A Clustered VLIW Architecture Based on Queue Register Files

Marcio Merino Fernandes

Doctor of Philosophy
University of Edinburgh
1998
To my parents,
Armirdo and Lourdes
Abstract

Instruction-level parallelism (ILP) is a set of hardware and software techniques that allow parallel execution of machine operations. Superscalar architectures rely most heavily upon hardware schemes to identify parallelism among operations. Although successful in terms of performance, the hardware complexity involved might limit the scalability of this model. VLIW architectures use a different approach to exploit ILP. In this case all data dependence analyses and scheduling of operations are performed at compile time, resulting in a simpler hardware organization. This allows the inclusion of a larger number of functional units (FUs) into a single chip. In spite of this relative simplification, the scalability of VLIW architectures can be constrained by the size and number of ports of the register file. VLIW machines often use software pipelining techniques to improve the execution of loop structures, which can increase the register pressure. Furthermore, the access time of a register file can be compromised by the number of ports, causing a negative impact on the machine cycle time. For these reasons, we understand that the register file required by a wide-issue unclustered machine could compromise the benefits of having parallel FUs, which have motivated the investigation of alternative machine designs.

This thesis presents a scalable VLIW architecture comprising clusters of FUs and private register files. Register files organized as queue structures are used as a mechanism for inter-cluster communication, allowing the enforcement of fixed latency in the process. This scheme presents better possibilities in terms of scalability as the size of individual register files is not determined by the total number of FUs, suggesting that the silicon area may grow only linearly with respect to the number of FUs. However, the effectiveness of such an organization depends on the efficiency of the code partitioning strategy. We have developed an algorithm for a clustered VLIW architecture integrating both software pipelining and code partitioning in a single procedure. Experimental results show it may allow performance levels close to an unclustered machine without communication constraints. Finally, we have developed silicon area and cycle time models to quantify the scalability of performance and cost for this class of architecture.
Acknowledgements

My greatest gratitude goes to Dr. Nigel Topham, my supervisor. This work would not exist without him.

Many thanks to Josep Llosa of UPC for his contributions and fruitful discussions, and Prof. Roland Ibbett, for his guidance during the first stages of this work.

Special people have made my years in Edinburgh a great time: Hazel, Ann, Susan, Thereza, Carol, Alberto, Goretti, Jitka, Nils, and many others, including fellow doctoral students, members of basketball clubs, the ceilidh crowd, and the local Brazilian society.

Finally, the unique opportunity for being here was made possible by the financial support from CAPES-Brasil, and my family.
Declaration

I declare that this thesis was composed by myself and that the work contained therein is my own, except where explicitly stated otherwise in the text. Some of the material in this thesis has already been published in:


(Marcio Merino Fernandes)
# Table of Contents

List of Figures v

List of Tables viii

Chapter 1 Introduction 1

1.1 Work Context .................................. 1
1.2 Work Overview ................................. 2
  1.2.1 Queue Register Files ........................ 3
  1.2.2 Unclustered VLIW Architectures ............ 3
  1.2.3 Clustered VLIW Architectures ............... 4
  1.2.4 Distributed Modulo Scheduling Algorithm .... 4
1.3 Thesis Structure ............................... 4

Chapter 2 Background 6

2.1 Instruction-Level Parallelism .................. 6
  2.1.1 Hardware-Centric ILP: Superscalar ........... 7
  2.1.2 Software-Centric ILP: VLIW ................ 8
2.2 VLIW Architecture ............................. 9
2.3 Register Files for VLIW Machines ............. 10
  2.3.1 Register File Requirements ................ 11
  2.3.2 Monolithic Register Files .................. 13
  2.3.3 Partitioned Register Files .................. 15
  2.3.4 Other Register File Organizations .......... 18
2.4 VLIW Compilation Issues ...................... 19
  2.4.1 Overview of the Compilation Process ......... 20
  2.4.2 VLIW Scheduling ........................... 25
  2.4.3 Register Allocation ........................ 28
  2.4.4 Code Generation ............................ 29
2.5 Commercial VLIW Machines .................... 30
### Chapter 3 Basic Experimental Framework

3.1 Machine Model .................................................. 33  
   3.1.1 Functional Units ........................................... 33  
   3.1.2 Local Register File ........................................ 36  
   3.1.3 Unclustered Machine ....................................... 37  

3.2 Workload ....................................................... 37  
   3.2.1 Perfect Club Benchmark ................................... 38  
   3.2.2 Selection of Loops and Compiler Optimizations ........ 39  
   3.2.3 Data Dependence Graph ................................... 40  

3.3 Modulo Scheduling Algorithm ................................. 41  
   3.3.1 Minimum Initiation Interval - MIII ....................... 43  
   3.3.2 Iterative Modulo Scheduling - IMS ....................... 46  
   3.3.3 Scheduling Example ....................................... 52  

3.4 Register Allocation .......................................... 53  

3.5 Output Information ............................................ 56  
   3.5.1 Direct Parameters ......................................... 56  
   3.5.2 Derived Parameters ....................................... 56  
   3.5.3 Results Presentation ..................................... 58  

### Chapter 4 Queue Register Files

4.1 QRF Organization ............................................... 59  

4.2 Transforming Multiple-Use Lifetimes .......................... 62  

4.3 Overhead Due to Copy Operations .............................. 66  

4.4 Allocating Lifetimes to a QRF ................................. 68  
   4.4.1 Q-Compatibility Test ...................................... 71  

4.4.2 Register Allocation Using the Q-Compatibility Test ..... 76  

4.5 Analytical Model for Register Files ........................... 79  

4.5.1 Silicon Area Model ......................................... 79  

4.5.2 Cycle Time Model .......................................... 80  

4.5.3 Comparing Register File Organizations .................... 81  

4.6 Summary of Results and Conclusions .......................... 83  

### Chapter 5 Unclustered Architectures

5.1 Increasing ILP with Loop Unrolling ........................... 85  

5.2 Experimental Framework Update ................................ 88  

5.3 Experimental Results .......................................... 91  

5.3.1 Performance Analysis ...................................... 91  

5.3.2 Machine Resources Analysis ................................. 94
Chapter 6  Clustered Architectures  \hfill 103
\hspace{1em} 6.1  Clustered Architecture Organization \hspace{1em} 103
\hspace{1em} 6.2  Experimental Framework Update \hspace{1em} 107
\hspace{1em} 6.3  Partitioning Heuristics \hspace{1em} 108
\hspace{1em} \hspace{1em} 6.3.1  Find Slot Function for a Clustered Machine \hspace{1em} 109
\hspace{1em} \hspace{1em} 6.3.2  Backtracking Function for a Clustered Machine \hspace{1em} 111
\hspace{1em} \hspace{1em} 6.3.3  Complexity of the New Heuristics \hspace{1em} 113
\hspace{1em} 6.4  Experimental Results \hspace{1em} 113
\hspace{1em} \hspace{1em} 6.4.1  Partitioning Effectiveness \hspace{1em} 114
\hspace{1em} \hspace{1em} 6.4.2  Performance Analysis \hspace{1em} 116
\hspace{1em} \hspace{1em} 6.4.3  Machine Resources Analysis \hspace{1em} 118
\hspace{1em} \hspace{1em} 6.4.4  Summary of Results and Conclusions \hspace{1em} 123

Chapter 7  Distributed Modulo Scheduling  \hfill 124
\hspace{1em} 7.1  Overview of the DMS Algorithm \hspace{1em} 125
\hspace{1em} \hspace{1em} 7.1.1  Chains of Move Operations \hspace{1em} 127
\hspace{1em} \hspace{1em} 7.1.2  Scheduling Chains and their Consumer Operations \hspace{1em} 133
\hspace{1em} \hspace{1em} 7.1.3  Unscheduling Operations \hspace{1em} 135
\hspace{1em} \hspace{1em} 7.1.4  Complexity of DMS \hspace{1em} 137
\hspace{1em} \hspace{1em} 7.1.5  Using DMS with Other Machine Models \hspace{1em} 138
\hspace{1em} 7.2  Experimental Framework Update \hspace{1em} 139
\hspace{1em} 7.3  Experimental Results \hspace{1em} 140
\hspace{1em} \hspace{1em} 7.3.1  Partitioning Effectiveness \hspace{1em} 140
\hspace{1em} \hspace{1em} 7.3.2  Performance Analysis \hspace{1em} 142
\hspace{1em} \hspace{1em} 7.3.3  Machine Resources Analysis \hspace{1em} 144
\hspace{1em} \hspace{1em} 7.3.4  Summary of Results and Conclusions \hspace{1em} 149

Chapter 8  Scalability of Performance and Cost for Clustered VLIW 150
\hspace{1em} 8.1  Scalability and Technology Trends \hspace{1em} 151
\hspace{1em} 8.2  Experimental Framework Update \hspace{1em} 152
\hspace{1em} 8.3  Experimental Results \hspace{1em} 152
\hspace{1em} \hspace{1em} 8.3.1  Performance Analysis \hspace{1em} 152
\hspace{1em} \hspace{1em} 8.3.2  Scalability of Performance \hspace{1em} 155
\hspace{1em} \hspace{1em} 8.3.3  Summary of Results and Conclusions \hspace{1em} 157
Chapter 9  Conclusions  159
  9.1  Main Findings  ............................................... 159
  9.2  Thesis Contribution  ......................................... 161
  9.3  Future Work  .................................................. 161

Bibliography  163
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>Basic structure of a VLIW processor</td>
<td>10</td>
</tr>
<tr>
<td>2.2</td>
<td>Example of an instruction word for a VLIW processor</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Register pressure resulting from loop unrolling</td>
<td>12</td>
</tr>
<tr>
<td>2.4</td>
<td>Register pressure resulting from software pipelining</td>
<td>12</td>
</tr>
<tr>
<td>2.5</td>
<td>Area of a monolithic register file</td>
<td>14</td>
</tr>
<tr>
<td>2.6</td>
<td>Subdividing a monolithic register file</td>
<td>16</td>
</tr>
<tr>
<td>2.7</td>
<td>Phases of an optimizing VLIW compiler</td>
<td>21</td>
</tr>
<tr>
<td>3.1</td>
<td>Hardware/Software codesign process</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Generic organization of a microprocessor pipeline</td>
<td>34</td>
</tr>
<tr>
<td>3.3</td>
<td>VLIW unclustered machine</td>
<td>37</td>
</tr>
<tr>
<td>3.4</td>
<td>Extracting loops from the benchmark</td>
<td>38</td>
</tr>
<tr>
<td>3.5</td>
<td>Innermost loop and data dependence graph</td>
<td>41</td>
</tr>
<tr>
<td>3.6</td>
<td>Modulo Reservation Table</td>
<td>47</td>
</tr>
<tr>
<td>3.7</td>
<td>Schedule produced by IMS</td>
<td>53</td>
</tr>
<tr>
<td>3.8</td>
<td>Register requirements - MaxLive</td>
<td>55</td>
</tr>
<tr>
<td>4.1</td>
<td>QRF block diagram</td>
<td>60</td>
</tr>
<tr>
<td>4.2</td>
<td>Using a queue to store a loop variant lifetime</td>
<td>61</td>
</tr>
<tr>
<td>4.3</td>
<td>Register storage</td>
<td>63</td>
</tr>
<tr>
<td>4.4</td>
<td>DDG transformation to include a copy operation</td>
<td>63</td>
</tr>
<tr>
<td>4.5</td>
<td>Inserting copy operations in a DDG</td>
<td>66</td>
</tr>
<tr>
<td>4.6</td>
<td>II Variation-Copy Op</td>
<td>67</td>
</tr>
<tr>
<td>4.7</td>
<td>SC Variation-Copy Op</td>
<td>67</td>
</tr>
<tr>
<td>4.8</td>
<td>Using one queue to store lifetimes of the same length</td>
<td>69</td>
</tr>
<tr>
<td>4.9</td>
<td>Using one queue to store lifetimes of distinct lengths</td>
<td>70</td>
</tr>
<tr>
<td>4.10</td>
<td>Dual-ported register cell</td>
<td>80</td>
</tr>
<tr>
<td>4.11</td>
<td>Implementation of a QRF using dual-ported register cells</td>
<td>81</td>
</tr>
<tr>
<td>4.12</td>
<td>Silicon area</td>
<td>83</td>
</tr>
<tr>
<td>4.13</td>
<td>Access time</td>
<td>83</td>
</tr>
</tbody>
</table>
5.1 Unclustered machine using a QRF .......................... 84
5.2 Optimizing the use of machine resources with loop unrolling .......................... 86
5.3 Number of loops in each class .................................. 90
5.4 Execution time-Fixed cycle .................................. 93
5.5 Loop speedup-Fixed cycle .................................. 93
5.6 Copy overhead-Class 1 ...................................... 94
5.7 Copy overhead-Class 3 ...................................... 94
5.8 IPC Dynamic-Class 1 ....................................... 95
5.9 IPC Dynamic-Class 3 ....................................... 95
5.10 Name space ............................................... 96
5.11 Storage locations ........................................... 96
5.12 Lifetimes sharing a queue .................................. 97
5.13 Maximum queue length .................................. 98
5.14 Silicon area ................................................. 100
5.15 Cycle time .................................................. 100
5.16 Execution time-Class 1 .................................. 101
5.17 Execution time-Class 3 .................................. 101

6.1 Clustered machine .......................................... 104
6.2 Communication between adjacent clusters using a CQRF .......................... 105
6.3 Cluster organization ......................................... 106
6.4 Loops with a larger II due to partitioning ........................................... 115
6.5 Communication distance after partitioning ........................................... 116
6.6 Number of cycles-Class 1 .................................. 117
6.7 Number of cycles-Class 3 .................................. 117
6.8 IPC Dynamic-Class 1 ....................................... 118
6.9 IPC Dynamic-Class 3 ....................................... 118
6.10 Total silicon area of register files ........................................... 120
6.11 Cycle time of reg. files .................................... 121
6.12 Machine cycle time ........................................ 121
6.13 Execution time-Class 1 .................................. 122
6.14 Execution time-Class 3 .................................. 122

7.1 Overview of IMS and DMS algorithms ........................................... 125
7.2 Options to create a chain .................................... 128
7.3 DDG transformation to insert a chain ........................................... 133
7.4 Typical use of a move operation ........................................... 139
7.5 Loops with a larger II due to DMS partitioning ........................................... 141
7.6 Communication distance after partitioning with DMS .............. 142
7.7 Number of cycles-Class 1 ........................................ 143
7.8 Number of cycles-Class 3 ........................................ 143
7.9 IPC Dynamic-Class 1 .............................................. 144
7.10 IPC Dynamic-Class 3 .............................................. 144
7.11 Total silicon area of register files ............................... 146
7.12 Cycle time of reg. files .......................................... 147
7.13 Machine cycle time .............................................. 147
7.14 Execution time-Class 1 ........................................... 148
7.15 Execution time-Class 3 ........................................... 148

8.1 Number of cycles-Class 1 ........................................... 153
8.2 Number of cycles-Class 3 ........................................... 153
8.3 Machine cycle time .............................................. 154
8.4 Execution time-Class 1 ............................................ 155
8.5 Execution time-Class 3 ............................................ 155
8.6 IPC vs Area-Class 1 ............................................. 156
8.7 IPC vs Area-Class 3 ............................................. 156
8.8 Performance vs Area-Class 1 .................................... 157
8.9 Performance vs Area-Class 3 .................................... 158
## List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Functional unit characteristics</td>
<td>36</td>
</tr>
<tr>
<td>3.2</td>
<td>Register file access port requirements</td>
<td>36</td>
</tr>
<tr>
<td>3.3</td>
<td>Loops extracted from the Perfect Club benchmark</td>
<td>39</td>
</tr>
<tr>
<td>3.4</td>
<td>Instruction mix of the selected loops</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Register file configurations</td>
<td>82</td>
</tr>
<tr>
<td>5.1</td>
<td>Unclustered machine configurations used in experiments</td>
<td>91</td>
</tr>
<tr>
<td>5.2</td>
<td>URV register requirements</td>
<td>98</td>
</tr>
<tr>
<td>5.3</td>
<td>UQV register requirements</td>
<td>99</td>
</tr>
<tr>
<td>6.1</td>
<td>Machine configurations used in experiments</td>
<td>114</td>
</tr>
<tr>
<td>6.2</td>
<td>URF register requirements</td>
<td>119</td>
</tr>
<tr>
<td>6.3</td>
<td>CQF register requirements</td>
<td>120</td>
</tr>
<tr>
<td>7.1</td>
<td>Machine configurations used in experiments with DMS</td>
<td>140</td>
</tr>
<tr>
<td>7.2</td>
<td>URF register requirements</td>
<td>145</td>
</tr>
<tr>
<td>7.3</td>
<td>CQF register requirements</td>
<td>145</td>
</tr>
<tr>
<td>8.1</td>
<td>SIA predictions of device capabilities (1997)</td>
<td>151</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

Performance of computer systems has evolved continuously since the first machine was built. The availability of faster machines encourages the development of new and sophisticated applications, leading to ever increasing performance requirements. Advances in computer design and implementation technology have allowed those improvements. Computer systems in 1998 are based on microprocessors, which have grown in performance at an annual rate of over 50% [45]. One of the latest trends in microprocessor architecture design is called Very Long Instruction Word (VLIW). Machines of this kind are able to exploit parallelism at the level of machine instructions. This thesis presents a clustered VLIW architecture model able to achieve high performance and exhibiting a good potential for scalability. It was developed using a hardware/software codesign process to design a number of features, including a novel register file organization based on queues, register allocation schemes, a clustered organization, and algorithms for code scheduling and partitioning.

1.1 Work Context

The microprocessor technology of 1998 relies on two basic approaches to improve performance. One is to increase clock rates, resulting in faster execution of machine operations. The other is instruction-level parallelism (ILP), a set of hardware and software techniques that allows parallel execution of machine operations [84]. Superscalar architectures [51] rely most heavily upon hardware schemes to identify parallelism among operations. Although this approach offers advantages in terms of code compatibility, the hardware complexity involved poses some limitations in terms of scalability. Increasing the number of functional units (FUs) in current superscalar microprocessors would require even more sophisticated schemes to find and schedule independent operations. Using a VLIW
architecture is another possibility to exploit ILP. In this case all data dependence analyses and scheduling of operations are performed at compile time, resulting in a simpler hardware organization. This allows the inclusion of a larger number of FUs into a single chip, increasing the possibilities of parallelism exploitation. In spite of this relative simplification, the scalability of VLIW machines can be constrained by the complexity and size of the required register file (RF). Ideally, a VLIW machine would have a number of parallel functional units connected to a common register file able to perform two reads and one write operation per FU in each cycle [14]. This implies that each FU requires three access ports to the register file.

The available processing power of a very wide issue machine can be fully exploited when executing loop structures, which in many cases accounts for the largest share of the total execution time of a program. Several compiling techniques have been developed to schedule loops in ILP machines. Software pipelining [16], for instance, is a scheme that allows the initiation of successive loop iterations before prior ones have completed. In this scheme, consecutive data values produced by the same operation may coexist, requiring distinct storage locations and thus increasing register pressure [64]. High register pressure results in register file requirements that are difficult to realize, taking into account current technology trends. The size of shared register files grows in proportion to the square of the number of ports, and hence also the number of FUs [13]. If software pipelining is performed it can grow in proportion to the cube of the number of FUs (Section 2.3.2). The size of the register file alone can be a problem in the machine design. Furthermore, the access time of such an RF can be compromised by the number of ports, causing a negative impact on the cycle time of the machine. For these reasons we understand that the register file required by a wide-issue unclustered machine can compromise the benefits resulting from aggressive ILP scheduling. This has motivated us to investigate alternative machine designs.

1.2 Work Overview

This thesis proposes a scalable VLIW architecture comprising clusters of functional units and private register files, using queue structures to implement a mechanism for inter-cluster communication. We believe this scheme presents better possibilities in terms of scalability as the size of individual RFs is not influenced by the total number of FUs, suggesting that the silicon area may grow only linearly with respect to the number of FUs. Technology trends indicate the future
possibility of building systems integrating powerful processors and main memory on a single chip [53]. This may address some design issues concerning the memory subsystem of our machine model, a problem also common to other microarchitectures. However, the effectiveness of such an organization also depends on the scheduling and code partitioning strategy. We have developed a scheme to produce software pipelined code for a clustered VLIW machine model aiming to achieve performance levels similar to an unclustered machine without communication constraints. The main developments and contributions of this research work are outlined in the next subsections.

1.2.1 Queue Register Files

Software pipelining generally increases register pressure in VLIW machines. The register file required in such cases may compromise scalability, which has motivated us to develop a Queue Register File (QRF). Register files organized by means of FIFO queues, with limited read and write access, are believed to be less complex than conventional organizations. That should be the case because it could be implemented using simple dual-ported individual register cells, and a possibly less complex address decoding logic. On the other hand, this simplification in hardware imposes new constraints on the register allocator, requiring new techniques to efficiently exploit this organization. Software pipelined loops generate a regular pattern of production and consumption of lifetimes. We have taken advantage of this fact to deduce and prove a Q-Compatibility Test to decide which lifetimes can share a given storage queue, based on their relative production and consumption order. The Q-Compatibility Test enables efficient register allocation to the QRF. Analytical models show that QRFs are in general more efficient than conventional organizations in terms of silicon area and access time.

1.2.2 Unclustered VLIW Architectures

We developed a VLIW machine model organized as a single cluster of functional units, all of them connected to a common register file. This model allow us to quantify the achievable ILP for the architecture and compilation techniques employed. Two types of register files have been used: a conventional RF, and a QRF. The advantage of using a QRF with software pipelined loops has been confirmed through experimental analysis. Nonetheless, it has also been confirmed that unclustered machines do not scale well, mainly due to the size and number of ports of the shared register file. In this case, all benefits achieved by an aggressive ILP scheduling can be lost due to a long register file cycle time.
1.2.3 Clustered VLIW Architectures

Including additional functional units in a unclustered VLIW architecture is not a straightforward design issue. New register file access ports can severely compromise the machine cycle time. This thesis proposes a VLIW architecture comprising clusters of functional units and private conventional register files. Each cluster should have a small number of FUs to avoid increasing the register file complexity. Clusters are interconnected using a bi-directional communication ring. We developed a scheme using QRFs to implement data communication between adjacent clusters. In this case a communication queue register file (CQRFs) is placed between two clusters. Sending a value from one cluster to another requires only one write and one read operation to the appropriate CQRF. We found through experimental analysis that the silicon area required to implement this scheme may grow only linearly with respect to the number of FUs.

1.2.4 Distributed Modulo Scheduling Algorithm

A clustered organization can address some of the issues related to the hardware complexity of a VLIW architecture. However, a single thread of control implies that operations scheduled in distinct clusters may be data dependent with each other, requiring inter-cluster communication. This might impose additional constraints on the scheduler and register allocator, possibly compromising the machine performance. We developed a software pipeline scheduling algorithm targeting clustered VLIW architectures. The scheme, called Distributed Modulo Scheduling, performs in a single step both scheduling and partitioning of operations among clusters. The objective is to produce code achieving performance levels close to a single cluster machine without communications constraints. Several experiments investigated the effectiveness of the scheme for machine configurations up to 10 clusters, and a total of 30 functional units. Furthermore, the scalability of the proposed clustered architecture was assessed, taking into account performance and cost aspects, along with future technology trends.

1.3 Thesis Structure

The thesis structure presented in this section generally reflects the chronological order in which this research work was carried out. Some of the findings and experiments performed in early stages of the work are omitted, being replaced by later developments. A number of hardware and software issues have been addressed, however the interrelation among them requires that they are presented
together. We tried to produce a single presentation format for the work methodology, design considerations, experimental results and discussions. Finally, a summary of the main objectives and contents of each chapter is described below:

- **Chapter 1**: Motivation and work overview.

- **Chapter 2**: Bibliographic survey related to this thesis. Topics discussed include ILP, hardware support for ILP, VLIW architectures, register file organizations, compilation and scheduling techniques for VLIW, and similar architectures developed elsewhere.

- **Chapter 3**: Description of the experimental framework used to perform quantitative analyses throughout the work. A basic machine model is defined, along with the benchmark loops employed. A software pipeline scheduling algorithm is at the core of the compilation process. A number of figures regarding performance and machine resources are generated.

- **Chapter 4**: A queue register file is proposed as an alternative organization to deal with high register pressure. A novel technique is presented to perform register allocation, which includes a theorem (Q-Compatibility Test) and the corresponding proof. Comparisons with conventional organizations are made in terms of silicon area and access time.

- **Chapter 5**: Defines an unclustered VLIW machine, using either a RF or QRF. The potential to exploit ILP, and the implications of using a shared register file are investigated by means of experimental analysis.

- **Chapter 6**: A clustered VLIW machine and the corresponding scheduling algorithm is proposed to address scalability issues. The main motivation is to keep register files small enough to result in a short cycle time. Queue register files are used to implement a communication mechanism between clusters.

- **Chapter 7**: Presents DMS, an integrated scheduling/partitioning algorithm targeting a clustered VLIW machine. The scheme is able to deal efficiently with communication constraints for a range of machine configurations.

- **Chapter 8**: Analyses the scalability of performance and cost of clustered VLIW machines. Compares several configurations of the proposed architecture, trying to predict its viability according to current technology trends.

- **Chapter 9**: Final conclusions and suggestions for future work.
Chapter 2
Background

2.1 Instruction-Level Parallelism

Instruction-level parallelism is a set of processor and compiler design techniques that allows a sequence of machine operations to be parallelized for execution on multiple pipelined functional units. The operations are similar to the ones usually found in a RISC microprocessor, such as memory loads and stores, additions, multiplications, and branch instructions. The main advantage of ILP is the possibility of exploiting parallelism with no need of code rewriting, working with existing programs. Sequential programming style still dominates the software base currently in use, and also new developments. This is unlikely to change in the foreseeable future, which emphasizes the commercial value of ILP.

Several studies have pointed out the existence of large amounts of available ILP to be exploited in existing programs. Some studies concluded that the available ILP is modest, ranging between 2 and 5. However those studies did not consider program transformations able to expose ILP [84], such as loop interchange, trace scheduling, loop unrolling, and software pipelining, among others [7]. Wall carried out an extensive study, using speculative execution, memory disambiguation and other techniques to enhance ILP [94]. He concluded the available parallelism ranges between 2 and 60, depending on the execution model employed. A new version of this study was later conducted, considering a much larger set of techniques to expose ILP [95]. That report presents simulations of test programs under 375 models of available parallelism. It was found that relying only on the technology available at the time (1993), it was possible to consistently obtain ILP between 4 and 10 for most of the programs. Using branch prediction with speculative execution the range would shift to 7-13. It was also concluded that vectorizable programs could attain much higher levels. Another study on available ILP focused on methods to eliminate control flow barriers [55].
found that the parallelism in non-numeric programs ranges between 18 and 400. Numeric applications could go even further. It is expected that new compiler optimizations will expose even larger amounts of parallelism to be exploited by aggressive machine configurations.

On the other hand, continuous improvements in VLSI design enables the integration of more functional units into a single chip. Furthermore, higher clock speeds may result in more deeply pipelined functional units. These factors contribute to increase the available hardware parallelism. The task of keeping an ILP processor busy can rely most heavily either on hardware or software schemes. This constitutes the basis upon which modern ILP processors can be classified, which is discussed in the next subsections.

2.1.1 Hardware-Centric ILP: Superscalar

Superscalar processors [51] have complex hardware structures to decide at runtime which operations have no dependences with each other, so they can be executed in parallel. Dynamic scheduling of operations are also performed by hardware. Scoreboarding [92] is a dynamic scheduling technique that allows instructions to execute out-of-order. Another approach, called Tomasulo Algorithm [93] combines out-of-order execution with register renaming. These and other related techniques try to avoid stalls in the pipeline by preventing data hazards. Hardware branch prediction schemes can also be implemented to avoid control hazards [90]. As a side effect, increasing the number of operations in flight (issued but not yet completed) can make the number of architectural visible registers insufficient, requiring register renaming techniques [98].

The possibility of having object code compatibility is one of the main advantages of superscalar processors, allowing applications to run in faster machines without reccompilation. For this reason, general purpose superscalar processors have reached the mainstream market. The drawback of this approach is that implementing those and other hardware schemes can be expensive in terms of silicon area and clock cycle. Contemporary machines of 1998 can issue about four operations per cycle [75, 37]. However, there is a general perception that hardware complexities may prevent the expected performance gains if the current instruction issue rate is increased by a significant factor. For this reason new ILP designs, in the form of VLIW processors, move into the compiler some of the tasks performed by hardware in superscalar architectures.
2.1.2 Software-Centric ILP: VLIW

VLIW machines provide hardware parallelism in the form of multiple and deep pipelined functional units. However, they have a relatively simple control logic, releasing more silicon area to implement functional units. This should result in higher levels of hardware parallelism than found in superscalar machines. Simpler hardware may result in lower cost per chip and less power consumption, important features for embedded computing and portable devices, among others. The counterpart of these advantages is the need of sophisticated compiler techniques to identify parallelism and schedule operations among functional units. The program for a VLIW machine specifies precisely which functional unit should execute a given operation, and when an operation should be issued in order to enforce dependence constraints [84]. Comparing to dynamic scheduling schemes, a compiler can work with a larger window of candidate operations to be parallelized. This improves the possibilities of keeping the available functional units busy. However, the compiling techniques involved are complex, still evolving and presenting challenges.

Detailed description of the target processor is necessary to achieve the best performance with static scheduling. In this case object code compatibility may not be possible among distinct machine generations, requiring program recompilation. Complexity and program size can result in long compilation times. To alleviate this problem the compiler can subdivide a program, performing tasks of manageable sizes [44]. The nature of some application fields make them less sensitive to this problem, such as scientific programs and digital signal processing (DSP) [23]. However, this is an important issue for general purpose computers and applications. A research group at IBM proposed a solution for this problem, organizing operations into tree-instructions [72]. Another work proposed dynamic scheduling of operations for VLIW machines [77]. Although a number of compiling issues are still open, VLIW architectures are beginning to establish itself in some niche markets, specially in the DSP area. The technology can also be effective to support multimedia applications, an area of increasing interest [76]. However, to have a broad impact on the mainstream market, VLIW processors must accelerate the non-vectorizable scalar code prevalent in most applications [86]. An indication of the viability of this technology is the announcement by Intel of the first general purpose VLIW-like processor [43], to be released in late 1999. The next sections of this chapter discuss VLIW architectures and related compilation issues, followed by a brief presentation of some commercial VLIW machines.
2.2 VLIW Architecture

A VLIW architecture is characterized by a wide instruction word controlling the action of all functional units. A single control unit can issue a new instruction every cycle. Data dependences and scheduling of instructions are resolved statically at compiling time, so the hardware has to perform no further checking to ensure program correctness. The first VLIW architecture was proposed by Fisher at Yale University [33]. Since then a growing interest in this technology has motivated a number of hardware and software developments to support the new paradigm.

The ideal VLIW machine has a number of concurrent FUs, connected to a register file able to perform two reads and one write operation per functional unit in each cycle [14, 21]. A control unit, instruction and data caches complete the basic VLIW design, as seen in Figure 2.1. The diagram shows a hypothetical machine with functional units capable to perform memory load and store access (L/S), arithmetic and logic operations (ADD), multiplication and divisions (MUL), and a branch unit (BR). Static scheduling and a single thread of control impose strict synchronization constraints among functional units, which should operate in lockstep [45, 40]. This may result in one single cache miss stalling all FUs, stressing the importance of the memory subsystem. However, the pattern of memory access of some DSP applications may result in a high rate of cache misses, motivating the use of alternative designs such as local memories or prefetching buffers [23]. Future trends suggest that it will be possible to integrate processing elements and main memory in a single chip [53], greatly simplifying this issue.

A long instruction word should contain, for each functional unit, the operation code, the source and the destination registers used, as shown in the example in Figure 2.2. No-operations (NOPs) may be inserted in the long instruction if there are not enough operations to be issued in parallel in a given cycle. Practical VLIW machines have been implemented using instruction words up to 1024 bits wide [83, 66]. Uncompressed encodings explicitly store NOPs in the instruction word. This simplifies the hardware organization, but at the expense of poor memory utilization due to increase in code size. Compressed encodings do not store NOPs, using variable size instructions, allowing greater effective memory bandwidth [17].

The need to execute some time-critical instructions might be known only at run time, which is often associated with the outcome of a branch operation. Nevertheless, the compiler can schedule those instructions speculatively, as long as some hardware support is provided to ignore the effects of executing unnecessary