains to emerge in more than just a few applications well suited to the reconfiguration proposed.
Chapter 7

Conclusion

7.1 Conclusions

We have studied the possibility of adding reconfigurability to superscalar processors. After a choice of how to add this reconfigurability, we focused on the functional units of the processor, while maintaining binary compatibility. All the issues and consequences of such an approach were explored, coalescing into a detailed design with precise timing results. These timings were used to configure a modified superscalar processor simulator to obtain quantitative speedups for the application of limited dynamic reconfiguration over a wide range of real-world applications.

The results presented in chapter 6 show that, with a careful design, an average improvement in performance of over 11% can be obtained over a wide range of applications in a mainstream general-purpose superscalar processor. The cost of these modifications can be roughly split into 3 groups: the cost of the modified tree, including the xALUs, and the decision algorithm has been precisely measured and is low. The cost of added wires for the forwarding paths and the reservation stations is difficult to estimate, but is almost certainly the greatest cost involved. This cost is the same or lower than simply adding the same number of static functional units, however. Finally, an increase in the complexity of the scheduler is expected, but should not be dramatic. These gains are made possible by the use of reconfiguration of some of the functional units of the processor, thus allowing it to adapt its hardware structure to the application it is executing. The sensitivity analysis showed the importance of the pipeline width and the number of load/store units, which affect the gains of dynamic reconfiguration. The low overall effect of xALU latency and the latency of the FPU’s multiplier were also confirmed, although the latter has a visible effect of the floating point benchmarks. The compact dynamic mainstream model showed a possible trade-off between the performance of integer and floating point benchmarks and complexity.

137
CHAPTER 7. CONCLUSION

Limiting the amount of reconfiguration allowed provides gains over a very wide range of applications as the performance penalty for using reconfigurable logic is minimized. Likewise, dynamic reconfiguration maintains binary compatibility, allowing all current programs to run, most of them with increased performance, using this limited reconfiguration.

There are no major problems to implement this design in a modern Floating Point Unit, as the detailed study and synthesis reports have shown. When the aim is to improve the performance of most applications, without introducing significant penalties in any application, great care must be taken in the analysis and design of the reconfigurable functional units, as the timing margins are very small.

The resulting reconfigurable processor shows gains over almost all the benchmarks in a very wide suite, acknowledged to represent most processor intensive applications today. The greatest gain is a speedup of over 56%, with only 2 benchmarks out of 26 showing no gain at all, and no losses.

In addition to the design of the FPU, the greatest complexity is in the decision algorithm to control the dynamic reconfiguration. An optimal solution is very complex, and impossible to derive in real-time. Adding the configuration information to the binary code would eliminate the need for a decision algorithm, but at the expense of binary compatibility. However, an algorithm which is both simple to implement and produces good results has been proposed and implemented. The proposed implementation can be executed in a single cycle in all but the fastest current processors.

The proposed limited reconfiguration is clearly interesting for mainstream superscalar processors, but shows less interest in top processors with many functional units, as the limits of parallelism that can be extracted from applications in a superscalar processor architecture are reached.

7.2 Contributions

Limiting the reconfiguration possibilities has advantages over the use of fully reconfigurable logic such as FPGAs, whose interest is mainly limited to applications in the digital signal processing domain providing enough parallelism to compensate for the slower logic speed. Limited reconfiguration is shown to allow gains in almost any application, with no drawbacks to the few applications which cannot benefit from it. An example of custom instructions to increase the performance of an application in the telecommunications domain has also been presented.

A examination of the reconfiguration possibilities in a processor’s functional units has been performed, focusing on the large floating point unit, and specifically on the compressor tree that is the foundation of all fast multipliers. The theoretical limits of modifications possible without altering the overall timing have been established, with the results verified experimentally.
7.3. PERSPECTIVES

The complexity of the control for dynamic reconfiguration has been exposed, tackled, and partially solved. An optimal solution is expressed mathematically as a relatively complex integer linear program whose resolution by numerical analysis is only possible for very small problems with current tools and methods. However, in this particular application, a simpler algorithm with measured performance close to the theoretical optimum was implemented and provided interesting speedups.

The performance of all the superscalar processor models was measured using a single processing thread, as this is still the main structure of most computing tasks today. The speedups obtained, an average of 11% due only to architectural enhancements, represent an interesting increase in performance, especially when the difficulties encountered with shrinking technologies are taken into account.

7.3 Perspectives

The dynamic reconfiguration of a superscalar processor’s FPU described in this thesis increases the resources that are made available for parallel execution. Thus, any technology that increases the usage of such parallelism, notably Simultaneous Multithreading, should see an increased benefit over the single-threaded performance analyzed in this thesis.

An implementation of this dynamic reconfiguration should evaluate the cost of the most aggressive design of the reconfiguration. This would allow reconfiguration when the FPU is in the third or fourth stage of its pipeline, and the use of all floating point operations exception multiplication, division and function evaluation at the same time as the xALUs.

In the same way that MMX instructions added to regular functional units allows some parallel processing of data, it might be interesting to provide the xALUs with the same capability, in essence another configuration in xALU mode. This would transform a superscalar processor into a small vector processor at little cost.

Implementing this dynamic reconfiguration in a VLIW processor, with the configuration information generated by the compiler in addition to the binary code, might be an interesting direction of research. In this case, binary compatibility is forfeited, but this is already often the case for VLIW processors. The upshot is a heavy simplification of the processor, notably for the scheduling, and there is no longer a need for dynamic reconfiguration control logic.

The application of limited reconfigurability to other domains, such as telecoms, video and signal processing, might yield interesting gains in performance, cost, or both.
Bibliography


BIBLIOGRAPHY


BIBLIOGRAPHY


[110] Apple G5 Processor is a PowerPC970. 
http://www.apple.com/g5processor

[111] AMD Athlon 64 Processor. 
http://www.amd.com/us-en/Processors/ProductInformation0,30,18,9485,9487,00.html


[113] ARM Processors and architecture. 
http://www.arm.com/products/CPUs/index.html

http://www.arm.com/miscPDFs/5049.pdf

http://www.bsd.org/

[116] Ilog CPLEX, a tool for solving linear optimization problems, 
http://www.ilog.com/products/cplex/

http://www.eembc.hotdesk.com

[118] Faraday Technology Corp., 
http://www.faraday-tech.com

http://www.freescale.com/webapp/sps/site/homepage.jsp?nodeId=018rH3bTdG

[120] Free Software Foundation. 
http://www.fsf.org

[121] GNU’s Not Unix, set of tools for unix/linux systems. 
http://www.gnu.org
[122] GNU gzip compression utility.  
http://www.gzip.org

[123] IBM Power4 Processors.  


[132] In-Stat/MDR Workstation and Server Processor Chart,  
http://www.mdronline.commpr/cw/cw_wks.html

[133] Intel IA-64 Architecture Overview, Hewlett-Packard,  
http://software.external.hp.com/software/I686software/IA64/arch.html


[135] Intel 64-bit Extensions.  
http://www.intel.com/technology/64bitextensions

[136] Intel SpeedStep technology,  

[137] Linus Torvalds, author of the Linux free Operating System. e.g. see  
http://web.mit.edu/invent/iow/torvalds.html

[138] (Microsoft) Windows Products and Technologies History,  
http://www.microsoft.com/windows/WinHistoryIntro.mspx

[139] Gordon Moore, former CEO of Intel Corp.  
http://www.intel.com/pressroom/kits/bios/moore.htm
[140] Online book on modern networks,  
http://irp.eme-enseignement.fr

[141] nVidia Corp.,  
http://www.nvidia.com

[142] nVidia GeForce 6800 GT,  
http://www.simhq.com/technology/technology_026a.html

http://www.tpc.org

http://www.spec.org/cpu2000/


[146] SPEC CPU 2000 Results published by SPEC.  

[147] SPEC CPU 2000 Binaries for the Alpha ISA.  
http://www.simplescalar.com/benchmarks.html

http://www.spec.org/cpu2000/docs/runrules.html

[149] Richard Stallman, advocate of free software.  
http://www.stallman.org/home.html

[150] Stretch Inc.  
http://www.stretchinc.com

[151] Synopsys Inc.  
http://www.synopsys.com

[152] Tensilica, Inc.  
http://www.tensilica.com

[153] United Microelectronics Corp. 0.18µm technology,  
http://www.umc.com/English/process/d.asp

[154] Xilinx Virtex-II Pro FPGA with embedded PowerPC 405 processors,  

[155] Xilinx FPGA and CPLD Design Tools,  
http://www.xilinx.com/products/design_resources/design_tool/index.htm
Appendix A

VHDL Schematics and Reports

This appendix contains the post-synthesis schematics for the various Very Large Scale Integration (VLSI) designs presented in the main body of the thesis, including the critical paths. The corresponding timing, area and power reports are also presented. All designs were written in VHDL, with the CSA compressor tree generated in Verilog by an approach similar to the Three-Greedy approach described in [84]. They were then synthesized on Unified Microelectronics Corporation’s [153] 0.18μm technology using Synopsys’ [151] Design Compiler and the Artisan libraries.

First, the balanced, fully unbalanced and optimally unbalanced versions of the compressor tree in the multiplier are presented. Then, an implementation of the decision algorithm threshold described in section 5.3.4 is detailed. The power reports are the result of a simple static analysis, but are sufficient to draw conclusions on the differences in power consumption between the different trees.

A.1 Multiplier Tree

A balanced, unmodified tree with only CSAs, as described in section 5.2.4 and figure 5.23 (top), is compared to an unbalanced tree where 3 CPAs have been added. Two cases are considered, with the difference being in the position of the CPAs in the tree. The first considers the fully unbalanced tree shown in figure 5.23 (bottom). The second attempts to match exactly the delay through the CPAs with the delay through the CSAs before joining the two, following the structure in figure 5.24 (bottom) to get an optimally unbalanced tree. There is no difference in structure between the two unbalanced solutions, so only the schematic for the optimally unbalanced tree is shown.

The difference in structure between the balanced and unbalanced cases
is highlighted by figures A.1 and A.2. This structure does not change with the position and timings of the CPAs, as only the connection between them and the CSA tree are affected.

Tables A.1, A.2, A.7 and A.10 show the reports for timing, area and power in the case of the balanced tree. They are used as references for all the results below.

Comparing the balanced and fully unbalanced trees produces an increase in delay of 5.4%, with a corresponding increase of 10.8% in area and 5% in power consumption. These numbers are compiled from tables A.3 to A.11.

However, as the timing reports in tables A.1 to A.6 show, there is no measurable increase in delay between the balanced and optimally unbalanced designs, as the difference of 0.01ns, or 0.24% is well withing the margin of error of Design Compiler, the synthesis tool used. The effect on the area, due to the greater complexity of a CPA over a CSA, is an increase of 7.5%, as shown by tables A.7 and A.9. The effect of adding 3 CPAs on power is shown in tables A.10 and A.12, with an increase of 3.8%.

The difference between the fully unbalanced and optimally unbalanced trees is about 0.20ns, which is added to the delay of the CSA tree (tables A.3 to A.6).
Figure A.1: Schematic of a 64-bit multiplier using an unmodified, balanced partial products compression tree. The critical path is shown in red. PPROD is the partial products generation, CSAB_INST is the balanced compressor tree, and add_68/plus is the final CPA.
Figure A.2: Schematic of a 64-bit multiplier using a partial products compression tree that has been unbalanced to avoid increasing the delay over the unmodified case. The critical path goes through the CSA branch, but both branches have very similar timing. The sel input allows fully independent switching of up to 4 multiplexers, although only 2 are used in this design. Leading to the unused sel[3] input. The critical path is shown in red.
A.1. MULTIPLIER TREE

Table A.1: Timing report for the multiplier using a balanced compressor tree. (1)
### Table A.2: Timing report for the multiplier using a balanced compressor tree.

<table>
<thead>
<tr>
<th>Component Description</th>
<th>Data Arrival Time</th>
<th>Max Delay</th>
<th>Output External Delay</th>
<th>Data Required Time</th>
<th>Slack (VIOLATED)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSAB_INST/U14758/Y (NAND2X4)</td>
<td>-4.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>CSAB_INST/U250/Y (AND2X2)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>CSAB_INST/U248/Y (OAI2BB2X4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>CSAB_INST/U5997/Y (NOR2X4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>CSAB_INST/CARRY[70] (csa_bal)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/B[70] (top_bal_DW01_add_127_0)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U789/Y (NAND2BX4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U50/Y (NAND4BX2)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U1103/Y (NOR3BX4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U995/Y (NAND2X4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U609/Y (NOR3X2)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U1107/Y (OA12BB1X4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U764/Y (NAND3BX4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U1473/Y (NAND2BX4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U896/Y (AO12X4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U893/Y (NOR2X4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U751/Y (XOR2X2)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U1503/Y (MXI2X4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/U806/Y (NAND2X4)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>add_68/plus/SUM[87] (top_bal_DW01_add_127_0)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>mul[87] (out)</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
<tr>
<td>Data arrival time</td>
<td>4.10</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>-4.10</td>
</tr>
</tbody>
</table>

The timing report shows that the slack time is violated by 4.10 units.
### A.1. MULTIPLIER TREE

---

**Report : timing**
- path full
- delay max
- max_paths 1

**Design : top_unbal_overest_muxes2**

**Version : 2003.06**

**Date : Tue Mar 30 15:44:30 2004**

---

**Operating Conditions:** typical  
**Library:** typical

**Startpoint:** b[5] (input port)  
**Endpoint:** mul[122] (output port)

**Path Group:** default  
**Path Type:** max

<table>
<thead>
<tr>
<th>Component</th>
<th>Wire Load Model</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_unbal_overest_muxes2</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>partial_prod</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>csa_unbal_overest</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2_DW01_add_128_0</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Point</th>
<th>Incr</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>input external delay</td>
<td>0.00</td>
<td>0.00 f</td>
</tr>
<tr>
<td>b[5] (in)</td>
<td>0.00</td>
<td>0.00 f</td>
</tr>
<tr>
<td>PPROD/b[5] (partial_prod)</td>
<td>0.00</td>
<td>0.00 f</td>
</tr>
<tr>
<td>PPROD/U81/Y (AND2X4)</td>
<td>0.12</td>
<td>0.12 f</td>
</tr>
<tr>
<td>PPROD/pp[3141] (partial_prod)</td>
<td>0.00</td>
<td>0.12 f</td>
</tr>
<tr>
<td>CSA_INST/pp[3141] (csa_unbal_overest)</td>
<td>0.00</td>
<td>0.12 f</td>
</tr>
<tr>
<td>CSA_INST/U796/Y (INX2)</td>
<td>0.07</td>
<td>0.19 r</td>
</tr>
<tr>
<td>CSA_INST/U14505/Y (HAND2BX4)</td>
<td>0.06</td>
<td>0.26 f</td>
</tr>
<tr>
<td>CSA_INST/U11199/Y (HAND2BX4)</td>
<td>0.09</td>
<td>0.35 r</td>
</tr>
<tr>
<td>CSA_INST/U8851/Y (XOR2X4)</td>
<td>0.18</td>
<td>0.53 r</td>
</tr>
<tr>
<td>CSA_INST/U6279/Y (XOR2X2)</td>
<td>0.21</td>
<td>0.74 r</td>
</tr>
<tr>
<td>CSA_INST/U6869/Y (XNOR3X4)</td>
<td>0.33</td>
<td>1.07 r</td>
</tr>
<tr>
<td>CSA_INST/U3062/Y (INX4)</td>
<td>0.06</td>
<td>1.12 f</td>
</tr>
<tr>
<td>CSA_INST/U14579/Y (HAND2BX4)</td>
<td>0.07</td>
<td>1.19 r</td>
</tr>
<tr>
<td>CSA_INST/U14580/Y (HAND2BX4)</td>
<td>0.04</td>
<td>1.23 f</td>
</tr>
<tr>
<td>CSA_INST/U11251/Y (OAI21X4)</td>
<td>0.09</td>
<td>1.33 r</td>
</tr>
<tr>
<td>CSA_INST/U5709/Y (INX8)</td>
<td>0.04</td>
<td>1.37 f</td>
</tr>
<tr>
<td>CSA_INST/U14581/Y (HAND2X4)</td>
<td>0.06</td>
<td>1.43 r</td>
</tr>
<tr>
<td>CSA_INST/U14582/Y (HAND2X4)</td>
<td>0.04</td>
<td>1.47 f</td>
</tr>
<tr>
<td>CSA_INST/U14583/Y (OAI21X4)</td>
<td>0.08</td>
<td>1.55 r</td>
</tr>
<tr>
<td>CSA_INST/U6345/Y (XNOR3X4)</td>
<td>0.28</td>
<td>1.83 f</td>
</tr>
<tr>
<td>CSA_INST/U6270/Y (INX4)</td>
<td>0.09</td>
<td>1.92 r</td>
</tr>
<tr>
<td>CSA_INST/U14646/Y (OAI21X4)</td>
<td>0.07</td>
<td>1.99 f</td>
</tr>
<tr>
<td>CSA_INST/U14648/Y (HAND2X4)</td>
<td>0.08</td>
<td>2.07 r</td>
</tr>
<tr>
<td>CSA_INST/U6971/Y (XOR2X4)</td>
<td>0.18</td>
<td>2.25 r</td>
</tr>
</tbody>
</table>

---

Table A.3: Timing report for the multiplier using a **fully unbalanced** compressor tree and 3 CPAs. (1)
<table>
<thead>
<tr>
<th>Component</th>
<th>Delay (ns)</th>
<th>Output Delay (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CSA_INST/U5875/Y (XNOR3X4)</td>
<td>0.32</td>
<td>2.58 r</td>
</tr>
<tr>
<td>CSA_INST/U6253/Y (XNOR3X2)</td>
<td>0.22</td>
<td>2.80 r</td>
</tr>
<tr>
<td>CSA_INST/SUM[57] (csa_unbal_overest)</td>
<td>0.00</td>
<td>2.80 r</td>
</tr>
<tr>
<td>U657/Y (MX2X4)</td>
<td>0.15</td>
<td>2.95 r</td>
</tr>
<tr>
<td>add_137/plus/A[57] (top_unbal_overest_muxes2_DW01_add_128_0)</td>
<td>0.00</td>
<td>2.95 r</td>
</tr>
<tr>
<td>add_137/plus/US92/Y (INVBX8)</td>
<td>0.04</td>
<td>2.99 f</td>
</tr>
<tr>
<td>add_137/plus/US94/Y (NAND2BX4)</td>
<td>0.08</td>
<td>3.07 r</td>
</tr>
<tr>
<td>add_137/plus/U431/Y (CLKINVBX8)</td>
<td>0.07</td>
<td>3.13 f</td>
</tr>
<tr>
<td>add_137/plus/U430/Y (NOR2BX4)</td>
<td>0.09</td>
<td>3.23 r</td>
</tr>
<tr>
<td>add_137/plus/U1109/Y (DAI21X4)</td>
<td>0.06</td>
<td>3.28 f</td>
</tr>
<tr>
<td>add_137/plus/US91/Y (NOR2X4)</td>
<td>0.08</td>
<td>3.36 r</td>
</tr>
<tr>
<td>add_137/plus/US90/Y (NOR2X4)</td>
<td>0.06</td>
<td>3.41 f</td>
</tr>
<tr>
<td>add_137/plus/US13/Y (NAND4BX4)</td>
<td>0.11</td>
<td>3.52 r</td>
</tr>
<tr>
<td>add_137/plus/US55/Y (NAND2BX4)</td>
<td>0.12</td>
<td>3.65 r</td>
</tr>
<tr>
<td>add_137/plus/US36/Y (NAND2X4)</td>
<td>0.06</td>
<td>3.71 f</td>
</tr>
<tr>
<td>add_137/plus/US48/Y (DAI21X4)</td>
<td>0.19</td>
<td>3.90 r</td>
</tr>
<tr>
<td>add_137/plus/U719/Y (NAND2X4)</td>
<td>0.05</td>
<td>3.96 f</td>
</tr>
<tr>
<td>add_137/plus/U1426/Y (NAND2BX4)</td>
<td>0.08</td>
<td>4.04 r</td>
</tr>
<tr>
<td>add_137/plus/U1096/Y (XOR2X4)</td>
<td>0.15</td>
<td>4.19 r</td>
</tr>
<tr>
<td>add_137/plus/U1464/Y (MXI2X4)</td>
<td>0.07</td>
<td>4.26 f</td>
</tr>
<tr>
<td>add_137/plus/U479/Y (DAI21X4)</td>
<td>0.06</td>
<td>4.32 r</td>
</tr>
<tr>
<td>add_137/plus/SUM[122] (top_unbal_overest_muxes2_DW01_add_128_0)</td>
<td>0.00</td>
<td>4.32 r</td>
</tr>
<tr>
<td>mul[122] (out)</td>
<td>0.00</td>
<td>4.32 r</td>
</tr>
<tr>
<td>data arrival time</td>
<td></td>
<td>4.32</td>
</tr>
<tr>
<td>max_delay</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>output external delay</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>data required time</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>data required time</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td>data arrival time</td>
<td>-4.32</td>
<td></td>
</tr>
<tr>
<td>slack (VIOLATED)</td>
<td>-4.32</td>
<td></td>
</tr>
</tbody>
</table>

Table A.4: Timing report for the multiplier using a *fully unbalanced* compressor tree and 3 CPAs. The critical path does not go through the CPAs, and about 0.20ns are added in the delay of the CSA tree compared to the *balanced* case, as the final adder starts at 2.95ns instead of 2.74ns in table A.2. (2)
A.1. MULTIPLIER TREE

Table A.5: Timing report for the multiplier using an optimally unbalanced compressor tree and 3 CPAs. (1)
Table A.6: Timing report for the multiplier using an optimally unbalanced compressor tree and 3 CPAs. The critical path does not go through the added CPAs, as expected. The increase of 0.01ns over the balanced case in tables A.1 and A.2 is within the error margins of Design Compiler. (2)
A.1. MULTIPLIER TREE

****************************************
Report : area
Design : top_bal
Version: 2003.06
Date : Mon May 17 18:04:02 2004
****************************************

Library(s) Used:
typical (File: /softs/dkits/umc18/art_core_v2002q1v1/synopsys/typical.db)

Number of ports: 255
Number of nets: 4606
Number of cells: 3
Number of references: 3

Combinational area: 842657.875000
Noncombinational area: 0.000000
Net Interconnect area: 239.343109

Total cell area: 842680.250000
Total area: 842897.187500

Table A.7: Area report for the multiplier using a balanced compressor tree.

****************************************
Report : area
Design : top_unbal_overest_muxes2
Version: 2003.06
Date : Tue Jul 13 16:46:22 2004
****************************************

Library(s) Used:
typical (File: /softs/dkits/umc18/art_core_v2002q1v1/synopsys/typical.db)

Number of ports: 772
Number of nets: 5967
Number of cells: 14
Number of references: 14

Combinational area: 933464.375000
Noncombinational area: 0.000000
Net Interconnect area: 273.328674

Total cell area: 933437.750000
Total area: 933737.687500

Table A.8: Area report for the multiplier using a fully unbalanced compressor tree and 3 CPAs.
Table A.9: Area report for the multiplier using an *optimally unbalanced* compressor tree and 3 CPAs.
Table A.10: Power report for the multiplier using a balanced compressor tree.
Report: power
- analysis effort low
Design: top_unbal_overest_muxes2
Version: 2003.06
Date: Tue Jul 13 16:56:18 2004

Library(s) Used:
typical (File: /softs/dkits/umd/art_core_v2002qvl/synopsys/typical.db)

Operating Conditions: typical  Library: typical
Wire Load Model Mode: segmented

<table>
<thead>
<tr>
<th>Design</th>
<th>Wire Load Model</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_unbal_overest_muxes2</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>partial_prod</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>csa_unbal_overest</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_add_66_2</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_add_66_1</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_add_66_0</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_add_128_0</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_mux_any_128_1_64_5</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_mux_any_128_1_64_4</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_mux_any_128_1_64_3</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_mux_any_128_1_64_2</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_mux_any_128_1_64_1</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_mux_any_128_1_64_0</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_mux_any_256_1_128_1</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_overest_muxes2 DW01_mux_any_256_1_128_0</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
</tbody>
</table>

Global Operating Voltage = 1.8
Power-specific unit information:
Voltage Units = 1V
Capacitance Units = 1.000000pf
Time Units = ns
Dynamic Power Units = 1mW (derived from V,C,T units)
Leakage Power Units = 1nW

Cell Internal Power = 2.0264 W (68%)
Net Switching Power = 952.8122 mW (32%)
---------
Total Dynamic Power = 2.9793 W (100%)
Cell Leakage Power = 4.0308 uW

Table A.11: Power report for the multiplier using a fully unbalanced compressor tree and 3 CPAs.
### A.1. MULTIPLIER TREE

<table>
<thead>
<tr>
<th>Design</th>
<th>Wire Load Model</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>top_unbal_6_mux</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>partial_prod</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>csa_unbal_6</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_add_66_2</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_add_66_1</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_add_66_0</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_add_128_0</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_mux_any_128_1_64_5</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_mux_any_128_1_64_4</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_mux_any_128_1_64_3</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_mux_any_128_1_64_2</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_mux_any_128_1_64_1</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
<tr>
<td>top_unbal_6_mux_DW01_mux_any_128_1_64_0</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
</tbody>
</table>

Global Operating Voltage = 1.8
Power-specific unit information:
- Voltage Units = 1V
- Capacitance Units = 1.000000pf
- Time Units = 1ns
- Dynamic Power Units = 1mW (derived from V,C,T units)
- Leakage Power Units = 1nW

Cell Internal Power = 2.0165 W (68%)
Net Switching Power = 929.0254 mW (32%)
Total Dynamic Power = 2.9456 W (100%)
Cell Leakage Power = 3.9163 uW

Table A.12: Power report for the multiplier using an *optimally unbalanced* compressor tree and 3 CPAs.
Appendix A. VHDL Schematics and Reports

Report: timing
- path full
- delay max
- max_paths 1
- sort_by group

Design: decmec
Version: 2003.06
Date: Mon May 3 16:42:18 2004

Operating Conditions: typical
Library: typical
Wire Load Model Mode: segmented
Startpoint: x_f[0] (input port)
Endpoint: r_k_out[0] (output port)
Path Group: default
Path Type: max

<table>
<thead>
<tr>
<th>Des/Clust/Port</th>
<th>Wire Load Model</th>
<th>Library</th>
</tr>
</thead>
<tbody>
<tr>
<td>decmec</td>
<td>UMC18_Conservative</td>
<td>typical</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Point</th>
<th>Incr</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>input external delay</td>
<td>0.00</td>
<td>0.00 r</td>
</tr>
<tr>
<td>x_f[0] (in)</td>
<td>0.00</td>
<td>0.00 r</td>
</tr>
<tr>
<td>U77/Y (OAI2BB1X4)</td>
<td>0.11</td>
<td>0.11 r</td>
</tr>
<tr>
<td>U78/Y (MXI2X4)</td>
<td>0.10</td>
<td>0.21 f</td>
</tr>
<tr>
<td>U81/Y (MXI2X2)</td>
<td>0.08</td>
<td>0.29 r</td>
</tr>
<tr>
<td>r_k_out[0] (out)</td>
<td>0.00</td>
<td>0.29 r</td>
</tr>
<tr>
<td>data arrival time</td>
<td>0.29</td>
<td></td>
</tr>
<tr>
<td>max_delay</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>output external delay</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>data required time</td>
<td>0.00</td>
<td></td>
</tr>
</tbody>
</table>

| data required time | 0.00 |
| data arrival time | -0.29 |
| slack (VIOLATED) | -0.29 |

Table A.13: Timing report for the threshold algorithm implemented in 0.18\(\mu\) technology.

A.2 Decision Algorithm

Figure A.3 shows the schematic for an implementation of the threshold decision algorithm. As the timing report in table A.13 shows, the critical path delay is 0.29ns, and can thus fit into a single cycle. Table A.14 shows the tiny cost of this implementation, a mere 20 gates, and table A.15 shows the associated power consumption. Although this power is an estimate, it shows the very small drain associated with this logic.
Figure A.3: Schematic of the \textit{threshold} algorithm implementation in 0.18\textmu m technology. Only two multiplexors and 20 gates are needed. The critical path of 0.29ns is shown in red.
Table A.14: Area report for the threshold algorithm implemented in 0.18µ technology.
### Table A.15: Power report for the threshold algorithm implemented in 0.18µ technology.

<table>
<thead>
<tr>
<th>Description</th>
<th>Value</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cell Internal Power</td>
<td>298.3652 µW</td>
<td>35%</td>
</tr>
<tr>
<td>Net Switching Power</td>
<td>550.6424 µW</td>
<td>65%</td>
</tr>
<tr>
<td>Total Dynamic Power</td>
<td>849.0076 µW</td>
<td>100%</td>
</tr>
<tr>
<td>Cell Leakage Power</td>
<td>1.5396 nW</td>
<td></td>
</tr>
</tbody>
</table>
Appendix B

Complete Decision Algorithm Example

This appendix contains a very simple example of the inputs and outputs of the reconfiguration decision algorithms considered for the case study in chapter 5.

The example is defined by traces of integer and floating point instructions, containing the arrival time and any dependencies for each instruction.

For the balance and threshold algorithms, the values of all the state variables at each moment in time $k$ and the final ending time are shown.

In the case of the integer linear programming method, the integer linear program derived from the input traces will be displayed, followed by the optimal solution as found by CPLEX and its details. The size of the integer linear program necessary will explain the utter simplicity of the example and the complexity involved in finding a solution for the longer traces whose results are presented in section 5.3.5.

Due to the simplicity of the example, the ending time is the same for all three methods, with no differences between the theoretical optimum and the other solutions.

B.1 Example Description

The processor considered has 1 ALU and 1 FPU which can be reconfigured as 2 $x$ALUs. The FU latencies are 1, 5 and 2 cycles, respectively. There are 4 instructions in total, 2 Int and 2 FP, with one of each arriving at times 0 and 1. The Integer instructions are identified by the integer linear program with numbers 0 and 2, while the FP instructions hold the numbers 1 and 3.

The optimal solution is to keep the FPU as an FPU and process the 2 integer instructions one at a time in the ALU. This gives a total number of cycles of 7—or a finish time of 6, as the second FP instruction issues at cycle 1.
Figure B.1: Timing chart of the Decision Algorithm Example. In the first 2 cycles, an integer and a FP instruction are executed, with latencies 1 and 5 cycles, respectively. The total time is 7 cycles, as instruction 3 finishes at time 6.

### B.2 Dynamic Solution

The dynamic solutions for both balance and threshold algorithms are shown in table B.1.

### B.3 Optimal Integer Linear Programming Solution

#### B.3.1 Integer Linear Program

The linear program representing this simple example is shown in tables B.2 to B.10. The program is 496 lines long, as every constraint must be explicitly written for each cycle. The text has been edited for readability and size. There are 179 variables and 1003 constraints. Note that lines starting with a "//" are comments and are ignored by CPLEX.

Figure B.1 shows the instructions issued and their finishing times according to the optimal solution.

#### B.3.2 Optimal Result

Table B.11 shows the optimal finish time (starting from 0), thus the number of cycles is equal to the finish time + 1. Additionally, table B.12 shows the values of all the variables that produced this optimal solution.
B.3. OPTIMAL INTEGER LINEAR PROGRAMMING SOLUTION

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_a[0]) = 1</td>
<td>(u_f[0]) = 1</td>
<td>(x_a[0]) = 0</td>
<td>(x_f[0]) = 0</td>
</tr>
<tr>
<td>(y_a[0]) = 0</td>
<td>(y_f[0]) = 0</td>
<td>(r[0]) = 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_a[1]) = 1</td>
<td>(u_f[1]) = 1</td>
<td>(x_a[1]) = 1</td>
<td>(x_f[1]) = 1</td>
</tr>
<tr>
<td>(y_a[1]) = 1</td>
<td>(y_f[1]) = 0</td>
<td>(r[1]) = 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_a[2]) = 0</td>
<td>(u_f[2]) = 0</td>
<td>(x_a[2]) = 1</td>
<td>(x_f[2]) = 1</td>
</tr>
<tr>
<td>(y_a[2]) = 1</td>
<td>(y_f[2]) = 0</td>
<td>(r[2]) = 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_a[3]) = 0</td>
<td>(u_f[3]) = 0</td>
<td>(x_a[3]) = 0</td>
<td>(x_f[3]) = 0</td>
</tr>
<tr>
<td>(y_a[3]) = 0</td>
<td>(y_f[3]) = 0</td>
<td>(r[3]) = 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_a[4]) = 0</td>
<td>(u_f[4]) = 0</td>
<td>(x_a[4]) = 0</td>
<td>(x_f[4]) = 0</td>
</tr>
<tr>
<td>(y_a[4]) = 0</td>
<td>(y_f[4]) = 0</td>
<td>(r[4]) = 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_a[5]) = 0</td>
<td>(u_f[5]) = 0</td>
<td>(x_a[5]) = 0</td>
<td>(x_f[5]) = 0</td>
</tr>
<tr>
<td>(y_a[5]) = 0</td>
<td>(y_f[5]) = 1</td>
<td>(r[5]) = 0</td>
<td></td>
</tr>
</tbody>
</table>

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_a[6]) = 0</td>
<td>(u_f[6]) = 0</td>
<td>(x_a[6]) = 0</td>
<td>(x_f[6]) = 0</td>
</tr>
<tr>
<td>(y_a[6]) = 0</td>
<td>(y_f[6]) = 1</td>
<td>(r[6]) = 0</td>
<td></td>
</tr>
</tbody>
</table>

Total simulation time: 7 cycles. Simulation stopped due to empty pipeline. All instructions accounted for (2int and 2fp)

Table B.1: Dynamic trace.


\textbf{APPENDIX B. COMPLETE DECISION ALGORITHM EXAMPLE}

Creating ILP with 1 ALUs, 1 RFPUs, (4 logical FUs) 2 xALUs/RFPU, for 7 cycles and 4 instructions.

\textbf{MINIMIZE} \(T\)

\textbf{SUCH THAT}

\[ \rho(u,t) = \sum_i x_{i,u,t} \]
\[ \rho(0,0) - x(0,0,0) - x(1,0,0) - x(2,0,0) - x(3,0,0) = 0 \]
\[ \rho(0,1) - x(0,0,1) - x(1,0,1) - x(2,0,1) - x(3,0,1) = 0 \]
\[ \rho(0,2) - x(0,0,2) - x(1,0,2) - x(2,0,2) - x(3,0,2) = 0 \]
\[ \rho(0,3) - x(0,0,3) - x(1,0,3) - x(2,0,3) - x(3,0,3) = 0 \]
\[ \rho(0,4) - x(0,0,4) - x(1,0,4) - x(2,0,4) - x(3,0,4) = 0 \]
\[ \rho(0,5) - x(0,0,5) - x(1,0,5) - x(2,0,5) - x(3,0,5) = 0 \]
\[ \rho(0,6) - x(0,0,6) - x(1,0,6) - x(2,0,6) - x(3,0,6) = 0 \]
\[ \rho(1,0) - x(0,1,0) - x(1,1,0) - x(2,1,0) - x(3,1,0) = 0 \]
\[ \rho(1,1) - x(0,1,1) - x(1,1,1) - x(2,1,1) - x(3,1,1) = 0 \]
\[ \rho(1,2) - x(0,1,2) - x(1,1,2) - x(2,1,2) - x(3,1,2) = 0 \]
\[ \rho(1,3) - x(0,1,3) - x(1,1,3) - x(2,1,3) - x(3,1,3) = 0 \]
\[ \rho(1,4) - x(0,1,4) - x(1,1,4) - x(2,1,4) - x(3,1,4) = 0 \]
\[ \rho(1,5) - x(0,1,5) - x(1,1,5) - x(2,1,5) - x(3,1,5) = 0 \]
\[ \rho(1,6) - x(0,1,6) - x(1,1,6) - x(2,1,6) - x(3,1,6) = 0 \]
\[ \rho(2,0) - x(0,2,0) - x(1,2,0) - x(2,2,0) - x(3,2,0) = 0 \]
\[ \rho(2,1) - x(0,2,1) - x(1,2,1) - x(2,2,1) - x(3,2,1) = 0 \]
\[ \rho(2,2) - x(0,2,2) - x(1,2,2) - x(2,2,2) - x(3,2,2) = 0 \]
\[ \rho(2,3) - x(0,2,3) - x(1,2,3) - x(2,2,3) - x(3,2,3) = 0 \]
\[ \rho(2,4) - x(0,2,4) - x(1,2,4) - x(2,2,4) - x(3,2,4) = 0 \]
\[ \rho(2,5) - x(0,2,5) - x(1,2,5) - x(2,2,5) - x(3,2,5) = 0 \]
\[ \rho(2,6) - x(0,2,6) - x(1,2,6) - x(2,2,6) - x(3,2,6) = 0 \]
\[ \rho(3,0) - x(0,3,0) - x(1,3,0) - x(2,3,0) - x(3,3,0) = 0 \]
\[ \rho(3,1) - x(0,3,1) - x(1,3,1) - x(2,3,1) - x(3,3,1) = 0 \]
\[ \rho(3,2) - x(0,3,2) - x(1,3,2) - x(2,3,2) - x(3,3,2) = 0 \]
\[ \rho(3,3) - x(0,3,3) - x(1,3,3) - x(2,3,3) - x(3,3,3) = 0 \]
\[ \rho(3,4) - x(0,3,4) - x(1,3,4) - x(2,3,4) - x(3,3,4) = 0 \]
\[ \rho(3,5) - x(0,3,5) - x(1,3,5) - x(2,3,5) - x(3,3,5) = 0 \]
\[ \rho(3,6) - x(0,3,6) - x(1,3,6) - x(2,3,6) - x(3,3,6) = 0 \]

\begin{center}
Table B.2: Integer Linear Program (1): Statement and equations for \(\rho_{u,t}\).
\end{center}
B.3. OPTIMAL INTEGER LINEAR PROGRAMMING SOLUTION

\[
\begin{align*}
\text{t}_i &= \sum_t \sum_u x_{i,u,t} \cdot (t + l_u) \\
t(0) &= 1x(0,0,0) - 5x(0,1,0) - 2x(0,2,0) - 2x(0,3,0) - 2x(0,0,1) - 6x(0,1,1) - 3x(0,2,1) - 3x(0,3,1) - 5x(0,0,2) - 9x(0,1,2) - 5x(0,2,2) - 5x(0,3,2) - 5x(0,0,3) - 10x(0,1,3) - 6x(0,2,3) - 6x(0,3,3) - 5x(0,0,4) - 9x(0,1,4) - 6x(0,2,4) - 6x(0,3,4) - 6x(0,0,5) - 10x(0,1,5) - 7x(0,2,5) - 7x(0,3,5) - 7x(0,0,6) - 11x(0,1,6) - 8x(0,2,6) - 8x(0,3,6) = 0 \\
t(1) &= 1x(1,0,0) - 5x(1,1,0) - 2x(1,2,0) - 2x(1,3,0) - 2x(1,0,1) - 6x(1,1,1) - 3x(1,2,1) - 3x(1,3,1) - 3x(1,0,2) - 7x(1,1,2) - 4x(1,2,2) - 4x(1,3,2) - 4x(1,0,3) - 8x(1,1,3) - 5x(1,2,3) - 5x(1,3,3) - 5x(1,0,4) - 9x(1,1,4) - 6x(1,2,4) - 6x(1,3,4) - 6x(1,0,5) - 10x(1,1,5) - 7x(1,2,5) - 7x(1,3,5) - 7x(1,0,6) - 11x(1,1,6) - 8x(1,2,6) - 8x(1,3,6) = 0 \\
t(2) &= 1x(2,0,0) - 5x(2,1,0) - 2x(2,2,0) - 2x(2,3,0) - 2x(2,0,1) - 6x(2,1,1) - 3x(2,2,1) - 3x(2,3,1) - 3x(2,0,2) - 7x(2,1,2) - 4x(2,2,2) - 4x(2,3,2) - 4x(2,0,3) - 8x(2,1,3) - 5x(2,2,3) - 5x(2,3,3) - 5x(2,0,4) - 9x(2,1,4) - 6x(2,2,4) - 6x(2,3,4) - 6x(2,0,5) - 10x(2,1,5) - 7x(2,2,5) - 7x(2,3,5) - 7x(2,0,6) - 11x(2,1,6) - 8x(2,2,6) - 8x(2,3,6) = 0 \\
t(3) &= 1x(3,0,0) - 5x(3,1,0) - 2x(3,2,0) - 2x(3,3,0) - 2x(3,0,1) - 6x(3,1,1) - 3x(3,2,1) - 3x(3,3,1) - 3x(3,0,2) - 7x(3,1,2) - 4x(3,2,2) - 4x(3,3,2) - 4x(3,0,3) - 8x(3,1,3) - 5x(3,2,3) - 5x(3,3,3) - 5x(3,0,4) - 9x(3,1,4) - 6x(3,2,4) - 6x(3,3,4) - 6x(3,0,5) - 10x(3,1,5) - 7x(3,2,5) - 7x(3,3,5) - 7x(3,0,6) - 11x(3,1,6) - 8x(3,2,6) - 8x(3,3,6) = 0 \\
\end{align*}
\]

Table B.3: Integer Linear Program (2): Equations for \(t_i\).
\begin{verbatim}
\T \geq t_i 
T - t(0) \geq 0 
T - t(1) \geq 0 
T - t(2) \geq 0 
T - t(3) \geq 0 
\hline
\rho_{u,t} \leq 1 \text{ for all } u,t 
\rho(0,0) \leq 1 
\rho(0,1) \leq 1 
\rho(0,2) \leq 1 
\rho(0,3) \leq 1 
\rho(0,4) \leq 1 
\rho(0,5) \leq 1 
\rho(0,6) \leq 1 
\rho(1,0) \leq 1 
\rho(1,1) \leq 1 
\rho(1,2) \leq 1 
\rho(1,3) \leq 1 
\rho(1,4) \leq 1 
\rho(1,5) \leq 1 
\rho(1,6) \leq 1 
\rho(2,0) \leq 1 
\rho(2,1) \leq 1 
\rho(2,2) \leq 1 
\rho(2,3) \leq 1 
\rho(2,4) \leq 1 
\rho(2,5) \leq 1 
\rho(2,6) \leq 1 
\rho(3,0) \leq 1 
\rho(3,1) \leq 1 
\rho(3,2) \leq 1 
\rho(3,3) \leq 1 
\rho(3,4) \leq 1 
\rho(3,5) \leq 1 
\rho(3,6) \leq 1 
\end{verbatim}

Table B.4: Integer Linear Program (3): Constraints on \( T \) and \( \rho_{u,t} \).
\[ \sum_{u,t} x_{i,u,t} = 1 \text{ for all } i \]

\[
x(0,0,0) + x(0,0,1) + x(0,0,2) + x(0,0,3) + x(0,0,4) + x(0,0,5) + x(0,0,6) +
x(0,1,0) + x(0,1,1) + x(0,1,2) + x(0,1,3) + x(0,1,4) + x(0,1,5) + x(0,1,6) +
x(0,2,0) + x(0,2,1) + x(0,2,2) + x(0,2,3) + x(0,2,4) + x(0,2,5) + x(0,2,6) +
x(0,3,0) + x(0,3,1) + x(0,3,2) + x(0,3,3) + x(0,3,4) + x(0,3,5) + x(0,3,6) +
x(0,3,6) = 1
\]

\[
x(1,0,0) + x(1,0,1) + x(1,0,2) + x(1,0,3) + x(1,0,4) + x(1,0,5) + x(1,0,6) +
x(1,1,0) + x(1,1,1) + x(1,1,2) + x(1,1,3) + x(1,1,4) + x(1,1,5) + x(1,1,6) +
x(1,2,0) + x(1,2,1) + x(1,2,2) + x(1,2,3) + x(1,2,4) + x(1,2,5) + x(1,2,6) +
x(1,3,0) + x(1,3,1) + x(1,3,2) + x(1,3,3) + x(1,3,4) + x(1,3,5) + x(1,3,6) = 1
\]

\[
x(2,0,0) + x(2,0,1) + x(2,0,2) + x(2,0,3) + x(2,0,4) + x(2,0,5) + x(2,0,6) +
x(2,1,0) + x(2,1,1) + x(2,1,2) + x(2,1,3) + x(2,1,4) + x(2,1,5) + x(2,1,6) +
x(2,2,0) + x(2,2,1) + x(2,2,2) + x(2,2,3) + x(2,2,4) + x(2,2,5) + x(2,2,6) +
x(2,3,0) + x(2,3,1) + x(2,3,2) + x(2,3,3) + x(2,3,4) + x(2,3,5) + x(2,3,6) = 1
\]

\[
x(3,0,0) + x(3,0,1) + x(3,0,2) + x(3,0,3) + x(3,0,4) + x(3,0,5) + x(3,0,6) +
x(3,1,0) + x(3,1,1) + x(3,1,2) + x(3,1,3) + x(3,1,4) + x(3,1,5) + x(3,1,6) +
x(3,2,0) + x(3,2,1) + x(3,2,2) + x(3,2,3) + x(3,2,4) + x(3,2,5) + x(3,2,6) +
x(3,3,0) + x(3,3,1) + x(3,3,2) + x(3,3,3) + x(3,3,4) + x(3,3,5) + x(3,3,6) = 1
\]

Table B.5: Integer Linear Program (4): Constraints on the \( x_{i,u,t} \).

\[
\sum_{i,u} x_{i,u,t} = I_w \text{ for all } t
\]

\[
x(0,0,0) + x(0,0,1) + x(0,0,2) + x(1,0,0) + x(1,0,1) + x(1,0,2) +
x(1,3,0) + x(2,0,0) + x(2,0,1) + x(2,0,2) + x(2,0,3) + x(2,0,4) + x(2,0,5) +
x(2,0,6) +
x(3,0,0) + x(3,0,1) + x(3,0,2) + x(3,0,3) + x(3,0,4) + x(3,0,5) + x(3,0,6) +
x(3,1,0) + x(3,1,1) + x(3,1,2) + x(3,1,3) + x(3,1,4) + x(3,1,5) + x(3,1,6) +
x(3,2,0) + x(3,2,1) + x(3,2,2) + x(3,2,3) + x(3,2,4) + x(3,2,5) + x(3,2,6) +
x(3,3,0) + x(3,3,1) + x(3,3,2) + x(3,3,3) + x(3,3,4) + x(3,3,5) + x(3,3,6) = 1
\]

Table B.6: Integer Linear Program (5): Constraints on the issue width.
\( \rho(u,t) \leq x_i(u,s), s=t, \ldots , t+l(u) \)
\( \rho(1,0) - x(1,0) \leq 0 \rho(1,0) - x(1,1) \leq 0 \)
\( \rho(1,0) - x(1,2) \leq 0 \rho(1,0) - x(1,3) \leq 0 \)
\( \rho(1,0) - x(1,4) \leq 0 \rho(2,0) - x(2,0) \leq 0 \)
\( \rho(2,0) - x(2,1) \leq 0 \rho(3,0) - x(3,0) \leq 0 \)
\( \rho(3,0) - x(3,1) \leq 0 \)
\( \rho(1,1) - x(1,1) \leq 0 \rho(1,1) - x(1,2) \leq 0 \)
\( \rho(1,1) - x(1,3) \leq 0 \rho(1,1) - x(1,4) \leq 0 \)
\( \rho(1,1) - x(1,5) \leq 0 \rho(2,1) - x(2,1) \leq 0 \)
\( \rho(2,1) - x(2,2) \leq 0 \rho(3,1) - x(3,1) \leq 0 \)
\( \rho(3,1) - x(3,2) \leq 0 \)
\( \rho(1,2) - x(1,2) \leq 0 \rho(1,2) - x(1,3) \leq 0 \)
\( \rho(1,2) - x(1,4) \leq 0 \rho(1,2) - x(1,5) \leq 0 \)
\( \rho(1,2) - x(1,6) \leq 0 \rho(2,2) - x(2,2) \leq 0 \)
\( \rho(2,2) - x(2,3) \leq 0 \rho(3,2) - x(3,2) \leq 0 \)
\( \rho(3,2) - x(3,3) \leq 0 \)
\( \rho(1,3) - x(1,3) \leq 0 \rho(1,3) - x(1,4) \leq 0 \)
\( \rho(1,3) - x(1,5) \leq 0 \rho(1,3) - x(1,6) \leq 0 \)
\( \rho(1,3) - x(1,7) \leq 0 \rho(2,3) - x(2,3) \leq 0 \)
\( \rho(2,3) - x(2,4) \leq 0 \rho(3,3) - x(3,3) \leq 0 \)
\( \rho(3,3) - x(3,4) \leq 0 \)
\( \rho(1,4) - x(1,4) \leq 0 \rho(1,4) - x(1,5) \leq 0 \)
\( \rho(1,4) - x(1,6) \leq 0 \rho(1,4) - x(1,7) \leq 0 \)
\( \rho(1,4) - x(1,8) \leq 0 \rho(2,4) - x(2,4) \leq 0 \)
\( \rho(2,4) - x(2,5) \leq 0 \rho(3,4) - x(3,4) \leq 0 \)
\( \rho(3,4) - x(3,5) \leq 0 \)
\( \rho(1,5) - x(1,5) \leq 0 \rho(1,5) - x(1,6) \leq 0 \)
\( \rho(1,5) - x(1,7) \leq 0 \rho(1,5) - x(1,8) \leq 0 \)
\( \rho(1,5) - x(1,9) \leq 0 \rho(2,5) - x(2,5) \leq 0 \)
\( \rho(2,5) - x(2,6) \leq 0 \rho(3,5) - x(3,5) \leq 0 \)
\( \rho(3,5) - x(3,6) \leq 0 \)
\( \rho(1,6) - x(1,6) \leq 0 \rho(1,6) - x(1,7) \leq 0 \)
\( \rho(1,6) - x(1,8) \leq 0 \rho(1,6) - x(1,9) \leq 0 \)
\( \rho(1,6) - x(1,10) \leq 0 \rho(2,6) - x(2,6) \leq 0 \)
\( \rho(2,6) - x(2,7) \leq 0 \rho(3,6) - x(3,6) \leq 0 \)
\( \rho(3,6) - x(3,7) \leq 0 \)

Table B.7: Integer Linear Program (6): Constraints linking the executability \( \rho_{u,t} \) to the availability \( \xi_{u,t} \).
B.3. OPTIMAL INTEGER LINEAR PROGRAMMING SOLUTION

\[ \alpha \ast \xi(u_n,t) + \sum_{\upsilon = u_n+1}^{u_n + \alpha} \xi(\upsilon,t) = \alpha \]
\[ 2\xi(1,0) + \xi(2,0) + \xi(3,0) = 2 \]
\[ 2\xi(1,1) + \xi(2,1) + \xi(3,1) = 2 \]
\[ 2\xi(1,2) + \xi(2,2) + \xi(3,2) = 2 \]
\[ 2\xi(1,3) + \xi(2,3) + \xi(3,3) = 2 \]
\[ 2\xi(1,4) + \xi(2,4) + \xi(3,4) = 2 \]
\[ 2\xi(1,5) + \xi(2,5) + \xi(3,5) = 2 \]
\[ 2\xi(1,6) + \xi(2,6) + \xi(3,6) = 2 \]

\[ \xi(u,t) = 1 \text{ for all } t \text{ if } u \text{ in ALU} \]
\[ \xi(0,0) = 1 \]
\[ \xi(0,1) = 1 \]
\[ \xi(0,2) = 1 \]
\[ \xi(0,3) = 1 \]
\[ \xi(0,4) = 1 \]
\[ \xi(0,5) = 1 \]
\[ \xi(0,6) = 1 \]

---

\[ \text{Dependency constraints} \]
\[ \text{INTInstruction } x(0,u,t) \text{ has no dependencies} \]
\[ \text{INTInstruction } x(2,u,t) \text{ has no dependencies} \]
\[ \text{FPInstruction } x(1,u,t) \text{ has no dependencies} \]
\[ \text{FPInstruction } x(3,u,t) \text{ has no dependencies} \]

Table B.8: Integer Linear Program (7): Constraints governing availability and dependencies between instructions.

<table>
<thead>
<tr>
<th>Arrival and Instruction type constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>x(2,0,0) = 0 x(2,1,0) = 0 x(2,2,0) = 0</td>
</tr>
<tr>
<td>x(2,3,0) = 0 x(0,1,0) = 0 x(0,1,1) = 0</td>
</tr>
<tr>
<td>x(0,1,2) = 0 x(0,1,3) = 0 x(0,1,4) = 0</td>
</tr>
<tr>
<td>x(0,1,5) = 0 x(0,1,6) = 0 x(2,1,0) = 0</td>
</tr>
<tr>
<td>x(2,1,1) = 0 x(2,1,2) = 0 x(2,1,3) = 0</td>
</tr>
<tr>
<td>x(2,1,4) = 0 x(2,1,5) = 0 x(2,1,6) = 0</td>
</tr>
<tr>
<td>x(3,0,0) = 0 x(3,1,0) = 0 x(3,2,0) = 0</td>
</tr>
<tr>
<td>x(3,3,0) = 0 x(1,0,0) = 0 x(1,0,1) = 0</td>
</tr>
<tr>
<td>x(1,0,2) = 0 x(1,0,3) = 0 x(1,0,4) = 0</td>
</tr>
<tr>
<td>x(1,0,5) = 0 x(1,0,6) = 0 x(1,2,0) = 0</td>
</tr>
<tr>
<td>x(1,2,1) = 0 x(1,2,2) = 0 x(1,2,3) = 0</td>
</tr>
<tr>
<td>x(1,2,4) = 0 x(1,2,5) = 0 x(1,2,6) = 0</td>
</tr>
<tr>
<td>x(1,3,0) = 0 x(1,3,1) = 0 x(1,3,2) = 0</td>
</tr>
<tr>
<td>x(1,3,3) = 0 x(1,3,4) = 0 x(1,3,5) = 0</td>
</tr>
<tr>
<td>x(1,3,6) = 0 x(3,0,0) = 0 x(3,0,1) = 0</td>
</tr>
<tr>
<td>x(3,0,2) = 0 x(3,0,3) = 0 x(3,0,4) = 0</td>
</tr>
<tr>
<td>x(3,0,5) = 0 x(3,0,6) = 0 x(3,2,0) = 0</td>
</tr>
<tr>
<td>x(3,2,1) = 0 x(3,2,2) = 0 x(3,2,3) = 0</td>
</tr>
<tr>
<td>x(3,2,4) = 0 x(3,2,5) = 0 x(3,2,6) = 0</td>
</tr>
<tr>
<td>x(3,3,0) = 0 x(3,3,1) = 0 x(3,3,2) = 0</td>
</tr>
<tr>
<td>x(3,3,3) = 0 x(3,3,4) = 0 x(3,3,5) = 0</td>
</tr>
<tr>
<td>x(3,3,6) = 0</td>
</tr>
</tbody>
</table>

Table B.9: Integer Linear Program (8): Constraints controlling the arrivals and types of instructions.
Table B.10: Integer Linear Program (9): Declaration that \( T \) and all \( t_i \) are integer, and that all other variables are binary—i.e., boolean.

Table B.11: Integer Linear Program Search Result. The last instruction finishes at time 6, so the optimal solution takes 7 cycles.
### B.3. Optimal Integer Linear Programming Solution

#### Table B.12: Integer Linear Program Solution.

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Solution Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>6.000000</td>
</tr>
<tr>
<td>rho(0,0)</td>
<td>1.000000</td>
</tr>
<tr>
<td>x(0,0,0)</td>
<td>1.000000</td>
</tr>
<tr>
<td>rho(0,1)</td>
<td>1.000000</td>
</tr>
<tr>
<td>x(2,0,1)</td>
<td>1.000000</td>
</tr>
<tr>
<td>rho(1,0)</td>
<td>1.000000</td>
</tr>
<tr>
<td>x(1,1,0)</td>
<td>1.000000</td>
</tr>
<tr>
<td>rho(1,1)</td>
<td>1.000000</td>
</tr>
<tr>
<td>x(3,1,1)</td>
<td>1.000000</td>
</tr>
<tr>
<td>t(0)</td>
<td>1.000000</td>
</tr>
<tr>
<td>t(1)</td>
<td>5.000000</td>
</tr>
<tr>
<td>t(2)</td>
<td>2.000000</td>
</tr>
<tr>
<td>t(3)</td>
<td>6.000000</td>
</tr>
<tr>
<td>xi(1,0)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(1,1)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(1,2)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(1,3)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(1,4)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(1,5)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(2,6)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(3,6)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(0,0)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(0,1)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(0,2)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(0,3)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(0,4)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(0,5)</td>
<td>1.000000</td>
</tr>
<tr>
<td>xi(0,6)</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

All other variables in the range 1-179 are zero.
Appendix C

Simplescalar Configurations

This appendix contains the configuration files used for all the simplescalar simulations. Tables C.2 to C.5 contain the configuration for the baseline mainstream model in table 6.2, while table C.6 shows only the changes related to the dynamic reconfiguration at the end of the file (table C.5).

The Simpoints, based on [79] and obtained from [145], and used for the main simulations of chapter 6, are reproduced in table C.1. These values were used for the -max:inst and -fastfwd commands instead of the ones in table C.2, which were used for the sensitivity analysis simulations in section 6.6.
Table C.1: Single standard simulation points, for 100 · 10^6 instruction intervals on Simplescalar using the Alpha ISA. The name column shows the name of the benchmark, and, in the cases where more than one input data set is specified by SPEC, the name of that input. SimPoint is the number of the interval to simulate, with interval numbers starting at 0. PC is the program counter where detailed simulation should start. Proc Name is the name of the procedure containing the code to be simulated, and xBB is the number of times the PC must be passed before starting simulation. The last three columns provide a more precise, but also more complex, way to define the Simpoints than simply fast-forwarding a set number of instructions. This is due to the somewhat variable length of system calls.

<table>
<thead>
<tr>
<th>name</th>
<th>SimPoint</th>
<th>PC</th>
<th>Proc Name</th>
<th>xBB</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp-ref</td>
<td>109</td>
<td>0x120026834</td>
<td>mm_fv_update_nonbon</td>
<td>744016</td>
</tr>
<tr>
<td>appli-ref</td>
<td>2180</td>
<td>0x120018520</td>
<td>buts</td>
<td>66533185</td>
</tr>
<tr>
<td>api-b-ref</td>
<td>3409</td>
<td>0x1200380ac</td>
<td>dcdtxf</td>
<td>60373732</td>
</tr>
<tr>
<td>art-110</td>
<td>341</td>
<td>0x12000f6b0</td>
<td>match</td>
<td>13958000</td>
</tr>
<tr>
<td>art-470</td>
<td>306</td>
<td>0x12006f560</td>
<td>match</td>
<td>15207000</td>
</tr>
<tr>
<td>equske-ref</td>
<td>813</td>
<td>0x120010240</td>
<td>phi0</td>
<td>20826187</td>
</tr>
<tr>
<td>facerc-ref</td>
<td>376</td>
<td>0x12002d1f4</td>
<td>$graphroutines$localmove</td>
<td>14827866</td>
</tr>
<tr>
<td>fma3d-ref</td>
<td>2542</td>
<td>0x1200e3140</td>
<td>scater_element_nodal_forces</td>
<td>117912000</td>
</tr>
<tr>
<td>galgel-ref</td>
<td>2492</td>
<td>0x12002db00</td>
<td>syshtn</td>
<td>954005882</td>
</tr>
<tr>
<td>lucas-ref</td>
<td>546</td>
<td>0x120021e0</td>
<td>ff_square</td>
<td>12058624</td>
</tr>
<tr>
<td>mesa-ref</td>
<td>1136</td>
<td>0x1200a3000</td>
<td>general_textured_triangle</td>
<td>35674746</td>
</tr>
<tr>
<td>mgrid-ref</td>
<td>3293</td>
<td>0x1200160b0</td>
<td>resid</td>
<td>192931860</td>
</tr>
<tr>
<td>sixtrack-ref</td>
<td>3044</td>
<td>0x120017894</td>
<td>thinst</td>
<td>5.01E+009</td>
</tr>
<tr>
<td>swim-ref</td>
<td>2080</td>
<td>0x120019130</td>
<td>calcl</td>
<td>66220853</td>
</tr>
<tr>
<td>wapwine-ref</td>
<td>3238</td>
<td>0x12001d6b0</td>
<td>zpmnm</td>
<td>144152442</td>
</tr>
<tr>
<td>bzip2-graphic-ref</td>
<td>719</td>
<td>0x120012a5c</td>
<td>spec_putc</td>
<td>105067386</td>
</tr>
<tr>
<td>bzip2-program-ref</td>
<td>459</td>
<td>0x12000d0d0</td>
<td>sortlt</td>
<td>58487615</td>
</tr>
<tr>
<td>buts-ref</td>
<td>978</td>
<td>0x120067744</td>
<td>gSort3</td>
<td>38227809</td>
</tr>
<tr>
<td>crafty-ref</td>
<td>775</td>
<td>0x120021730</td>
<td>SwapXray</td>
<td>4859567</td>
</tr>
<tr>
<td>eon-rushmeier-ref</td>
<td>404</td>
<td>0x12004e1b4</td>
<td>viewingHit</td>
<td>247289484</td>
</tr>
<tr>
<td>gpa-ref</td>
<td>675</td>
<td>0x120057050</td>
<td>CollectGarb</td>
<td>93060237</td>
</tr>
<tr>
<td>gcc-00-166-ref</td>
<td>390</td>
<td>0x1200d157c</td>
<td>gen_rtx</td>
<td>653108</td>
</tr>
<tr>
<td>gcc-00-200-ref</td>
<td>737</td>
<td>0x1200e6b0</td>
<td>refers_to_regno_p</td>
<td>11774021</td>
</tr>
<tr>
<td>gcc-00-expr-ref</td>
<td>37</td>
<td>0x1201916b0</td>
<td>validate_change</td>
<td>4890037</td>
</tr>
<tr>
<td>gcc-00-integrate-ref</td>
<td>5</td>
<td>0x1201198e0</td>
<td>find_single_use_in_loop</td>
<td>18527</td>
</tr>
<tr>
<td>gcc-00-scall-ref</td>
<td>208</td>
<td>0x120100e54</td>
<td>insert</td>
<td>1364643</td>
</tr>
<tr>
<td>gzip-graphic-ref</td>
<td>654</td>
<td>0x12009c00</td>
<td>fill_window</td>
<td>51781632</td>
</tr>
<tr>
<td>gzip-log-ref</td>
<td>266</td>
<td>0x1200628b0</td>
<td>inflate_codes</td>
<td>3790025</td>
</tr>
<tr>
<td>gzip-program-ref</td>
<td>1190</td>
<td>0x120009660</td>
<td>longest_match</td>
<td>390710000</td>
</tr>
<tr>
<td>gzip-random-ref</td>
<td>624</td>
<td>0x1200014c</td>
<td>deflate</td>
<td>125799992</td>
</tr>
<tr>
<td>gzip-source-ref</td>
<td>335</td>
<td>0x1200ea224</td>
<td>deflate</td>
<td>15436691</td>
</tr>
<tr>
<td>mcf-ref</td>
<td>554</td>
<td>0x12000911c</td>
<td>price_out_impl</td>
<td>947090000</td>
</tr>
<tr>
<td>parser-ref</td>
<td>1147</td>
<td>0x12001edc</td>
<td>region_valid</td>
<td>13246540</td>
</tr>
<tr>
<td>perlhk-diffmail-ref</td>
<td>142</td>
<td>0x1200797e4</td>
<td>recompact</td>
<td>43446291</td>
</tr>
<tr>
<td>perlhk-makerand-ref</td>
<td>12</td>
<td>0x12008268c</td>
<td>Perl_runops_standard</td>
<td>14706905</td>
</tr>
<tr>
<td>perlhk-perfect-ref</td>
<td>6</td>
<td>0x12008268c</td>
<td>Perl_runops_standard</td>
<td>2928227</td>
</tr>
<tr>
<td>perlhk-splitmail-ref</td>
<td>451</td>
<td>0x12007f9b8</td>
<td>recompact</td>
<td>94451770</td>
</tr>
<tr>
<td>twdf-ref</td>
<td>1067</td>
<td>0x120041094</td>
<td>ucxsc1</td>
<td>12839464</td>
</tr>
<tr>
<td>vortex-one-ref</td>
<td>272</td>
<td>0x12006289c</td>
<td>Mem_GetWord</td>
<td>73809217</td>
</tr>
<tr>
<td>vortex-three-ref</td>
<td>565</td>
<td>0x1200336b8</td>
<td>Part_Delete</td>
<td>32672892</td>
</tr>
<tr>
<td>vortex-two-ref</td>
<td>1025</td>
<td>0x12005e66c</td>
<td>Mem_NewRegion</td>
<td>628896</td>
</tr>
<tr>
<td>vpr-route-ref</td>
<td>477</td>
<td>0x120025c80</td>
<td>get_heap_head</td>
<td>841083335</td>
</tr>
</tbody>
</table>
Table C.2: *Baseline mainstream* configuration file for Simplescalar. (1)
# extra branch mis-prediction latency
-fetch:mplat 3

# speed of front-end of machine relative to execution core
-fetch:speed 1

# optimistic misfetch recovery
-fetch:mf_compat false

# branch predictor type {nottaken|taken|perfect|bimod|2lev|comb}
-bpred comb

# bimodal predictor config (<table size>)
-bpred:bimod 2048

# 2-level predictor config (<l1size> <l2size> <hist_size> <xor>)
-bpred:2lev 1 1024 8 0

# combining predictor config (<meta_table_size>)
-bpred:comb 1024

# return address stack size (0 for no return stack)
-bpred:ras 32

# BTB config (<num_sets> <associativity>)
-bpred:btb 512 4

# speculative predictors update in {ID|WB} (default non-spec)
-bpred:spec_update ID

# instruction decode B/W (insts/cycle)
-decode:width 8

# instruction issue B/W (insts/cycle)
-issue:width 8

# run pipeline with in-order issue
-issue:inorder false

# issue instructions down wrong execution paths
-issue:wrongpath true

# instruction commit B/W (insts/cycle)
-commit:width 8

# register update unit (RUU) size
-ruu:size 256

# load/store queue (LSQ) size
-lsq:size 32

# l1 data cache config, i.e., {{config}|none}
-cache:dl1 dl1:512:64:4:1

# l1 data cache hit latency (in cycles)
-cache:dl1lat 1

Table C.3: Baseline mainstream configuration file for Simplescalar. (2)
# 12 data cache config, i.e., {<config>|none}
-cache:dl2 ul2:2048:128:8:1

# 12 data cache hit latency (in cycles)
-cache:dl2lat 5

# 11 inst cache config, i.e., {<config>|dl1|dl2|none}
-cache:il1 il1:512:64:4:1

# 11 instruction cache hit latency (in cycles)
-cache:il1lat 1

# 12 instruction cache config, i.e., {<config>|dl2|none}
-cache:il2 dl2

# 12 instruction cache hit latency (in cycles)
-cache:il2lat 5

# flush caches on system calls
-cache:flush false

# convert 64-bit inst addresses to 32-bit inst equivalents
-cache:icompress false

# memory access latency (<first_chunk> <inter_chunk>)
-mem:lat 100 10

# memory access bus width (in bytes)
-mem:width 16

# instruction TLB config, i.e., {<config>|none}
-tlb:itlb itlb:64:4096:64:1

# data TLB config, i.e., {<config>|none}
-tlb:dtlb dtlb:64:4096:64:1

# inst/data TLB miss latency (in cycles)
-tlb:lat 30

# total number of integer ALU's available
-res:ialu 3

# total number of memory system ports available (to CPU)
-res:memport 4

# total number of floating point multiplier/dividers available
-res:gpfp 2

# total number of extra (programmable) integer ALU's available
-res:xialu 0

Table C.4: Baseline mainstream configuration file for Simplescalar. (3)
APPENDIX C. SIMPLESCALAR CONFIGURATIONS

# Use Dynamic reconfiguration?
-use_dyn_fu false

# how many IALUs per GPFPU reconfigured
-res: dyn_fu_factor 1

# XIALU latencies (<operation latency> <issue latency>)
-xialu:lat 2 1

# DynFU reconfiguration latency (in addition to wait until free)
-dyn_fu:lat 0

# FPU latencies (<operation latency> <issue latency>)
-gpfpu:lat 4 1

# FPU latencies (<operation latency> <issue latency>)
-gpfpu:mul_lat 4 1

# profile stat(s) against text addr's (mult uses ok)
-pcstat <null>

# operate in backward-compatible bugs mode (for testing only)
-bugcompat false

Table C.5: Baseline mainstream configuration file for Simplescalar. (4)

# Use Dynamic reconfiguration?
-use_dyn_fu true

# how many IALUs per GPFPU reconfigured
-res:dyn_fu_factor 4

# XIALU latencies (<operation latency> <issue latency>)
-xialu:lat 2 1

# DynFU reconfiguration latency (in addition to wait until free)
-dyn_fu:lat 0

# FPU latencies (<operation latency> <issue latency>)
-gpfpu:lat 4 1

# FPU latencies (<operation latency> <issue latency>)
-gpfpu:mul_lat 5 1

Table C.6: Differences in the dynamic mainstream configuration file for Simplescalar compared to the baseline mainstream configuration.
# Use Dynamic reconfiguration ?
-use_dyn_fu true

# how many IALUs per GPFPU reconfigured
-res:dyn_fu_factor 3

# XIALU latencies (<operation latency> <issue latency>)
-xialu:lat 2 1

# DynFU reconfiguration latency (in addition to wait until free)
-dyn_fu:lat 0

# FPU latencies (<operation latency> <issue latency>)
-gpfpu:lat 4 1
-gpfpu:mul_lat 4 1

Table C.7: Differences in the optimal dynamic mainstream configuration file for Simplescalar compared to the baseline mainstream configuration.
Appendix D

List of Acronyms

**ADD** Processor instruction to execute an ADDition.

**ADSL** Asymmetric Digital Subscriber Line.

**ALU** Arithmetic and Logic Unit.

**ASIC** Application Specific Integrated Circuit.

**ATM** Asynchronous Transfer Mode.

**BAS** Broadband Access Server.

**BSD** Berkeley Software Distribution.

**C** A programming language.

**C++** Another programming language, based on C.

**CISC** Complex Instruction Set Computer.

**CMOS** Complementary Metal Oxide Semiconductor.

**CPA** Carry Propagate Adder.

**CPLD** Complex Programmable Logic Device.

**CPU** Central Processing Unit.

**CSA** Carry Save Adder.

**DIV** Processor instruction to execute a DIVision.

**DSL** Digital Subscriber Line.

**DSLAM** DSL Access Multiplexer.

**DSP** Digital Signal Processor.
APPENDIX D. LIST OF ACRONYMS

**DW** DesignWare.

**EEMBC** Embedded Microprocessor Benchmark Consortium.

**EPIC** Explicitly Parallel Computing (another name for VLIW).

**FFT** Fast Fourier Transform.

**FP** Floating Point.

**FPGA** Field Programmable Gate Array.

**FPU** Floating Point Unit.

**FU** Functional Unit.

**GPP** General Purpose Processor.

**I/O** Input/Output.

**IA-64** Intel Architecture-64 (bits).

**IC** Integrated Circuit.

**IEEE** Institute of Electrical and Electronics Engineers.

**ILP** Instruction Level Parallelism.

**IP** Internet Protocol (actually version 4).

**IPC** Instructions Per Cycle.

**IPv6** Internet Protocol, version 6.

**ISA** Instruction Set Architecture.

**ISP** Internet Service Provider.

**JTAG** Joint Test Action Group.

**L2TP** Layer 2 Tunneling Protocol.

**LNS** L2TP Network Server.

**LSU** Load/Store Unit.

**MIP** Mixed Integer Program.

**MMX** Multimedia Extensions.

**MOS** Metal Oxide Semiconductor.

**MSB** Most Significant Bit.
MUL Processor instruction to execute a MULtiplication.

NOP No OPeration.

PC Personal Computer.

PLD Programmable Logic Device.

RADIUS Remote Authentication Dial-In User Service.

RFPU Reconfigurable Floating Point Unit.

RISC Reduced Instruction Set Computer.

ROM Read-Only Memory.

SMT Simultaneous MultiThread.


SUB Processor instruction to execute a SUBtraction.

TCP Transmission Control Protocol.

TLP Thread Level Parallelism.

UDP User Datagram Protocol

UMC United Microelectronics Corporation.

VHDL VLSI Hardware Description Language.

VLIW Very Long Instruction Word.

VLSI Very Large Scale Integration.

VPN Virtual Private Network.

xALU extra ALU.
Marc EPALZA
Chemin Bizot 4
1208 Genève
(022) 346 48 69
epalza@capp.ch

Né le 27-02-1976
Célibataire
Nationalités suisse, britannique et espagnole

Ingénieur R&D micro-électronique/informatique

Profil Personnel
Travaillant avec passion dans un métier que j’aime, je termine une thèse dans le domaine de l’architecture des processeurs. Une collaboration de deux ans avec une entreprise de design de circuits intégrés télécom m’a permis d’acquérir une certaine expérience du travail en entreprise et des outils. Un fort intérêt personnel pour l’informatique m’a aussi permis d’augmenter mes connaissances dans ce domaine.

Personne consciencieuse et de confiance, créative et efficace, rapide et responsable, j’aime analyser les problèmes avec rigueur pour trouver des solutions innovantes et réalistes. Doué d’un esprit d’initiative et de synthèse, j’adore relever de nouveaux défis.

Mes compétences et intérêts me poussent à exercer des fonctions de R&D ou de chef de projet dans les domaines de la haute technologie et de l’informatique. Je m’intéresse aussi beaucoup aux aspects non techniques de l’industrie, tels que la gestion et la finance. Mon expérience académique me donne aussi un penchant pour la formation.

Compétences-clés

- Architecture des ordinateurs
- Technologies internet
- Architecture des processeurs
- Télécommunications
- Programmation scientifique
- Informatique

Expérience Professionnelle
Assistant Doctorant au LTS3 (Laboratoire de Traitement des Signaux 3, ancien LSI)
Fin de thèse sur l’architecture des processeurs.

2001-2003 TRANSWITCH SA, Ecublens (Circuits intégrés pour les télécommunications)
Membre Assistant du Staff Technique, continuation de la thèse dans un milieu industriel. Travail sur l’intégration de processeurs dans des circuits télécoms, des points de vue matériel et logiciel.

2000-2000 ECOLE POLYTECHNIQUE FÉDÉRALE, Lausanne (EPFL)
Assistant au LSI (Laboratoire de Systèmes Intégrés)

Formation
2000 Diplôme d'ingénieur électricien EPF, orientation micro-électronique et traitement numérique des signaux.
ECOLE POLYTECHNIQUE FÉDÉRALE, Lausanne (EPFL)
Projet de diplôme chez Motorola (Genève) – Vérification formelle des circuits intégrés (« Model checking »).

Informatique
Connaissances hardware (PC, stations) et software (Windows, Unix/Linux), bureautique (MS Office, FrameMaker) et programmation scientifique C, C++, ASM).

Langues
Français : Langue maternelle
Espagnol : Courant
Anglais : Bilingue
Allemand : Moyen
Italien : Bonne compréhension
Japonais : Débutant
Annexe au curriculum vitae de Marc Epalza

Connaissances Techniques

**Informatique**

Assemblage, réparation et modification de PCs
Programmation de logiciels embarqués

Systèmes d’exploitation : Windows, Unix/Linux
Langages de programmation : C, C++, SystemC, assembleur RISC, Pascal, Basic
Langages de scripts : Unix/Linux shell scripting, makefiles
Bureautique : Connaissance générale de MS Office, LaTeX
Méthodologie : UML, GNU & outils « open source »

**Microélectronique**

HDLs : VHDL, Verilog
Architecture des processeurs (x86, RISC, VLIW)
Architecture des ordinateurs
Architecture d’un système basé sur les microprocesseurs pour le domaine des télécommunications.
Architecture des logiciels embarqués pour ce système.

**Stages et Projets de semestre**

Optimisation des processeurs au niveau des jeux d'instruction et des flots de données.
1999: 10e place au concours de programmation de l’ACM (Association for Computing Machinery) à Freiburg (D).
1999 : Consultant en informatique à l’ONU (Genève) – Recyclage d’ordinateurs pour les programmes d’aide à Tchernobyl
Stage : programmation pour un CD multimédia sur la Perse, Poesis sàrl, Lausanne.

**Cours et Séminaires**

Advanced Digital Systems Design, EPFL, Octobre 2002