A Clustered VLIW Architecture Based on Queue Register Files

Marcio Merino Fernandes

Doctor of Philosophy
University of Edinburgh
1998

EX BIBL. UNIV. EDINBURGH

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:

- M. Fernandes, J. Llosa, and N. Topham. Distributed Modulo Scheduling. In *HPCA-5, 5th IEEE International Symposium on High Performance Computer Architecture*, Orlando, USA, 1999.

- M. Fernandes, J. Llosa, and N. Topham. Partitioned schedules for clustered VLIW architectures. In *IPPS'98, 12th IEEE/ACM International Parallel Processing Symposium*, Orlando, USA, 1998.

- M. Fernandes, J. Llosa, and N. Topham. Allocating lifetimes to queues in software pipelined architectures. In *EURO-PAR'97, 3rd International Euro-Par Conference*, Passau, Germany, 1997.

- M. Fernandes, J. Llosa, and N. Topham. Extending a VLIW architecture model. Technical Report ECS-CSG-34-97, University of Edinburgh, Department of Computer Science, 1997.

- M. Fernandes, J. Llosa, and N. Topham. Using queues for register file organization in VLIW architectures. Technical Report ECS-CSG-29-97, University of Edinburgh, Department of Computer Science, 1997.

Table of Contents

List of Figures

List of Tables

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]. It was 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 recompilation. 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

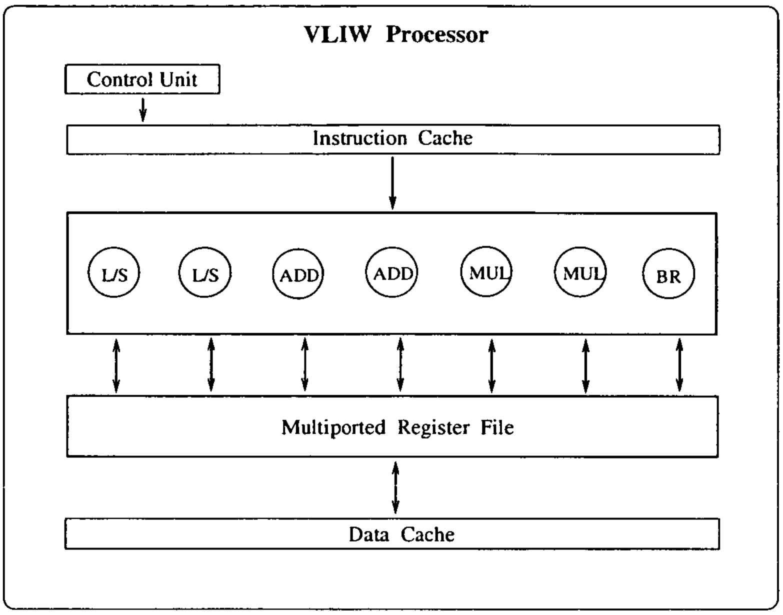


Figure 2.1: Basic structure of a VLIW processor

Single instruction word controlling all functional units

L/S 1	L/S 2	ADD 1	ADD 2	MUL 1	MUL 2	BR
LD R3, 0(R1)	LD R5, 0(R2)	ADD R10, R8, R9	NOP	MUL R20, R17, R18	NOP	JR R30

Figure 2.2: Example of an instruction word for a VLIW processor

operations [21, 83, 52]. *Predicated execution* determines the execution of an operation according to a Boolean input [47]. Some schemes employ extra *poison bits* to indicate if the contents of a register is valid [45]. Other methods buffer the result of an speculative instruction until deciding if it is needed or not. Branches can be eliminated from an acyclic region of a control flow graph using *if-conversion* [4] and predicated execution, as showed by Dehnert [18]. These and other special hardware features can be used to optimize the performance of a VLIW machine.

2.3 Register Files for VLIW Machines

Performance reasons dictate that operations other than memory access should be *register-register* [45]. Those operations use destination and source operands stored in the on-chip register file. Memory operations can be made through load and store operations. All functional units of a VLIW machine operate in lockstep. If one of them stalls due to a memory operation, all the others must also stop executing. Data communication among functional units should always take place thorough a centralized register file. A VLIW processor should provide a register file with enough capacity and bandwidth to support the intended instruction issue. Occasional on-chip cache access to deal with spill code could be tolerated, however frequent cache misses can seriously compromise the machine performance. For this reason the design of the register file is one of the crucial aspects of a VLIW machine, being able to determine the machine cycle time [25].

In this section we show that register file requirements for wide-issue VLIW machines are high and complex. As a result, conventional designs may not be well suited to address the problem. Thus, performance and scalability issues may lead to alternative hardware organizations and compilation techniques

2.3.1 Register File Requirements

There are two main factors that make register file requirements for VLIW machines complex:

- Number of registers
- Number of access ports

Compiler optimizations employed to exploit ILP machines often require a large number of registers. Predication and speculative execution generate extra data values that must be kept without knowing if they are necessary or not. Loop unrolling and software pipelining can also increase register pressure dramatically.

The *lifetime* of a value is the time span ranging from its definition up to the last use of it. It can also be referred to as *lifetime length.* The precise definition of the first and the last cycle of a lifetime depends on architecture details, and is not relevant at this point of the discussion. Two types of lifetimes can be found in a loop: A *loop invariant* lifetime spans the whole loop execution. Usually it is initialized before entering loop execution and remains fixed until exiting. For this reason they need only one storage position. On the other hand, a *loop variant* is a lifetime produced by successive iterations of a loop, usually having its value changed.

Loop unrolling [20] is a well known compiler optimization that replicates the body of a loop some number of times. This allows simultaneous scheduling of more than one iteration at a time, resulting in a larger number of operations to exploit machine resources. New lifetimes are also generated, possibly requiring distinct storage positions, as shown in the example in Figure 2.3.

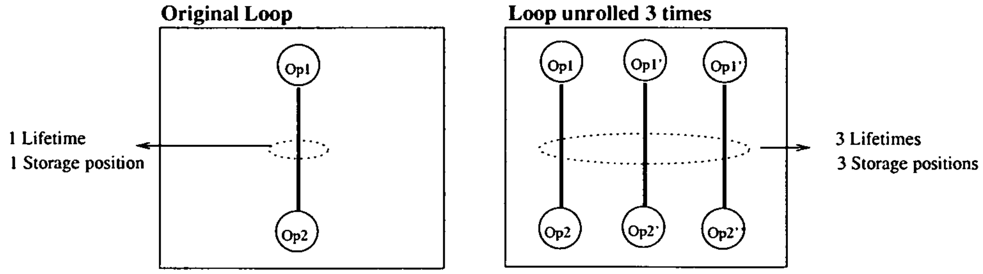


Figure 2.3: Register pressure resulting from loop unrolling

Software pipelining can also be very demanding due to the overlapping of lifetimes produced by the same operation from distinct loop iterations. In this scheme a new iteration starts before prior ones have fully completed. A given lifetime length can be longer than the time between the initiation of two or more successive iterations. In this case an operation produces a new value before previous ones have been consumed, thus requiring distinct storage locations (Figure 2.4).

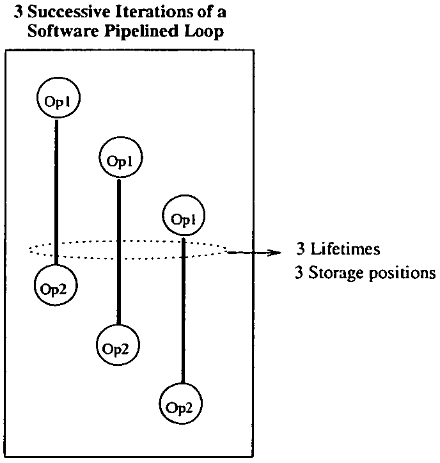


Figure 2.4: Register pressure resulting from software pipelining

Register requirements for ILP architectures have been studied by several authors. A theory for assessing register requirements of pipelined processors with various issue widths was developed by Mangione-Smith and others [69]. The technique is intended to help in evaluating trade-offs in machine designs. The study concentrates on the execution of innermost loops from scientific programs. Llosa performed a quantitative analysis on register requirements of software pipelined loops and their effect on performance [64]. This study focused on loop variants as they account for the most significant fraction of register requirements for numeric applications. It was found that 64 registers are enough to avoid spill code for at least 90% of the loops, however a lot more is required by a few loops [65]. Furthermore, those loops account for a significant fraction of the total execution time of the benchmark, which emphasizes the relative importance of them. Farkas produced a study on register files for dynamically scheduled ILP processors, with some findings that can also be extended to VLIW machines [25]. It was concluded that a four-issue machine requires at least 80 registers. An eight-issue processor would require at least 128 registers. Those and other works have confirmed that aggressive exploitation of ILP requires a large number of registers.

It has been shown that ILP machines executing non-numeric applications would require RFs with a small number of access ports [71]. However, numeric applications, often the target of wide-issue machines, are much more demanding. The number of access ports required by a register file for VLIW architectures further complicates its implementation and performance. As already said, each functional unit requires 2 read and 1 write register file access ports to sustain the achievable issue. The area of shared register files grows in proportion to the *square* of the number of ports, and hence also the number of FUs [13, 25], which has motivated a number of alternative organizations, as shown in the next sections.

2.3.2 Monolithic Register Files

A *monolithic register file* can be implemented using a register cell with multiple read and write ports. It allows multiple access to the same register in any given clock period [58]. This is the organization used by most of the ILP machines built until 1998. However it can constitute a barrier for scalability. If software pipelining is performed, as often happens in ILP machines, we have found that the register file size can grow in proportion to the *cube* of the number FUs.

Let us assume that a p ported register file, containing r registers of d bits each is fully connected to a collection of f functional units, each having a latency of l cycles. The silicon area required to implement such a register file is shown diagrammatically in Figure 2.5. It can be inferred that, for some constant value K, the expression to calculate the area A of that register file is:

$$A = Krdp^2 = \Theta(rp^2) \qquad (2.1)$$

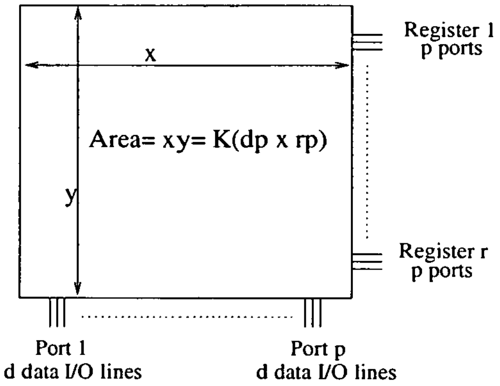


Figure 2.5: Area of a monolithic register file

In a software pipelined loop each operation in flight reserves n register names, where n is the number of software pipe-stages straddled by each lifetime. The number of operations in flight is determined by the product of the instruction issue width and the pipeline lengths. This equals the number of independent functional units, f, multiplied by the number of pipeline stages of each FU, l. Thus, assuming that n is a constant, the number of registers r required to execute a software pipelined loop in a VLIW machine is:

$$r = nfl = \Theta(fl) \tag{2.2}$$

To sustain an average issue rate of s instructions per cycle it may be necessary to have a register bandwidth of at least $2s$ reads and s writes per cycle, requiring 2 read ports and 1 write port per instruction. Under the reasonable assumption that $s = \Theta(f)$ we can say:

$$p = \Theta(f) \tag{2.3}$$

Using equations 2.2 and 2.3 the register file area can be expressed in terms of the number of functional units:

$$A = \Theta(lf^3) \tag{2.4}$$

We have found that the area of a register file to support software pipelining execution is proportional to the cube of the number of functional units. This result clearly shows that designs employing a large multiported register file can constrain the scalability of VLIW architectures.

A further complicating factor is the access time of multiported register files: it grows approximately linearly with the number of ports [23]. The register file may well determine the cycle time of a VLIW machine. One study found that when the issue rate scales up from 4 to 8, the performance improvement achieved is only 20%. The main reason for this is the complexity of the enlarged register file, specially in regard to the number of access ports [25].

We have shown that conventional register file organizations may not be suitable for ILP architectures. This could be even more problematic for wide-issue VLIW machines, which has motivated the development of decentralized architectures, as presented in the next section.

2.3.3 Partitioned Register Files

As discussed in Section 2.3.1, exploiting a high degree of parallelism also requires parallel access to a possibly large set of registers. A single multiported register file is the simplest solution to the problem, however access time and silicon area may inhibit its use by wide-issue VLIW machines. The technology available in 1998 allows one to build register files with around 10-15 ports at reasonable cost and speed [23]. This configuration would be suitable for machines with up to 5 FUs, however higher degrees of hardware parallelism are already possible. To overcome this problem, some processor architectures may incorporate distributed or partitioned register files, each of them providing access to a smaller set of functional units. This reduces the port requirements of individual RFs, and should also reduce the size of them. Multiple banked register file organizations can be used by dynamically scheduled processors. One way to deal with the new organization is by using register renaming [96]. Statically scheduled processors require complex compiling techniques to distribute operands among RF banks, each of them dependent on the architectural model adopted.

As defined in [58], in a *distributed* RF configuration each set of functional units has direct access to one register file only (Figure 2.6a), resulting in a clustered structure [50]. Access to non-local register files may stall the processor or require register copy operations, as discussed later in this section. A *partitioned* register file provides less connectivity between FUs and registers (Figure 2.6b), however each FU has still direct access to any register [50]. Copy operations are not necessary, although conflict access may arise. This approach has been successfully used in vector processors [58].

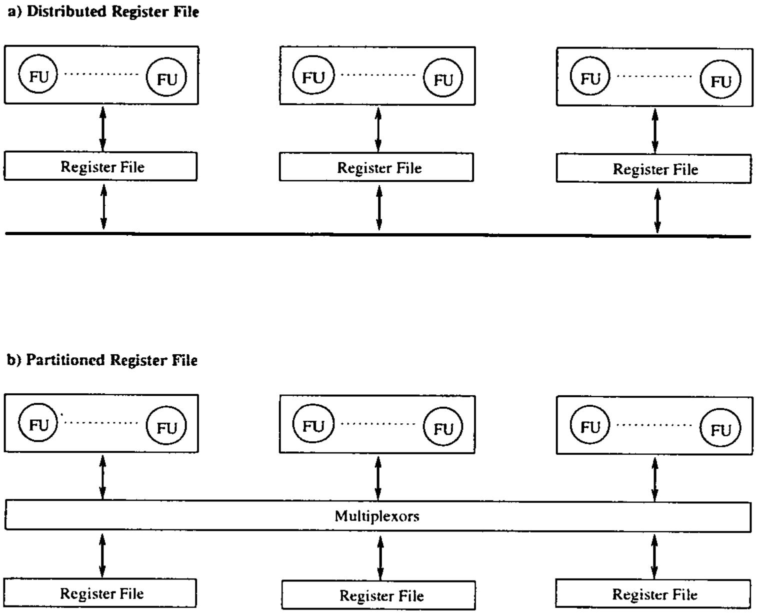


Figure 2.6: Subdividing a monolithic register file

Significant performance degradation may occur if code partitioning is not properly done. According to Faraboschi [23], three distinct architectural scenarios may be possible when register files are partitioned or distributed:

1. The register file clustering is *architecturally invisible*. In this case the compiler assumes a unified register space. Local register access operations occur as usual. However, access to a register file located in another cluster may be necessary. In this case, hardware support should be available to stall the processor while register contents are moved across non-local elements. This approach does not impose extra complexities to the compiler, however significant performance penalties may occur due to excessive stalls.

2. The register file clustering is *architecturally visible*, with *complete connectivity* between FUs and RFs. In this case local and non-local RF access have distinct latencies. A non-local access is actually implemented using a copy operation, which must be scheduled together with the operation requiring the access to the RF. This creates an indivisible "operation-copy" pair. The advantage here is the possibility of minimizing the overhead due to copy operations by scheduling them out of the critical path.

3. The register file clustering is *architecturally visible*, with *limited connectivity* between FUs and RFs. This case is similar to the previous one, however the compiler must schedule copy operations explicitly, but not necessarily together. Better schedules might be produced, at a cost of a more complex compilation process.

Previous works have proposed distributed or partitioned register files for wide-issue machines. Whatever organization is used, compilation for these type of machines is difficult, resulting in severe performance loss if not properly handled.

The *Multiflow Trace* VLIW machines were commercially available as general purpose systems in the late 1980s. Configurations capable of issuing up to 28 operations simultaneously were built [66]. The architecture was designed using clusters of functional unit and *distributed* private register files. Global shared buses were used to connect non-local register banks, which increases operation latencies. The approach used to minimize the amount of data transfer latency is derived from the *Bottom Up Greedy (BUG)* algorithm [22]. It is used as a pre-scheduling step to assign operations to functional units and register banks [66]. The actual latency of an operation is determined by the register bank of the destination operand. If it is local, no extra delay is necessary. Otherwise, the functional unit has also to perform a sort of copy operation to access the non-local register file using the global bus.

Capitanio and others proposed a *Limited Connectivity* VLIW architecture, employing *distributed* register files [14]. Although originally called "partitioned", this structure is more closely related to our definition of "distributed" register file. We attribute this denomination conflict to the lack of a widely accepted classification of register file organizations. The processor in this machine model is partitioned into clusters of functional units fully connected to a private register file. Communication between clusters take place through global buses. The compiler schedules move operations when non-local register file access is required. The compilation process uses a three step scheduling process: code is first generated for an ideal VLIW, assuming a monolithic register file. Then an algorithm is applied to distribute data among clusters, minimizing a given cost function related to communication delays. Finally the required move operations are inserted and the code is recompacted.

The *Transport Triggered Architecture (TTA)* is a VLIW machine using *partitioned* register files [50]. No extra copy or move operations are needed, however the register allocator has to prevent access conflicts due to a limited number of ports. A conventional register allocator maps architectural registers to machine registers. Further actions must be taken if the number of physical registers is insufficient. The register allocator for a TTA must also take into account the limited number of ports. The authors proposed several methods for this task.

Another approach has being reported recently by HP Laboratories [19]. The target machine is a clustered VLIW architecture, using distributed register files. The code partitioning strategy can be seen as an extension of the techniques used by the Multiflow architecture. However, a distinct architecture model allows the compiler to explicitly schedule copy operations between clusters. The algorithm distributes operations among clusters, trying to avoid the inclusion of copy operations in the critical path. This should minimize increases in the schedule length due to partitioning. Register allocation is also taken into account during the partitioning phase, avoiding further complications due to eventually required spill code.

2.3.4 Other Register File Organizations

Alternative register file organizations have been proposed in addition to the ones described above. The Cydra 5 was designed as a VLIW machine to achieve high performance when executing innermost loops [83]. A set of *rotating register files* supports the execution of software pipelined loops. The rotating register file is addressed using an iteration frame pointer (IFP), which is decremented on each iteration [18]. The result is that a particular register reference actually refers to a distinct physical register on each iteration. The rotating register file concept is an effective technique to deal with overlapped lifetimes produced by software pipelining schedules. However, it does not address access conflicts and register port requirements, which can be high for wide issue machines. Furthermore, the Cydra 5 architecture employed a crossbar interconnection among FUs and RFs, which is not a scalable solution.

The regularity in memory access patterns found in some classes of applications, like DSP, has motivated the design of other storage structures. Aloqeely and Chen proposed *queues* and *stacks* to store values not requiring random access [5]. They are implemented using chains of shift registers. Storage structures resembling *FIFO* (first-in first-out) or *LIFO* (last-in first-out) queues may reduce the access time and hardware costs. This can be accomplished because there is no need of address generation and decoding logic to access intermediate positions. However, those organizations require register allocation schemes more complex than conventional ones. A similar approach was proposed using circular queues [9], also using shift registers. This implementation may impose further constraints to the register allocator: values must be written to and read from fixed physical locations, at the end points of the storage structure. It can be possible that the first logical value in a queue is not stored in the first physical location in the queue, requiring extra cycles to perform shift operations. This may delay the schedule of an operation dependent on that value. This problem can be tackled using a structure called *sequential read-write memory (SRWM)* [46]. A register bit is used to control which memory location should be accessed, eliminating the need for global shifts of values. These structures have been reported as more efficient in terms of silicon area and power consumption than the ones using shift registers [36]. All of those works proposed register allocation schemes exploiting particular characteristics of the application programs, enabling the use of the non-conventional RF organizations.

2.4 VLIW Compilation Issues

The ultimate goal of a compiler is to produce code that minimizes the total execution time of a program (*runtime*). Compiler optimizations for sequential RISC processors accomplish this by minimizing the *instruction count* (number of operations executed). For ILP processors the correlation between these two factors is not necessarily the same. Some schemes actually increase the number of operations executed in order to minimize runtime. Producing code for ILP architectures requires knowledge of the available parallelism at both software and hardware levels. This allows the compiler to transform the program in order to optimize the use of machine resources, reducing the *cycle count* (number of cycles to complete the program execution).

Programs are often represented as graph structures, which expose data dependence among operations, and also opportunities for parallelism exploitation. A target machine model, similar to the actual hardware of the target machine, should feed the compiler with the available hardware parallelism. A detailed machine model description [42] may allow the compiler to produce better quality code. However, this exposes one of the drawbacks of statically scheduled ILP machines: a new recompilation is required for every distinct configuration of a given architecture.

Compilers for ILP architectures must find enough parallelism to exploit the available machine resources. This involves several program analyses and transformations. Some designs focus on optimizing code for numeric applications, often accelerating loop execution, using techniques such as software pipelining [16].

General purpose machines must deal with non-numeric (scalar) code. Although some attempts have produced acceptable results [32], research is far from complete in this area.

2.4.1 Overview of the Compilation Process

The quality of compiler parallelization techniques can potentially make a difference of an order of magnitude in the performance of processors exploiting ILP. In order to parallelize a program, three tasks must be performed by a compiler:

- Analyze the program to determine dependences between instructions.
- Perform optimizations to remove those dependences.
- Schedule instructions to be executed in parallel.

A simplified representation of the phases constituting an optimizing compiler for a VLIW machine can be seen in Figure 2.7. The front end takes the source code and performs lexical, syntactical, and semantic analysis [3], translating the program into an *intermediate code.* Control and data flow analysis is then performed, providing the information required to apply machine-independent and machine-dependent *optimization* techniques [7]. Parallelism among instructions can be represented by a data dependence graph, which is used by sophisticated code scheduling techniques. Finally, the code generator produces the object code for the target architecture. Optimization techniques can interact with each other, so the order in which they are performed can change significantly the final effect. For this reason, compiler implementations employing the same optimizations can adopt a distinct *phase ordering.*

2.4.1.1 Dependence analysis

Optimizing compilers rely most heavily in a technique called *dependence analysis* [8]. A dependence is a relationship between two computations that places constraints on their execution order. Dependence analysis is used to determine whether a particular program transformation can be applied without changing the program behaviour.

Deciding which operations can execute in parallel requires knowledge about which operations *must* follow other ones. A *dependence* exists between two operations if interchanging their order changes the results. In the following examples we assume that executing Op_1 before Op_2 ensures the correct semantics. Dependence analysis can be used to verify whether that ordering can be changed.

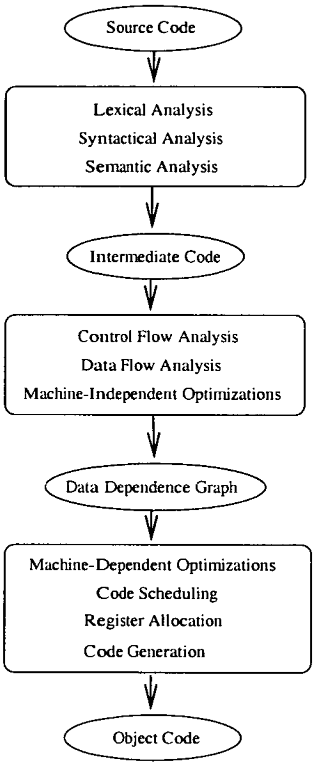


Figure 2.7: Phases of an optimizing VLIW compiler

Dependences can be one of two types: *data dependence* and *control dependence* [8]. Data dependences can be further subdivided into three types:

- *True dependence:* It is said that Op_2 has a true dependence on Op_1 if Op_1 writes a variable that is read by Op_2. In the following example the dependence exists because of R_1, which *must* be calculated (Op_1) *before* it is used (Op_2). True dependences are due to the program semantics, imposing a serialization in the program execution. However, a technique called *data value speculation* may avoid this constraint by predicting the values that flow among data dependent instructions [60, 38].

$$\begin{aligned} Op_1: &\quad \boldsymbol{R_1} = R_2 * 4 \\ Op_2: &\quad R_3 = \boldsymbol{R_1} + 5 \end{aligned}$$

- *Antidependence:* It is said that Op_2 has a antidependence on Op_1 if Op_2 writes a variable that is read by Op_1. This example shows an antidependence because of R_5, which must be read (Op_1) *before* it is written over (Op_2). This dependence can be avoided if Op_1 and Op_2 use two distinct memory locations for R_5.

$$\begin{array}{ll} Op_1: & R_4 = \boldsymbol{R_5} \text{ - } 1 \\ Op_2: & \boldsymbol{R_5} = R_6 * 2 \end{array}$$

- *Output dependence:* An output dependence refers to two operations Op_1 and Op_2 writing the same variable. The example shows two instructions using the same variable (R_7), to store the result of both computations. Output dependences can also be avoided using distinct memory locations.

$$\begin{array}{ll} Op_1: & \boldsymbol{R_7} = R_8 + 1 \\ Op_2: & \boldsymbol{R_7} = R_9 \text{ - } 3 \end{array}$$

Scalar variable references explicitly refer to a name, with each statement being executed at most once. This simplifies the dependence analysis process. In loops each statement may be executed many times, thus a more elaborated dependence analysis is required. Dependences between operations from the same iteration are called *intra-iteration* dependences. Other complex dependences may be found in loop structures:

- *Loop-carried dependence*: They occur between operations from distinct iterations. If on a given iteration i the loop refers to an element with index $i-k$, the dependence *distance* is said to be k. The following example shows a simple loop without any dependence within a single iteration. However there is a dependence between two iterations: Op_2 reads a variable ($A[i-2]$) written by Op_1 from the second previous iteration. In this case the dependence distance is said to be 2.

$$\begin{array}{ll} & \text{Do i=1, 100} \\ Op_1: & A[i] = C[i] + 1 \\ Op_2: & B[i] = A[i-2] \text{ - } 5 \end{array}$$

- *Recurrence.* This is a particular form of loop-carried dependence. It occurs when a variable is defined based on the value of that variable in an earlier iteration. The following example shows a recurrence with distance 1. In this case Op_1 reads a variable produced by itself one iteration before.

Do i=1, 100

Op_1 : $A[i] = A[i-1] + 5$

The last type of dependence described refers to the program control flow:

- *Control dependence:* This type of dependence occurs when a given Op_1 determines whether Op_2 should be executed or not. A typical example of a control dependence is the conditional construct *if*, as the following example shows. Control dependences also impose a serialization in the program execution, which may be avoided with *branch prediction* schemes [45].

  Op_1 : if ($R_{10} = 5$)

  Op_2 : then $R_{11} = R_{12}$

  Control dependences can be converted into data dependences, using a transformation called if-conversion [4]. If-conversion can be applied by using predicated instructions. In this case, an instruction is executed only if the value of a third operand is equal to zero. A predicated instruction (in pseudo-assembly) for the above example is shown below. The instruction copies the contents of register R_{11} into R_{12}, according to the value of R_{40}.

  CMOVZ R_{11}, R_{12}, R_{40}

2.4.1.2 Optimizations

Optimizing a program often requires some sort of *transformation*, which may involve inclusion, elimination, and reordering of instructions. A compiler must perform three steps to apply an optimization [7]:

- Decide the region of a program to apply a given optimization.
- Verify that the required transformation does not change the program semantics.
- Transform the program.

Optimizations can be classified into *machine-independent* and *machine-dependent.* They can be further classified into local (within a single basic block) and global (across basic blocks). A *basic block* is a sequence of instructions with no branches into or out of the block, apart from the entry and exit boundaries. The goals of these optimizations are to improve the execution speed and reduce the size of program. For ILP machines they can also increase the amount of parallelism to be exploited by the scheduler. Some of the most common machine-independent optimizations [67] are:

- Constant propagation
- Forward/Backward copy propagation
- Memory copy propagation
- Arithmetic common subexpression elimination
- Redundant load/store elimination
- Dead code removal

Loop structures can be the most significant factor affecting the total execution time for many classes of applications [59]. This has motivated the development of several loop-oriented optimizations [7]. Some of them are machine-independent, capable of reducing loop overhead, improving register usage and data cache locality, among other features. A few of them are listed below:

- Invariant code removal
- Global variable migration
- Induction variable strength reduction
- Induction variable elimination

Machine-dependent optimizations take into account hardware resources of the target machine to make program transformations to further expose parallelism and exploit efficiently machine resources. The following are included among these optimizations:

- Static branch predication
- Speculative Execution
- Loop unrolling
- Loop interchange
- Loop distribution
- Software pipelining

2.4.2 VLIW Scheduling

Scheduling algorithms can be classified into three types [84], according to the control flow graph of the region being scheduled:

- Local scheduling
- Global acyclic scheduling
- Global cyclic scheduling

2.4.2.1 Local scheduling

This class of algorithms work with a single *basic block* at a time. A number of efficient techniques have been proposed to schedule basic blocks. One of the most popular is *list scheduling* [2], a scheme that schedule operations according to a given priority list, such as *highest-level-first.* However, local schedulers have a fundamental problem that prevents them from being used effectively with ILP machines: the size of a basic block. Several studies have confirmed that on average the size of a basic block ranges between 5-20 operations, limiting the possibilities of parallelism exploitation. It has been found that limiting parallelism extraction to a single basic block would yield a maximum speedup between two and four [94, 59], an unacceptable limitation for wide-issue VLIW machines. For this reason high performance can only be achieved if ILP is exploited across multiple basic blocks.

2.4.2.2 Global Acyclic Scheduling

Global scheduling operates on multiple basic blocks simultaneously, identifying *windows* of operations to be scheduled. A window is typically composed by entire procedures or regions from a procedure [86]. Global acyclic scheduling selects regions with no back edges in the control flow, a structure typically found in loops. Doing so they target mainly the loop-free stretches of code prevalent in many general purpose programs. Possibly the most well known algorithm of this class is *trace scheduling* [32]. A trace scheduler selects regions of code that could be taken according to the output of branch operations. These regions are then scheduled as if they were a single basic block. The larger the size of a region the better the possibilities of finding parallelism. However there is an implicit trade-off: a large region requires a long compilation time, and may also result in inefficient schedules due to wrong paths speculatively taken. Traces are scheduled according to their execution frequency, which can be determined using profile information or static branch prediction. The scheduler attempts to optimize the execution time of frequently executed traces, at the expense of the least frequent ones. The insertion of compensation code might be necessary in order to correct the outcome of wrongly executed paths. That may generate excessive code replication, resulting in code size explosion. A detailed implementation of this algorithm can be found in [22].

Superblock Scheduling is an algorithm derived from trace scheduling [49]. A superblock is a trace without control entries into it, except at the top, although it still allows intermediate exit points. Intermediate entry points are eliminated using tail duplication, a technique that creates a copy of the trace below the entry point, redirecting the control path to it if necessary. Each trace in a superblock is scheduled using list scheduling.

Hyperblock Scheduling also create structures with a single entry at the top, and possibly multiple exits [68]. The control flow is if-converted [4] to remove control dependences, resulting in a code with a single entry point and multiple exits. Then list scheduling is performed, followed by reverse if-conversion. The later may result in code size explosion, as portions of the schedule in which m predicates are active yield 2^m versions of the code.

2.4.2.3 Global cyclic scheduling

Algorithms of this type use basic blocks taken from multiple iterations of a loop structure. Efficient schemes have been proposed to perform cyclic scheduling of numeric applications, as most of their execution time is often spent executing loops. Trace scheduling [32] can also be used to schedule loops. In this scheme back edges in the loop control flow graph are eliminated by performing loop unrolling [20] of the loop body. Although effective, this strategy can generate code size explosion, and also inefficient processing at the start and end points of each series of unrolled iterations. Other acyclic scheduling algorithms can also be used in a similar way, however they also show this limitation.

A scheme specially developed to schedule loop structures for ILP machines is *software pipelining* [16]. The concept is similar to a hardware pipeline: successive iterations start before previous ones have completed. This is possible because the execution of operations from distinct loop iterations is overlapped, taking advantage of the available hardware parallelism. Software pipelining algorithms have to deal with machine resource constraints and data dependences among operations in the loop body. In this context, generating optimal schedules of loops with arbitrary data dependence graphs is known to be a NP-complete problem [54].

Optimal schedules can be generated using *integer linear programming*, a technique that employs precise definitions of objectives and constraints of the schedule. Some techniques assume the use of unlimited machine resources [35]. Others are realistic enough to model resource constraints and minimize register requirements [39]. Although very effective to find the best possible schedule, the complexity of methods based on integer linear programming prevent them from being used in production compilers. However, they can be a valuable tool to evaluate the effectiveness of other approaches.

Software pipelining algorithms of practical use must rely on heuristics to produce near-optimal solutions in most of the cases. *Modulo Scheduling* is a class of software pipelining algorithms targeting *innermost* loops. A basic schedule of one single iteration is generated, which is issued at fixed intervals, called *Initiation Interval* (II). The basic schedule is structured in order to preserve data dependences among operations, even if the II is much smaller than the basic schedule length. During the *steady state* a new iteration starts and another one finishes every II cycles. The basic schedule must adhere to the *modulo constraint*:

> *When the basic schedule is initiated at II intervals no machine resource should be oversubscribed. It should be noticed that an operation holding a given resource at cycle c will hold the same resource at regular intervals $c + II$. This is equivalent to saying that the resource is required every c mod II cycles, with mod denoting the modulo operator.*

The *minimum* initiation interval achievable is based on two factors:

- *Machine resources* required by the computations of one iteration of the loop body. These resources are functional units, buses, and register file ports, among others.

- *Recurrence circuits* in the loop data dependence graph. A recurrence occurs when a given operation has a direct or indirect dependence upon the same operation from a previous iteration.

The first approach to modulo scheduling was proposed in [80]. This algorithm was targeted at machines with simple resource usage patterns and loops with no recurrence circuits. An extension of this algorithm, able to deal with complex recurrences and machine usage patterns, is *Iterative Modulo Scheduling (IMS)* [79]. Like several other modulo scheduling algorithms, IMS uses a variation of list scheduling to produce the basic schedule of one iteration. The algorithm allows *backtracking* (unschedule and reschedule of operations) to deal with the extra complexity involved. Loops with arbitrary control flow can also be scheduled using *if-conversion* [4], as long as hardware support is available. A detailed description of this algorithm is shown in Section 3.3. This approach can also be used to deal with further constraints: *Slack scheduling*, for instance, incorporates heuristics to shorten lifetimes. which helps to minimize register requirements [48].

An algorithm proposed by Lam uses a *hierarchical reduction* scheme to convert code fragments containing control constructs into single nodes [54]. This allows control flow structures to be modulo scheduled without special hardware support for predicate execution. This work also proposed a new optimization called *modulo variable expansion (mve)*. The technique allows register allocation to be performed without any special hardware support, such as rotating register files [18].

Another technique, called *swing modulo scheduling*, produces efficient schedules in terms of initiation interval, stage count and register requirements [63]. It also compares favourably against other schemes in terms of complexity, requiring low compilation time.

Loops with conditional branches have been the topic of some research work [18, 54]. As schedules presenting a single II can be inefficient according to the path taken at run time, algorithms to generate schedules with different II have been proposed [91, 97]. Finally, an attempt to use modulo scheduling efficiently with non-numeric applications is reported in [57].

2.4.3 Register Allocation

Optimizing compilers of the 1990s usually perform register allocation using efficient techniques such as *graph coloring* [15]. Ideally, the number of available registers should be enough to keep all the live values in on-chip storage. If that is not the case, some values should be stored in the main memory and reloaded when required, a process called *spill code* [11]. However, the difference in access times between registers and main memory can be very high. For this reason, spill code should be minimized or completely avoided in order to achieve high performance levels.

In conventional (sequential) processors, register allocation might be a more important step than instruction scheduling. As there is no parallelism involved, the performance is determined by the number of operations executed, and not the ordering among them. One factor that could compromise the performance is an insufficient number of registers [15]. For this reason, register allocation was regarded as a more important step, and usually performed before scheduling in early compilers. However, for multiple-issue microprocessors the phase ordering between these two steps is not a clear choice [84]. Performing register allocation *before* scheduling may minimize the use of spill code. However, this might result in the introduction of unnecessary edges in the data dependence graph to be scheduled, preventing parallelism exploitation. The alternative is to perform register allocation *after* scheduling, at the expense of possible excessive spill code. For statically scheduled processors this can result in a severe performance penalty. Situation may arise when the entire processor has to stall to perform a single memory access.

Multiple-issue machines already strive to find ILP among operations, so extra dependence constraints should be avoided [94]. On the other hand, these machines can generate high register pressure [69]. Both phase-ordering alternatives above described can potentially result in performance degradation. For this reason some sort of cooperation between these two tasks has been suggested [73]. Some approaches have adopted a multi-pass procedure: pre-scheduling, register allocation (introducing spill code if necessary), and final scheduling [66].

Performing register allocation before modulo scheduling a loop can place unacceptable constraints in the scheduler [81]. For this reason it is often performed after the scheduling phase. In the Cydra 5 machine a failure to allocate registers would result in incrementing the II and completely rescheduling the loop [18]. Dealing with the issue of overlapped lifetimes imposes further complications in the process, requiring non-conventional techniques. This can be addressed using rotating register files [18], which implements in hardware a sort of register renaming scheme transparent to the compiler. If hardware support is not available, modulo variable expansion (mve) can be used [54, 81]. In this case the code of the loop body must be unrolled a number of times to ensure that no lifetime is longer than the replicated kernel. Register allocation for modulo scheduled loops is described in detail in [81].

2.4.4 Code Generation

The generation of modulo scheduled code is affected by a number of issues: if the program structure is a *do-loop* or *while-loop*, the type of hardware support provided, whether the loop has live-in or live-out scalar variables, and the register allocation scheme employed. Distinct schemes can be used according to these factors, with variable implications in performance and code size [82].

The first and last few loop iterations are scheduled in the prologue and epilogue stages, and could also be the only ones for loops with small iteration counts. Dealing with them requires a mechanism to enable the execution of subsets of the kernel code, as the instruction pipeline is not completely filled in those stages. Using a rotating register file along with support for predicate execution allows the implementation of a scheme called *kernel-only* code [82]. In this case a single copy of the kernel is sufficient to execute the entire modulo scheduled loop, preventing code size explosion.

Generating code using a machine with a conventional register file may require the use of mve [54]. The drawback of this technique is the possibility of code size explosion, which can result in a high frequency of instruction-cache misses. This can be avoided using a rotating register file [18], or any other scheme able to perform a sort of dynamic register renaming.

2.5 Commercial VLIW Machines

The first VLIW processors built were the so-called attached array processor, of which the best known were produced by Floating Point Systems [84]. The next generation of products were the minisupercomputers Trace and Cydra. A growing interest in VLIW architectures has been shown in the late 1990s, particularly for specialized applications such as DSP and multimedia [23]. New products employing this technology have been released by Philips and Texas Instruments, among other companies.

Multiflow Trace computers were produced from 1984 to 1990. They were offered as general purpose machines, relying on a trace scheduling compiler to find ILP in a large class of applications [66]. Although successful in finding parallelism in systems applications, compilation time limitations made the machine best suited for scientific code. The machine was organized into clusters of functional units and register files, with issue-rate ranging from 7 to 28 instructions per cycle.

Cydra 5 was a minisupercomputer developed by Cydrome between 1984 and 1988. It was designed to serve high-performance scientific computing [18]. It can be seen as a heterogeneous multiprocessor system [83]. Interactive processors are responsible for all non-numeric computations such as operating system, compilation, I/O, etc. In addition, a numeric processor is used to execute scientific computations. The numeric processor was implemented as a VLIW architecture with seven functional units, using software pipelining scheduling techniques [18].

Trimedia is a family of programmable multimedia processors from Philips

Semiconductors. TM-1000 is the first product from this family, designed to concurrently process video, audio, graphics, and communication data. The architecture is based upon a high performance VLIW CPU core [76], consisting of 27 functional units. However, only 5 instructions can be issued simultaneously, mainly due to a limited number of register file ports (15 read and 5 write ports). Hardware support to implement guarded instructions is exploited by the compiler to eliminate branches, and thus increase ILP identification.

Texas Instruments VelociTi is a VLIW architecture. The TMS320C601 is DSP processor from this family [89]. Its CPU has 8 independent functional units running at 200 MHz. The CPU has two identical data paths with four functional units each. Each data path has a register file with 10 read and 6 write ports. A cross path allows read operations from the other register file. The compiler for this architecture performs software pipelining and loop unrolling, among other optimizations.

Chapter 3

Basic Experimental Framework

This chapter describes the basic structure of the experimental framework used to develop the VLIW architecture proposed in this thesis. We have adopted a *hardware/software codesign* methodology [34], to develop a machine model and compilation techniques to accelerate the execution of loop intensive applications. Results and conclusions obtained from experimental analysis have supported all stages of the project. The framework organization reflects the main research topics in which work was conducted to complete the thesis, as listed below:

- VLIW architectures
- Modulo scheduling
- Register allocation

The basic input to the experimental framework is an *innermost loop*, represented by operations and the corresponding data dependencies information among them. The output produced consists of a modulo schedule, performance and machine resources analysis. This information is used to guide further improvements in hardware or software aspects, restarting the process shown in Figure 3.1.

The framework was implemented using the C++ language and the LEDA library routines [70], which are particularly useful for graph manipulation. This chapter describes only those aspects which are common to all the experiments reported. Additional capabilities are introduced in the relevant chapter, such as new register file organizations and heuristics for the scheduling algorithm. The next sections present a detailed description of the components of the experimental framework.

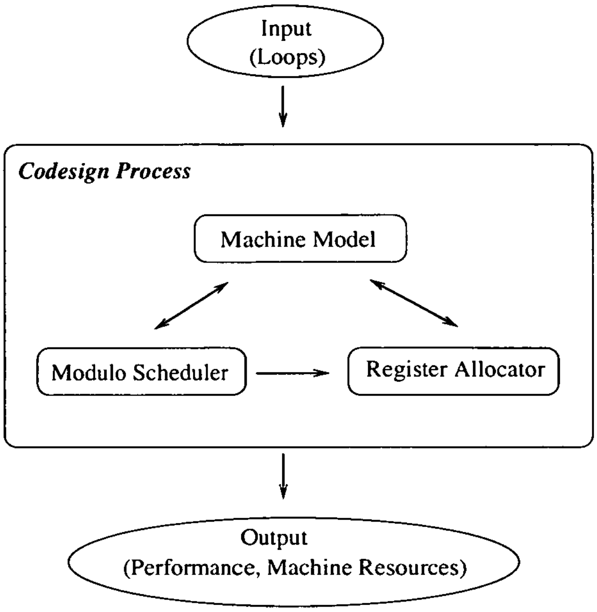


Figure 3.1: Hardware/Software codesign process

3.1 Machine Model

The machine model used by the experimental framework consists of a collection of two basic components: *functional units* and *register files*. The framework provides enough flexibility to change some characteristics of these elements, allowing distinct machine configurations to be considered.

3.1.1 Functional Units

We call a **functional unit (FU)** an element of a microprocessor data-path capable of performing actual computations or memory access operations. Modern microprocessors use a technique called *pipelining* to implement functional units [45]. A pipeline is similar to an assembly line, in which a task is subdivided into simpler subtasks. Although an operation requires the completion of all steps in succession, the pipeline can work on distinct operations in parallel. Depending on the pipeline organization, it is possible to have as many operations simultaneously *in flight* as the number of *pipeline stages*. We assume the use of a **register-register** architecture [45], which implies that memory access can only be made through load and store operations. For any other operation, the destination and source operands are stored in the on-chip register file. A simplified organization of a pipeline can be seen in Figure 3.2, showing the five main stages of a pipeline:

1. *Instruction Fetch (IF)* : Fetch an instruction from memory into the instruction register.

2. *Instruction Decode/Register Fetch (ID)* : Decode the instruction and read the input operands from the register file. These operations can be done in parallel.

3. *Execution (EX)* : Execute an instruction, which can be an ALU (arithmetic and logic unit) computation, memory address calculation, or branch operation.

4. *Memory Access (MEM)* : Access to memory (usually cache) is performed at this stage, as specified by load and store operations. It also updates the PC (program counter).

5. *Write Back (WB)* : Write the result into the register file.

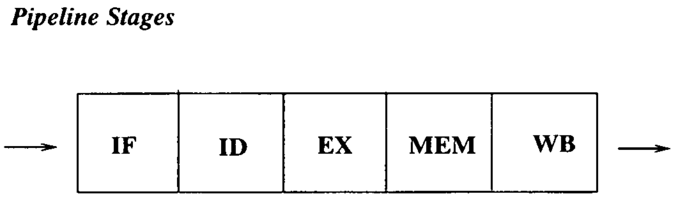


Figure 3.2: Generic organization of a microprocessor pipeline

Each pipeline stage, except execution, takes one cycle to complete. Thus, the machine cycle may be determined by the slowest of those pipeline stages. Real implementations may adopt other organizations, combining or further subdividing stages to allow a shorter cycle. The interested reader can refer to [45] for a detailed discussion on pipelining techniques, as it is outside the scope of this thesis.

We define the latency of an operation as the total number of cycles required to issue, execute, and make the result available for use. It is possible for a functional unit to execute more than one type of operation, each of them possibly requiring distinct latencies to complete. RISC and VLIW processors usually use a technique called *bypassing* [1]. Bypassing allows forwarding operands directly from the producer to the consumer operation. Doing so, it is possible for an operation to use a value before it has actually been written in the register file.

Real implementations have separate FUs to perform integer and floating point operations. For the sake of simplicity our machine model uses the same FUs to perform both types of operations. We consider four types of functional units:

- L/S: Executes memory load and store operations, transferring values between the main memory and the register file.

- ADD: Adder unit, executing addition, subtraction, type conversion, conditional branch, compare, const (load immediate), and absolute operations.

- MUL: Multiplier unit, executing multiplication, division, square root and modulus operations.

- COPY: Auxiliary functional unit, used to duplicate and move values between register files. It is capable of reading one value from a register file and writing it back to one or two other register locations. Specially designed to support architecture features as defined in Sections 4.2 and 7.2.

All functional units are fully pipelined, being able to start a new operation at any cycle. Operations may have distinct latencies, as shown in Table 3.1. This is due to the pipeline execution stage, which consists of one or more stages, according to the instruction being executed. Those latencies do not include the first two pipeline stages as they are common to all operations and do not appear on the critical path, except after a misprediction. The presence of bypassing hardware, to forward results before the write-back stage, is assumed. Throughout this thesis we use the term **standard** when referring to the functional units usually found in other microprocessors: L/S, ADD, and MUL. Although performing a simple operation, the Copy FU is not included in that group as it has been specially designed for this architecture.

In order to consider memory operations (loads and stores) with fixed latency, we assume a *perfect cache* hit ratio. Hence we have not considered as yet a memory system coupled with the set of functional units and register file. It is well known that providing the required memory bandwidth for a large number of functional units is one of the main issues in the design of high performance microprocessors. However, the issues addressed by this work are scheduling and register file organizations. Designing an efficient memory system is a problem common to most architecture designs, thus we have avoided increasing the level of detail at this stage of the work. Furthermore, current technology trends indicate the future possibility of building systems integrating powerful processors and main memory on a single chip [53]. This might address some the issues posed by wide-issue VLIW machines.

Functional Unit	Instruction	Latency	Issue Interval
L/S	Load	2	1
L/S	Store	1	1
ADD	Addition	3	1
ADD	Subtraction	3	1
ADD	Conversion	3	1
ADD	Branch	1	1
ADD	Compare	1	1
ADD	Const	1	1
ADD	Absolute	1	1
MUL	Multiplication	4	1
MUL	Division	17	1
MUL	Modulus	17	1
MUL	Square root	30	1
COPY	Copy values	1	1

Table 3.1: Functional unit characteristics

3.1.2 Local Register File

As mentioned earlier, 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]. The simplest design option is to use a multiported register file with R read ports and W write ports, an organization called a **monolithic register file** (Section 2.3.2).

In this basic version of the experimental framework we assume that all functional units are connected to a monolithic register file, called the **local register file**. It should provide the bandwidth required by the functional units (Section 3.1.1). Specifically, most FUs require 2 read and 1 write ports. The only exception is the COPY functional unit. which requires 1 read and 2 write ports, as shown in Table 3.2.

Functional unit	Read Ports	Write Ports
L/S	2	1
ADD	2	1
MUL	2	1
COPY	1	2

Table 3.2: Register file access port requirements

3.1.3 Unclustered Machine

The first machine model built in the experimental framework comprises a number of functional units connected to a monolithic register file, using the components described in Sections 3.1.1 and 3.1.2. We call this hardware organization an Unclustered Machine. As an example, a simple VLIW unclustered machine could be organized using 1 L/S, 1 ADD, and 1 MUL functional unit, as shown in Figure 3.3. The local register file requires 6 read and 3 write ports.

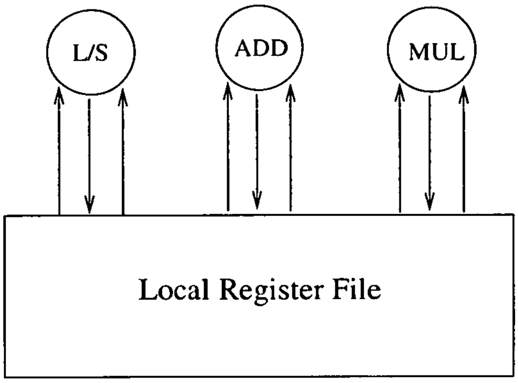


Figure 3.3: VLIW unclustered machine

This organization is the natural choice of design to implement a VLIW machine. It can be seen as a direct extension of a superscalar processor, without some of its complexities. Although it works well for a moderate number of functional units, it presents scalability problems, as previously discussed. We have used this basic architecture for comparison purposes with the new machine organization proposed in this thesis.

3.2 Workload

An innermost loop is the basic input to the experimental framework. All eligible innermost loops from the Perfect Club Benchmark [10] that are suitable for software pipelining are used. The total number of selected loops is 1258. They were obtained using the ICTINEO compiler [6], which performed all data dependencies analysis and optimizations necessary to use modulo scheduling techniques. The compiler generates information regarding loop operations and data dependencies. These are in turn taken by the experimental framework, which reconstructs the corresponding data dependence graph (*DDG*) to be used by the modulo scheduling algorithm (Figure 3.4).

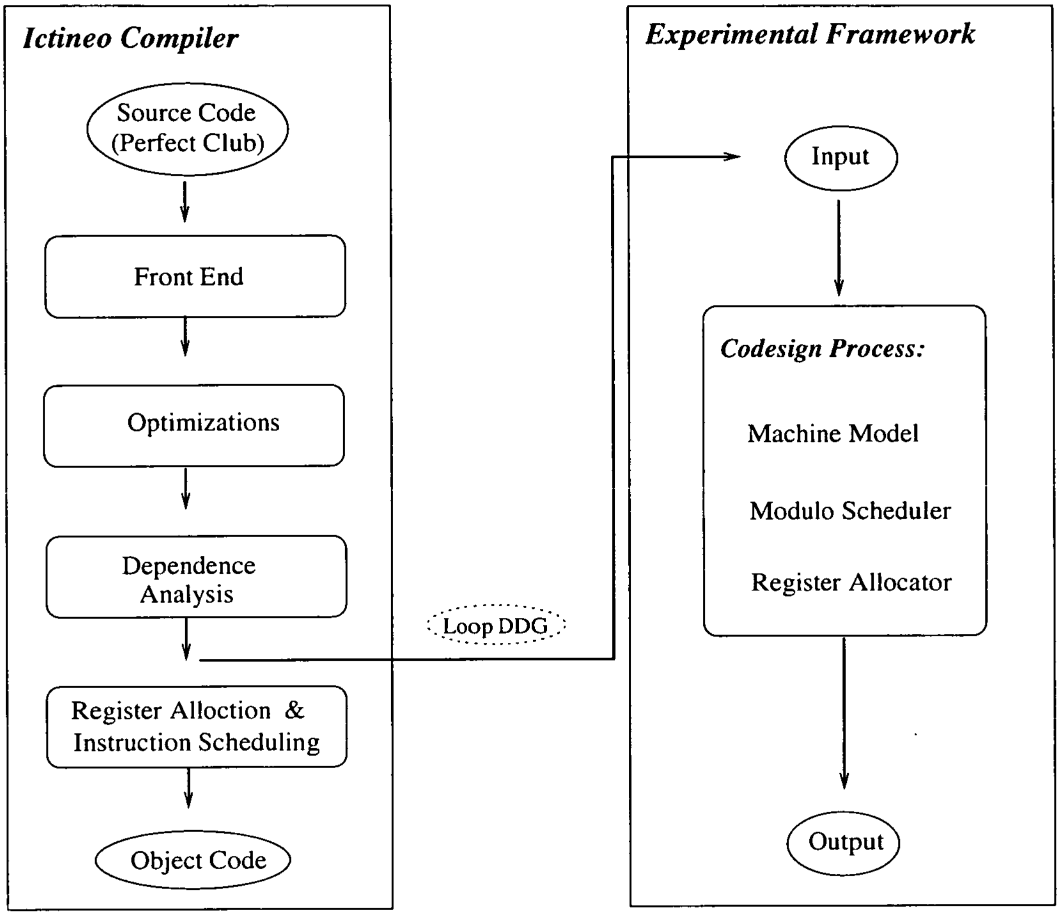


Figure 3.4: Extracting loops from the benchmark

3.2.1 Perfect Club Benchmark

The Perfect Club Benchmark [10] is composed of 13 programs, containing approximately 60,000 lines of code written in Fortran-77. They are numeric intensive programs selected from science and engineering applications. Only loops without subroutine calls and without conditional exits were selected. Although some techniques have been developed to deal with early exits [57], using them is out of the scope of this thesis. The selected 1258 loops represent 78% of the total execution time of the benchmark, when executed on a HP-PA 7100 computer [61]. The data in Table 3.3 shows the number of loops extracted from each program in the benchmark. It also shows the fraction of the total execution time spent in those loops.

Those loops present a varied *instruction mix*, including all the instructions listed in Table 3.1. However the static instruction count reveals that four in-

Program	No. Loops	Execution Time - %
ADM	151	79
SPICE	57	9
QCD	90	43
MDG	31	62
TRACK	49	30
BDNA	152	69
OCEAN	74	97
DYFESM	104	98
MG3D	80	70
ARC2D	139	95
FLO52	81	92
TRFD	25	97
SPEC77	225	85
Total	1258	78

Table 3.3: Loops extracted from the Perfect Club benchmark

structions are responsible for about 96% of the total instruction count, as seen in Table 3.4. This analysis was used to define the type and characteristics of the functional units employed by the machine model.

Instruction	Static Count - %
Load	21
Store	15
Addition	24
Multiplication	36
Total	96

Table 3.4: Instruction mix of the selected loops

3.2.2 Selection of Loops and Compiler Optimizations

As already said, the loops used by the experimental framework were extracted using the ICTINEO compiler [6], a research tool developed at *The Universitat Politecnica de Catalunya.* It performs a number of optimizations, including the following:

- Constant value propagation
- Common subexpression elimination

- Strength reduction/Induction variable recognition
- Dead code removal
- Invariant removal
- Privatization
- Interprocedural analysis

All loops suitable for modulo scheduling [79] are identified. Then if-conversion [4] is performed to eliminate conditional structures from the loop body. In addition, sophisticated data dependence analysis is performed in the innermost loop. This includes symbolic analysis of array subscripts, a technique capable of identifying loop-carried dependencies. A dependence graph is produced containing information about the operation associated with each node, and two edge attributes: dependence type and dependence distance. These attributes are described in Section 3.2.3. Information regarding loop invariant lifetimes is also supplied, which is required to estimate register requirements.

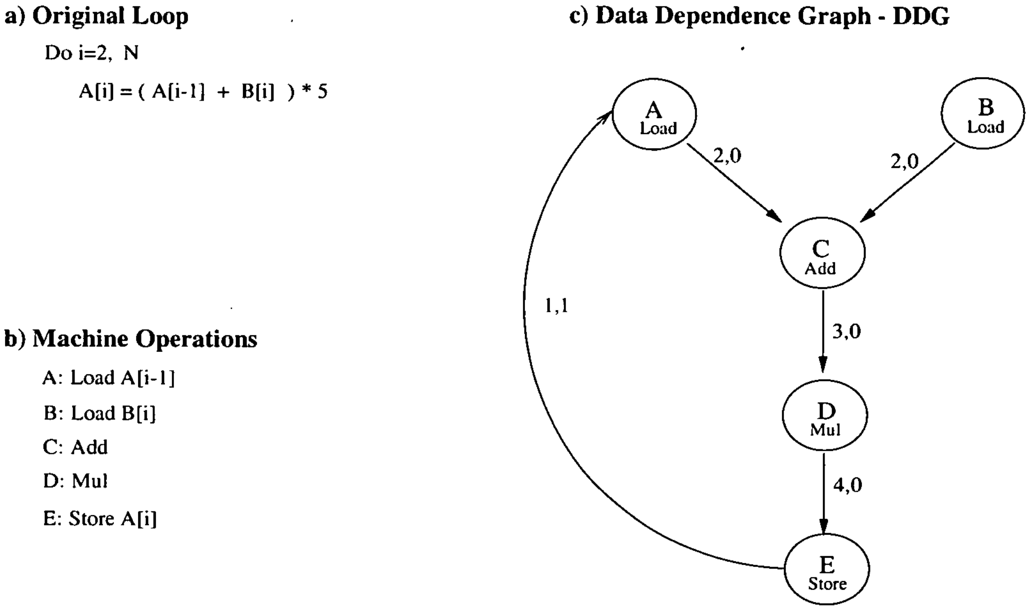
3.2.3 Data Dependence Graph

A **data dependence graph** (*DDG*) can be used to represent the dependencies among loop operations. Let the data dependence graph be represented by *DDG(N;E)*, where N is the set of **nodes** and E is the set of **directed edges**. Each node $v \in N$ represents an operation in the loop body. Each edge $e = (u, v) \in E$ represents a dependence between two operations u, v. It is said that u is the **source (predecessor)**, and v is the **target (successor)** operation. The target operation depends on the source operation, as discussed in Section 2.4.1.1. The number of edges leaving a node u is called *out-degree(u)*. The number of edges entering a node u is called *in-degree(u)*.

There are two attributes associated with each edge e: λ_e and δ_e. The first one, λ_e, is the number of time units the source operation u takes to execute, also known as the *delay*. The second attribute, δ_e, represents the dependence *distance* between them, as defined in Section 2.4.1.1.

A *circuit* in the data dependence graph indicates the existence of a recurrence. Every operation on a recurrence circuit must all be part of the same *strongly connected component (SCC)*. A SCC is the largest sub-graph of the *DDG* such that a path exists from every node to every other node. Subdividing a DDG into SCC can be useful to reduce the computational complexity of some procedures usually found in optimizing compilers, reducing the compilation time.

The example in Figure 3.5 shows an innermost loop (a) and the corresponding machine operations (b). The data dependence graph (c) represents the required order of execution of those instructions. Each edge has a pair of values, representing the delay and the distance of each dependence, respectively. The delay refers to the latency of the functional unit executing the source operation. In this example we use the values as defined in Table 3.1. The only dependence distance that is not equal to zero is the one originating in node E. It enforces that the load operation (node A) starts executing only after the completion of the store operation (node E) *from one previous iteration.*



a) Original Loop

```
Do i=2, N
    A[i] = ( A[i-1] + B[i] ) * 5
```

b) Machine Operations

A: Load A[i-1]

B: Load B[i]

C: Add

D: Mul

E: Store A[i]

c) Data Dependence Graph - DDG

Figure 3.5: Innermost loop and data dependence graph

3.3 Modulo Scheduling Algorithm

This section describes the implementation of the code scheduling process adopted by the experimental framework. The core algorithm used is Iterative Modulo Scheduling (IMS) [79]. A number of intermediate code optimizations are performed by the ICTINEO compiler [6] before an innermost loop can be modulo scheduled. These include the elimination of redundant loads and stores, if-conversion of branches, and minimization of anti- and output dependencies. Other possible optimizations depend on the hardware support available. A more detailed discussion of this issues can be found in [79].

Given a data dependence graph *DDG* representing an innermost loop, the code scheduling process is summarized by the Algorithm 3.1. The first step calculates the minimum initiation interval, which will be used in the first invocation of the IMS algorithm. If the algorithm fails to find a valid schedule, the II is increased and IMS is reinvoked. This process is repeated until a valid schedule is found.

The schedule of a single iteration can be divided into stages of II cycles each. The number of stages in one iteration is called **stage count** (*SC*). During each of the first (*SC-1*) stages, a new iteration starts without the first one having ended yet. This phase is called the **prologue**. From the *SC-th* stage onwards, one iteration starts and another one finishes every II cycles, a phase called the **kernel** or **steady state**. The last (*SC-1*) do not start any new iteration, but only execute instructions from the last iterations started during the kernel phase. This phase is called the **epilogue**. The last steps of the scheduling algorithm consist of generating code for the prologue and epilogue phases. The code for these stages can be directly derived from the kernel code produced by IMS. As already said, the scheme called kernel-only code prevents code size explosion, however it requires hardware support for both predicated execution and register renaming of loop variants (such as rotating register files or an equivalent scheme). In this work we assume the existence of the required hardware support, thus kernel-only code is generated.

Algorithm 3.1 *Modulo Scheduling*

```
Schedule(DDG)
  Calculate_MII(MII)
  /* Initialize II to the MII */
  II= MII
  completed = 0
  /* Perform IMS until a valid schedule is found */
  /* If necessary, increase the II */
  While (not completed)
    completed= IMS(II)
    If (not completed)
      II= II + 1
  }
  Generate Prologue
  Generate Epilogue
```

3.3.1 Minimum Initiation Interval - MII

The minimum initiation interval, *MII*, is a lower bound on the smallest possible value of II for which a modulo schedule exists. The *MII* can be calculated by analysing the *DDG* representing the computations of the loop body. One lower bound is derived from the resource usage requirements of these computations, *ResMII*. The other one, *RecMII*, is defined according to the latency of recurrent circuits in the *DDG*. The *MII* must be equal to or greater than both lower bounds, being calculated using Algorithm 3.2.

Algorithm 3.2 *Calculate MII*

```
Calculate_MII(MII)
  /* Based on machine constraints */
  Calculate_ResMII(ResMII)
  /* Based on recurrence constraints */
  /* Start with the minimum acceptable value, to reduce the compilation time */
  candidate= ResMII
  completed = 0
  While (not completed) {
    completed= Calculate_RecMII(candidate)
    /* Candidate MII too small */
    If (not completed)
      ++candidate
  }
  RecMII= candidate
  /* Based on both machine and recurrence constraints */
  MII= max(ResMII, RecMII)
```

3.3.1.1 Calculating ResMII

The *ResMII* is calculated by totalling the usage of machine resources required by one iteration of the loop. In this experimental framework we take into account only the usage of functional units, to calculate *ResMII*. Thus the most heavily used FU determines the *ResMII*, which can be calculated using Algorithm 3.3. As we are assuming the use of fully pipelined functional units, each operation holds a FU during one cycle only. The II must be an integer, so the value calculated for *ResMII* is rounded up to next integer.

Algorithm 3.3 *Calculate ResMII*

```
Calculate_ResMII(ResMII)
  /* Compute the number of operations to be issued by each type of FU */
  forall FU of type i do
    usage[i]= 0
  forall operation $u \in DDG$ {
    if u uses FU of type i
      ++usage[i]
  }
  ResMII= 0
  forall FU of type i do {
    /* Compute the number FUs of each type */
    resource_count = number of FUs of type i
    /* MII based on a given type of FU */
    $candidate = \left\lceil \frac{usage[i]}{resource\_count} \right\rceil$
    if (candidate > ResMII)
      ResMII= candidate
  }
```

3.3.1.2 Calculating RecMII

The *RecMII* imposes a lower bound on the II due to recurrence circuits in the loop *DDG*. A loop contains a recurrence if an operation in one iteration has a direct or indirect dependence upon the same operation from a previous iteration. An *elementary circuit* in a *DDG* is a path through the graph which starts and ends at the same node, and which does not visit any vertex on the circuit more than once. An elementary circuit c indicates a recurrence in the *DDG*. Let $delay(c)$ be the sum of the delays along the circuit c, and $distance(c)$ the sum of distances. It can be said that $delay(c)$ is the minimum time interval between the issue of an operation on the circuit, and the same operation $distance(c)$ iterations later. Considering that all iterations have the same schedule, only delayed by II cycles, the elementary circuit imposes the following lower bound on the II:

$$delay(c) \ \leq \ II \times distance(c) \tag{3.1}$$

The *RecMII* is determined by considering the worst-case across all elementary circuits in the *DDG*. We have adopted the approach proposed in [48] to determine the *RecMII.* The algorithm *ComputeMinDist* calculates, for every pair of operations $(u, v) \in DDG$, the minimum time interval between the schedule of both operations *from the same iteration* (Algorithm 3.4). The main data structure of the algorithm is the matrix $MinDist[u, v]$. An entry $[u, v]$ specifies the minimum time interval between operations u and v. If there is no path from u to v in the DDG, the value of entry $[u, v]$ is set to $-\infty$. If $MinDist[u, u]$ is positive for any u, it means that u must be scheduled later than itself, which is impossible. This indicates that II is to small. If all diagonal entries are negative, it indicates a slack around all recurrence circuits, resulting from a II higher than necessary. The goal of the algorithm is to find the minimum II for which there are no positive entries and at least one entry equal to zero in the diagonal.

Algorithm 3.4 *Calculate RecMII*

```
Calculate_RecMII(candidate)
  II= candidate
  /* Initialize the distance matrix with the minimum delay */
  /* between pairs of dependent operations u and v */
  forall operation u ∈ DDG {
    forall operation v ∈ DDG {
      MinDist[u,v]= $-\infty$
      forall edge e(u,v) ∈ DDG
        MinDist[u,v]= max(MinDist[u,v], ($\lambda_e - II \times \delta_e$ ))
    }
  }
  /* Now consider all possible paths via an intermediate node w */
  forall operation w ∈ DDG {
    forall operation u ∈ DDG {
      forall operation v ∈ DDG
        dist= MinDist[u,w] + MinDist[w,v]
        if (dist > MinDist[u,v]) {
          MinDist[u,v]= dist
          /* Candidate II too small-that would result in a impossible */
          /* constraint: a scheduling delay between the same operation */
          if (u==v) and (dist > 0)
            return 0
        }
    }
  }
  return 1
```

The algorithm complexity is $O(N^3)$, which is expensive for *DDG* with a large number of nodes. This problem can be minimized if small subsets of the *DDG* are used. The *RecMII* can be calculated for each strongly connected component of the graph. The highest value computed determines the *RecMII* of the *DDG*. This strategy is used by the experimental framework to optimize the algorithm running time. Several algorithm executions may be necessary until the *RecMII* is found. The first invocation uses *ResMII* as a candidate *RecMII*. This is acceptable as we are interested in finding the *MII*, and not the actual *RecMII*.

3.3.2 Iterative Modulo Scheduling - IMS

Iterative modulo scheduling uses a goal-directed search for a legal schedule at the candidate II. The strategy employed is similar to *list scheduling* using *height-based* priorities [2]. However, it is possible that a partial schedule results in a dead-end state. In this case no additional operation can be scheduled, unless the II is increased, which should be avoided as much as possible. IMS tries to break dead-end states using *backtracking*: previously scheduled operations are ejected from the partial schedule. Backtracking allows the scheduling process to resume from a different path in the search for a valid schedule. Those unscheduled operations will be rescheduled, possibly in a distinct slot from the previous one. Repeatedly scheduling and rescheduling operations lends the term *iterative* to the algorithm [79]. If the search for a valid schedule fails after a large number of steps, it is assumed that no solution exists for the candidate II. When this happens, the II is increased and IMS reinvoked. The number of scheduling steps attempted before increasing the II is controlled by a parameter called *budget*. We have used a budget equal to *three* times the number of operations in the *DDG*. This was based on an evaluation published in [79] and also our own observations. During the early stages of this research work we tried using smaller and larger values for the budget parameter. A smaller budget sometimes resulted in unnecessary increase of the II. On the other hand, in most of the cases increasing the budget was not effective to avoid increasing the II, only causing compilation delays due to additional backtracking. The Algorithm 3.5 describes the main steps of IMS, followed by a detailed description of those steps.

IMS uses a structure called Modulo Reservation Table (MRT) to keep track of machine resource usage during the scheduling process [54, 79]. MRT records when a particular resource is in use by an operation at a given cycle. As already said, this experimental framework only keeps track of FU usage. Thus each FU of the machine model has a total of II reservation slots, as seen in Figure 3.6. This simple representation is possible because of the modulo scheduling nature: an operation holding a given resource at cycle *c* will hold the same resource every

II cycles. Thus the MRT needs to be only II long. As operations are scheduled, the corresponding slots are marked as *used.* Scheduling an operation in a given cycle is legal only if it does not result in an attempt to use a slot more than once.

MRT for II= 5

L/S	ADD	MUL
used		used
	used	
used		
	used	

Figure 3.6: Modulo Reservation Table

Algorithm 3.5 *Iterative Modulo Scheduling*

```
IMS(II)
  budget= 3 × (No. operations in DDG)
  Create_Priority_List(List)
  While (List not empty) and (budget > 0) {
    Get(List,OP)
    /* mintime is the earliest start time for OP according to */
    /* currently scheduled predecessors */
    mintime= Earliest_time(OP)
    /* Select a valid slot */
    slot= Find_Slot(OP, mintime)
    /* According to the slot chosen, unschedule all operations due to */
    /* resource and dependence conflicts with scheduled successors */
    Backtracking(OP, slot)
    Schedule(OP, slot)
    /* Keep track of machine resources usage */
    Update MRT
    Remove(List,OP)
    budget= budget-1
    If (List is empty)
      Return 1
    If (budget == 0)
      Return 0
  }
```

3.3.2.1 Creating a priority list

IMS uses a *height* based priority function [2] to define the order in which operations are selected for scheduling. An operation with higher priority than another means it has *less* scheduling options to prevent lengthening the schedule. Furthermore, the height-based priority defines a topological sort in which a predecessor operation will have higher priority than all of its successors.

A sorted priority list can be built using Algorithm 3.6. It assumes that two pseudo-operations, *Start* and *Stop* are included in the *DDG*. These operations are never scheduled, being only used to support the strategy to deal with recurrences in the DDG. *Start* and *Stop* are respectively made predecessor and successor of all original operations in the *DDG*. In order to deal with recurrence cycles, the strong connected components of the DDG are identified. Doing so, each SCC can be viewed as a super-vertex, resulting in a acyclic graph. Creating a priority list involves a successive calls to the function *HeightR* (Algorithm 3.7), using the *Start* operation as the first argument. Every operation is assigned a priority, based on the distance to the *Stop* operation. Finally, the list of operations is sorted in decreasing order of priorities.

Algorithm 3.6 *Create Priority List*

```
Create_Priority_List(List)
  /* Identify cyclic regions of the DDG */
  Identify_SCC(DDG)
  /* Initialize operations for the graph traversal process */
  forall operation OP ∈ DDG {
    priority[OP]= $-\infty$
    visited[OP]= false
    Include(List, OP)
  }
  /* Start graph traversal */
  HeightR(START, List)
  /* Sort operations in descendent order of priorities */
  Sort(List)
```

The procedure *HeightR* computes, for each operation *OP*, the longest path from *OP* to the end of the graph, a *Stop* operation with priority zero. The height of operations not belonging to a recurrence circuit is computed in a post-order fashion, by means of a depth-first search (DFS) of a tree rooted at the *Start* operation. On the other hand, dealing with SCC requires to keep track of the first vertex (root) visited of each SCC. To facilitate this, the vertices of an SCC are collected on a stack during the DFS traverse of the DDG. Once all operations belonging to a SCC have been visited, a call to the procedure *FinalizeSCC* (Algorithm 3.8) computes the corresponding priorities.

Algorithm 3.7 *Compute Height*

```
HeightR(OP, List)
  visited[OP]= true
  if OP has no successor
    priority[OP]= 0
  else {
    /* DFS traversal */
    forall successor of OP {
      if (visited[successor] == false) {
        HeightR(successor, List)
      }
      priority[OP]= Max(priority[OP], (priority[successor] +
          $\lambda_{e(OP,successor)} - \delta_{e(OP,successor)} \times II$ ))
    }
  }
  /* Stack used to keep track of operations belonging to the same SCC */
  If (OP $\in$ $SCC_i$)
    Push(OP, stack)
  /* All operations of the SCC have been visited */
  If (OP is the root of $SCC_i$ ) {
    FinalizeSCC
    /* Update stack */
    While (Top(stack) $\in$ $SCC_i$)
      Pop(stack)
  }
```

Algorithm 3.8 *Finalize SCC*

```
FinalizeSCC()
  /* Get the fixed-point solution for the heights of all vertices of the SCC */
  first= deepest op $\in$ stack | op $\in$ $SCC_i$
    Repeat {
      For (op= first) to Top_Of_Stack
        forall successor of op
          priority[op]= Max(priority[op], (priority[successor] +
              $\lambda_{e(op,successor)} - \delta_{e(op,successor)} \times II$ ))
    until no priority[op] changes }
```

3.3.2.2 Earliest time to schedule an operation

The algorithm *Earliest_time* finds the earliest cycle when an operation *OP* can be scheduled. This procedure enforces correct schedules from the viewpoint of dependence constraints. However it takes into account only the currently scheduled *predecessors* of *OP*, as shown by Algorithm 3.9. Dependence conflicts with *successors* will be addressed by the backtracking process.

Algorithm 3.9 *Earliest time*

```
Earliest_time(OP)
  mintime= 0
  forall predecessor of OP {
    If (predecessor is scheduled at cycle=c) {
      candidate= $c + \lambda_{e(predecessor,OP)} - (II \times \delta_{e(predecessor,OP)})$
      if (candidate > mintime)
        mintime= candidate
    }
  }
  Return mintime
```

3.3.2.3 Find slot

This procedure finds a valid slot to schedule an operation. It enforces correct schedules from a resource usage viewpoint, using the MRT structure described in Section 3.3.2. Algorithm 3.10 tries to find a *resource free* slot to schedule *OP* in the range between *mintime* and *maxtime*. *Mintime* is the earliest cycle to schedule *OP*, as described in Section 3.3.2.2. *Maxtime* is set to $(mintime + II - 1)$. The value of *maxtime* is defined based on the observation that it is redundant to consider more than II contiguous time slots. If a resource free slot cannot be found in that range, the algorithm will relax this constraint to assign a schedule slot for *OP*. If *OP* was never scheduled before, the chosen slot will be at cycle *mintime*. Otherwise, it will be *one cycle later* than it was previously scheduled. In this case the operation currently scheduled in the chosen slot will be unscheduled during the backtracking process.

Algorithm 3.10 *Find Slot*

```
Find_Slot(OP, mintime)
  /* Limit range of possible slots */
  maxtime= mintime + II -1
  currtime= mintime
  While (currtime $\leq$ maxtime) {
    /* Find a resource free slot */
    find free slot in MRT at cycle=currtime
    if (slot found)
      Return slot
    else
      ++currtime
  }
  /* Relax the "resource free" condition */
  If (OP never scheduled)
    /* Choose a slot in the first possible cycle */
    choose slot in MRT at cycle=mintime
  else
    /* Choose a slot one cycle later than previously scheduled */
    choose slot in MRT at cycle= $OP_{previous\_slot} + 1$
  Return slot
```

3.3.2.4 Backtracking

Once a slot is found to schedule *OP*, resource and dependence conflicts may arise, which would require some operations to be unscheduled. The backtracking procedure described by Algorithm 3.11 checks dependence conflicts due to scheduled *immediate successors* of *OP*. The algorithm calculates the earliest cycle in which a successor can be scheduled, which we call *correct* time. Any successor scheduled *before* the correct time must be ejected from the partial schedule as its operands will not be available. There is no need to check for eventual dependence conflicts with scheduled predecessors, as this is taken into account by the *Earliest_time* procedure (Section 3.3.2.2). Operations may be also unscheduled due to resource usage conflicts, as discussed in Section 3.3.2.3.

EX BIBL. UNIV. EDINBURGH

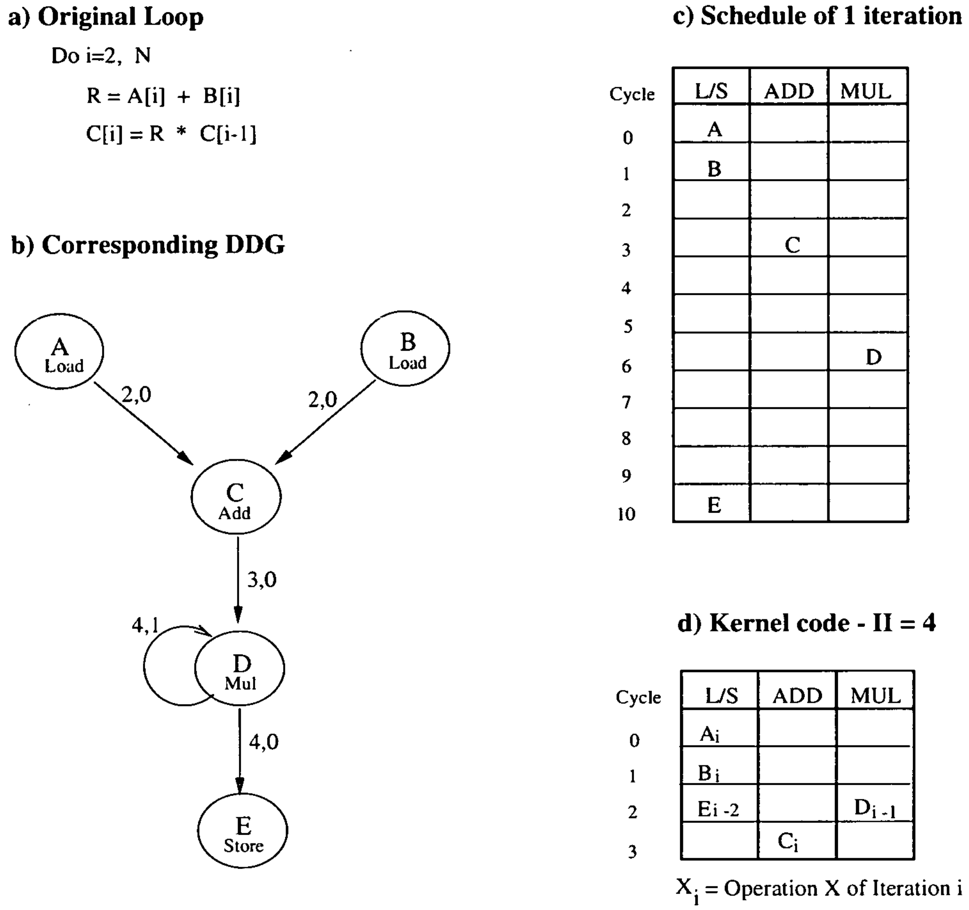
Algorithm 3.11 *Backtracking Process*

```
Backtracking(OP, slot)
  s= slot cycle
  forall successor of OP {
    /* Compute the correct time in which successors should be scheduled */
    If (successor is scheduled at cycle c) {
      correct= $s + \lambda_{e(OP,successor)} - (II \times \delta_{e(OP,successor)})$
      if (correct > c) {
        /* Scheduling cycle too early to meet dependence constraints */
        Unschedule(successor)
        Update MRT
        /* Return successor to the list of unscheduled operations */
        Include(List,successor)
      }
    }
  }
  /* Unschedule all operations due to resource conflicts */
  forall op having a resource conflict in slot {
    Unschedule(op)
    Update MRT
    Include(List,op)
  }
```

3.3.3 Scheduling Example

This section shows an example of the scheduling of a simple innermost loop using IMS. It is assumed a machine model comprising 3 standard FUs: 1 L/S, 1 ADD, and 1 MUL. The loop source code and the corresponding *DDG* are shown in Figures 3.7a and 3.7b, respectively. It requires the execution of 3 memory operations, 1 addition, and 1 multiplication. Hence, the most heavily used function unit is the L/S, determining a *ResMII* of 3 cycles. One of the multiplication operands is the result of the same operation from the previous iteration. This is translated into the only recurrent circuit seen in the *DDG*, an edge starting and ending at the multiply operation. Because the latency of the multiply operation is 4 cycles, successive executions of this operation requires an interval of at least 4 cycles. Thus the *RecMII* of the *DDG* is 4, which also determines the *MII*.

In this example IMS manages to find a valid modulo schedule within the *MII*. In Figure 3.7c we show the schedule of operations for a single iteration of the loop, which takes 11 cycles to complete. It can be verified that starting the same schedule every II cycles does not violate either machine or dependence constraints. A compact representation of the kernel code is shown in Figure 3.7d. Each operation has a subscript indicating the iteration it belongs to.



a) Original Loop

```
Do i=2, N
   R = A[i] + B[i]
   C[i] = R * C[i-1]
```

b) Corresponding DDG

c) Schedule of 1 iteration

Cycle	L/S	ADD	MUL
0	A		
1	B		
2			
3		C	
4			
5			
6			D
7			
8			
9			
10	E		

d) Kernel code - II = 4

Cycle	L/S	ADD	MUL
0	A_i		
1	B_i		
2	E_{i-2}		D_{i-1}
3		C_i	

X_i = Operation X of Iteration i

Figure 3.7: Schedule produced by IMS

3.4 Register Allocation

Once the scheduling process is completed, register allocation is performed. We call lifetime the number of cycles during which a value must be stored during loop execution. Lifetimes can be related to one of two types of variables:

- *Loop-invariants*: These variables are used by every loop iteration, but never modified. They are also called *scalar lifetimes*, and can be seen as a single value during loop execution. Thus a single physical register may be enough to store a loop-invariant lifetime.

- *Loop-variants*: A new value of this type of variable is produced in each loop iteration, generating a *vector of lifetimes*. Loop-variant lifetimes from distinct iterations can overlap due to software pipelining, requiring distinct storage locations.

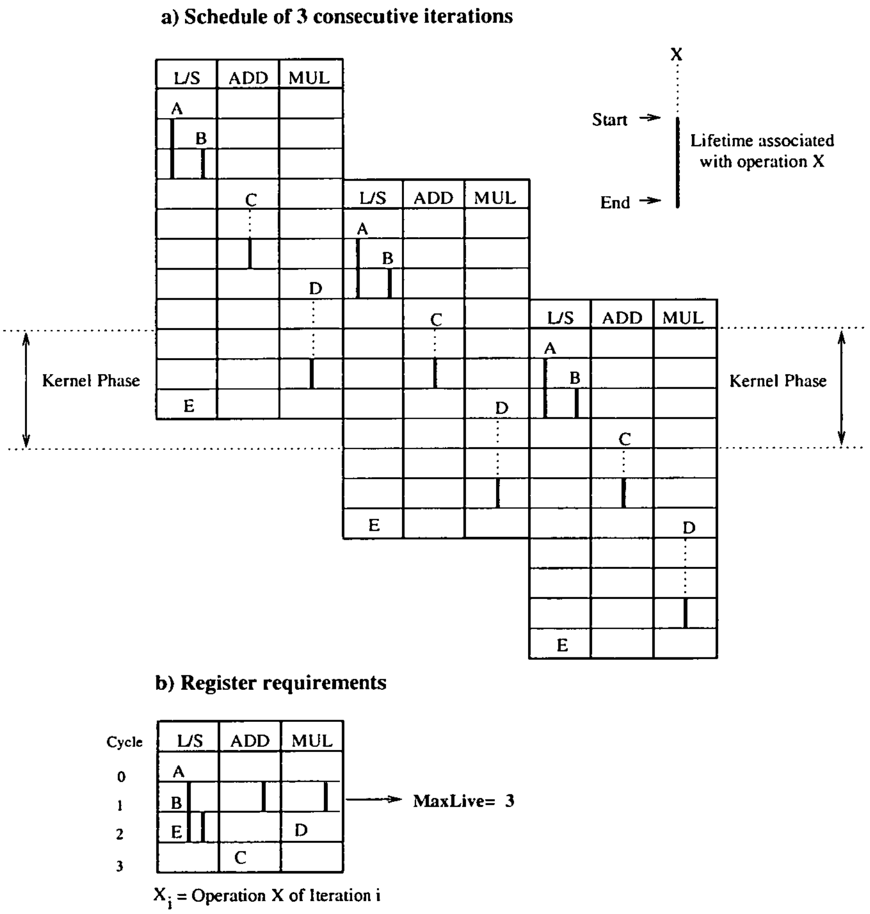
It can be said that the **length** of a loop-variant lifetime is the number of cycles ranging from its production to its last consumption. The lifetime length can be calculated in different ways, according to the architectural model in use. We have adopted the **end-begin** definition: A lifetime starts at the last cycle of the producing operation, and finishes the cycle before the last consumer starts. This definition has been adopted in order to support the functionality of a queue register file (Chapter 4).

The register allocator is not constrained by a finite number of physical registers. Hence, instead of performing actual register allocation to the RF, we compute a lower bound on the number of registers required. We call *MaxLive* the highest number of values that must be stored at any given cycle of the schedule [81]. *MaxLive* indicates how many physical locations are necessary to keep those values. A scheme to calculate *MaxLive*, considering both, loop variants and invariants, is described by Algorithm 3.12.

Algorithm 3.12 *Maximum number of live registers*

```
MaxLive()
  /* Each loop invariant holds a register during every cycle of the schedule */
  for i= 0 to (II - 1)
    live[i]= number of loop invariants
    /* Compute the start cycle and length of each loop variant */
    forall OP ∈ DDG {
    lf_start= (starting cycle of OP) + (latency of OP) -1
    forall successor using a value produced by OP {
      lf_length= (starting cycle of successor) - lf_start +
          (II × $\delta_{e(OP,successor)}$)
      /* Identify cycles in which a lifetime requires a register */
      for i= lf_start to (lf_start + lf_length -1)
        ++live[i mod II]
    }
  }
  /* Total number of registers required is defined by the cycle */
  /* in which the largest number of registers are needed */
  MaxLive= max(live[i])
```

The schedule of 3 consecutive loop iterations of the example presented in Section 3.3.3 is shown in Figure 3.8a. It is intended to illustrate a *MaxLive* calculation. The steady state starts with the second iteration. The maximum number of lifetimes coexisting at any given cycle occurs in the third cycle of the kernel phase. In this cycle the value of *MaxLive* is 3, which can also be seen in the compact notation presented in Figure 3.8b. It should be noticed that a given scheduling slot at cycle *i* corresponds to scheduling slot (*i mod II*) in the compact representation. This example considers only loop variant lifetimes. Each eventual loop invariant lifetime would increase *MaxLive* by one.


a) Schedule of 3 consecutive iterations

b) Register requirements

L/S
ADD
MUL
A
B
C
D
E
X
Start
End
Lifetime associated with operation X
Kernel Phase
Kernel Phase
Cycle
0
1
2
3
MaxLive= 3
X_i = Operation X of Iteration i

Figure 3.8: Register requirements - MaxLive

3.5 Output Information

A standard set of information is produced for each loop scheduled assuming a particular machine model. These data can be used individually, or most often by means of statistical analysis regarding the full benchmark set. The data generated can be divided into two groups, called *direct* and *derived* parameters.

3.5.1 Direct Parameters

The following parameters are derived directly from the application of IMS on an innermost loop:

- *Modulo Schedule*
- II, *MII, ResMII, RecMII*: As previously defined.
- *Schedule length*: Number of cycles of the schedule for one iteration.
- *Stage Count:* As previously defined, and calculated using the following expression:

$$\left\lceil \frac{Schedule\ length}{II} \right\rceil \tag{3.2}$$

- *Instruction Count*: Number of instructions scheduled.
- *Iteration Count*: Total number of times the body of the innermost loop is executed.

3.5.2 Derived Parameters

Derived parameters provide an insight on the architecture performance and the required machine resources. They are calculated using the direct parameters:

- X: This parameter states the total execution time of a modulo scheduled innermost loop. The kernel, prologue and epilogue phases are taken into account. It is assumed that kernel-only code is generated. Given a loop i in a benchmark set composed of L loops; let N_i be the iteration count, let II_i be the initiation interval, and let SC_i be the number of stages in the software pipeline schedule. The total execution time of loop i, X_i, is given by the following expression:

$$X_i = II_i \times (SC_i + N_i - 1) \tag{3.3}$$

- $IPC_{dynamic}$: The dynamic issue takes into account the total execution time of a loop, including the kernel, prologue, and epilogue stages. The weight of a loop is determined by the execution time when $IPC_{dynamic}$ is calculated for the complete benchmark. The execution time of a loop is mostly determined by the II and the iteration count, which is used to calculate the cycle count. Given a loop i in a benchmark set composed of L loops; let N_i be the iteration count, let II_i be the initiation interval, let SC_i be the number of stages in the software pipeline schedule, and let O_i be the number of useful operations in the schedule. Then, $IPC_{dynamic}$ can be calculated as follows:

$$IPC_{dynamic} = \frac{\sum_{i=1}^{L} N_i O_i}{\sum_{i=1}^{L} II_i \left(SC_i + N_i - 1\right)} \tag{3.4}$$

- $II_{speedup}$: We have defined this parameter to measure the gain in performance execution of the *kernel code* when a given machine model A scales up to a machine model B. It is calculated using the following expression:

$$II_{speedup} = \frac{II_{machineA}}{II_{machineB}} \tag{3.5}$$

- SC_{var}: This parameter accounts for the stage count variation when distinct machine models are used, calculated according to the following expression:

$$SC_{var} = SC_{machineB} - SC_{machineA} \tag{3.6}$$

- *Register requirements*: The basic version of the experimental framework assumes the use of a conventional register file. It is calculated as a lower bound for the number of registers required to store loop variant and invariant lifetimes without using spill code. This is done using Algorithm 3.12 to compute *MaxLive*. The model assumes that once a value is stored in a given register, it remains there until the last use.

3.5.3 Results Presentation

Unless otherwise stated, all data reporting machine resources refers to *dynamic* analysis. The data presented refers to the machine resources required to execute the *loops* accounting for at least 99% of the total *execution time* of all loops from our benchmark set (Section 3.2). The remaining 1% of the execution time is usually spent in a few very large loop bodies with a large iteration count. These loops cannot be identified beforehand, so they are not discarded before scheduling. Hence, the machine resources reported refers to the maximum requirements of the loops necessary to make the 99% fraction. In practice, instead of attempting to software pipeline large loops, a production compiler could split them into smaller ones, using techniques such as *loop distribution* [7]. Doing so, the new machine requirements would possibly be lower. Finally, when static data is presented instead, it will refer to the resources accounting for 99% of the loops from the benchmark (1246 loops).

Chapter 4

Queue Register Files

We have shown in Section 2.3 that software pipelining increases register pressure. Modulo scheduling algorithms capable of minimizing register requirements have already been proposed, presenting some advantages over conventional schemes [48, 63]. However, the typical number of available registers is still insufficient for some loops, thus requiring spill code. Increasing the number of registers in a conventional RF organization requires more address generation and decoding hardware. It may also increase the cycle time due to longer wires [5]. Furthermore, register files for wide-issue ILP architectures require a large number of access ports, which may result in a long machine cycle time [25]. It is possible to limit the number of access ports, but this may also compromise parallelism exploitation. Those factors suggest that new register file organizations may be necessary to implement high performance ILP architectures.

4.1 QRF Organization

The regular pattern of production and consumption of loop variant lifetimes has motivated us to use queue structures as storage elements. A *FIFO* queue can be used to store consecutive loop variants, which are written in the tail and read from the head of the queue. If we constrain each definition of a lifetime to a single use, there is no need to access intermediate queue positions, simplifying the hardware organization [5, 46]. It is possible to transform a multiple use lifetime into several single use lifetimes (Section 4.2). In this chapter we propose the use of a queue register file (QRF) to support the execution of software pipelined loops in VLIW machines. A QRF having *one* queue of p elements consists of a storage array surrounded by supporting circuits to select the current write and read positions. A number of data access ports complete the basic structure of the QRF (Figure 4.1). A multiple-queue QRF can be built using the same organization.

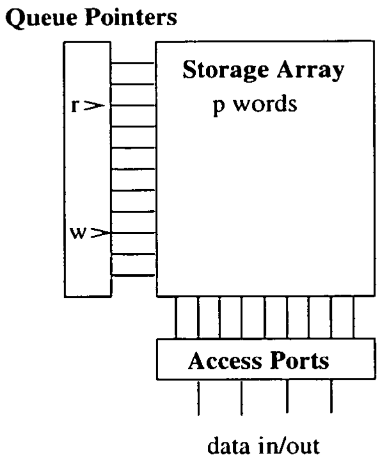


Figure 4.1: QRF block diagram

During the design of the VLIW architecture proposed in this thesis we have found that QRF organizations allow at least three advantages over conventional RFs:

- *Silicon area*: As previously discussed the hardware complexity of a queue register file should be lower than of a conventional RF, and likewise the silicon area required. This may also improve the *cycle time*, which has been confirmed using an analytical timing model (Section 4.5).
- *Inter-Cluster communication*: We have developed a scheme using QRFs to implement data communication between adjacent clusters in a distributed VLIW architecture (Section 6.1). This results in a low-latency communication mechanism, which can be efficiently exploited by the novel partitioning algorithm described in Chapter 7.
- *Compilation issues*: In a QRF a data value is allocated to a specific queue instead of to a specific register. Experimental analysis has shown that the shift from register names to queue names reduces dramatically the pressure on the size of the *register name space* [28]. Furthermore, the functionality of a QRF is similar to a rotating register file [82, 83]. This may improve the machine performance in two aspects: kernel-only code can be used (Section 2.4.4), and register allocation can be performed without modulo variable expansion (Section 2.4.3).

The concept of allocating loop variant values to a queue is illustrated in Figure 4.2. It shows a simple innermost loop, the corresponding *DDG*, and the modulo schedule of 3 consecutive iterations (parts a, b, and c, respectively). Values produced by operation A can be stored in a queue until they are consumed by operation C . A single queue (corresponding to a single register name) is enough to store those values, as seen in Figure 4.2d. If a conventional register file was used instead, two distinct register names would be required because the II is smaller than the length of that lifetime. In this case, using the same location to store all lifetimes would result in lifetimes being overwritten before being used. It can be seen in the figure that successive definitions of lifetimes produced by operation A matches successive consumptions by C (Figure 4.2e). This regular pattern ensures that the first element in the queue is always the value required by the next read operation.

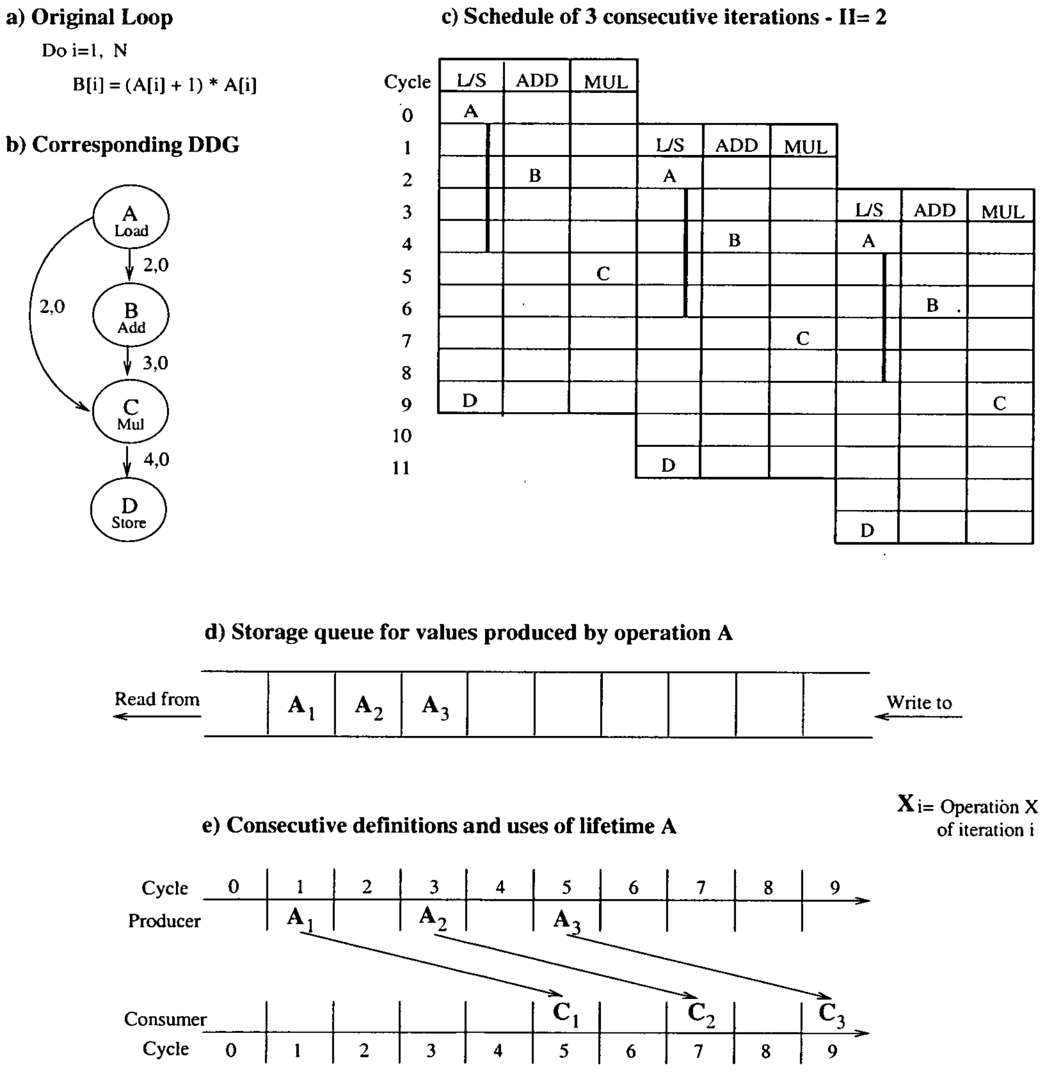


Figure 4.2: Using a queue to store a loop variant lifetime

It should be noticed that in a conventional register file organization random access to any register is allowed, thus simplifying the register allocation process. However, in a QRF this issue is further complicated because of its limited addressing capabilities. We have developed a new register allocation strategy for a QRF. It is based on the regular access pattern of loop variant lifetimes produced by a modulo scheduled loop. This scheme is described in Section 4.4, and is essential to take full advantage of a QRF.

In the next section we address the issue of multiple-use lifetimes, which we define as a *pre-condition* to employ a QRF. Then we discuss register allocation schemes for this particular organization. The chapter ends with a possible hardware implementation of a QRF, including analytical models for the silicon area and cycle time. These parameters are used to compare a QRF to a conventional register file.

4.2 Transforming Multiple-Use Lifetimes

The proposed QRF model assumes that access to physical locations is controlled by two elements, called the *r-pointer* and the *w-pointer.* They determine the current read and write positions, respectively. Every time data is written in the queue, the *w-pointer* moves back one position. Similarly, a read operation moves the r-pointer to the next read position. This operational mode implies that *data can be read only once* from a QRF.

A value produced by a given operation may be consumed more than once, as shown in the *DDG* of Figure 4.3a. Once a value is written into a conventional register file, it can be read as many times as necessary (Figure 4.3b). However the *read-once* limitation of a QRF requires that multiple-use lifetimes must be stored in distinct locations, one for each use (Figure 4.3c). We call this situation as *replicated writes*, which may result in at least two problems:

- *Problem 1*: The instruction format should allow a single instruction to specify an unbounded (possibly large) number of destination queues.

- *Problem 2*: The QRF should allow simultaneous write access to an unbounded number of queues.

We propose instead the use of **copy operations** to eliminate the need for replicated writes. A copy operation has one input and two output operands. It is capable of reading one register value, and copying it back to two other storage

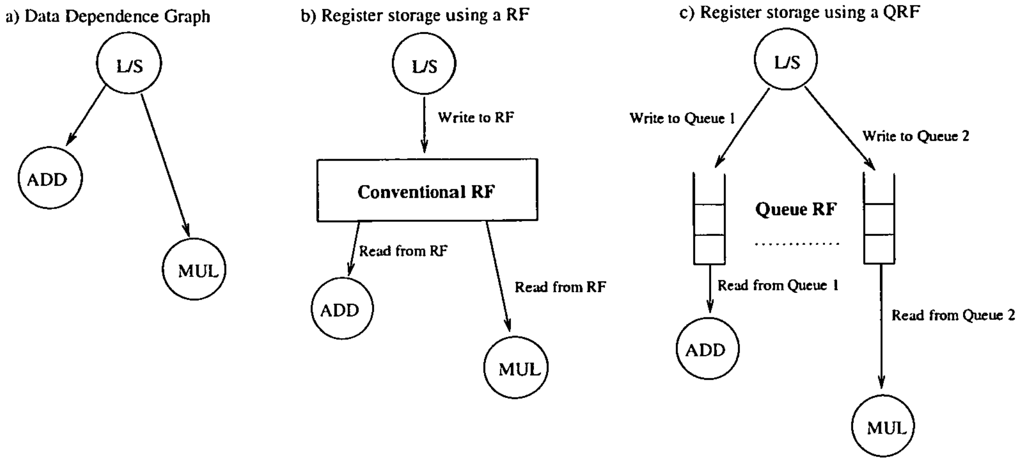


Figure 4.3: Register storage

locations. Replicated writes can be eliminated by transforming the data dependence graph to include copy operations. One copy operation can transform one *dual-use* lifetime into two *single-use* lifetimes, as shown by the diagram in Figure 4.4. In this transformation a new node is inserted in the *DDG*, referring to the copy operation. The two original edges are replaced by three new edges: one from the producer to the copy operation, and two others from the copy operation to each one of the consumers, respectively. New attributes are set according to the *delay* and *distance* values of the replaced edges.

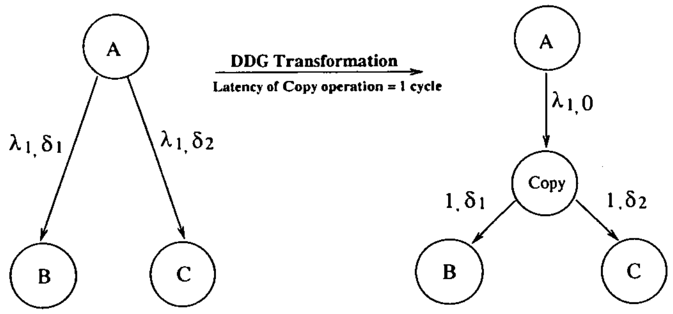


Figure 4.4: DDG transformation to include a copy operation

Successively applying this simple transformation allows one to transform any multiple-use lifetime into a number of single lifetimes. We have designed an algorithm able to generate a *balanced* subgraph after the inclusion of copy operations. The root of this subgraph is the original producer, and the leaves are the original consumers. This reduces the eventual delay resulting from the inclusion of copy operations in the critical path of the *DDG*. The scheme is described by Algorithms 4.1 and 4.2, which creates a modified *DDG* from the original one. These algorithms assume a latency (delay) of *one* cycle to execute a copy operation. An application of the algorithm, transforming a four-use lifetime into four single-use lifetimes, is shown by the example in Figure 4.5

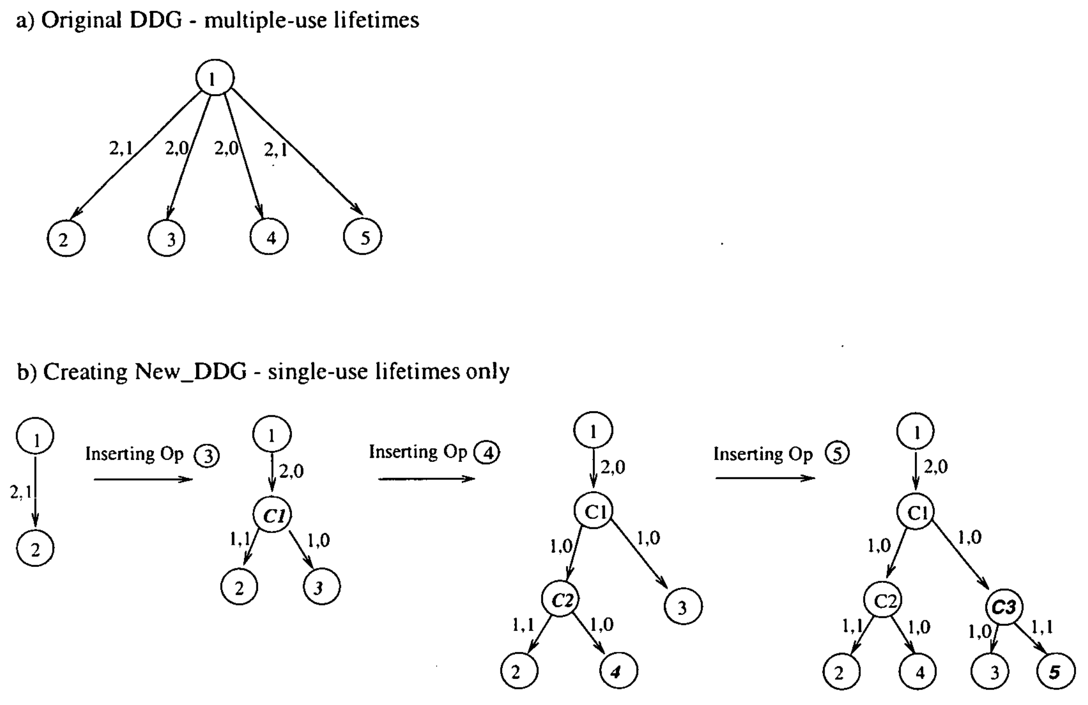
In the experimental framework a consistency check is performed after this transformation, comparing the original data dependence graph against the modified version. This is done to ensure that the semantics of the original program is preserved after the *DDG* transformation.

Algorithm 4.1 *Choose Edge*

```
Choose_Edge(New_DDG, source)
  /* Perform a breadth-first search until until the first edge */
  /* without a copy operation as a target is found */
  Append(NewList,source)
  found= 0
  While not found {
    CurrentList= NewList
    Clear(NewList)
    forall $u \in CurrentList$ {
      forall out_edge(u,e) {
        if $(target(e) \neq copy)$ {
          found= 1
          Return(e)
        }
        else
          Append(NewList, target(e))
      }
    }
  }
```

Algorithm 4.2 *Inserting Copy Operations*

```
Insert_Copy(DDG, New_DDG)
  /* Insert original nodes in new DDG */
  forall operation $u \in DDG$
    Insert(New_DDG,u)
  /* Insert edges in new DDG */
  forall edge $e \in DDG$ {
    if out_degree(source(e)) = 1
      /* Single-use lifetime-Copy not necessary */
      new_edge= e
      Insert(New_DDG, new_edge)
    else { /* Multiple-use lifetime-Insert copy operation */
      Insert(New_DDG,copy)
      /* Choose the insertion point of the copy operation according to */
      /* the original pair of producer and consumer operations */
      split_edge= Choose_Edge(New_DDG, source(e))
      /* Edge from the original producer to the copy operation */
      new_edge= (source(split_edge), copy)
      $\lambda_{e(new\_edge)} = \lambda_{e(split\_edge)}$
      $\delta_{e(new\_edge)} = 0$
      Insert(New_DDG, new_edge)
      /* Edge from the copy operation to the original consumer */
      new_edge= (copy, target(split_edge))
      $\lambda_{e(new\_edge)} = 1$
      $\delta_{e(new\_edge)} = \delta_{e(split\_edge)}$
      Insert(New_DDG, new_edge)
      /* Edge from the copy operation to the new consumer */
      new_edge= (copy, target(e))
      $\lambda_{e(new\_edge)} = 1$
      $\delta_{e(new\_edge)} = \delta_e$
      Insert(New_DDG, new_edge)
    }
  }
```


a) Original DDG - multiple-use lifetimes

b) Creating New_DDG - single-use lifetimes only

Figure 4.5: Inserting copy operations in a DDG

4.3 Overhead Due to Copy Operations

In terms of hardware, the use of copy operations requires an extra FU, capable of copying a value from one register and writing it back to two other storage locations. This should be simple to implement. In the experimental framework that function is performed by the *Copy* FU, as described in Section 3.1.1. We assume a latency of one cycle for this operation. Although a simple function is performed, a significant overhead overhead results from the use of Copy FU: extra *access ports* are required.

In terms of software the introduction of copy operations may increase the total execution time of a loop. That would be the result of a higher stage count or initiation interval. Copy operations inserted in the critical path of the *DDG* will increase the *schedule length* of a single iteration. A longer schedule length may result in a higher *SC*, requiring longer prologue and epilogue phases. It might also happen that the required number of copy operations makes the Copy FU the most heavily used machine resource, increasing the II. Increasing the II may cause a higher impact on the execution time than increasing the *SC*, specially for loops with a large iteration count. This can be inferred from the expression for the execution time of a loop (Section 3.5.2). It is also possible that a copy operation is inserted in a recurrent circuit, increasing the *RecMII.*

We have performed a number of experiments to evaluate the effect of introducing copy operations in the *DDG* of innermost loops. Machine models and the workload used in the evaluation are as described in Chapter 3. In this section we only present results referring to an unclustered machine comprising of 4 FUS: 1 L/S, 1 ADD, 1 MUL, 1 Copy. We have found similar results and conclusions for other machine configurations, as reported in [27].

The chart in Figure 4.6 shows that around 94% of the loops can be scheduled within the same II that would be otherwise possible without using copy operations. The remaining fraction of loops requires a higher II, an increase of one cycle in most of the cases. Similar results were found for the stage count variation (Figure 4.7): the value of *SC* remains the same for 89% of the loops. An eventual increase of one stage occurs for 10% of the loops. The average stage count, with and without copy operations, is 3.0 and 3.1, respectively.

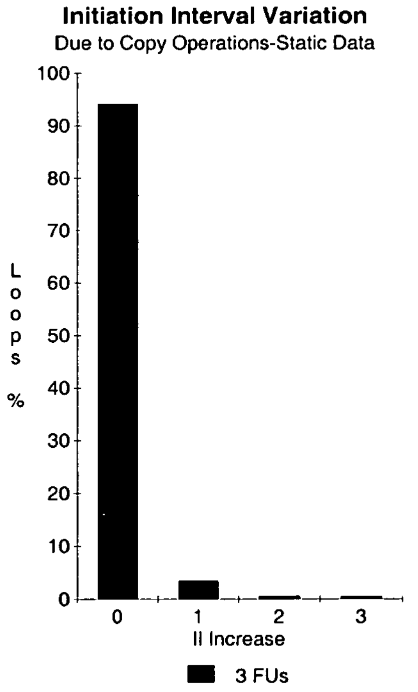


Figure 4.6: II Variation-Copy Op

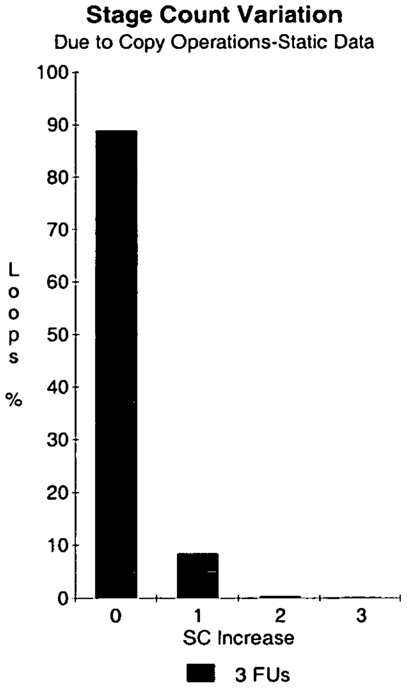


Figure 4.7: SC Variation-Copy Op

The combined effect of variations in those parameters has been measured by calculating the number of cycles required to execute *all* loops of the benchmark using copy operations. This value was compared with the time required to execute the same set of loops *without* using copy operations. For this machine configuration, we have found that copy operations increase the total execution time by 1.5%, confirming that the performance penalty due to this transformation is acceptable. In the next chapters more extensive analyses of this matter will be presented.

4.4 Allocating Lifetimes to a QRF

Allocating lifetimes to a queue is more complicated because of the limited addressing capability. We will focus the discussion on the problem of allocating *loop variant lifetimes* generated by a modulo scheduled loop as this is the main motivation to develop a QRF. We have identified two sufficient conditions under which two or more lifetimes can share a storage queue in the QRF proposed in Section 4.1. We shall call them Q-Compatibility Conditions, which are listed below:

1. There can be *at most* one read and one write access operation to the queue at any given cycle.
2. Lifetimes must be written and read from a queue in exactly the same order.

A straightforward solution is to allocate a single lifetime to each queue. In this case a queue would act as a buffer, storing successive productions of a given lifetime (Figure 4.2). The advantage over a conventional register file is that only one register name is required (the name of the queue), even if the lifetime length spans more than II cycles. However it is easy to identify situations in which a queue can accommodate more than one lifetime, hence optimizing the use of machine resources.

Two or more loop variant lifetimes of the *same length* may share a single queue. It can be verified that the production and consumption order of *all lifetimes* are identical during the kernel stage of a modulo schedule. This is sufficient to meet the second Q-Compatibility condition. The first condition can be met if that set of lifetimes have distinct *starting* cycles. Using this condition results in a relatively small search space to identify which lifetimes can share a common queue. It only involves grouping lifetimes according to their lengths, an approach used in [5]. As an example Figure 4.8a shows the kernel code for a hypothetical loop, which is repeated every 8 cycles. It can be seen that lifetimes produced by operations A and B have the same length (2 cycles), and distinct starting cycles. The diagram in Figure 4.8b shows, cycle by cycle, the production and consumption of those values. This definition-use pattern is repeated throughout the kernel code execution.

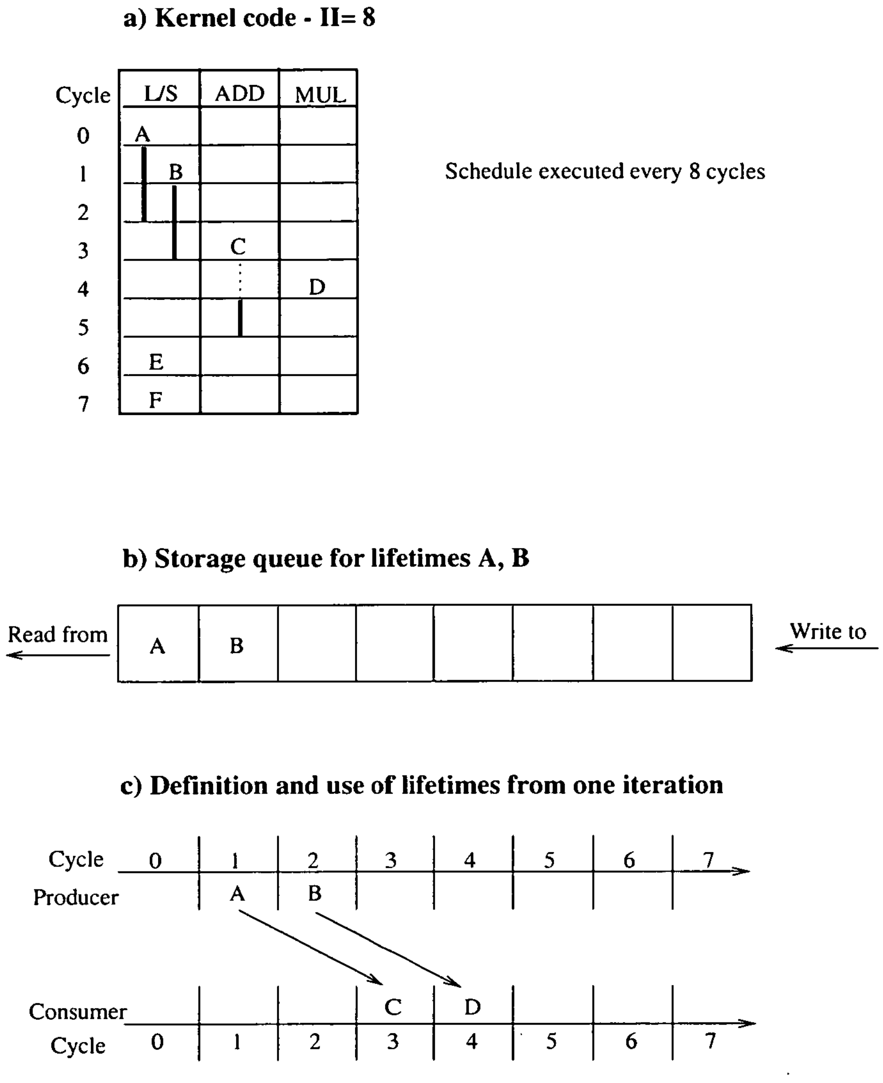

a) Kernel code - II= 8
Cycle
L/S
ADD
MUL
0
1
2
3
4
5
6
7
A
B
C
D
E
F
Schedule executed every 8 cycles
b) Storage queue for lifetimes A, B
Read from
A
B
Write to
c) Definition and use of lifetimes from one iteration
Cycle
Producer
Consumer
0
1
2
3
4
5
6
7

Figure 4.8: Using one queue to store lifetimes of the same length

Furthermore, the above example suggests that lifetimes of *distinct lengths* can share a single queue. In Figure 4.9b we show that three lifetimes produced by the modulo schedule of Figure 4.9a can share the same queue. Lifetimes A, B and C have distinct lifetimes of 2, 2, and 1 cycles, respectively. But they also have identical production and consumption orders, and distinct starting cycles, meeting both Q-compatibility conditions.

The allocation scheme shown in the last example can indeed optimize the use of machine resources. However a simplistic approach to the problem would require working through a huge search space, a complex problem for a large number of lifetimes. Therefore, it is important to reduce the size of the search space in order to find a practical allocation method. We have developed a set

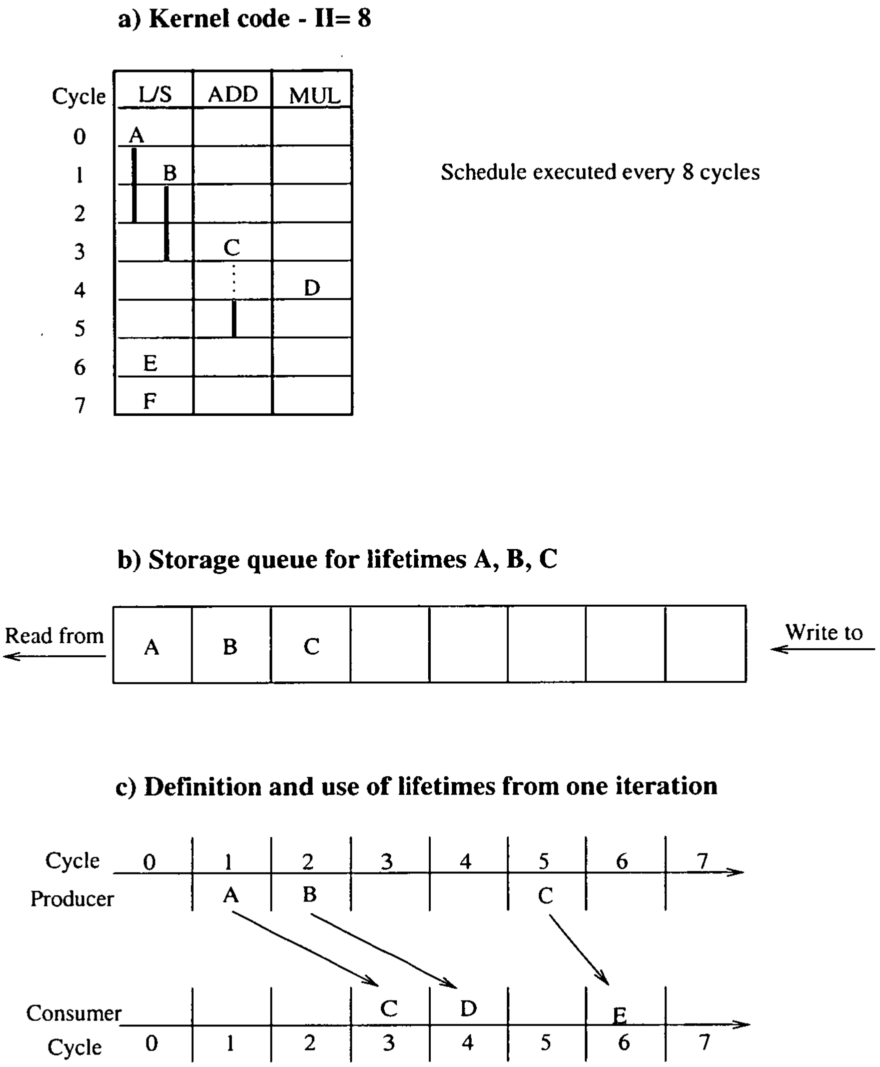

a) Kernel code - II= 8
Cycle
L/S
ADD
MUL
0
1
2
3
4
5
6
7
A
B
C
D
E
F
Schedule executed every 8 cycles
b) Storage queue for lifetimes A, B, C
Read from
A
B
C
Write to
c) Definition and use of lifetimes from one iteration
Cycle
0
1
2
3
4
5
6
7
Producer
A
B
C
Consumer
C
D
E
Cycle
0
1
2
3
4
5
6
7

Figure 4.9: Using one queue to store lifetimes of distinct lengths

of constraints under which two lifetimes can share the same storage queue [26]. We also show how this condition can be evaluated through a simple and practical compile-time test. We shall call it Q-Compatibility Test, which is formally stated and proved in the next section. To the best of our knowledge no other work has used a similar approach. Schemes based on rotating register files are the most similar to this one. Although a rotating register file can be seen as set of queues, each of them can store only distinct instances of the same lifetime.

4.4.1 Q-Compatibility Test

In a modulo-scheduled loop each computation generates a new value every Initiation Interval (II) cycles. Each value has a fixed lifetime which begins at some start-point and terminates at some end-point within the schedule.

Definition 4.4.1 (Lifetimes) *On each iteration of a loop every computation* a *produces a new value which exists over a period defined by the pair* $\langle S_a, S_a + L_a - 1\rangle$*, where* S_a *is the start-point and* $S_a + L_a - 1$ *is the end-point of that value. We say that* L_a *is the* lifetime *of computation* a.

Definition 4.4.2 (Vector lifetimes) *In a modulo-scheduled loop every computation* a *produces a vector of lifetimes* A*:*

$$A \equiv \{\langle a_n, a_n + L_a - 1\rangle : a_n = S_a + n.II\}_{n \geq 0}$$

Definition 4.4.3 (Q-Compatibility) *Let two computations* a *and* b *have start-points* S_a *and* S_b*, and have lifetimes* L_a *and* L_b*. The values produced by* a *and* b *can share the same destination queue if the relative order in which they produce values is identical to the relative order in which those values are consumed by their successor computations, and their start-points are different.*

It is now necessary to formulate a simple way of determining the compatibility of any pair of computations. We do this by formulating a proposition which encapsulates our definition of Q-Compatibility and then we prove that there exists a simple relationship between lifetimes, start-points and Initiation Interval which can be used in a scheduler to determine Q-Compatibility. The proofs we develop use modulo arithmetic and rely on the following four lemmas.

Lemma 4.4.1 *For all integers* x*,* y *and* n*,* $x \equiv y \Rightarrow [x]_n \equiv [y]_n$.

Lemma 4.4.2 *For all integers* x*,* y *and* n*,* $[x + ny]_n \equiv [x]_n$.

Lemma 4.4.3 *For all integers* x *and* n*,* $x > 0 \Rightarrow [x]_n \leq x$.

Lemma 4.4.4 *For all integers* x *and* n*,* $0 \leq x < n \Rightarrow [x]_n \equiv x$.

We now formulate a proposition based on Definition 4.4.3 which provides us with a formal criterion for queue compatibility.

Proposition 4.4.1 *The two computations* a *and* b *are Q-compatible if, and only if:*

$$\begin{aligned} \forall_{i,j\geq 0} \;:\; & a_i > b_j \Rightarrow a_i + L_a > b_j + L_b && (4.1)\\ \wedge \;\; & a_i < b_j \Rightarrow a_i + L_a < b_j + L_b && (4.2)\\ \wedge \;\; & a_i \neq b_j && (4.3)\end{aligned}$$

This proposition, although an accurate formulation of Definition 4.4.3, cannot be used directly when scheduling a loop as it contains universal quantifiers. These imply a large, possibly unbounded, search space for i and j. The following theorem defines an alternative, and computationally efficient, test for Q-Compatibility.

Theorem 4.4.1 (Q-Compatibility Test) *Two computations* a *and* b*, with start-times S_a and S_b, and lifetimes L_a and L_b such that $L_a \geq L_b$, are Q-compatible if $L_a - L_b < [S_b - S_a]_{II}$.*

Proof

To prove Theorem 4.4.1 we must demonstrate that $\models P \Rightarrow Q$, where

$$\begin{aligned} Q &\equiv \forall_{i,j\geq 0} \;:\; R_1 \wedge R_2 \wedge R_3 && (4.4)\\ R_1 &\equiv a_i > b_j \Rightarrow a_i + L_a > b_j + L_b && (4.5)\\ R_2 &\equiv a_i < b_j \Rightarrow a_i + L_a < b_j + L_b && (4.6)\\ R_3 &\equiv a_i \neq b_j && (4.7)\\ P &\equiv L_a - L_b < [S_b - S_a]_{II} && (4.8)\end{aligned}$$

We now show that this formula holds using proof by contradiction:

1. Let there exist interpretations of P and Q which render $P \Rightarrow Q$ false. Hence, there must exist values of i, j, S_a, S_b, L_a, L_b, and II such that P is true and Q is false.

2. From the definition of Q in equation (4.4) any one of R_1, R_2 and R_3 can be false for Q to be false. We consider these cases in steps 3, 4 and 5.

3. Let R_1 be false, then there must exist $i, j \geq 0$ such that $a_i > b_j$ and $a_i + L_a \leq b_j + L_b$. Thus:

   $$\exists_{i,j\geq 0} \;:\; a_i + L_a \leq b_j + L_b$$

   and so

   $$\exists_{i,j\geq 0} \;:\; L_a - L_b \leq b_j - a_i$$

But $a_i > b_j$, so $b_j - a_i < 0$, and hence

$$L_a - L_b \leq b_j - a_i < 0 \tag{4.9}$$

However, it is assumed that $n > L_a \geq L_b$, consequently R_1 cannot be false if P is true.

4. Let R_2 be false, then there must exist $i, j \geq 0$ such that $a_i < b_j$ and $a_i + L_a \geq b_j + L_b$. Thus:

$$\exists_{i,j \geq 0} : \quad a_i + L_a \geq b_j + L_b$$

and so

$$\exists_{i,j \geq 0} : \quad L_a - L_b \geq b_j - a_i \tag{4.10}$$

We know from Definition 4.4.2 that $a_i = S_a + i.II$ and $b_j = S_b + j.II$, so we may write:

$$b_j - a_i = S_b - S_a + II(j - i) \tag{4.11}$$

By Lemma 4.4.1 and equation (4.11) we may write:

$$[b_j - a_i]_{II} = [S_b - S_a + II(j - i)]_{II} \tag{4.12}$$

By Lemma 4.4.2, equation (4.12) can be reduced to:

$$[b_j - a_i]_{II} = [S_b - S_a]_{II} \tag{4.13}$$

Recall that for R_2 to be false we must satisfy the following inequalities:

$$\exists_{i,j \geq 0} : \quad L_a - L_b \geq b_j - a_i > 0$$

But since proposition P is assumed to be true, then:

$$L_a - L_b < [S_b - S_a]_{II}$$

Since both equation (4.10) and proposition P must both hold, we can write:

$$[S_b - S_a]_{II} > L_a - L_b \geq b_j - a_i > 0$$

Eliminating $L_a - L_b$, we get:

$$[S_b - S_a]_{II} > b_j - a_i > 0 \tag{4.14}$$

From equation (4.13) we know that $[S_b - S_a]_{II} = [b_j - a_i]_{II}$, and hence substituting for $[S_b - S_a]_{II}$ in equation (4.14) we get:

$$[b_j - a_i]_{II} > b_j - a_i > 0 \tag{4.15}$$

This contradicts Lemma 4.4.3 so R_2 cannot be false.

5. Let R_3 be false, then $\exists_{i,j\geq 0} \; : \; a_i = b_j$ and consequently $[b_j - a_i]_{II} = 0$. From equation (4.13) we can further deduce that $[S_b - S_a]_{II} = 0$. However, if P is true we can say that:

$$L_a - L_b < [S_b - S_a]_{II} = 0$$

But as an assumption of the theorem we have:

$$L_a - L_b \geq 0$$

This represents a contradiction, so R_3 cannot be true when P is true.

We have thus demonstrated that none of R_1, R_2 or R_3 can be false if P is true, which in turn means that Q cannot be false if P is true. □

We have therefore shown that whenever the inequality in Theorem 4.4.1 is satisfied, the condition expressed in Proposition 4.4.1 is also satisfied. This provides us with a guarantee that the Q-Compatibility test from Theorem 4.4.1 will always indicate incompatibility for a pair of lifetimes that are incompatible. We would also like to guarantee that whenever our Q-Compatibility test indicates incompatibility, then so does Proposition 4.4.1. This would demonstrate equivalence between Theorem 4.4.1 and Proposition 4.4.1 and show that Theorem 4.4.1 is an exact test.

Theorem 4.4.2 (Exactness) *Two computations* **a** *and* **b**, *with start-times S_a and S_b, and lifetimes L_a and L_b such that $L_a \geq L_b$, are Q-compatible* if and only if $L_a - L_b < [S_b - S_a]_{II}$.

Proof

To prove this theorem we must show that $\models P \Rightarrow Q$ and $\models \neg P \Rightarrow \neg Q$, for P and Q defined by equations (4.8) and (4.4) respectively. Theorem 4.4.1 established $\models P \Rightarrow Q$, so it only remains to show $\models \neg P \Rightarrow \neg Q$.

1. Let us assume that there exist interpretations of P and Q such that $\neg P \Rightarrow \neg Q$ is false. Therefore P must be false and Q must be true.

   Recall that Q is defined as:

   $$\forall_{i,j\geq 0} \; : \quad R_1 \wedge R_2 \wedge R_3$$

   In statements 2, 3 and 4 we consider three cases which cover all possible relative values of a_i and b_j. These are $a_i < b_j$, $a_i > b_j$, and $a_i = b_j$.

2. Assume $a_i < b_j$, then R_1 and R_3 are both true. However, R_2 is only true if:

$$\forall_{i,j \geq 0} : \quad a_i + L_a < b_j + L_b \tag{4.16}$$

Under these conditions we can assert:

$$\forall_{i,j \geq 0} : \quad L_a - L_b < b_j - a_i \tag{4.17}$$

and

$$\forall_{i,j \geq 0} : \quad b_j - a_i > 0 \tag{4.18}$$

From the definition of $\neg P$ we know that:

$$L_a - L_b \geq [S_b - S_a]_{II} \tag{4.19}$$

Substituting for $[S_b - S_a]_{II}$ according to equation (4.13) in equation (4.19) we get:

$$L_a - L_b \geq [b_j - a_i]_{II} \tag{4.20}$$

Combining equations (4.20) and (4.17) yields the following proposition:

$$\forall_{i,j \geq 0} : \quad b_j - a_i > L_a - L_b \geq [b_j - a_i]_{II} \tag{4.21}$$

However, every computation in a schedule is executed exactly once per II cycles when the loop is in kernel mode (i.e. after the pipeline is primed, but before the shutdown phase). Consequently it is axiomatic that:

$$\exists_{i,j \geq 0} : \quad b_j - a_i < II \tag{4.22}$$

Equations (4.18) and (4.22) tell us that there exist i and j such that:

$$II > b_j - a_i > 0$$

Hence, by Lemma 4.4.4 we can assert:

$$[b_j - a_i]_{II} \equiv b_j - a_i \tag{4.23}$$

Substituting for $[b_j - a_i]_{II}$ in equation (4.21) using equation (4.23) yields:

$$b_j - a_i > b_j - a_i$$

This is a contradiction and so there exist values of i and j for which R_2 cannot be true when $\neg P$ is true and $a_i < b_j$. Under these conditions $\neg R_2$ cannot be false and consequently $\neg Q$ cannot be false.

3. Assume $a_i > b_j$, then R_2 and R_3 are both true. However, R_1 is only true if:

$$\forall_{i,j\geq 0} : \quad a_i + L_a > b_j + L_b \tag{4.24}$$

Under these conditions we can assert:

$$\forall_{i,j\geq 0} : \quad L_a - L_b > b_j - a_i < 0 \tag{4.25}$$

From the definition of P in equation (4.8) we know that $\neg P$ is given by:

$$\neg P \quad \equiv \quad L_a - L_b \geq [S_b - S_a]_{II} \tag{4.26}$$

Substituting for $[S_b - S_a]_{II}$ in equation (4.26) using equation (4.13) we get:

$$L_a - L_b \geq [b_j - a_i]_{II} \tag{4.27}$$

As $a_i > b_j$, we know $[b_j - a_i]_{II} \neq 0$, so we may write:

$$L_a - L_b \neq 0 \tag{4.28}$$

It is an assumption of the theorem that $L_a \geq L_b$. Let us therefore assume $L_a = L_b$, and thus $L_a - L_b = 0$. This directly contradicts equation (4.28) and we may conclude that if $\neg P$ is true when $a_i > b_j$ then R_1 must be false. Hence $\neg R_1$ must be true and $\neg Q$ cannot be false.

4. Assume $a_i = b_j$. Then R_1 and R_2 are both true but R_3 is false. Hence $\neg Q$ is true. Thus when $a_i = b_j$ and $\neg P$ is true, $\neg Q$ is always true.

5. We have shown that whatever the relative values of a_i and b_j it is not possible for $\neg Q$ to be false when $\neg P$ is true. Thus $\models \neg P \Rightarrow \neg Q$. □

4.4.2 Register Allocation Using the Q-Compatibility Test

In the experimental framework register allocation is performed assuming an *unlimited* number of queues and queue positions. However, it tries to minimize the usage of the most critical resource (number of queues) by allocating as many lifetimes as possible to a single queue. If values are allocated to queues instead of to individual registers, the number of distinct queues can be seen as the size of the *name space*.

The register allocation process starts after the modulo schedule is generated. In the first step all loop variant lifetimes are identified, as well as their start time and length. Then an interference matrix is built, specifying for every pair of lifetimes if it can share the same storage queue (Algorithm 4.3). The Q-Compatibility test presented in Section 4.4.1 is used to create the matrix, operating on each pair of lifetimes.

Algorithm 4.3 *Check Compatibility*

```
Check_Compatibility(lf, compatible, no_lf)
  no_lf= 0;
  forall op $\in$ DDG {
    /* Compute the start cycle and length of each lifetime */
    forall successor of op {
      ++no_lf
      new(lf[no_lf])
      lf.start[no_lf]= (starting cycle of op) + (latency of op) -1
      lf.length[no_lf]= (starting cycle of successor) - lf.start[no_lf] +
          (II $\times \delta_{e(op,successor)}$)
    }
  }
  /* Initialize the compatibility matrix for every pair of lifetimes */
  for i= 1 to (no_lf -1) {
    for j= (i+1) to no_lf {
      if (lf.length[i] - lf.length[j]) < ((lf.start[j] - lf.start[i]) mod II) {
        compatible[i][j]= 1
        compatible[j][i]= 1
      }
      else {
        compatible[i][j]= 0
        compatible[j][i]= 0
      }
    }
  }
```

Finally, a single pass procedure is executed, trying to allocate each lifetime to a previously assigned queue. If that is not possible due to incompatibilities, the lifetime is assigned to a new queue. This procedure is described in Algorithm 4.4.

Algorithm 4.4 *Register allocation to a QRF*

```
QRF_Allocation()
  Check_Compatibility(lf, compatible, no_lf)
  no_queues= 0;
  /* Allocate every lifetime to a queue */
  for i= 1 to no_lf {
    allocated= 0;
    /* Try to allocate lifetime i to one of the existing queues */
    while not allocated {
      /* Try to allocate lifetime i to queue j */
      for j= 1 to no_queues {
        conflict= 0
        /* Check compatibility between lifetime i and all other lifetimes */
        /* previously allocated to queue j */
        forall lf[k] ∈ queue[j]
          if compatible[i][k] == 0
            conflict= 1
        if not conflict { /* Allocation possible */
          allocate lf[i] to queue[j]
          allocated= 1
        }
        break
      }
    }
    /* Allocation not possible in any of the existing queues-use a new one */
    if not allocated {
      ++no_queues
      allocate lf[i] to queue[no_queues]
    }
  }
```

As already said, the procedure above described assumes the availability of an unlimited number of queues, each of them having as many storage positions as necessary. If those resources were limited, the algorithm should provide a mechanism to balance the distribution of lifetimes among queues. This is specially important to prevent having more live values than the actual queue capacity. We have found that even employing this simple scheme the required capacity of a queue remains low in most of the cases, as shown in Section 5.3.2.4. Effectively dealing with a finite number of storage positions would require the introduction of spill code [15], however this is out of the scope of this thesis. A discussion on spilling mechanisms for software pipelined loops can be found in [65].

4.5 Analytical Model for Register Files

An accurate comparison between conventional and queue register file organizations should consider the *silicon area* required to implement them. A further refinement would include the *cycle time* allowed by those implementations. In this thesis we have used the *analytical model* for multiported register files described in [62]. The input parameters taken by that model include the number of registers, the number of read and write ports, and the width of the registers (64 bits in this thesis). Queue register files can also be analysed using the same model. In this case the number and size of each queue is also taken into account. A brief overview of the analytical model is presented in the next subsections.

4.5.1 Silicon Area Model

We assume the VLSI technology used to implement the register file is *scalable CMOS*. Dimensions can be expressed by means of a technology dependent parameter, called λ. Modern implementations in 1998 employ $\lambda = 0.25\mu m$ processes [88]. The micron measurement refers to the *distance* between circuits on a microprocessor. Generally, a smaller λ results in a faster microprocessor.

The overall size of a multiported RF is mainly determined by the size of the memory cells, which are replicated and together represent between 85% and 95% of the total area. Other components such as decoders and read/write drivers for the data lines account for the remaining fraction. The diagram of Figure 4.10 shows a dual-ported (1R,1W) register cell that could be used to implement a multiported register file. One transistor, a select line and a data line is required by *each port*, in order to access the register cell. The area of the cell grows approximately with the square of the number of ports added because each additional port increases both dimensions of the cell. The memory portion is a pair of cross-coupled inverters consisting of four transistors, resulting in a minimum height of 41λ. The memory portion can accommodate 3 select lines running horizontally across the cell. Thus, the height does not increase until more than 3 ports are implemented. After this, each additional port adds 8λ to the height. The width of the dual-ported cell is 50λ. Each additional read port adds 14λ to the width, while additional write ports add 22λ. The analytical model used in this thesis also considers the area of other elements of the register file, which can be found in [62].

A *queue cell* of N registers can be implemented using a collection of N dual-ported register cells, like the one described above. To build a multiported QRF,

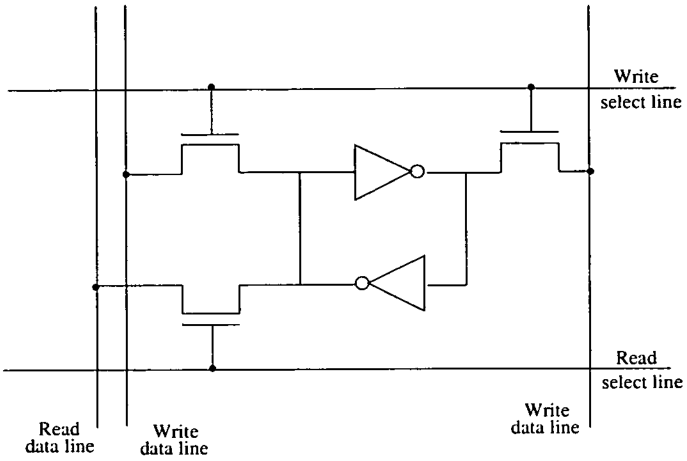


Figure 4.10: Dual-ported register cell

each queue cell can be considered as the memory portion. For each access port, an additional select line is required (Figure 4.11). Note that since each register cell of the queue cell only requires two select lines, a N element queue cell can accommodate up to N select lines without growing in height. Hence, the cell height does not grow until more than N ports are implemented. This usually allows for a *smaller area* in comparison to a multiported conventional register file of the same capacity. In that case, each additional port increases the size of individual memory cells. Other elements of the queue register file occupy a relatively smaller area, which is also taken into account by the model used in this work, as described in [62].

4.5.2 Cycle Time Model

The cycle time of register files and queue files is also modelled in [62]. The timing model uses the technology parameters of CACTI [99]. The access time to a multiported register file is determined by the length of the word-lines, the length of the bit lines and, to a lesser extent, by decode time. Other components, such as sense amplifier or precharge delays, contribute minimally to the cycle time, but are included in the model for completeness and accuracy. The length of each word-line depends on the width of the bit-cell, which depends on the number of ports, and the number of bits per register. Conversely, the length of each bit-line depends on the height of the bit-cell, which varies with the number of ports and the number of registers. Finally, decoder dimensions are determined by the number of registers and the height of the memory array, which in this layout is

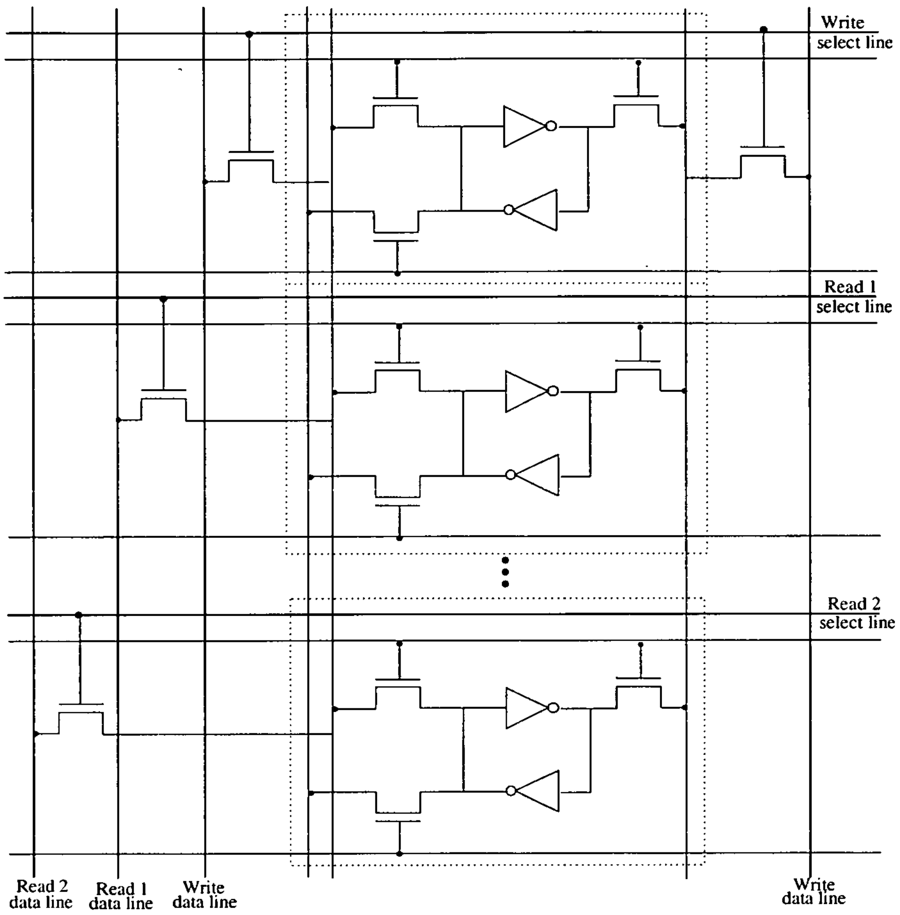


Figure 4.11: Implementation of a QRF using dual-ported register cells

computed as the product of the number of registers and the height of a bit-cell. They are implemented using a multi-level predecoding scheme. Hence, there is a predecoded address bus running parallel to the bit-lines, as in the CACTI model.

4.5.3 Comparing Register File Organizations

We have used both models above presented to make a preliminary comparison between RF and QRF organizations. We compare structures having the same number of storage locations and access ports. The main objective at this point is to compare register files with similar *storage capacity*. Evaluating the *effectiveness* of those structures would require the analysis of performance figures associated with those configurations, which will be done in the next chapters.

In this evaluation we assume that the number of access ports in either case is based on the number of functional units sharing the register file. Ten configurations ranging from 3 to 30 FUs have been evaluated. Each functional unit

FUs	Register File Configuration				
	Access		RF	QRF	
	Read Ports	Write Ports	Registers	Queues	Queue Length
3	6	3	32	4	8
6	12	6	64	8	8
9	18	9	96	12	8
12	24	12	112	14	8
15	30	15	128	16	8
18	36	18	144	18	8
21	42	21	160	20	8
24	48	24	176	22	8
27	54	27	192	24	8
30	60	30	208	26	8

Table 4.1: Register file configurations

requires two read and one write ports. The total number of storage locations for a conventional register file was arbitrarily chosen. It is an approximation of the register requirements of a VLIW machine using a multiported register file, as reported in Section 5.3.2. We assume that each queue in the QRF has 8 locations. Thus the number of queues is chosen in order to match the total number of storage locations of a RF. The parameter values used in this analysis are summarized in Table 4.1.

It can be seen in Figure 4.12 that the silicon area required to implement a QRF is always smaller than the area of a RF. Furthermore, the rate of increase is much lower as the number of functional units scales up. Similar results were observed when analysing the cycle time resulting from both implementations (Figure 4.13). Those results have confirmed that in general a QRF is more efficient than the equivalent RF in terms area and cycle time. However the analysis presented did not take into account the functionality of those register files. Distinct performance levels may be result from from conventional and queue register files. These issues are addressed in the next chapters.

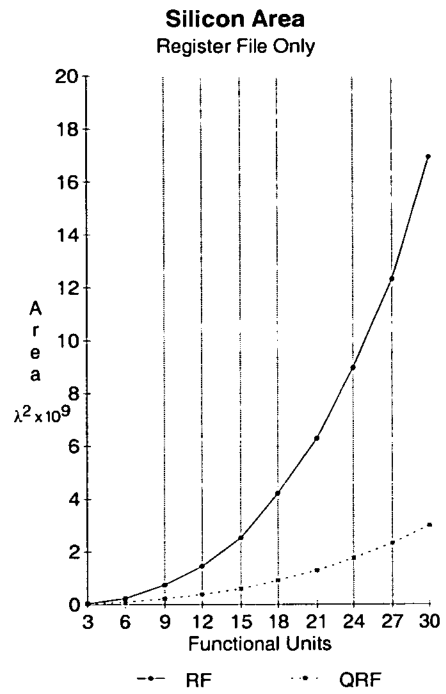


Figure 4.12: Silicon area

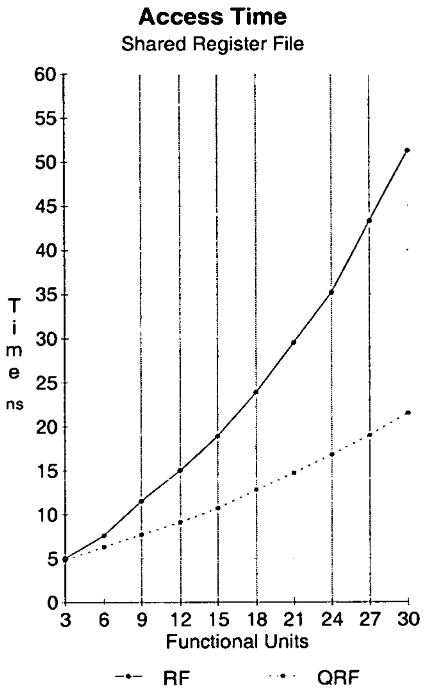


Figure 4.13: Access time

4.6 Summary of Results and Conclusions

The following list summarizes the main results and conclusions of this chapter:

- A queue register file, as defined in this chapter, presents a number of advantages to support the execution of modulo scheduled loops.
- A novel strategy was proposed for the allocation of loop variant lifetimes to a QRF. The scheme is based on the *Q-Compatibility Test.*
- The proposed QRF requires the introduction of copy operations in the *DDG*. Experimental analysis have shown that the performance penalty due to this transformation is acceptable.
- In general QRFs are more efficient than conventional RFs in terms of silicon area and cycle time. However further experiments are required to assess the performance level allowed by those implementations.

Chapter 5

Unclustered Architectures

Our main motivation to design a queue register file is the register requirements resulting from the execution of software pipelined loops in a VLIW machine. We have shown in Chapter 4 that a QRF organization have some advantages over a RF in terms of silicon area and access time. In this chapter we present a number of experimental results comparing two types of *unclustered* architectures, which differ only by the register file organization. The basic characteristics of an unclustered machine are described in Section 3.1.3. The experiments compare the performance and required machine resources of both organizations. Machine configurations ranging from 3 to 30 functional units have been considered, which connect either to a RF or to a QRF (Figure 5.1). The number of access ports of each register file is determined by the number and type of the functional units (Section 5.3).

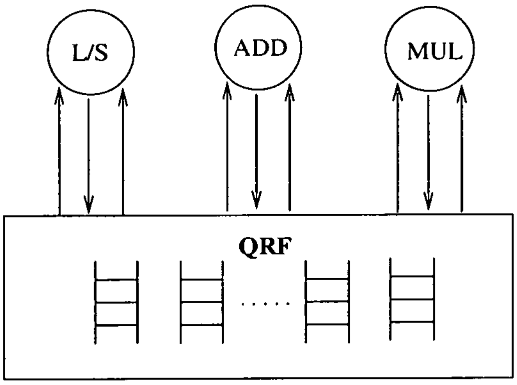


Figure 5.1: Unclustered machine using a QRF

It is well known that many programs cannot take full advantage of the available hardware parallelism. Accordingly, one objective of the experiments was to measure the *performance improvement* achieved when the benchmark is executed in wider-issue machines. As discussed in Section 4.3, the use of a QRF may result in some performance degradation due to the introduction of copy operations in the loop *DDG*. This set of experiments extends the investigation on that issue. Early results suggested that using a QRF instead of a RF favours scalability [28, 27], which is possibly complemented by benefits in silicon area and access time (Section 4.5). Hence, another objective of the experimental evaluation was to quantify the *machine resources* required to achieve a given performance level.

Optimizing the use of a wide-issue VLIW machine requires finding large amounts of ILP. However, that is not always available in the body of single loop iteration.We have used *loop unrolling*, performed prior to modulo scheduling, to address this issue, which is described in the next section. Then the experimental framework is updated with the QRF organization and related parameters. The chapter finishes presenting experimental results and related conclusions.

5.1 Increasing ILP with Loop Unrolling

Requiring that the II be an integer can result in sub-utilization of machine resources. That situation happens because the II should be rounded up to the next integer. As discussed in Section 3.3.1, in this experimental framework the II can be determined by the most heavily used functional unit. Consider a resource constrained loop with 3 instructions using the L/S functional unit. Assuming that the machine model has *2* FUs of this type results in a $MII = 1.5$, and thus $II = 2$. However this implies in one idle L/S slot every 2 cycles. A particular situation occurs when $MII < 1$. Suppose the same loop is scheduled in a machine model with *4* FUs of that type. In this case $MII = 0.75$, $II = 1$, and *1* L/S slot is idle every 4 cycles.

Resource constrained loops having $MII < 1$ do not have enough operations in the loop body to use all available functional units. One way to minimize this problem is to perform **loop unrolling** [20] of the loop body prior to modulo scheduling. Loop unrolling replicates the original loop body multiple times, eliminating unnecessary branch instructions. This results in a larger basic block, increasing the possibilities of finding ILP.

Returning to the above examples, distinct unroll factors can be used to avoid having idle L/S FUs. Unrolling the loop *twice* for the machine model with 2 L/S FUs result in $MII = 3$. On the other hand, for the machine model with 4 L/S FUs the loop can be unrolled four times, in which case $MII = 3$. In both cases the schedule has no idle L/S slots, achieving full processing power. Although both schedules have the same II, the execution time of *individual* iterations is different. In the first case 2 iterations are completed every 3 cycles. In the second, 4 iterations are completed in the same amount of time, because the machine has twice as many L/S FUs. These examples are summarized in Figure 5.1. A study on the benefits of using loop unrolling with modulo scheduling can be found in [56].

Scheduling Parameters	Machine Model A 2 L/S FUs		Machine Model B 4 L/S FUs	
Unroll Factor	1	2	1	4
L/S Operations	3	6	3	12
MII	1.5	3	0.75	3
II	2	3	1	3
Cycles/Iteration	2	1.5	1	0.75
FUs Utilization	75%	100%	75%	100%

Figure 5.2: Optimizing the use of machine resources with loop unrolling

This research work has considered wide-issue machine models employing up to 30 FUs. For this reason loop unrolling has been used to increase the ILP available in small loops. Furthermore, unrolling has also been used to minimize the negative effects of rounding the *ResMII* up to the next integer [78]. Although performance gains can be achieved with unrolling, side effects may also occur, which might compromise the achieved benefits. In this work two issues are of particular concern, being investigated through experimental analysis:

- Unrolling may generate a loop containing too many operations and complex dependence chains, making it difficult and time consuming to find a valid schedule with IMS.

- Unrolling may further increase the register pressure.

In the experimental framework loop unrolling is performed only if the *ResMII* is the dominating factor determining the *MII*. The number of times a loop body is unrolled is called **unroll factor** (u). It is chosen according to the performance degradation incurred when the *ResMII* is rounded up to the next integer. The unroll factor u is determined in order to minimize a tolerance value, called $U_{tolerance}$, which can be calculated by means of the following expression [78]:

$$U_{tolerance} = \frac{\lceil ResMII \times u \rceil}{ResMII \times u} - 1 \tag{5.1}$$

In general the maximum unroll factor used in this experimental framework is 10. This avoids creating loops with too many operations. So the unroll factor is chosen between 1 and 10, whichever results in the smallest $U_{tolerance}$. Unroll factors that would result in a *DDG* with over 250 operations are not considered. The reasons for this are twofold: possible performance gains are not so significant because the wasted fraction is small compared to the II. Furthermore, a large number of operations may result in a higher II due to excessive backtracking, which also increases the scheduling time. However, it might happen that for very small loops an unroll factor of ten is still insufficient to have $ResMII \geq 1$. In this cases an unroll factor as high as 40 can be used. Loop unrolling is also used to support the partitioning algorithm when small loops are scheduled, as discussed in Section 7.1.4. We have found that the criterion used to perform loop unrolling does not result in significant performance degradation or excessive use of machine resources [27]. Thus we have adopted it as a standard feature of the experimental framework, being performed whenever necessary.

Unrolling an original *DDG* m times produces a new graph which we shall call *UDDG*. For each vertex u representing an instruction $u_i \in DDG$, *UDDG* has m vertices u^j corresponding to distinct iterations of the same operation. Likewise, for each dependence edge $e(u, v) \in DDG$, m dependences $e(u^i,\ v^{(i+\delta(u,v))\ mod\ m}) \in UDDG$ are created. The loop unrolling scheme [85] used by the experimental framework is described in Algorithm 5.1. If the number of iterations of the original loop is N, the number of iterations performed by its unrolled version is approximately $(N \div m)$. So we have defined a new parameter called II_{si} to express the number of cycles required to execute a *single iteration* of the original loop body. This parameter is useful to compare schedules of the same loop having distinct unroll factors, and is calculated by means of the following expression:

$$II_{si} = \frac{II}{m} \tag{5.2}$$

Algorithm 5.1 *Loop Unrolling*

```
Unroll(DDG, UDDG, m)
  /* Replicate each operation according to the unroll factor m */
  forall operation $u \in DDG$ {
    for i=0 to m-1 {
      $u^i = u$
      Insert(UDDG, $u^i$)
    }
  }
  /* Insert edges in order to preserve the original data dependences */
  forall edge $e(u,v) \in DDG$ {
    for i=0 to m-1 {
      $new\_edge = e(u^i,\ v^{(i+\delta(u,v)) \ mod \ m})$
      $\delta_{new\_edge} = \left\lfloor \frac{i+\delta(u,v)}{m} \right\rfloor$
      $\lambda_{new\_edge} = \lambda_e$
      Insert(UDDG, new_edge)
    }
  }
```

5.2 Experimental Framework Update

So far the experimental framework considers only an unclustered machine model using a conventional register file, as described in Section 3.1.2. In this section that model is extended to consider also the use of a QRF. A machine using a QRF requires Copy FUs, and thus extra access ports to the register file (see Table 3.2).

Only *loop variant* lifetimes are considered when register allocation is performed for a QRF, which is done using the scheme described in Section 4.4.2. Although some alternatives have been considered to allocate loop invariants [27], they have not been as yet implemented. Furthermore, loop invariants account only for a small fraction of the total register requirements, thus this simplification should not affect the results significantly. An unclustered machine using a conventional register file (RF) to perform register allocation of loop variants only is called URV. A similar architecture using a QRF is called UQV. Whenever necessary the suffix nn is used to indicate the number of functional units of a given configuration. Copy FUs are not taken into account to compute the total number of FUs as they do not perform any useful computation from the user's perspective. Thus, although two machines denoted *URVnn* and *UQVnn* have the same number of standard FUs, *UQVnn* has also Copy FUs, requiring additional access ports to the QRF. Additional *output parameters* related to the QRF are generated by the experimental framework, including the following:

- *Number of queues:* Total number of queues required to allocate loop variant lifetimes produced by a modulo schedule. This parameter can also be viewed as the size of the register *name space* for a machine using a QRF.

- *Queue length:* Maximum number of live values that must coexist in a queue at a given cycle. The length corresponds to a lower bound on the required storage capacity of a queue.

- *Queue locations:* Total number of storage positions required by a modulo scheduled loop. It is calculated summing up all queue lengths resulting from the register allocation. Although it may not be necessary to use all locations at the same time, the characteristics of a QRF require this capacity.

- *Queue sharing:* Number of lifetimes produced by *distinct* operations sharing a queue at any given cycle.

Some loops allow a higher degree of performance improvement than others as the machine model scales up. In the experimental framework this is particularly the case for resource constrained loops. Recurrence constrained loops do not benefit from extra functional units. The reason for this is that unrolling is not performed in loops having $ResMII < RecMII$. Simply unrolling the loop may reduce the impact of rounding the *RecMII* up to the next integer [56]. However this does not help to fully utilize extra machine resources, as happens when $ResMII < 0$. Achieving significant performance gains in loops with $ResMII < RecMII$ requires sophisticated techniques, such as blocked back-substitution [87]. This technique unrolls a loop m times, reducing the *RecMII* by the same factor. However its implementation is outside the scope of this thesis.

For this reason we understand that resource constrained loops should be the main target of wide-issue machine configurations. We have subdivided the benchmark loops into three classes in order to produce a more precise analysis:

- *Class 1:* Contains all 1258 loops.

- *Class 2:* Contains only resource constrained loops, thus $ResMII \geq RecMII$. They fully benefit from extra functional units. Although *RecMII* is a fixed parameter, $ResMII$ depends on the number of FUs. Thus the number of loops in this set is machine dependent.

- *Class 3:* Contains only loops *without* recurrence circuits, characterized by $RecMII = 0$. The number of loops in this set is constant, regardless of the machine configuration. All class 3 loops also belongs to class 2.

The data in Figure 5.3 shows the number of loops in each class. Class 1 contains 1258 loops, while class 3 contains 753 loops, which is machine independent in both cases. The number of class 2 loops ranges from 1102 (for 3 FUs) to 807 (for 30 FUs). The chart shows that the number of class 2 loops converges to the number of class 3. If a very large number of functional units was available the condition $ResMII < 1$ would always hold for those loops. Thus the II would always be rounded to 1. A class 3 loop has $RecMII = 0$, which is also rounded to one. Thus for a very large number of functional units the condition to be a class 2 loop, $ResMII \geq RecMII$, is equivalent to $ResMII = RecMII$, which results in the same loops being included in both sets.

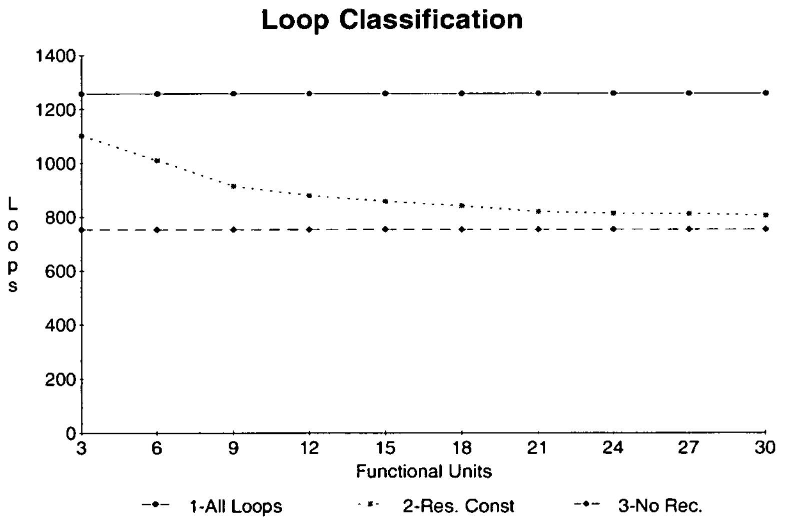


Figure 5.3: Number of loops in each class

When reporting performance results we often present only the results for classes 1 and 3. Although we are mostly interested in the loops of class 2, the number of them is not fixed for distinct machine configurations. We avoid comparing sets of distinct sizes by using class 3 loops instead. Nonetheless the conclusions obtained should be the same as the characteristics of both sets are the same regarding the parameters of interest. Unless otherwise stated the results present machine models scaling by increments of 3 standard functional units, plus extra Copy FUs if required.

5.3 Experimental Results

This section presents experimental results regarding unclustered architectures, using either a conventional or a queue register file. A total of 30 machine configurations were considered: 10 URV and 20 UQV, each group ranging from 3 to 30 functional units. Two type of UQV machine models have been used, differing only by the number of Copy FUs available. One configuration, called *UQV1* has a fixed proportion of *1* Copy to *3* standard FUs. The other, called *UQV2*, provides *at least 1* Copy for each group of *6* standard FUs. The queue register files used by these machines are referred as *QRF1* and *QRF2*, respectively. These machine configurations are summarized in Table 5.1.

Unclustered Machine Configurations			
Functional Units	URV	UQV1	UQV2
L/S	1-10	1-10	1-10
ADD	1-10	1-10	1-10
MUL	1-10	1-10	1-10
Copy	-	1-10	1-5
Register File	RF	QRF1	QRF2
Read ports	6-60	7-70	7-65
Write ports	3-30	5-50	5-40

Table 5.1: Unclustered machine configurations used in experiments

Innermost loops taken from the Perfect Club Benchmark were used in the experiments (Section 3.2). Loop unrolling prior to modulo scheduling was performed according to the criteria described in Section 5.1. We have subdivided the presentation of results into two main topics: performance, and machine resources.

5.3.1 Performance Analysis

In this section the performance of the machine models defined in Section 5.3 is analysed and compared. First the potential for parallelism exploitation is investigated, followed by an investigation of the impact caused by the use of copy operations with a queue register file. The analysis presented in this section is focused on scheduling issues, so it assumes a *fixed* cycle time for all machine configurations. In Section 5.3.2.5 a reasonable estimation for the register file cycle time is presented, which is used to calculate the actual execution time of each loop.

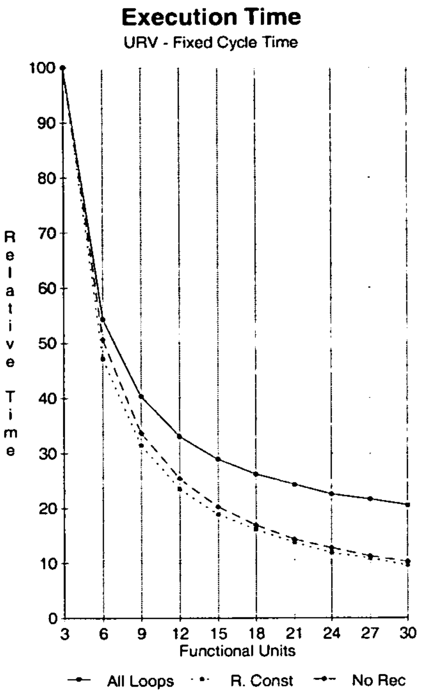
5.3.1.1 ILP Exploitation

Implementing a wide-issue VLIW machine cannot be justified unless it produces significant performance improvements for the target applications. Accordingly, this first set of experiments investigates the performance improvement achieved by scaling up the number of functional units in URV machine models. We have found that the total number of cycles required to execute all loops from a given class can be significantly reduced using wider-issue machines, as shown in the chart in Figure 5.4. The results are relativized using as a baseline the execution time of the corresponding class of loops in a URV03 machine. In this case the number of cycles required to execute all loops of classes 1, 2 and 3 is 3.6×10^{10}, 3.4×10^{10}, and 1.3×10^{10}, respectively.

It should be noticed that the execution time of classes 1 and 2 do not differ by a large factor for the smallest configuration, which shows that a large fraction of the execution time is spent in resource constrained loops. As the number of functional units increases, so does the gap between both curves, showing that class 2 loops take more advantage of the extra functional units. This is most clearly concluded by comparing the achieved speedup for each class of loops (Figure 5.5). Classes 2 and 3 can achieve linear speedups as they consist of vectorizable loops. The speedup of class 1 is sublinear because the recurrence constrained loops found in this class do not take full advantage of extra functional units.

5.3.1.2 Impact of Copy Operations on Performance

As previously discussed the use of a QRF may delay loop execution due to the introduction of copy operations. In order to further investigate this issue we have extended the analysis presented in Section 4.3, now considering additional machine configurations and performing loop unrolling when necessary. The analysis compares performance execution using three machine models: URV, UQV1, and UQV2, as previously defined. A comparison of the execution time of all loops is shown in Figure 5.6. Small performance degradation in relation to configuration URV is observed when UQV1 is used. If UQV2 is used instead, the total execution time increases by a larger factor. This confirms that Copy FUs can be the critical resource for some loops, determining the II. The overhead tends to increase as the machine configuration scales up. The main reason for this is that more loops become recurrence constrained as more functional units are used. In this case more loops are affected by the introduction of Copy operations in a recurrence circuit. Furthermore, a wider-issue machine may require a higher unroll factor. Thus, more Copy operations are necessary, possibly increasing the II. A



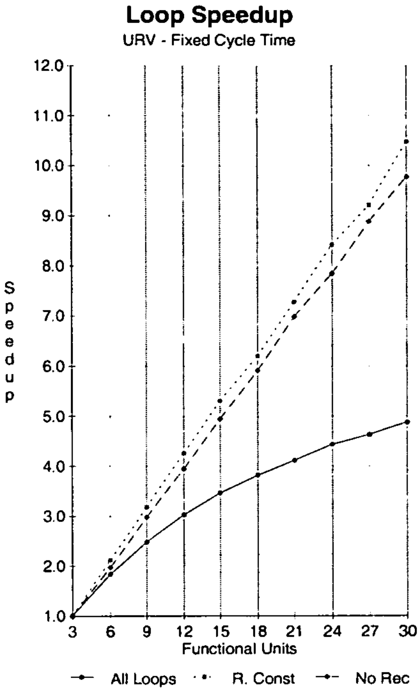


Figure 5.4: Execution time-Fixed cycle

Figure 5.5: Loop speedup-Fixed cycle

smaller overhead is observed when only loops without recurrences are considered (Figure 5.7). In this case the extra execution time is only due to loops whose critical resource is the Copy FU.

The value of $IPC_{dynamic}$ also shows small variations between machine models for both sets of loops, as seen in Figures 5.8 and 5.9, respectively. However the performance loss is larger when UQV2 is used, for the reasons above discussed. A linear increase is observed for loops without recurrences, which is also achieved for resource constrained loops. When all loops are included, the growth of $IPC_{dynamic}$ is sublinear because recurrence constrained loops do not use all the available FUs.

We have concluded that it is possible to achieve high speedup levels using any of the architecture models proposed, specially for loops that are resource constrained. It might be possible that additional techniques make it feasible to accelerate an even larger class of loops. We have found that using a QRF does not result in significant performance degradation, as long as an adequate number of Copy FUs is provided. However the inclusion of extra functional units implies new access ports, which increases the complexity of the QRF, possibly compromising performance. Hence, a trade-off is involved in the performance optimization of UQV machines, between scheduling flexibility and number of access ports.

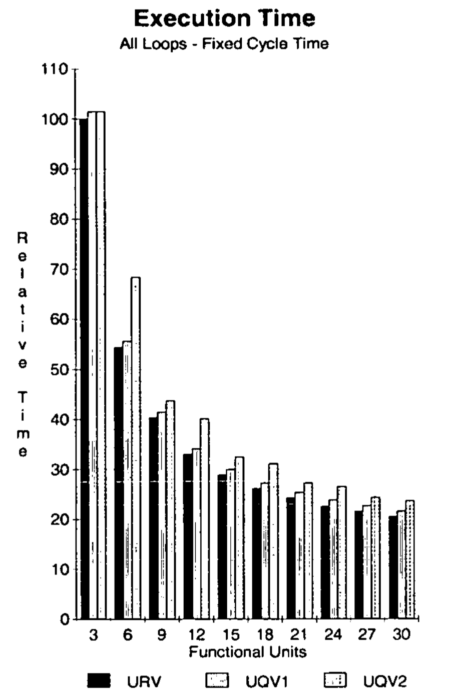


Figure 5.6: Copy overhead-Class 1

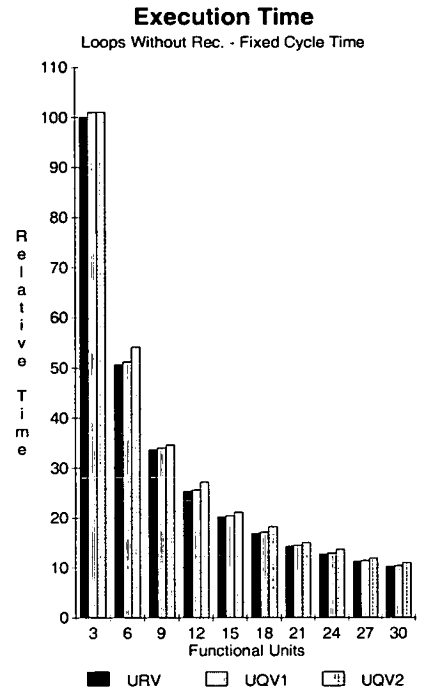


Figure 5.7: Copy overhead-Class 3

5.3.2 Machine Resources Analysis

The experiments shown in this section compare the machine resources required to achieve the performance levels reported in Section 5.3.1. An issue of particular interest is the scalability of the model using a queue register file. As defined in Section 5.2, all data refer to dynamic measurements, accounting for the loops responsible for 99% of the total execution time of the benchmark.

5.3.2.1 Name Space

As already discussed, the maximum number of live values at a given cycle determines the required number of distinct storage locations. In a conventional register file each storage location requires a distinct register name. However, in a queue register file all registers comprising a given queue are referred to by the same register name. Thus, the register name space problem is shifted from register names to queue names. A smaller name space size may require less bits to specify the address of instruction operands, possibly simplifying the instruction word format and the register file implementation.

The data in Figure 5.10 compares the size of the name space of both organizations. The growth of the name space for a RF is almost linear as the number of

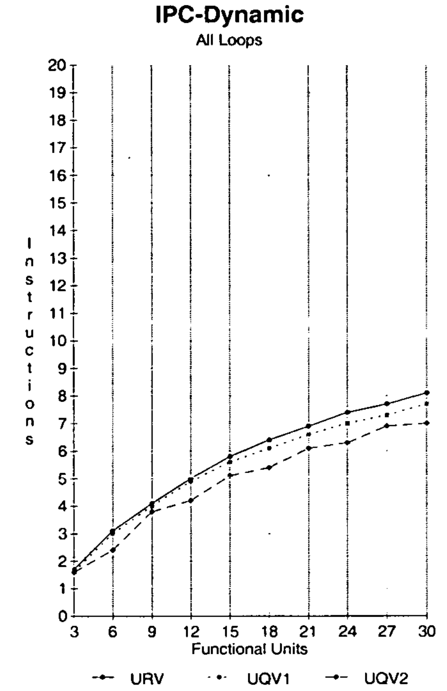


Figure 5.8: IPC Dynamic-Class 1

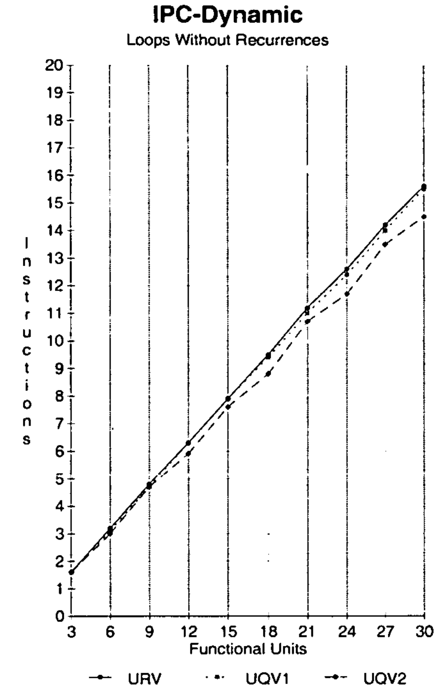


Figure 5.9: IPC Dynamic-Class 3

functional units scales up. On the other hand, for QRFs the name space increases constantly by a small factor across the machine models. It should be noticed that the size of the name space is smaller if less Copy FUs are used, as can be inferred by comparing the results for UQV1 and UQV2.

5.3.2.2 Storage Locations

We define the number of storage locations as the lower bound on the register file capacity. For a RF it is determined by MaxLive. However for a QRF it is computed according to the longest size of each queue during loop execution. The longest size of a queue can be viewed as its MaxLive value. As expected, the requirements are higher for a queue register file (Figure 5.11). Although register requirements for UQV2 are lower than for UQV1, they are always higher than for RF. Multiple-use lifetimes are the main reason for this difference. As discussed in Section 4.2, a multiple-use lifetime requires only one storage location in a RF. If a QRF is used instead, multiple locations are required. Furthermore, a QRF requires copy operations to move data between queues, using additional storage locations to implement the data transfer. The somewhat irregular shapes of the curves are due to distinct unroll factors. The main criteria determining the

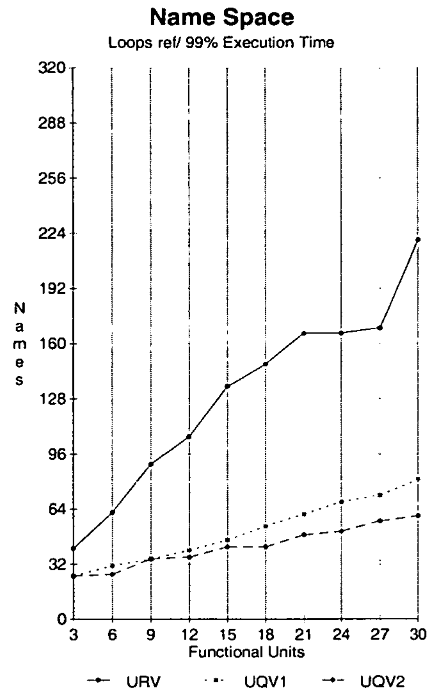


Figure 5.10: Name space

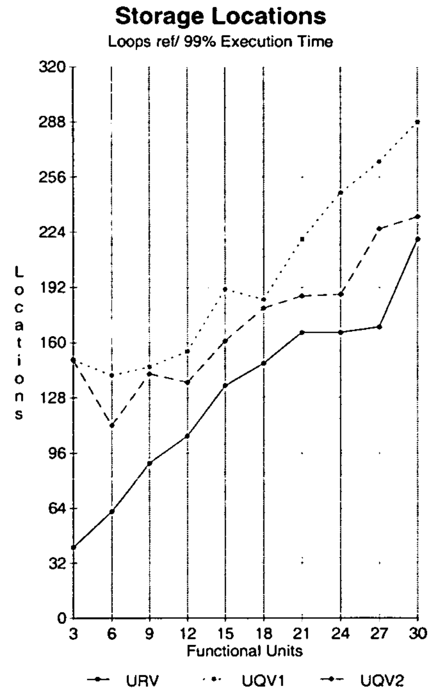


Figure 5.11: Storage locations

extent of unrolling is performance optimization, which depends on the number of functional units. Distinct unroll factors may result in large differences in register requirements. In any case the experiments have confirmed that a QRF requires more storage locations than the equivalent RF.

5.3.2.3 Queue Sharing

The chart of Figure 5.12 shows the number of lifetimes *produced by distinct operations* sharing the same queue. It was measured over all the benchmark loops for UQV1 configurations. The data is subdivided into three cumulative sets: *more than one*, *two*, and *three* distinct lifetimes in a queue. The results are normalized based on the total number of cycles in which one or more live values are stored in a queue. It can be seen that sharing does occur, being the reason for a reduction in the size of the name space of UQV machines. Similar results were observed for UQV2 configurations.

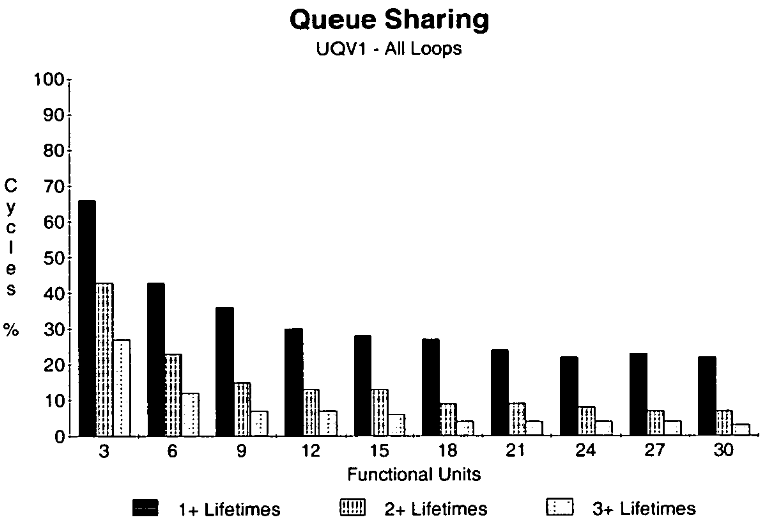


Figure 5.12: Lifetimes sharing a queue

5.3.2.4 Register File Silicon Area

We have used the hardware model presented in Section 4.5 to compare the silicon area of the register files required to implement the architecture models described in this chapter. All results are based on CMOS technology using $\lambda = 0.8\mu m$. We assume that MaxLive determines the number of registers in a conventional RF. Using a similar approach for a QRF would not be appropriate. In a real implementation the size of each queue is finite, possibly constant, a constraint that is not taken into account when calculating the maximum number of live values. We have measured the dynamic length of every queue used by both organizations, UQV1 and UQV2, as shown in Figure 5.13. The maximum size of each queue ranges between 7 and 10, for those loops accounting for 99% of the total execution time.

We have defined the specification of each register file as follows: The size of the *name space* (Figure 5.10) determines the *number of registers* for a RF, or the *number of queues* for a QRF. The *size* of each queue is based on the maximum *queue length* for the corresponding configuration (Figure 5.13). The exact value of those parameters can be found in Tables 5.2 and 5.3. It is assumed that each register location is *64 bits* wide. It is clear that the capacity of QRF configurations are more than enough in terms of total storage locations. However, that allows flexibility to the scheduler and register allocator.

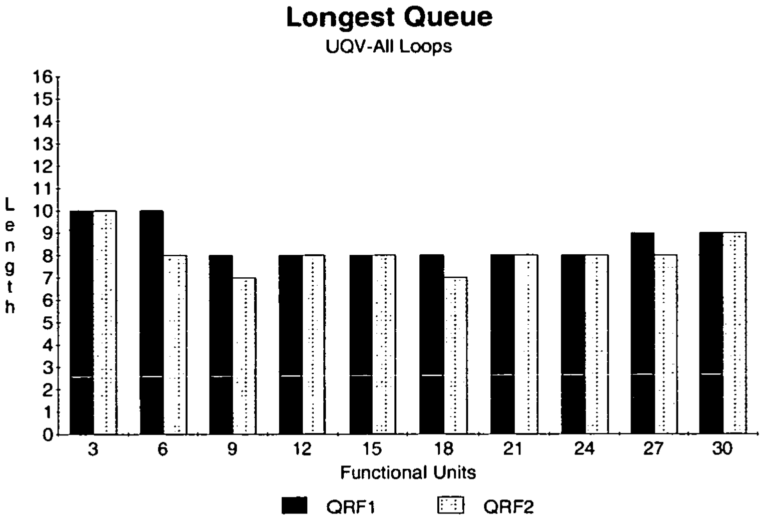


Figure 5.13: Maximum queue length

RF Parameters			
	Capacity	Ports	
FUs	Registers	Read	Write
3	41	6	3
6	62	12	6
9	90	18	9
12	106	24	12
15	135	30	15
18	148	36	18
21	166	42	21
24	166	48	24
27	169	54	27
30	220	60	30

Table 5.2: URV register requirements

QRF Parameters								
	QRF1				QRF2			
	Capacity		Ports		Capacity		Ports	
FUs	Queues	Length	Read	Write	Queues	Length	Read	Write
3	25	10	7	5	25	10	7	5
6	31	10	14	10	26	8	13	8
9	35	8	21	15	35	7	20	13
12	40	8	28	20	36	8	26	16
15	46	8	35	25	42	8	33	21
18	54	8	42	30	42	7	39	24
21	61	8	49	35	49	8	46	29
24	68	8	56	40	51	8	52	32
27	72	9	63	45	57	8	59	37
30	81	9	70	50	60	9	65	40

Table 5.3: UQV register requirements

All figures referring to silicon area are presented in λ^2 units (Section 4.5). The data in Figure 5.14 shows that QRF1 organizations uses more silicon area than a conventional RF, although the differences are not large in most of the cases. QRF2 and RF organizations have use areas up to 12 FUs. However, if more functional units are used there is an increasing advantage for QRF2.

5.3.2.5 Register File Cycle Time

A number of critical paths can determine the cycle time of a wide-issue dynamically scheduled processor. The first one refers to the number of access ports of the register file. The second one refers to a number of structures used for dynamic instruction-scheduling [24]. As reported in [25], the machine cycle time of both superscalar and VLIW processors may be determined by the cycle time of the register file. The DEC Alpha 21264, for instance, has a partitioned integer register file because it is on a critical timing path [41]. Considering that VLIW processors have no hardware for instruction scheduling, it would be reasonable to estimate the machine cycle time according to the register files. Thus, we will use this approach through the remainder of this thesis.

The analytical model presented in Section 4.5 was used to estimate the cycle time of the machine configurations considered in this analysis. We have found that in most of the cases the access time of RF and QRF1 organizations are similar (Figure 5.15). Although QRF1 has more access ports than RF, that is compensated by the lower complexity of a queue structure. QRF2 allows a shorter

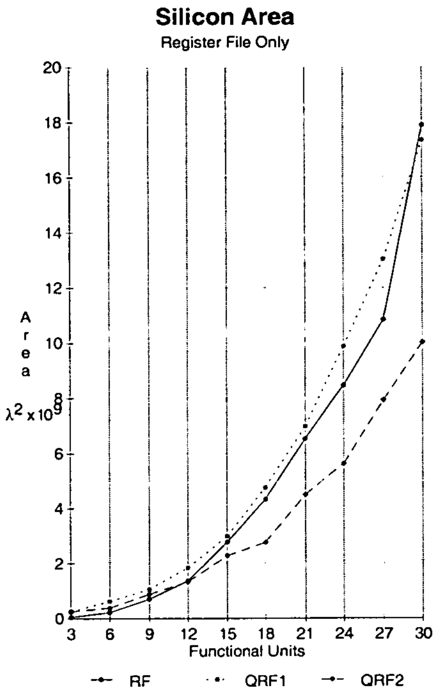


Figure 5.14: Silicon area

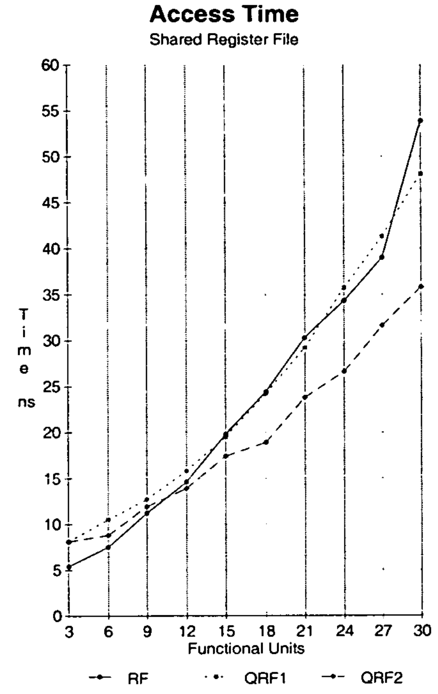


Figure 5.15: Cycle time

cycle time than QRF1 because it has less access ports. However, as shown in Section 5.3.1.2 this option results in a higher performance penalty due to a smaller number of Copy FUs. Although those results assume a fixed cycle time, that is not the case in real implementations. In fact, scheduling for a machine having less Copy FUs and less access ports may actually result in a better performance.

We have used the cycle time values calculated for each configuration to weight the execution times reported in Section 5.3.1, which assume a fixed cycle time for all machines. This should provide a more accurate insight on the actual performance. The results are normalized using as a baseline the total execution time of the corresponding set of loops in a URV03 machine, measured in ns. In this case, the execution time for all loops of classes 1, 2 and 3 is 2×10^{11} ns, 1.8×10^{11} ns, and 7×10^{10} ns, respectively.

The chart in Figure 5.16 shows the total execution time of all benchmark loops. The most important finding is that *real* performance improvement only occurs when configuration URV03 scales up to URV06. In this case there is a reduction in the total execution time over the previous configuration. URV configurations allow better performance than UQV up to 12 FUs. The performance of URV15 and UQV15 are similar, but for more than 15 FUs UQV has a better performance than URV machines. Nonetheless the implementation of any of those machines cannot be justified as their *absolute performance* is *worse* than URV06 configuration. Using more than 6 FUs only *increases* the total execution time. This means that the improvements resulting from aggressive ILP scheduling are surmounted by a longer cycle time, which is due to high register requirements.

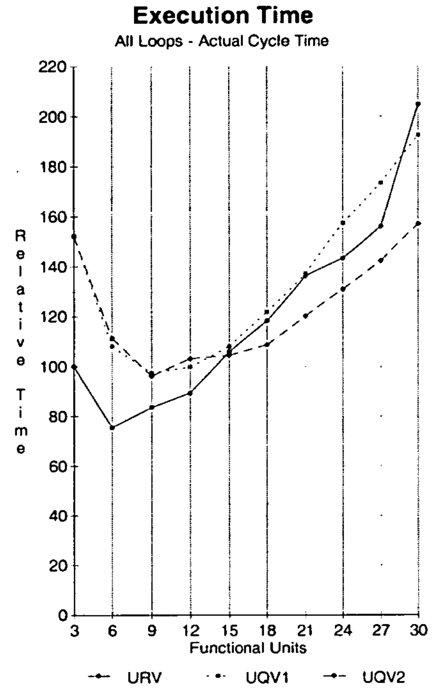


Figure 5.16: Execution time-Class 1

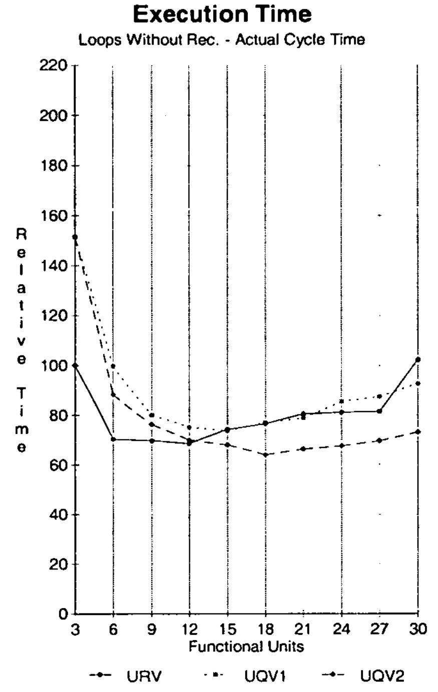


Figure 5.17: Execution time-Class 3

Distinct conclusions can be drawn if only loops without recurrences are considered. Actual performance improvement occurs when the number of FUs increases from 3 to 6 (Figure 5.17). However the lowest execution time is produced by the UQV18 machine, although URV06 (a much simpler implementation) has similar performance. UQV machines of more than 18 FUs do not allow further performance improvements. It should be noticed that, for this class of loops, the absolute performance of UQV machines does not degrade by a large factor with wider-issue machines, as observed in the other case. It could be said that the improvements of ILP scheduling approximately matches the performance penalty resulting from a complex register file. Nonetheless, the results suggest once again that implementations of unclustered machines with more than 6 FUs are not well justified.

5.3.3 Summary of Results and Conclusions

The following list comprises the main conclusions drawn from the experimental analysis presented in this chapter:

- Wide-issue VLIW machines can be efficiently exploited by means of aggressive scheduling of resource constrained loops. This can be done using a combination of software pipelining with loop unrolling.

- Although recurrence constrained loops do not allow the same level of improvement, resource constrained loops constitute a significant part of the full benchmark set.

- The introduction of copy operations alone in the loop *DDG* does not compromise the scheduling quality to a large extent. A more important constraint is the number of available Copy functional units.

- QRF organizations have a clear advantage over conventional RFs in terms of name space, considering both aspects, absolute value and scalability. However, a QRF requires more storage locations as the required size of a queue (around 8 locations) is not fully utilized.

- The number of Copy FUs determines if a QRF is preferable than an RF in terms of silicon area and cycle time. Having 1 Copy FU for each 3 standard FUs requires too many additional access ports, making the RF a better choice. This conclusion is reversed if 1 Copy FU is used for each 6 standard FUs.

- Wide-issue unclustered machines require large and complex register files, resulting in much longer cycle times. This can completely overshadow performance gains resulting from aggressive ILP scheduling.

- There is a significant potential for ILP exploitation in VLIW machines. However unclustered designs over 6 FUs fail to deliver the expected performance due to larger register files.

Chapter 6

Clustered Architectures

We have found in Section 5.3.1 that scheduling techniques targeting wide-issue VLIW machines can take advantage of the available ILP found in loops. However a centralized register file would be required if all functional units are included in a single cluster. The number of registers and access ports of such organization result in a large access time, which can overshadow the performance gains obtained from ILP scheduling. We have shown in Section 5.3.2.5 that unclustered machines of more than 6 FUs do not deliver the expected performance, either using conventional or queue register files. Those findings have motivated the development of a clustered VLIW architecture. In this organization each cluster should contain a number of FUs small enough to result in a short register file cycle time.

6.1 Clustered Architecture Organization

The overall structure of the clustered VLIW architecture proposed in this thesis is shown in Figure 6.1. It comprises a conventional superscalar CPU, plus a collection of VLIW clusters connected in a bi-directional ring topology. The front-end processor is responsible for executing all parts of a program except the innermost loops, which are scheduled for parallel execution on the VLIW compute-engine. The clustered VLIW processor executes the innermost loops of an application compiled using a modified version of IMS Algorithm [79], which is shown in Section 6.3. In this thesis we focus exclusively on the performance and cost considerations of the VLIW compute-engine. Optimization of the front-end is also important, but plays a lesser role when scalability is the prime concern.

Each cluster contains an instruction processor capable of issuing instruction parcels to the pipelined functional units from a statically compiled loop schedule. For convenience the functional units are grouped within a cluster, connecting to a *conventional* register file.

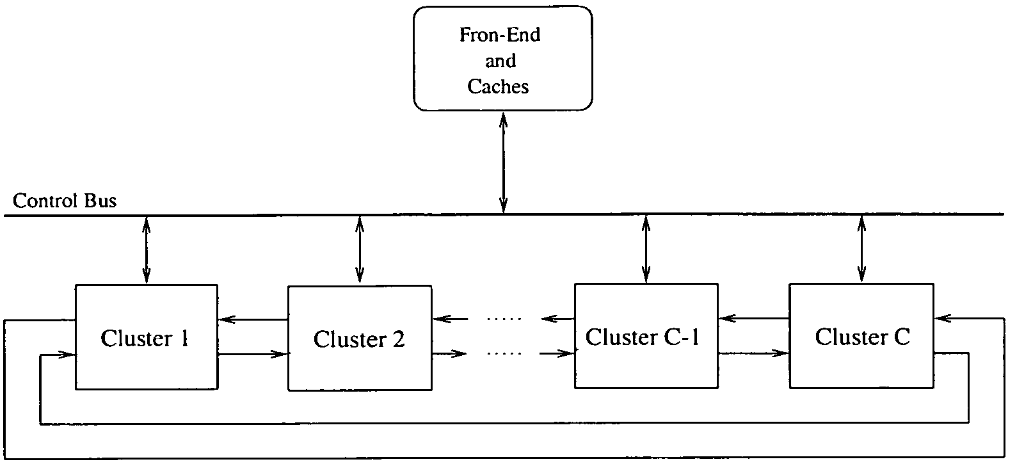


Figure 6.1: Clustered machine

It was shown in Section 4.4 that loop variant lifetimes can be allocated to a queue register file. However, loop invariant lifetimes also require a special allocation scheme, due to the read-once limitation of our QRF model. We believe that such scheme can be implemented using copy operations to write a value back to its source queue after a read operation. As shown in Section 4.5.3, the silicon area and cycle time of conventional and queue register files are similar for configurations up to 6 functional units. Thus, we understand that in these cases the small disadvantage of a conventional register file is compensated by the flexibility of a standard design. Doing so, the proposed VLIW architecture could also be used with other scheduling techniques targeting non-loop structures. This are the main motivations to adopt a conventional register file to store values that are *produced and consumed* in the *same* cluster. Hence, all intra-cluster communication takes place via a Local Register File (LRF), while inter-cluster communication takes place via one of the Communication Queue Register Files (CQRF).

A CQRF is a queue register file (as defined in Chapter 4) placed between two adjacent clusters, providing read-only access to one of them, and write-only access to the other (Figure 6.2). Sending a value from one cluster to another requires only a pair of write/read operations to the appropriate CQRF. The analytical model presented in Section 4.5 shows that the cycle time of QRFs and RFs are similar for a small number of FUs. If the machine cycle time is determined by one of those components [25] it might be feasible to assume the same latency for an operation accessing either the LRF or the CQRF, allowing flexibility to the scheduler.

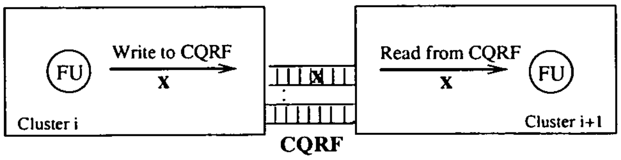


Figure 6.2: Communication between adjacent clusters using a CQRF

Each cluster of the proposed organization has two CQRFs to communicate with its two neighbouring clusters (Figure 6.3a). Instruction results can be written to the LRF or to the cluster CQRFs. Source operands can be read from the LRF and also from the CQRFs of the adjacent clusters (Figure 6.3b). The functional units obtain their operands from the LRF and the CQRFs, via the *operand multiplexing*, comprising a 3-input multiplexer per functional unit read port. We have found that the cluster configuration resulting in the lowest cycle time for either a LRF or a CQRF comprises 3 standard and 1 Copy FUs (Section 6.4.3.2). Hence this is the configuration we have used in the experiments, unless stated otherwise. Those four functional units in each cluster would require a maximum bandwidth of 5 writes and 7 reads per cycle. This bandwidth requirement may be directed at the LRF and/or one or more CQRFs. The only restriction is that in each cycle each queue can be read at most once, and written at most once. In this study we assume a perfect memory system, capable of servicing each load and store operation within the latencies set out in Table 3.1. The code generator maps the lifetimes that span a cluster boundary onto the corresponding CQRF. An important feature of the architecture and its scheduler is that nearest-neighbour communication requires no explicit instruction to communicate the value.

We have adopted a bi-directional ring topology because of the advantages of symmetry and strictly nearest-neighbour interconnects, which ensures the same latency for any communication operation. Rings also exhibit natural broadcast properties that facilitate the implementation of scheduling and partitioning algorithms, which is a crucial aspect in this kind of architecture. Extending the connectivity using other structures such as 2D-Mesh or Torus, for instance, would offer greater flexibility to the scheduler. However, that would increase the hardware complexity and silicon area, and also result in variable communication latency due to a non-symmetric structure.

The ring topology lends itself well to a single-chip implementation. Although the nearest-neighbour communications require very short wires, the furthest distance between any two clusters is approximately the width of the chip. This would

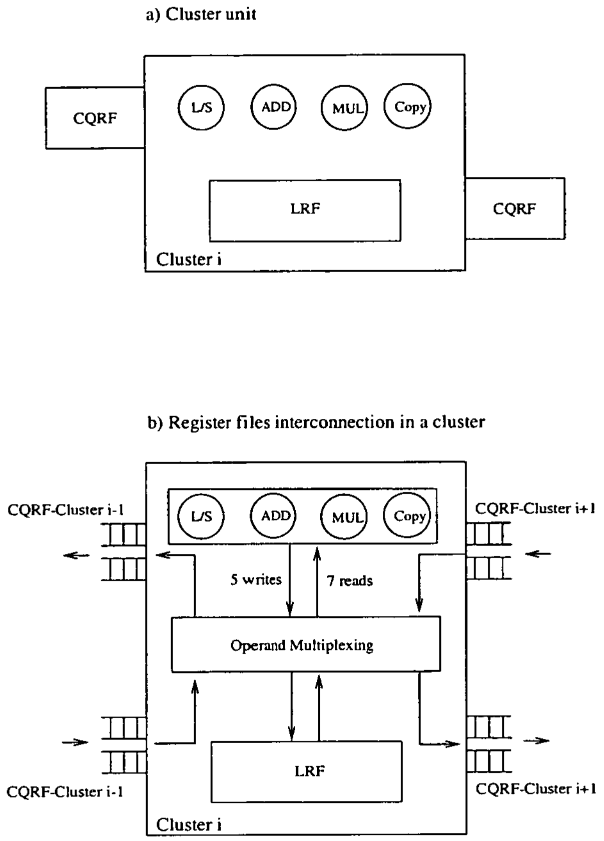


Figure 6.3: Cluster organization

make it difficult to operate with a global synchronous clock, and is where we see an additional benefit of using buffered inter-cluster communication. Each cluster could operate from its own clock, phase-locked to a master clock but potentially skewed with respect to the clocks of its neighbouring clusters. Local clock synchronization can be used to bring cluster clocks into sync with each other, or alternatively asynchronous data transfer could be used across cluster boundaries.

In spite of the distribution of functional units among clusters, the proposed architecture model still assumes a *single thread* of control. This will almost certainly involves data exchange among FUs located in distinct clusters. Compiling for a clustered architecture involves *code partitioning* in order to meet communication constraints (a bi-directional communication ring, in this case). The ideal operation assignment would result in the same II achievable for an unclustered architecture. However communication constraints may force an operation to be scheduled in a given cluster. It might happen that no available slot exists to schedule that operation in the required cluster. In this case the only alternative would be increasing the II, which will delay the loop execution. For this reason code partitioning is a crucial issue for the effectiveness of a clustered VLIW architecture. In this thesis we propose two schemes to perform this task: a simple one, described in this chapter, and a more elaborate one, shown in Chapter 7.

The remainder of this chapter presents an update to the experimental framework in order to consider the clustered architecture model. Then, a new heuristic is presented to perform code partitioning of a modulo scheduled loop. Finally, a number of experiments are reported, comparing the performance and machine resources of clustered and unclustered architectures.

6.2 Experimental Framework Update

As defined in previous chapters the experimental framework considers only unclustered VLIW machines, either using a conventional or a queue register file. In this section we extend its capabilities to consider a clustered machine model, as described in Section 6.1. In this model groups of functional units and a conventional register file (LRF) are grouped into clusters, which in turn are interconnected by means of a bidirectional ring of queue register files (CQRFs).

The use of LRFs allows the allocation of *loop invariant* values without any special scheme. Lifetimes of this type are allocated to the LRF of every cluster where they are used. Hence, lifetime duplication may occur. Therefore from now on loop invariants are also taken into account when register requirements are estimated. Loop variant lifetimes can be allocated to the LRF or to one of the CQRFs. If a value is produced and consumed in the *same* cluster, then it is allocated in the LRF. If it is consumed in one of the *adjacent* clusters, then it is allocated in the corresponding CQRF. The code partitioning algorithm must ensure the existence of a communication channel between every pair of producer/consumer operations. As will be shown later in Section 7.1.4, the *DDG* transformation to eliminate multiple-use lifetimes simplifies the partitioning task.

We shall call URF an unclustered machine using a conventional register file to perform allocation of both loop variant and invariant lifetimes. Clustered architectures will be referred as CQF, also performing register allocation for both types of lifetimes. If necessary the suffix nn is used to indicate the number of *standard* functional units, which excludes Copy FUs. Unless otherwise stated it should be assumed that each cluster comprises 3 standard FUs (1 L/S, 1 ADD, and 1 MUL) and 1 Copy FU. Thus each LRF and CQRF must provide 7 read and 5 write ports.

The machine model description now includes some new parameters:

- Number of clusters
- Cluster configuration, which is defined by the number and type of functional units, and the type of register file.
- Communication topology
- Latency of inter-cluster communication

The output parameters are the same as described in previous chapters. However, machine resources such as number of registers and queues, among others, are now estimated individually for each LRF and CQRF. Specific requirements of individual register files often differ among clusters. We let the most demanding LRF determine the capacity of all other LRFs, using a similar approach to estimate the size of CQRFs.

Finally, it should be noticed that the scheduling algorithm employed so far cannot target a clustered machine with limited connectivity. We have enhanced the IMS algorithm with new heuristics, making it capable of producing code for this kind of architecture, as described in Section 6.3.

6.3 Partitioning Heuristics

The original version of Iterative Modulo Scheduling [79] used by the experimental framework assumes that all functional units connect to a centralized register file. However this is not the case for the clustered machine model defined in Section 6.1. The limited connectivity among FUs prevents the assignment of operations to some clusters, according to definition and use of lifetimes.

If two operations with a *true data dependence* are scheduled in indirectly-connected clusters it is said that a **communication conflict** occurs. One way to address this problem is to perform code partitioning before modulo scheduling, ensuring that no communication conflict arises. This problem can be seen as a k-way graph partitioning problem [31], where the cost function to be minimized represents the *MII.* Once the partitioning is completed, the scheduling process can proceed, taking into account the assignment of operations to clusters.

We have developed an alternative approach, performing both scheduling and partitioning in a *single step* [29]. The main motivation for integrating both tasks into one single procedure is the possibility of reducing the compilation time without compromising the quality of the schedule. This should be possible because a whole step is eliminated (partitioning), while the additional complexity introduced into the other one (scheduling) is comparatively smaller than of its original tasks. The strategy adopted includes *further heuristics* to the IMS algorithm, relying on its backtracking capabilities to break dead-end states. The effect of those heuristics is to limit the space of choices to schedule an operation, as defined below:

- An operation is scheduled in a given cluster only if there are no *communication conflicts* with previously scheduled operations, i.e. the clusters to which the communicating operations belong to must be directly interconnected. We do not as yet consider the introduction of operations to transfer a value between indirectly connected clusters, implying that an operation can only send or receive data from an operation scheduled either in the same cluster or in one of the adjacent ones.
- Communication conflicts can prevent an operation from being scheduled in any of the clusters, leading to a *backtracking* process to unschedule conflicting operations. The unscheduled operations are then rescheduled, taking into account the new communication constraints arising from the new partial schedule.

These new heuristics require modifications in two of the original procedures of the IMS algorithm: *Find_Slot and Backtracking* (Section 3.3). The updated version of those functions are described in Sections 6.3.1 and 6.3.2, and are used when scheduling for the clustered machine model described in this chapter.

6.3.1 Find Slot Function for a Clustered Machine

The original procedure tries to find a valid slot to schedule an operation OP, enforcing correct schedules from a resource usage viewpoint. If a free slot cannot be found within the range $[mintime..maxtime]$, the algorithm will relax this constraint to assign a scheduling slot for OP. In this case the operation currently scheduled in the chosen slot will be unscheduled during the backtracking process. The new version, called *Find_Slot_Clustered* includes a further constraint to return a valid slot: no communication conflict should arise if operation OP is indeed scheduled in the returned slot. The scheme is described by Algorithm 6.3.1. Communication conflicts are checked between OP and all of its *scheduled predecessors* and *successors*. They should be scheduled either in the same or in an adjacent cluster to the candidate slot. The checking procedure described by Algorithm 6.2

uses a parameter called *NrClusters*, expressing the number of clusters of the machine configuration. If a valid slot cannot be found under these conditions, the algorithm will relax the constraints, choosing a slot as in the the original version. In this case, the backtracking process may also unschedule operations due to communication conflicts.

Algorithm 6.1 *Find slot for a clustered machine*

```
Find_Slot_Clustered(OP, mintime)
  /* Limit range of possible slots */
  maxtime= mintime + II -1
  currtime= mintime
  While (currtime $\leq$ maxtime) {
    /* Find a resource free slot */
    find free slot in MRT at cycle=currtime
    if (slot found)
      /* Verify if communicating operations would be located */
      /* in directly connected clusters */
      conflict= check_communication_conflict(OP,slot)
      if (conflict == 0)
        Return slot
    else
      ++currtime
  }
  /* Resource free slot not found-relax this condition */
  If (OP never scheduled)
    /* Choose a slot in the first possible cycle */
    choose slot in MRT at cycle=mintime
  else
    /* Choose a slot one cycle later than previously scheduled */
    choose slot in MRT at cycle= $OP_{previous\_slot}$ + 1
  Return slot
```

Algorithm 6.2 *Check Communication*

```
Check_Communication(OP, slot)
  /* Assuming that OP is scheduled in a slot at a given cluster, verify if */
  /* all operations sending and receiving values from OP would be */
  /* located in directly connected clusters */
  conflict= 0;
  Let $C_a$ be the cluster which slot is located in
  forall predecessor of OP {
    if (predecessor is scheduled at cluster $C_b$) {
      /* Verify the number of hops between the cluster of OP */
      /* and its predecessor, assuming a bi-directional ring */
      gap= $abs(C_a - C_b)$
      if (gap > 1) and (gap $\neq$ (NrClusters - 1) ) {
        conflict= 1
        Return conflict
      }
    }
  }
  forall successor of OP {
    if (successor is scheduled at cluster $C_c$) {
      /* Verify the number of hops between the cluster of OP */
      /* and its successor, assuming a bi-directional ring */
      gap= $abs(C_a - C_c)$
      if (gap > 1) and (gap $\neq$ (NrClusters - 1) ) {
        conflict= 1
        Return conflict
      }
    }
  }
  Return conflict
```

6.3.2 Backtracking Function for a Clustered Machine

The new version of the backtracking algorithm also unschedules operations due to communication conflicts. This will be the case if a conflict free slot could not be found by the *Find_Slot_Clustered* procedure. The scheme described in Algorithm 6.3 is a direct extension of the *Check_Communication* procedure. Every predecessor or successor of OP, scheduled in a cluster not directly connected to where OP is scheduled, is ejected from the partial schedule. These operations will then be rescheduled, possibly in another cluster, trying to avoid communication conflicts.

Algorithm 6.3 *Backtracking for a clustered machine*

```
Backtracking_Clustered(OP, slot)
  s= slot cycle
  forall successor of OP {
    If (successor is scheduled at cycle c) {
      correct= $s + \lambda_{e(OP,successor)} - (II \times \delta_{e(OP,successor)})$
      if (correct > c) {
        Unschedule(successor)
        Update MRT
        /* Return successor to the list of unscheduled operations */
        Include(List,successor)
      }
    }
  }
  /* Unschedule pred. and suc. of OP due to communication conflicts */
  Let $C_a$ be the cluster which OP is scheduled in
  forall predecessor of OP {
    if (predecessor is scheduled at cluster $C_b$) {
      gap= $abs(C_a - C_b)$
      if (gap > 1) and (gap $\neq$ (NrClusters - 1) )
        /* Clusters non-directly connected /*
        Unschedule(predecessor)
        Update MRT
        Include(List,predecessor)
      }
    }
  }
  forall successor of OP {
    if (successor is scheduled at cluster $C_c$) {
      gap= $abs(C_a - C_c)$
      if (gap > 1) and (gap $\neq$ (NrClusters - 1) )
        /* Clusters non-directly connected /*
        Unschedule(successor)
        Update MRT
        Include(List,successor)
      }
    }
  }
  forall op having a resource conflict in slot {
    Unschedule(op)
    Update MRT
    Include(List,operations)
  }
```

6.3.3 Complexity of the New Heuristics

The order of complexity of the IMS algorithm is not affected by the introduction of new heuristics, as only further constraints are added to the existing ones when choosing a valid time slot to schedule an operation. It has been shown that the empirical computational complexity of IMS is $O(N^2)$, with N representing the number of operations to be scheduled [79]. The new computation involved to check communication conflicts consists of simple comparisons between cluster identifiers. In practice the number of comparisons is small, as it involves only immediate predecessors and successors of an operation. In general there are at most two· predecessors. The number of successors may be larger, however we have noticed that in most of the cases it is less than 5. Furthermore, using the scheduling techniques proposed in this thesis, an operation can have *at most* two successors. This follows from the *DDG* transformation used to eliminate multiple-use lifetimes. It is expected that the backtracking frequency may increase in some cases, requiring a longer compilation time. Nonetheless, we have not changed the *budget* parameter limiting the number of scheduling steps for a given II. Communication conflicts may require a larger II than would be achieved if a single cluster machine had been used instead. Small or no variations at all in the II would mean that only a tolerable increase occurs in the backtracking frequency. Accordingly, we evaluate the effectiveness of the new heuristics by measuring the II increase due to code partitioning. This, and other analyses, are presented in the next section.

6.4 Experimental Results

This section presents experimental results comparing clustered and unclustered architectures. A total of 14 machine configurations were considered: 7 URF and 7 CQF models, each set of seven ranging from 3 to 21 functional units. Unclustered machine models have a centralized conventional register file (RF). Clustered machines can have between 1 and 7 clusters. Each cluster has 3 standard FUs and 1 Copy FU, 1 local register file (LRF), and 2 communication register files (CQRF). Register allocation of both loop variant and invariant lifetimes is performed for all models. Those machine configurations are summarized in Table 6.1.

Innermost loops taken from the Perfect Club Benchmark were used in the experiments (Section 3.2). Loop unrolling prior to modulo scheduling was performed according to the criteria described in Section 5.1. The presentation of results is subdivided into three main topics: partitioning effectiveness, performance, and

Machine Configurations			
Number of Clusters: 1-7			
Functional Units	URF	CQF	Single Cluster
L/S	1-7	1-7	1
ADD	1-7	1-7	1
MUL	1-7	1-7	1
Copy	-	2-7	1
Register Files	URF	CQF	Single Cluster
RF	1	-	-
LRF	-	1-7	1
CQRF	-	4-14	2

Table 6.1: Machine configurations used in experiments

machine resources. It should be noticed that a clustered machine composed of a single cluster is identical to the equivalent unclustered version. Thus, it does not include Copy FUs, CQRFs, and does not require transformations in the *DDG*.

6.4.1 Partitioning Effectiveness

In this section we evaluate the effectiveness of the partitioning algorithm investigating two issues: increase in the II, and distribution of operations among clusters.

6.4.1.1 Overhead on the II Due to Partitioning

An ideal algorithm would always produce a valid schedule with minimum II. As reported in [79], and confirmed by our experiments, the IMS algorithm achieves this objective for a high fraction of loops when targeting a single cluster machine. The *MII* should ideally be the same for a single- or multi-cluster machine, assuming that both have the same number of standard FUs. Although not always possible, due to communication constraints, a good scheduling/partitioning scheme would minimize the performance degradation incurred.

We have compared the II of schedules for clustered and unclustered machines. The data in Figure 6.4 shows the fraction of loops with an increase in the II due to partitioning. The overhead observed for machines with 6 and 9 FUs is only due to the introduction of copy operations in the *DDG*. These machines have 2 or 3 clusters only, which makes the bi-directional communication ring equivalent to a cross-bar. Thus the partitioning process is not constrained by the communication system. Only a slightly larger overhead occurs for 12 FUs, showing the algorithm is very effective when scheduling for 4 clusters. A more significant increase is observed for 15 FUs, although it is still possible to schedule around 85% within the same II. Larger machine configurations are more demanding for the partitioning scheme, which is reflected by an accentuated increase in the fraction of loops having some performance degradation. The low frequency of loops requiring a higher II for configurations up to 5 clusters also shows that the extra *backtracking* is tolerable, when it occurs.

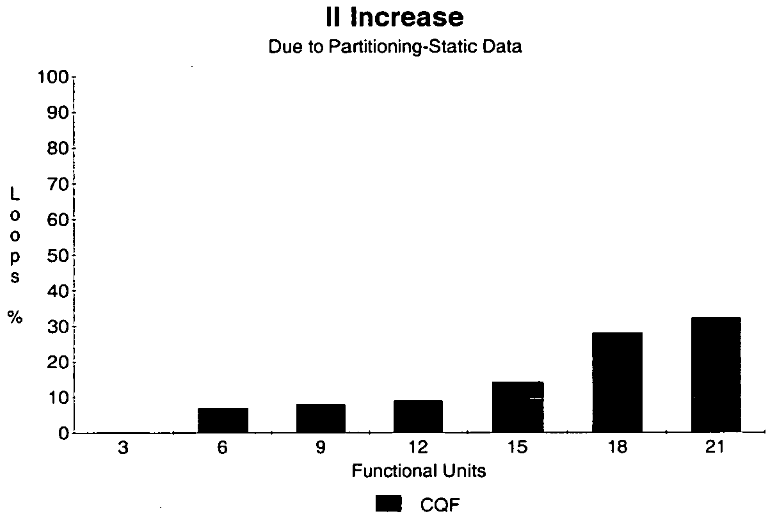


Figure 6.4: Loops with a larger II due to partitioning

We have found that in many cases the II increase is of *one cycle* only. The impact caused by this may be minimal, but may also be significant, depending on the II value. The results presented provide an insight into the partitioning effectiveness, however they cannot show the actual performance degradation. The effect of increasing the II and the *SC* can be better quantified by analysing the execution time and the IPC, which is shown in Section 6.4.2.

6.4.1.2 Communication Distance

We define the **communication distance** to be the number of cluster boundaries each value crosses on its way from producer to consumer. A distance of zero refers to both operations being executed in the same cluster. A distance of one refers to the consumer operation being executed in one of the adjacent clusters to the producer. In these cases the communication process involves only access to the LRF or the CQRF, respectively. The data in Figure 6.5 shows the distribution of communication distances for several machine configurations, measured over all lifetimes created. Local communication occurs most frequently if only 2 clusters are used, shown by the higher frequency of distance 0. Equivalent frequencies of distances 0 and 1 are observed for 3 clusters. If 4 or more clusters are used the frequency of distance 1 is approximately twice as high as the frequency of distance zero. Hence, it can be said, empirically, that the probability of a consumer operation being located in the producer cluster is half of the probability of being located in either one of the adjacent clusters. This shows the algorithm manages to balance the distribution of operations among clusters, avoiding concentrating too many operations in the same cluster. If this was the case, over-utilization of some clusters, and sub-utilization of others would occur, possibly increasing the II.

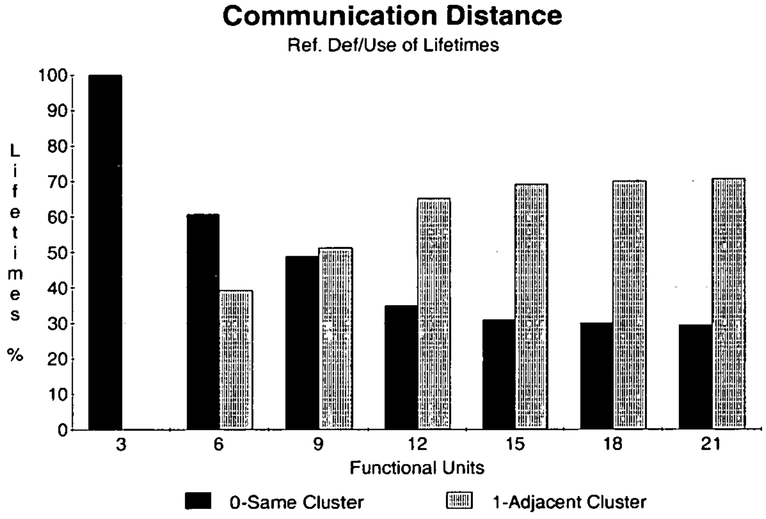


Figure 6.5: Communication distance after partitioning

6.4.2 Performance Analysis

The actual impact of the partitioning process can be quantified by comparing the execution time of both architecture models, clustered and unclustered. The data in Figure 6.6 compares the execution time of the full benchmark set, assuming a fixed cycle time for all configurations. The results are relativized using as baseline the execution time of the corresponding class of loops in a URF03 machine. The number of cycles required to execute all loops is 3.6×10^{10}, 3.4×10^{10}, and 1.3×10^{10}, for classes 1, 2, and 3, respectively.

Small performance degradation with clustering was observed for machine models up to 15 FUs (5 clusters). However for 18 FUs the number of cycles does not decrease by a significant factor. Furthermore, it *increases* if 21 FUs are used. This suggests that the proposed partitioning scheme is effective for at most 5 clusters. Although no increase in the number of cycles was observed if only loops without recurrences are considered, some performance degradation was observed for 6 and 7 cluster machines (Figure 6.7). However it should be noticed that very small differences were observed up to 5 clusters.

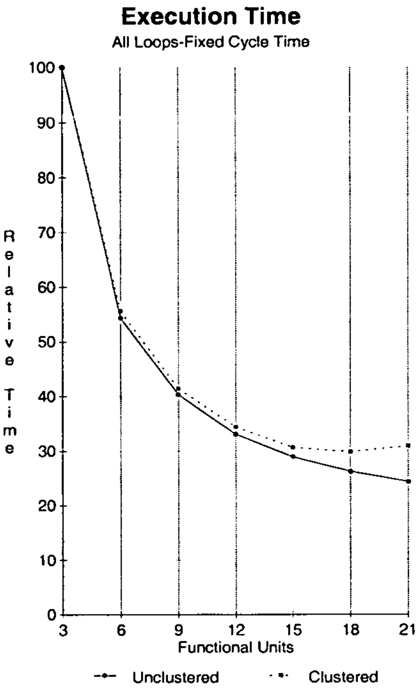


Figure 6.6: Number of cycles-Class 1

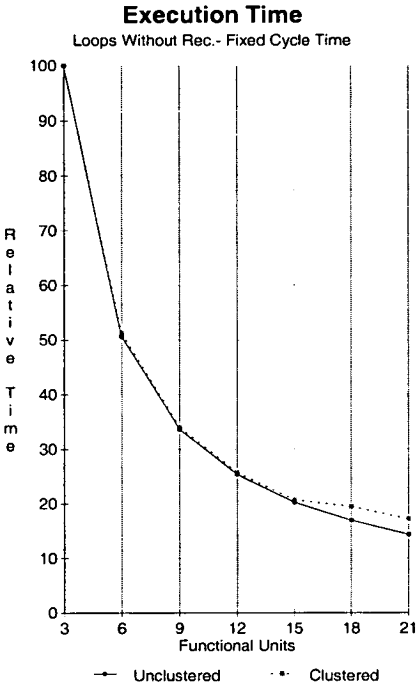


Figure 6.7: Number of cycles-Class 3

Similar conclusions can be drawn by analysing the values of $IPC_{dynamic}$ for both sets of loops, as seen in Figures 6.8 and 6.9, respectively. Very small performance degradation occurs for machines up to 5 clusters. If only loops without recurrences are considered, clustered and unclustered machines present virtually the same performance. However, it can be seen once again that the algorithm is not appropriate for 6 or more clusters.

We have concluded that the quality of the schedules produced by the integrated scheduling/partitioning algorithm is good for machine models up to 5 clusters. But we have also shown in Section 5.3.2.5 that scheduling quality can only translate into actual performance if accompanied by a short machine cycle time. It is expected that the small register files required by clustered machines

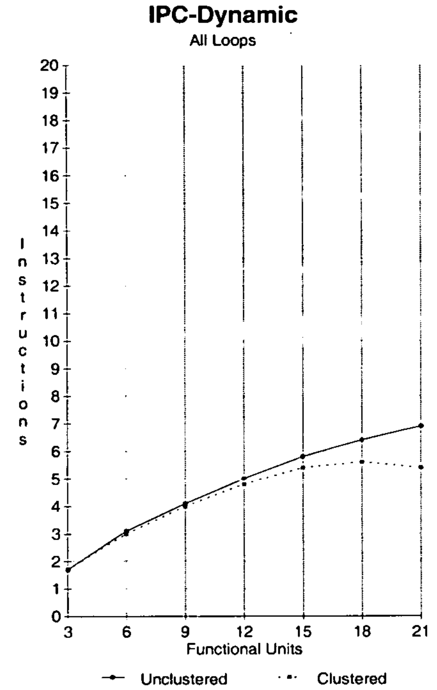


Figure 6.8: IPC Dynamic-Class 1

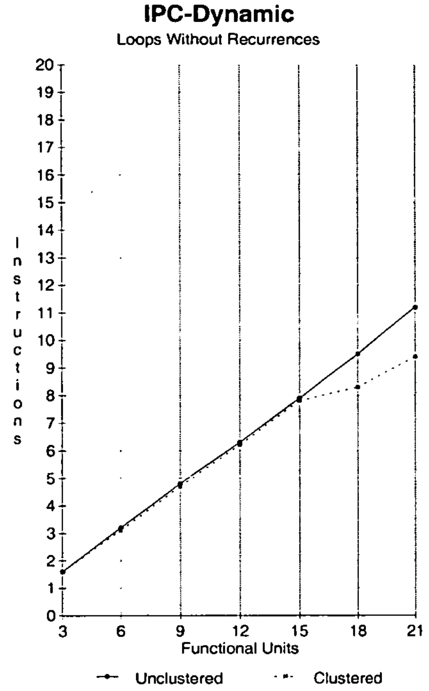


Figure 6.9: IPC Dynamic-Class 3

will minimize this parameter. A more detailed analysis of this issue is presented in Section 6.4.3.2.

6.4.3 Machine Resources Analysis

In this section we compare the machine resources required to achieve the performance levels reported in Section 6.4.2. We focus on the silicon area and cycle time analysis, which is used to estimate the actual performance of both machine models. As was done in the last chapter, all data refer to dynamic measurements, accounting for the loops responsible for 99% of the total execution time of the benchmark.

6.4.3.1 Register File Area

The hardware model presented in Section 4.5 was used to compare the silicon area of the register files required to implement the architecture models described in this chapter. As previously defined, MaxLive determines the number of registers in a conventional RF. As seen in Table 6.2, the register requirements for URF machines are slightly higher than for URV models (Table 5.2), which is due to the allocation of loop invariants.

URF Register File Parameters			
	Capacity	Ports	
FUs	Registers	Read	Write
3	71	6	3
6	69	12	6
9	102	18	9
12	108	24	12
15	137	30	15
18	152	36	18
21	176	42	21

Table 6.2: URF register requirements

In a clustered machine the total area occupied by register files depends on the area of the LRF and the CQRFs of each cluster. The capacity of each LRF is determined by the MaxLive value in the corresponding cluster. A similar approach to the one employed in Section 5.3.2.4 was used to estimate the size of each CQRF: the size of the name space determines the number of queues, and the maximum length of *any* of them determines the length of *all other* queues. As expected, we have found that register requirements are not homogeneous throughout the clusters. We have used the *highest* requirements found to determine the parameters of all other register files. Although this approach does not minimize the use of machine resources, it allows some flexibility to the scheduling/partitioning algorithm. Nonetheless we have not found large differences among the register requirements of all clusters. The exact value of the parameters used to calculate the area of *each* LRF and CQRF of a clustered machine is reported in Table 6.3. It is assumed that each register location is *64 bits* wide. The columns labelled *No* indicate the number of LRFs and CQRFs required by each machine configuration.

The chart in Figure 6.10 shows the *total* silicon area required to implement the register files of each machine configuration. All figures referring to silicon area are presented in λ^2 units. The area used by register files in an unclustered machine of 6 FUs is smaller than required by the equivalent clustered machine. Similar space is occupied by configurations of 9 FUs. Clustered machines have a clear advantage regarding this parameter for 12 of more FUs, a difference that tends to increase as more FUs are used. This is possible because the required capacity of LRFs and CQRFs increases by a much smaller factor than it does for the RFs of an unclustered machine, as shown in Tables 6.2 and 6.3.

CQF Register File Parameters									
	LRF				CQRF				
	No	Capacity	Ports		No	Capacity		Ports	
FUs		Registers	Read	Write		Queues	Length	Read	Write
3	1	71	6	3	0	-	-	-	-
6	2	63	7	5	4	11	8	7	5
9	3	45	7	5	6	9	9	7	5
12	4	38	7	5	8	9	8	7	5
15	5	34	7	5	10	9	8	7	5
18	6	50	7	5	12	11	13	7	5
21	7	47	7	5	14	13	13	7	5

Table 6.3: CQF register requirements

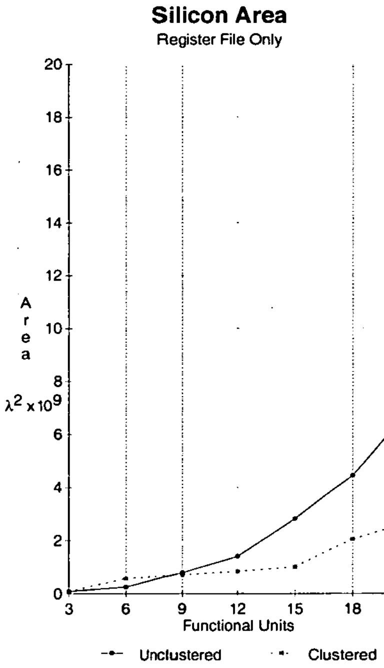


Figure 6.10: Total silicon area of register files

6.4.3.2 Register File Cycle Time

The cycle time of register files can also be estimated using the analytical model presented in Section 4.5. The data in Figure 6.11 show the cycle time of RFs, LRFs, and CQRFs across distinct numbers of functional units. It can be seen that LRFs and CQRFs have *similar* cycle times for all machine configurations, which is around 6 ns. This confirms our expectations that the inter-cluster communication mechanism using CQRFs does not result in significant delays. It is important to notice that the cycle time of LRFs and CQRFs hardly increases for larger configurations, which is an important feature in terms of scalability. This is not the case for the cycle time of centralized RFs, which has a value always higher than LRFs and CQRFs, growing approximately linearly to the number of FUs. Although LRFs and CQRFs require extra access ports to support Copy operations, they are still more efficient than a centralized RF supporting all standard functional units.

We have estimated the cycle time of a clustered machine based on the cycle time of LRF and CQRFs, whichever is higher. URF machines have the cycle time determined by the RF [25]. The data in Figure 6.12 shows that the cycle time of a clustered machine is *always lower* than the corresponding unclustered version.

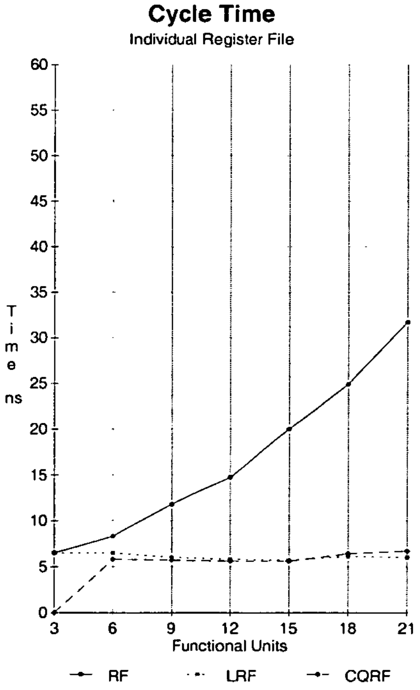


Figure 6.11: Cycle time of reg. files

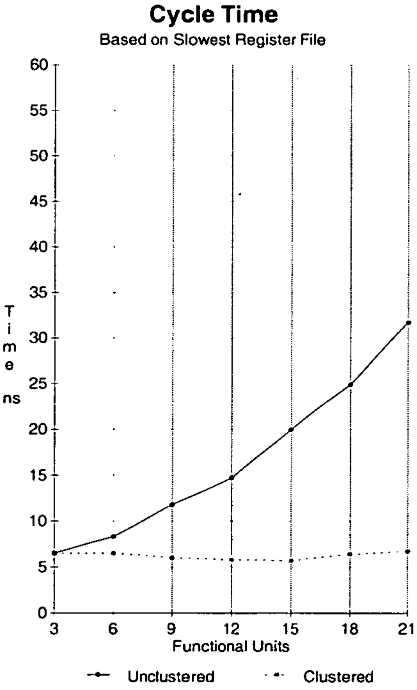


Figure 6.12: Machine cycle time

The performance results reported in Section 6.4.2 assume a fixed cycle time for all machine configurations. Although an unrealistic assumption, it is valid to evaluate the scheduling/partitioning effectiveness. However, the actual cycle time should be taken into account to have an insight on the real machine performance.

We have used the cycle time values calculated for each configuration to weight those execution times. The results are normalized using as baseline the total execution time of the corresponding set of loops in a URF03 machine, measured in ns. In this case, the execution time for all loops of classes 1, 2 and 3 is 2.3×10^{11} ns, 2.2×10^{11} ns, and 8.5×10^{10} ns, respectively.

The chart in Figure 6.13 shows the total execution time of all benchmark loops. Real performance gains were observed for machine configurations up to 15 FUs. This is not the case for unclustered machines, which only allow performance gains up to 6 FUs. Similar results were observed if only loops without recurrences are considered (Figure 6.14). The only difference is that unclustered machines still allow a small performance improvement up to 15 FUs, although the degree of improvement is small.

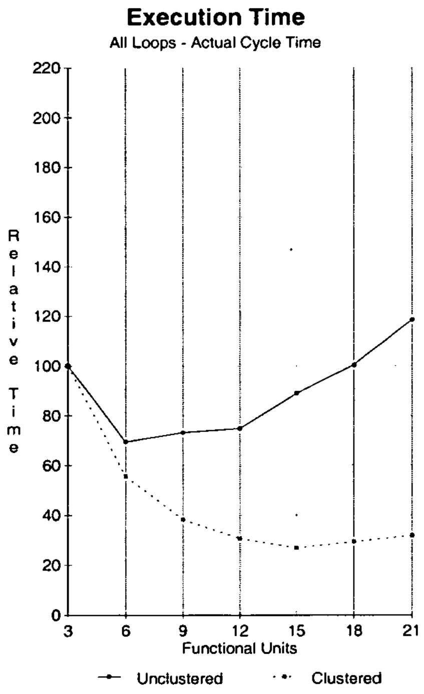


Figure 6.13: Execution time-Class 1

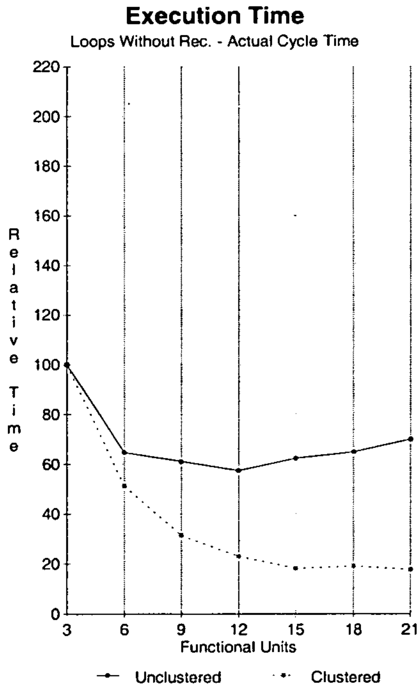


Figure 6.14: Execution time-Class 3

Those results suggest that clustered machines allow *real* performance gains for configurations up to 5 clusters (15 FUs). No improvements were observed for larger configurations, which is due to limitations in the partitioning scheme. For the configurations studied we have found that clustered machines have always better performance than unclustered organizations.

6.4.4 Summary of Results and Conclusions

The main conclusions obtained from the experimental analysis presented in this section are summarized in the following list:

- The integrated scheduling/partitioning scheme proposed is effective for machine configurations up to 5 clusters. The resulting overhead on execution time and IPC values is minimal for these configurations.

- The register file requirements of individual clusters are reasonable, only growing by a small factor as the machine model scales up. For this reason the total silicon area of a clustered machine is smaller than the equivalent unclustered organization for 9 or more functional units.

- The cycle time of LRFs and CQRFs are similar for all of the configurations studied. Hence, the proposed communication system using CQRFs between adjacent clusters does not cause any significant extra delay.

- Small register file requirements of individual clusters make the cycle time of a clustered machine always shorter than the equivalent unclustered organization. Furthermore, there is no significant variation in the cycle time value across the range of configurations considered.

- The short cycle time of a clustered machine allows performance gains obtained from aggressive ILP scheduling to be translated into a shorter execution time. In general this is not the case of unclustered machines.

- The techniques presented in this chapter have shown the feasibility of a VLIW clustered architecture using queue register files as an inter-cluster communication mechanism. The proposed model presents distinct advantages in terms of scalability, enabling the use of additional functional units without a significant penalty in the machine cycle time. Actual performance gains were observed for machine models up to 5 clusters (15 FUs), which was achieved using the single scheduling/partitioning procedure. However the proposed scheme is not effective for 6 or more clusters. Performance improvements beyond that level would require further enhancements to the algorithm, or even a new approach to the problem.

Chapter 7

Distributed Modulo Scheduling

A clustered machine model was proposed in the last chapter in order to reduce the size and number of ports of individual register files, and thus improve the machine cycle time. A key issue in this architecture model is the partitioning of operations among clusters, which must conform to the communication system. The scheme proposed in Section 6.3 is a variation of the original IMS algorithm. It performs both scheduling and partitioning in a single step, ensuring that no communication conflicts are present in the final schedule. Experimental results have shown the strategy is effective for machine models up to 5 clusters. In those cases the performance degradation due to partitioning is acceptable. Furthermore, the required register files result in short access times, which allows effective performance gains through successive machine upgrades.

However, the proposed scheduling/partitioning scheme is not effective when dealing with machine models of 6 or more clusters. The communication system allows data dependent operations to be scheduled either in the same or in adjacent clusters. Intuitively one should expect an increasing difficulty in balancing the distribution of operations among clusters for larger machine configurations. In this chapter we present a new algorithm, called Distributed Modulo Scheduling (DMS), to deal with the problem [30]. DMS is derived from the modified version of IMS described in Section 6.3, targeting the same machine model and compilation environment. It can also be classified as a *single step* scheduling/partitioning algorithm. However, the new capabilities allow it to produce efficient schedules for a larger number of clusters. This is possible because the algorithm can schedule *intermediate* copy operations between a pair of producer/consumer operations scheduled in *non-adjacent* clusters. The DMS algorithm, and the corresponding experimental analyses, are presented in the next sections.

7.1 Overview of the DMS Algorithm

As already discussed, a clustered machine model introduces new communication constraints to the scheduling algorithm, in addition to machine and dependence constraints.

We have used the IMS algorithm as a basic structure to develop a scheme able to deal with distributed functional units and register files. As seen in Figure 7.1, IMS has one basic strategy to schedule an operation OP, which is described in detail in Section 3.3. On the other hand, the DMS algorithm has three basic strategies: First it tries to schedule OP in such a way that no communication conflict arises with scheduled predecessors and successors of OP. If that is not possible, it tries to insert *move* operations between OP and its scheduled predecessors, using a structure we call a chain. If the partial schedule has enough machine resources to schedule the required chains, OP can be scheduled. Other operations may be unscheduled due to machine and dependence conflicts. If the schedule of chains cannot be completed, OP is simply scheduled in a given slot and all conflicting operations are unscheduled.

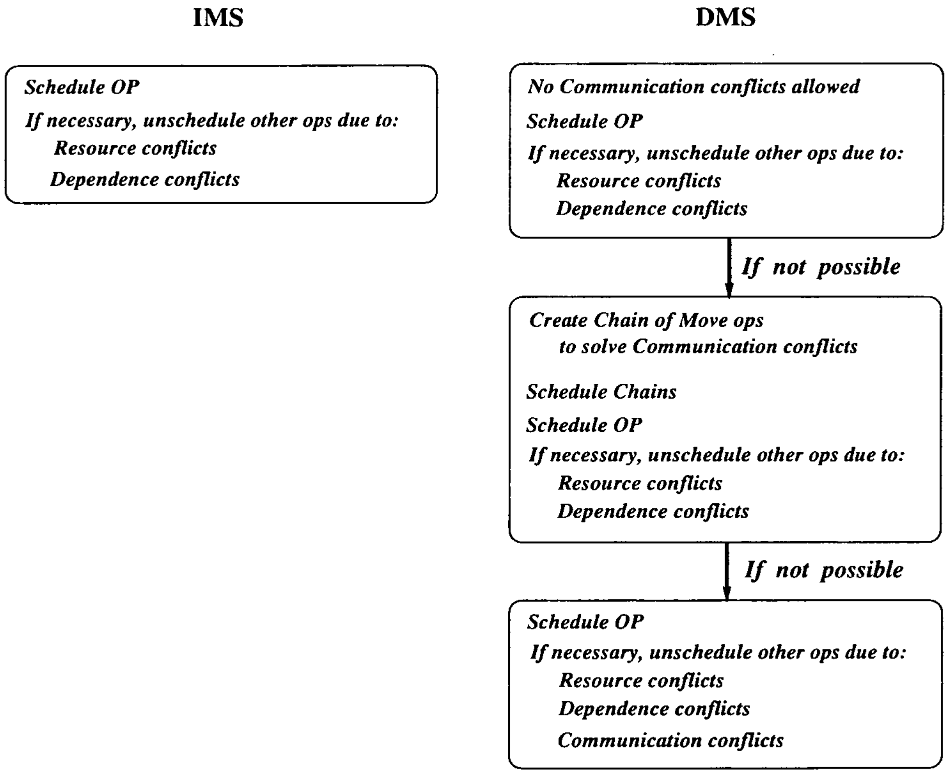


Figure 7.1: Overview of IMS and DMS algorithms

A high level description of DMS is shown below (Algorithm 7.1), with further details and related algorithms presented in the next sections.

Algorithm 7.1 *Distributed Modulo Scheduling*

```
DMS(II)
  budget= 3 × (No. operations in DDG)
  Create_Priority_List(List)
  While (List not empty) and (budget > 0) {
    Get(List,OP)
    /* Earliest start time for OP, according to scheduled predecessors */
    mintime= Earliest_time(OP)
    /* Select a valid slot, without communication conflicts */
    slot= Find_Slot_Clustered(OP, mintime)
    If (valid slot) {
      /* Unschedule operations due to resource and dependence conflicts */
      Backtracking_Clustered(OP, slot)
      Schedule(OP, slot)
    }
    else { /* Chains required-select cluster for OP according to set of chains */
      Cluster_OP= Create_Chains(Chain_List)
      If (Cluster_OP ≠ 0) { /* Valid set of chains found */
        Schedule_Chains(Chain_List)
        /* Find slot for OP at the chosen cluster */
        mintime= Earliest_time(OP)
        slot= Find_Slot_DMS(OP, mintime, Cluster_OP)
        /* Unschedule operations due to resource and dependence conflicts */
        Backtracking_Clustered(OP, slot)
        Schedule(OP, slot)
      }
      else { /* Select a valid slot, ignoring communication conflicts */
        slot= Find_Slot(OP, mintime)
        /* Unschedule operations due to resource, dependence, */
        /* and communication conflicts */
        Backtracking_Clustered(OP, slot)
        Schedule(OP, slot)
      }
    }
  }
  Update MRT; Remove(List,OP); budget= budget-1
  If (List is empty)
    Return 1
  If (budget == 0)
    Return 0
```

7.1.1 Chains of Move Operations

The first part of the DMS algorithm avoids communication conflicts, scheduling pairs of producer/consumer operations in the same or adjacent clusters. However we have found that it is increasingly difficult to do that having 5 or more clusters. That has led us to consider the use of move operations between data-dependent operations located in non-adjacent clusters. All discussions and algorithms regarding the introduction of move operations assume that the topology of the communication system among clusters is a bi-directional ring (Section 6.1). It is also assumed that the machine model has N clusters, sequentially identified by an integer in the range [1..N].

We define a **chain** as a string of move operations scheduled in the clusters between OP and one of its predecessors. A **move** operation simply reads one value from a register file, and writes it back to another one. Thus, if it is scheduled in the required cluster, it is possible to move operands between a pair of producer/consumer operations located in non-directly connected clusters.

Given a candidate cluster to schedule OP, and the cluster of its predecessor, there are two possibilities to create a chain, each of them following *opposite* directions (Figure 7.2). The bi-directional ring of queues used to connect the clusters allows this flexibility.

7.1.1.1 Creating a Chain

Initially every cluster can be considered to schedule OP. Depending on the *candidate* cluster, OP can require *at most* two chains, as it can have no more than two scheduled predecessors. As discussed above, for each candidate cluster there are two possibilities to create a chain. Thus, at most four possible chains should be considered in each candidate cluster. This ensures that the number of options to be investigated is finite and relatively small, scaling linearly with the number of clusters. The scheme described in Algorithm 7.2 considers all clusters to schedule OP. In the first attempt only clusters having a free slot for OP are considered, which is verified with a call to the function *Find_Slot_DMS*. If a valid slot cannot be found, the constraint is relaxed, and thus every cluster is considered, which might generate backtracking. All possible chains are evaluated before choosing the best one, according to some criteria that we have defined. The chosen set of chains defines the cluster in which OP will be scheduled *(Cluster_OP)*.

A crucial aspect in the implementation of these algorithms are the data structures used to keep track of machine resource usage. Although we have implemented a simple scheme in our experimental framework, more sophisticated tech-

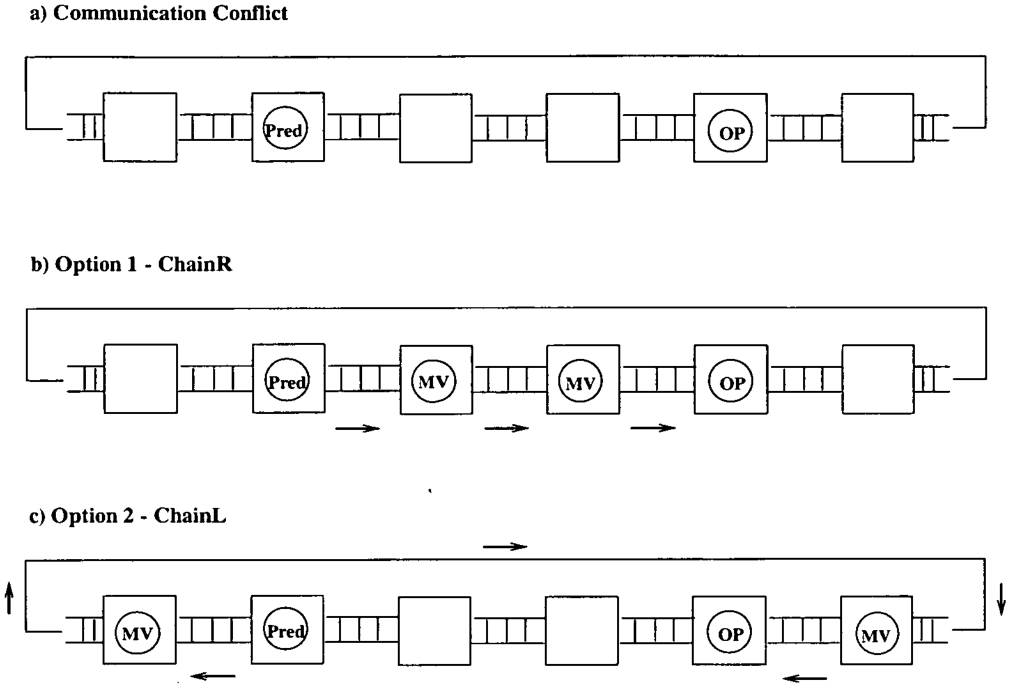


Figure 7.2: Options to create a chain

niques exist to model machine descriptions. As an example, the scheme proposed in [42] allows efficient low-level representations to be derived from a high-level language, which is a desirable feature for compiler writers. The low-level representation, based on AND/OR trees, is obtained by applying a number of transformations on the high-level description. Experiments have shown significant reductions in both memory requirements and number of resource checks per scheduling attempt.

Algorithm 7.2 *Create Chains*

```
Create_Chains(OP, Chain_List)
  /* Investigate the possibility of creating the necessary chains assuming */
  /* that OP can be scheduled in any cluster with a resource free slot */
  forall cluster C {
    /* search for a resource free slot for OP */
    slot= Find_Slot_DMS(OP, C)
    If (free slot found) {
      Candidate_Chains(OP, C, Candidate_List[C])
  }

  /* Define a cluster to schedule OP, according to the chosen set of chains */
  Cluster_OP= 0
  Chain_List= void
  forall cluster C {
    If Better_Chain(Candidate_List[C], Chain_List) {
      Chain_List= Candidate_List[C]
      Cluster_OP= C
    }
  }

  If (Cluster_OP $\neq$ 0)
    Return Cluster_OP
  else {
    /* Not possible to schedule OP in a resource free slot-relax the condition */
    forall cluster C {
      Candidate_Chains(OP, C, Candidate_List[C])
    forall cluster C {
      If Better_Chain(Candidate_List[C], Chain_List) {
        Chain_List= Candidate_List[C]
        Cluster_OP= C
      }
    }
    Return Cluster_OP
  }
```

7.1.1.2 Candidate Chains

For each predecessor *pred* of an operation *OP*, two possible chains can be built, each of then starting from *pred* and following opposite directions. Those chains are called *ChainR* (right-hand side), and *ChainL* (left-hand side). The scheme described in Algorithm 7.3 build, for each predecessor of OP, the corresponding ChainR and ChainL, assuming that OP will be scheduled in cluster C. If the pair *pred/OP* is set to be scheduled without communication conflicts, no chain is necessary. This can be verified by calculating the *gap* between both clusters.

Algorithm 7.3 *Candidate Chains*

```
Candidate_Chains(OP, C, Candidate_List)
  /* Create a set of chains connecting OP with all of its predecessors */
  /* Investigate both paths for each chain, based on the bi-directional ring */
  Candidate_List= void
  Forall scheduled predecessor of OP {
    $C_p$ = (Predecessor cluster)
    gap= abs(C - $C_p$)
    If (gap > 1) and (gap $\neq$ (NrClusters -1)) {
      /* Chain following the right direction, from the predecessor to OP */
      ChainR= Set of move ops between (Predecessor $\rightarrow$ OP)
      /* Chain following the left direction, from the predecessor to OP */
      ChainL= Set of move ops between (OP $\leftarrow$ Predecessor)
      /* Decide which is the best of both chains, and also if it is valid */
      /* A valid chain implies in free slots to schedule move operations */
      If Better_Chain(ChainL, ChainR) {
        If (Valid_Chain(ChainL))
          Include(Candidate_List, ChainL)
      }
      else {
        If Valid_Chain(ChainR)
          Include(Candidate_List, ChainR)
      }
    }
  }
```

7.1.1.3 Choosing the Best Chain

The process of choosing the best option to schedule a chain depends basically on verifying the current state of machine resource usage. First of all, a set of chains has to be valid to be considered for scheduling. A *set of chains* consists of all chains that would be required to schedule OP in a given cluster C. A set of chains is said to be *valid* if the partial schedule has enough free *resource free* slots to schedule *all* of its move operations. The proposed scheme *does not* unschedule operations to release slots for a new chain. During the algorithm development we have found this alternative would increase the backtracking frequency without resulting in actual benefits. Verifying if a set of chains is valid can be done using Algorithm 7.4.

Algorithm 7.4 *Valid Chain*

```
Valid_Chain(Chain_Set)
  /* Verify if there are free slots to schedule all move operations */
  /* belonging to a set of chains */
  Forall Move ops $\in$ Chain_Set
    Assume they are scheduled in the respective cluster

  /* Minimum number of slots for move operations that */
  /* would be left in any cluster */
  bound_free_slots= $\infty$
  Forall cluster C {
    free_slots= (No. remaining Move slots in cluster C)
    If (free_slots < bound_free_slots)
      bound_free_slots= free_slots
  }
  If (bound_free_slots $\geq$ 0)
    Return 1
  else Return 0
```

It might happen that more than one valid set of chains exists to address a given communication conflict. In this case the best option is selected according to the following criteria:

1. Choose the set of chains that, once scheduled, maximizes the number of free slots left available for move operations in *any* cluster.

2. If two or more sets of chains are still equivalent regarding the above condition, choose the one composed by the smallest number of move operations.

The first of these conditions is intended to facilitate the scheduling of another eventually required chain. The second one aims to minimize the number of move operations used. We have found that the DMS algorithm is more sensitive to the first one, as it reduces the frequency of backtracking. Thus the first condition has been assigned a higher priority than the second one. A scheme to compare two sets of chains according to these criteria is described by Algorithm 7.5.

Algorithm 7.5 *Better Chain*

```
Better_Chain(Chain_Set1, Chain_Set2)
  /* Identify the best one between two sets of chains based */
  /* on machine resources to schedule move operations */
  total_mv1= 0
  /* Total number of move operations required by set 1 */
  Forall Move ops ∈ Chain_Set1 {
    Assume they are scheduled in the respective cluster
    total_mv1++
  }
  /* Minimum number of free move slots left by set 1 */
  bound_free_slots1= ∞
  Forall cluster C {
    free_slots= (No. remaining Move slots in cluster C)
    If (free_slots < bound_free_slots1)
      bound_free_slots1= free_slots
  }

  total_mv2= 0
  /* Total number of move operations required by set 2 */
  Forall Move ops ∈ Chain_Set2 {
    Assume they are scheduled in the respective cluster
    total_mv2++
  }
  bound_free_slots2= ∞
  Forall cluster C {
    free_slots= (No. remaining Move slots in cluster C)
    If (free_slots < bound_free_slots2)
      bound_free_slots2= free_slots
  }

  /* Choose the best set based on the above parameters */
  If (bound_free_slots1 > bound_free_slots2)
    Return 1
  else {
    If (bound_free_slots1 = bound_free_slots2) {
      If (total_mv1 < total_mv2)
        Return 1
      else Return 0
    }
    else Return 0
  }
```

7.1.2 Scheduling Chains and their Consumer Operations

Once a valid set of chains is chosen, it can be scheduled without concern about finding a free slot for each move operation. A valid chain implies that the availability of machine resources has been verified before. However the data dependence graph must be modified to include the new move operations, and also to update data dependencies. Inserting a chain between a pair of producer/consumer operations implies that they are not directly dependent any more. The first move operation is directly dependent on the producer operation. Thus, the attributes of this dependence are the same as the ones of the original dependence. All other move operations are successively dependent on the previous one. The original consumer operation is dependent on the last move operation of the chain. These *DDG* transformations are illustrated in Figure 7.3.

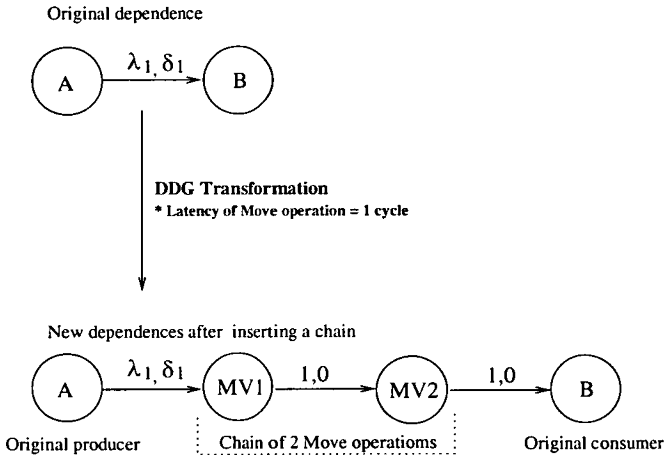


Figure 7.3: DDG transformation to insert a chain

The first step required to schedule a set of chains is to update the *DDG*, as described above. Then move operations are sequentially scheduled, starting from the first one after the producer operation. This ordering must be enforced in order to determine the correct scheduling time of each of them. That ensures correctness from the point of view of dependence constraints. As already discussed, resource constraints should not arise, thus a free slot should always be available to schedule a move operation in the proper cluster. A high level description of this procedure is shown by Algorithm 7.6.

Algorithm 7.6 *Schedule Chains*

```
Schedule_Chains(Chain_List)
  Forall chain ∈ Chain_List {
    Forall move ∈ chain {
      /* Update DDG with move operations */
      Include(DDG, move)
      /* Move operation is scheduled according to immediate successor */
      mintime= Earliest_time(move)
      /* A resource free slot for move should always be found */
      cluster= (cluster which move must be scheduled in)
      slot= Find_Slot_DMS(move, mintime, cluster)
      Schedule(move, slot)
      Update MRT
    }
  }
```

A modified procedure, called *Find_Slot_DMS*, is used to find a free slot to schedule a move operation. The only difference between this function and the version used by the IMS algorithm is that the returned slot is located in a cluster specified beforehand. This procedure, described in Algorithm 7.7, is also used to find a slot to schedule move operations. In this case the slot returned must be in a cluster as defined by the procedure *Create_Chains*.

Algorithm 7.7 *Find Slot DMS*

```
Find_Slot_DMS(OP, mintime, cluster)
  maxtime= mintime + II -1
  currtime= mintime
  While (currtime ≤ maxtime) {
    find free slot in MRT belonging to cluster at cycle=currtime
    if (slot found)
      Return slot
    else
      ++currtime
  }
  If (OP never scheduled)
    choose slot in MRT belonging to cluster at cycle=mintime
  else
    choose slot in MRT at cycle=previous_slot(OP) + 1
  Return slot
```

7.1.3 Unscheduling Operations

Special attention must be paid to the process of unscheduling operations. It might happen that an operation being ejected from the partial schedule is related to a chain. In this case it may be necessary to unschedule other operations in order to prevent communication conflicts among the remaining scheduled operations. Distinct actions must be taken, according to the type of the operation being unscheduled, which can be one of the following:

1. *The original producer of a chain:* Unschedule the operation and *all* chains starting from itself. Unschedule the *consumer* operation at the end of each of those chains. Unschedule all chains leading to this consumer operation.

2. *A move operation:* Unschedule *all* move operations of the corresponding chain. Unschedule the *consumer* operation at the end of the chain. Unschedule all chains leading to this consumer operation.

3. *The original consumer of a chain:* Unschedule the operation. Unschedule all chains leading to itself.

4. *Other types of operations:* Unschedule the operation only.

So far both procedures employed to perform backtracking use a simple function called *Unschedule* to eject an operation from the partial schedule. This function simply unschedules the operation specified. However, the use of move operations may require other operations to be unscheduled, as discussed above. We have defined a new function called *Unschedule_DMS*, capable of unscheduling all required operations, as shown in Algorithm 7.8. If a chain of move operations is unscheduled, the *DDG* must be updated, restoring the original data dependencies between the original producer/consumer pair. Once unscheduled, the chain must be removed from the *DDG*, as it is not known if it will be necessary again. Hence, an updated version of *Backtracking_Clustered* must be used by the DMS algorithm. This new version employs the routine *Unschedule_DMS* instead of *Unschedule*.

Algorithm 7.8 *Unscheduling of operations for DMS*

```
Unschedule_DMS(op)
  Unschedule(op)
  /* Verify if op is directly on indirectly dependent on a chain */
  /* If so, unschedule all involved operations */
  If (op = original producer of a chain) {
    Forall chain starting at op {
      Forall move ∈ chain
        Unschedule(move)
      consumer= operation at the end of chain
      Unschedule(consumer)
      Forall chain2 ending at consumer {
        Forall move ∈ chain2
          Unschedule(move)
      }
    }
  }

  If (op = move operation ∈ chain) {
    Forall move ∈ chain
      Unschedule(move)
    consumer= operation at the end of chain
    Unschedule(consumer)
    Forall chain2 ending at consumer {
      Forall move ∈ chain2
        Unschedule(move)
    }
  }

  Unschedule(op)
  If (op = original consumer of a chain) {
    Forall chain ending at op {
      Forall move ∈ chain
        Unschedule(move)
    }
  }
```

7.1.4 Complexity of DMS

The basic structure of the DMS and IMS algorithms is similar. Although the worst-case complexity of IMS is exponential in N, the empirical computational complexity has been estimated as $O(N^2)$ [79], with N representing the number of operations to be scheduled. IMS verifies a set of conditions before scheduling an operation, which may lead to other operations being unscheduled. DMS has to verify a more complex set of conditions, possibly unscheduling a larger number of operations. However these conditions are not dependent on N, but rather in the number of clusters and predecessors of the operation being scheduled. The extra computational cost involved in the verification of a more complex set of conditions is fixed, consisting of numeric comparisons among a finite number of elements, as shown in Section 7.1.1.1.

It is expected that the additional constraints used by DMS could increase the number of unscheduled operations. However, we have found through experimental analysis that the overhead on the II due to the partitioning performed by DMS is tolerable in most of the cases (Section 7.3.1). Those results suggest that on average the backtracking frequency of IMS and DMS is similar for machine configurations up to 8 clusters. Hence, although we have not performed a formal analysis on this issue, the experimental results suggest that the empirical complexity of DMS is also $O(N^2)$ for these machine configurations.

When the backtracking frequency does increase, it is due to an insufficient number of slots to schedule the required move operations, a situation occurring most often for machines with a large number of clusters. Increasing the number of clusters complicates the partitioning process for two reasons:

- Each additional cluster stretches the maximum possible distance between a pair of producer/consumer operations by 1. Thus, extra move operations are necessary to fill the gap.
- Each additional cluster increases the number of available functional units. This results in a smaller II if the loop being scheduled is resource constrained. A smaller initiation interval implies less free slots to schedule a move operation in any single cluster, reducing the chances of successfully scheduling a chain.

When small loops are scheduled in a wide-issue machine the II is typically 1 or 2. In those cases the number of available slots for move operations in a single cluster is not enough to schedule the required chains. Hence we have used loop unrolling to increase the II, which results in more slots per cluster. This strategy allows scheduling of most of those loops without performance degradation.

We have also found that the *DDG* transformation to eliminate multiple-use lifetimes is an important factor to support both partitioning schemes proposed in this thesis. Limiting to 2 the number of predecessors of each operation simplifies the task of distributing operations evenly among clusters with limited connectivity. Multiple-use lifetimes would force the producer and all consumer operations to be scheduled in at most 3 adjacent clusters, in order to avoid using move operations. If moves are required, there will be a high concentration of them around the cluster of the producer operation. Both situations can restrict the scheduler, potentially requiring an increase in the II, which can be avoided using the transformation described in Section 4.2.

This restriction is not necessary if they are transformed into several single-use lifetimes.

7.1.5 Using DMS with Other Machine Models

Although the DMS algorithm has been specially developed targeting the architecture model described in Section 6.1 we believe it could also be used with other clustered VLIW architectures. Although some other issues may arise due to particular features of an architecture, we understand that it should present three basic characteristics in order to use DMS:

- Directly-connected clusters should communicate through a mechanism able to ensure fixed timing constraints, known precisely at compile time. For performance reasons this latency should be similar to the cycle time of the cluster private register file. In our model this is accomplished by using a CQRF between clusters. In addition, each cluster must have at least one FU able to perform move operations.

- The number of possible paths to create a chain should be small, in order to avoid considering an excessive number of options. Using a bi-directional ring, for instance, limits the number of options in two. By contrast, a three dimensional mesh of clusters may present several options, which would probably cause a negative impact on the scheduling time.

- The instruction set we use assumes that an operation can have a true data dependence with at most two predecessors. This is the case in most of the instruction sets currently in use, with operations taking one or two operands. This ensures that the number of predecessors of *OP* should be limited to two. However the number of successors can be very large. A load operation, for instance, can have its target operand used by several other arithmetic operations. As discussed in Section 7.1.4, transforming all multiple-use lifetimes into single-use ones is an important feature supporting the effectiveness of DMS. Hence we believe that some sort of *DDG* transformation should be made in order to limit the number of immediate data dependent successors of any operation.

7.2 Experimental Framework Update

As previously defined in Chapter 6, the experimental framework can also consider clustered machine models. New heuristics have been also introduced in the IMS algorithm, in order to avoid communication conflicts. In this chapter we have introduced DMS, a new modulo scheduling algorithm targeting the clustered machine model defined in Section 6.1. Hence, from now on DMS will be used by the experimental framework whenever scheduling is performed for a clustered machine. The unclustered machine still employs the original version of IMS. As done before, machine resources will be estimated taking into account loop variant and invariant lifetimes.

A key feature of DMS is the possibility of using chains of move operations. We define a move operation as the process of reading a value from one register location and copying it back to another one. In practice it moves a value between distinct CQRFs, allowing communication between non-adjacent clusters (Figure 7.4). In the experimental framework we assume that those operations are executed by a Copy FU (Section 3.1.1). The only difference is that a move operation has only *one* result operand, instead of two in a copy operation. Therefore, no additional functional unit or register file access port is required to execute a move operation. However, it can increase the utilization rate of Copy FUs, which may require a higher II. This and other issues were investigated thorough experimental analyses, as presented in the next section.

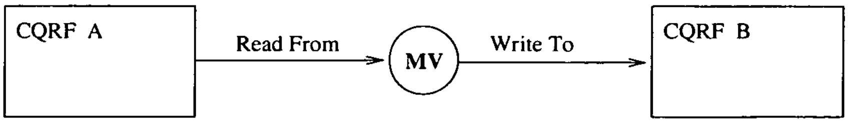


Figure 7.4: Typical use of a move operation

7.3 Experimental Results

This section presents experimental results comparing clustered and unclustered architectures. This analysis is similar to the one presented in Chapter 6, the main difference being the use of the DMS scheduling/partitioning algorithm for clustered architectures. The effectiveness of the strategies employed in the last chapter limited the analysis to machine configurations up to 7 clusters (21 FUs). The results suggested that no further performance gains would be possible beyond that level. However better results should be expected by using DMS, so in this chapter a total of 20 machine configurations are considered: 10 URF and 10 CQF models, each set of ten ranging from 3 to 30 functional units. Clustered machines can have between 1 and 10 clusters. Each cluster has 3 standard and 1 Copy FUs, 1 local register file (LRF), and 2 communication register files (CQRF). These machine configurations are summarized in Table 7.1.

Machine Configurations			
Number of Clusters: 1-10			
Functional Units	URF	CQF	Single Cluster
L/S	1-10	1-10	1
ADD	1-10	1-10	1
MUL	1-10	1-10	1
Copy	-	2-10	1
Register Files	URF	CQF	Single Cluster
RF	1	-	-
LRF	-	1-10	1
CQRF	-	4-20	2

Table 7.1: Machine configurations used in experiments with DMS

As in the previous chapters, innermost loops taken from the Perfect Club Benchmark were used in the experiments, with loop unrolling performed according to the criteria described in Section 5.1. The presentation of results is subdivided into three main topics: partitioning effectiveness, performance, and machine resources.

7.3.1 Partitioning Effectiveness

In this section the effectiveness of the partitioning algorithm is evaluated by analysing variations in the II, and also the distribution of operations among clusters.

7.3.1.1 Overhead on the II Due to Partitioning

As already discussed, a good scheduling/partitioning algorithm should minimize eventual increases of the II in relation to the values otherwise achieved for the corresponding unclustered machine. The data in Figure 7.5 shows the fraction of loops scheduled without increasing the II due to DMS partitioning. Overheads for machines with 6 and 9 FUs are only due to the introduction of copy operations in the *DDG*, because the partitioning process is not constrained by the communication system. Over 80% of the loops do not present any overhead for machine models up to 24 FUs (8 clusters). This result is considerably better than obtained using the simpler heuristics presented in Section 6.4.1, as seen in the chart. Furthermore, the overhead increase is less accentuated when DMS is used, suggesting the algorithm may adapt well to wider-issue machines. When the II increases it is mainly because the Copy FUs became the most heavily used resources, due to an excessive number of move operations (Section 7.1.4). That could be improved with additional hardware support. The ultimate effect of increasing the II or the *SC* can be estimated by analysing the total execution time and IPC values, which is presented in Section 7.3.2.

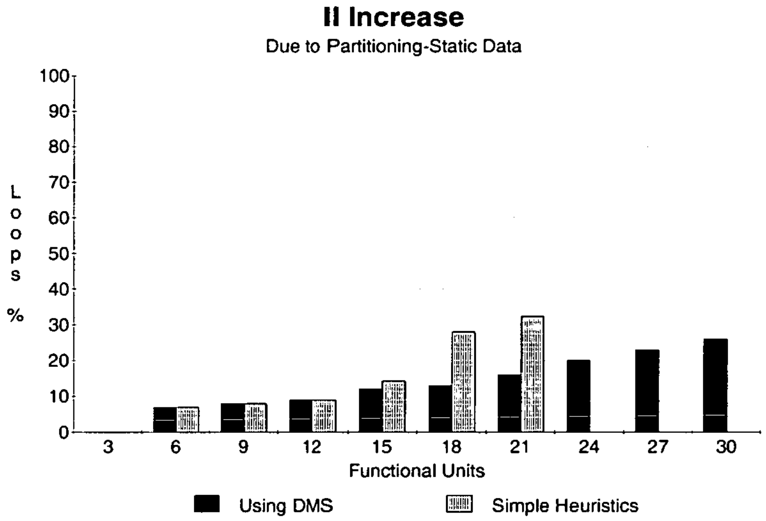


Figure 7.5: Loops with a larger II due to DMS partitioning

7.3.1.2 Communication Distance

As defined in Section 6.4.1.2, the communication distance is the number of clusters boundaries each value crosses on its way from producer to consumer. The data in

Figure 7.6 shows the distribution of communication distances for several machine configurations, measured over all lifetimes created. Similar results to the ones found in Section 6.4.1.2 were observed, with distances 0 and 1 accounting for most of the cases. Once again the frequency of distance 1 is approximately twice as high as the the frequency of distance 0. In practice this indicates that DMS effectively balance the distribution of operations and lifetimes among clusters. An increasing number of distances greater than 1 can be observed for 18 or more FUs (6 or more clusters). In all those cases this is only possible due to move operations, which are scheduled in intermediate clusters between the original producer/consumer pair. This is simply not possible using the heuristics described in Section 6.3, and shows the mechanism through which DMS achieves better results than scheme.

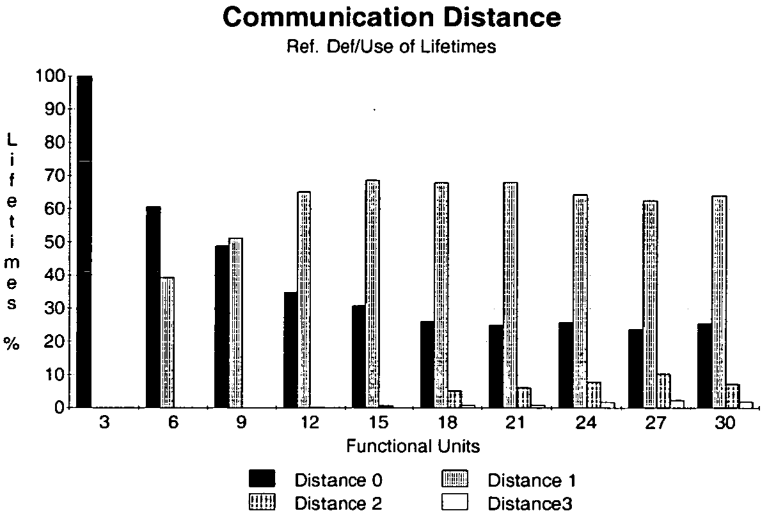


Figure 7.6: Communication distance after partitioning with DMS

7.3.2 Performance Analysis

In this section we compare the total execution time for sets of loops in both architecture models, assuming fixed cycle time for all configurations. All results are normalized using as a baseline the execution time in a URF03 machine (see values in Section 6.4.2. Results for the full benchmark set are shown in Figure 7.7. It can be seen that only small performance degradation occurs for up to 21 FUs (7 clusters). However, no further improvement can be achieved for wider-issue clustered machines. On the other hand, very small performance losses due to partitioning were observed for loops without recurrences (Figure 7.8) scheduling in *all* machine configurations. Minimal differences between clustered and unclustered machines occur up to 7 clusters. Furthermore, the results suggests that DMS may be effective for even wider-issue machines.

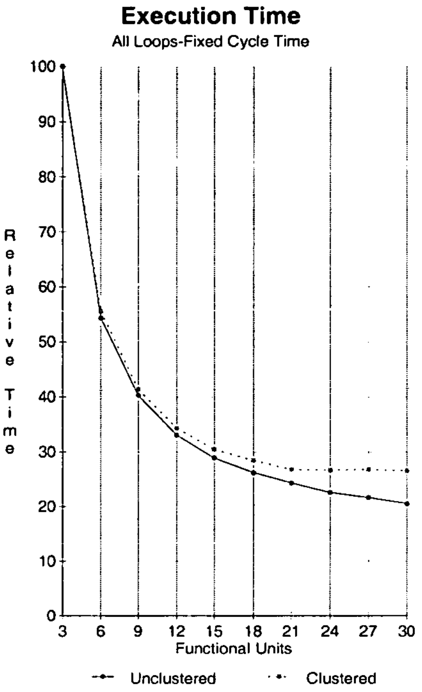


Figure 7.7: Number of cycles-Class 1

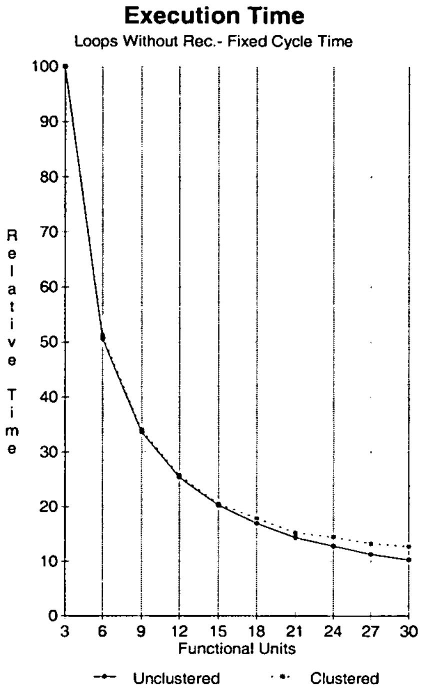


Figure 7.8: Number of cycles-Class 3

Similarly, the value of $IPC_{dynamic}$ improves for machines up to 21 FUs, levelling beyond that point (Figure 7.9). Loops without recurrences allow improvements for the whole range of machine models, as seen in Figure 7.10.

The results presented suggest that the DMS algorithm is effective for machine configurations of at least 7 clusters, or even further if a more restricted set of loops is used. These conclusions will be extended in Section 7.3.3.2, where the actual cycle time of each machine configuration will be used to calculate the execution time.

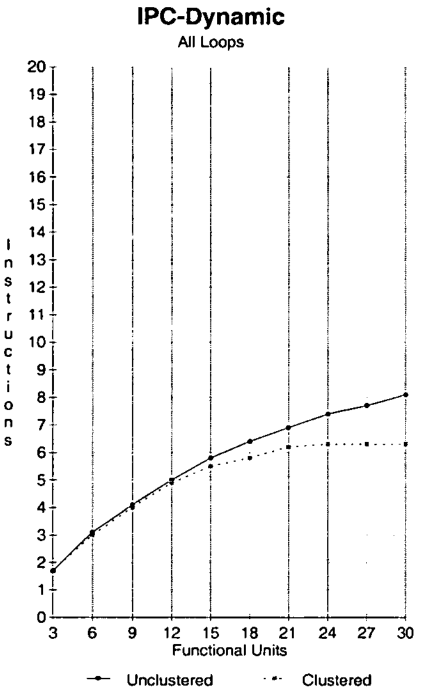


Figure 7.9: IPC Dynamic-Class 1

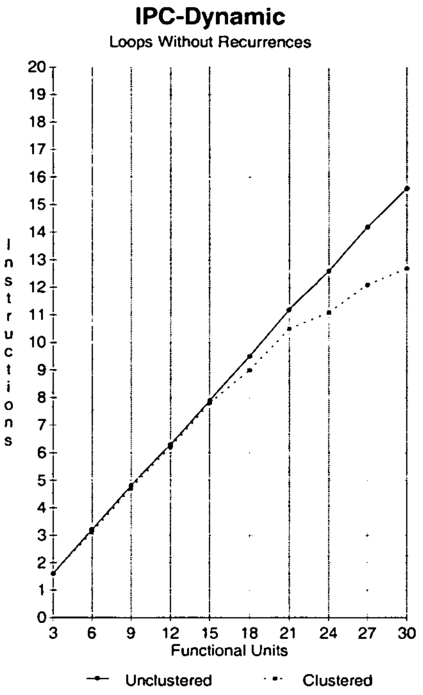


Figure 7.10: IPC Dynamic-Class 3

7.3.3 Machine Resources Analysis

In this section we estimate the machine resources required to achieve the performance levels reported in Section 7.3.2, focusing on the silicon area and cycle time analysis. All data refer to dynamic measurements, accounting for the loops responsible for 99% of the total execution time of the benchmark.

7.3.3.1 Register File Area

The value of MaxLive can be used to determine the number of registers in a conventional RF, which is used by unclustered machines (Table 7.2).

Register requirements for clustered machines are estimated as described in Section 6.4.3.1. The exact value of the parameters used to calculate the area of *each* LRF and CQRF of a clustered machine is shown in Table 7.3. It is assumed that each register location is *64 bits* wide. The columns labelled *No* indicates the number of LRFs and CQRFs required by each machine configuration, respectively.

The chart in Figure 7.11 shows the *total* silicon area required to implement the register files of each machine configuration. Once again the results show that the area of an unclustered machine of 6 FUs is smaller than the equivalent clustered machine. Both areas are similar for 9 FUs. Clustered machines of 12 or more FUs

URF Register File Parameters			
	Capacity	Ports	
FUs	Registers	Read	Write
3	71	6	3
6	69	12	6
9	102	18	9
12	108	24	12
15	137	30	15
18	152	36	18
21	176	42	21
24	178	48	24
27	172	54	27
30	221	60	30

Table 7.2: URF register requirements

CQF Register File Parameters									
LRF					CQRF				
	No	Capacity	Ports		No	Capacity		Ports	
FUs		Registers	Read	Write		Queues	Length	Read	Write
3	1	71	6	3	0	-	-	-	-
6	2	63	7	5	4	11	8	7	5
9	3	45	7	5	6	9	9	7	5
12	4	38	7	5	8	9	8	7	5
15	5	34	7	5	10	9	7	7	5
18	6	33	7	5	12	10	9	7	5
21	7	30	7	5	14	11	10	7	5
24	8	29	7	5	16	9	8	7	5
27	9	32	7	5	18	12	10	7	5
30	10	33	7	5	20	13	24	7	5

Table 7.3: CQF register requirements

(4 or more clusters) are clearly more efficient in terms of area. The advantage tends to increase as the machine model scales up. It can be seen in Table 7.3 that register requirements of individual LRFs and CQRFs remain at the same level for the whole range of configurations. This suggests that the silicon area may grown proportionally to the number of functional units, which is not the case for an unclustered machine.

We have estimated register requirements not taking into account any further optimization to minimize resource usage. Hence, it might happen that some configurations have the capacity of all register files determined by the requirements of just a few loops (1 or 2, most often). For instance, it was defined that the size of each queue is 24 for a clustered machine having 30 FUs. This is the requirement of one loop only. If we do not take it into account, the required queue length would be reduced to 12.

After performing a detailed analysis on the most demanding loops of the benchmark we have concluded that it may be possible to achieve similar performance levels having all CQRFs implemented with 8 queues of 8 locations each. A few loops require additional resources, and when it happens it could be minimized if a better resource allocation was performed. To address these cases, it may be possible to develop a strategy to add some degree of register-pressure sensitivity to DMS. Furthermore, spill-code could be used to deal with the remaining cases. Thus, we understand that the proposed configuration is a realistic target to be pursued, which will result in further improvements in the silicon area and cycle time.

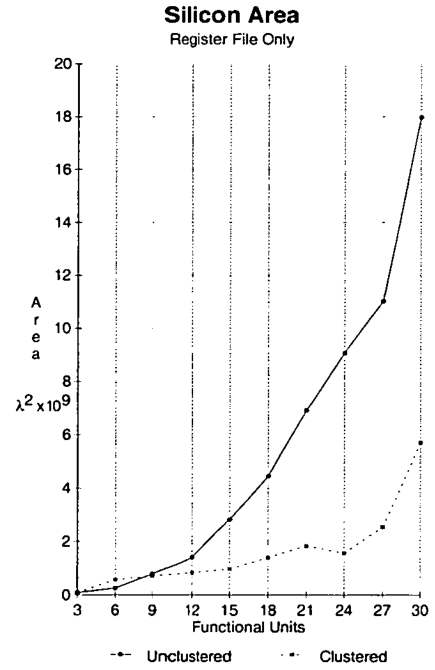


Figure 7.11: Total silicon area of register files

7.3.3.2 Register File Cycle Time

Estimates for the expected cycle time of RFs, LRFs, and CQRFs can be found in Figure 7.12. It can be seen that LRFs and CQRFs have equivalent cycle times for all machine configurations, which is around 6 ns. A significant difference is only observed for the CQF30 machine, which is due to CQRFs designed with over-dimensioned queues. Apart from that, we have found that the cycle times of LRFs and CQRFs are very similar for all machine configurations. As previously found, the cycle times of centralized RFs are always higher than LRFs and CQRFs, growing approximately linearly to the number of FUs.

As in Chapter 6, we have determined the cycle time of a clustered machine based on the cycle time of LRF and CQRFs, whichever is higher. Unclustered machines have the cycle time determined by the RF. It can be seen in Figure 7.13 that the cycle time of a clustered machine is *always lower* than the corresponding unclustered version.

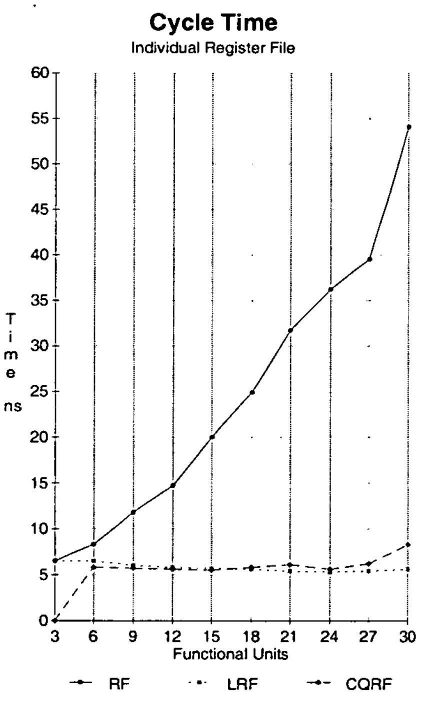


Figure 7.12: Cycle time of reg. files

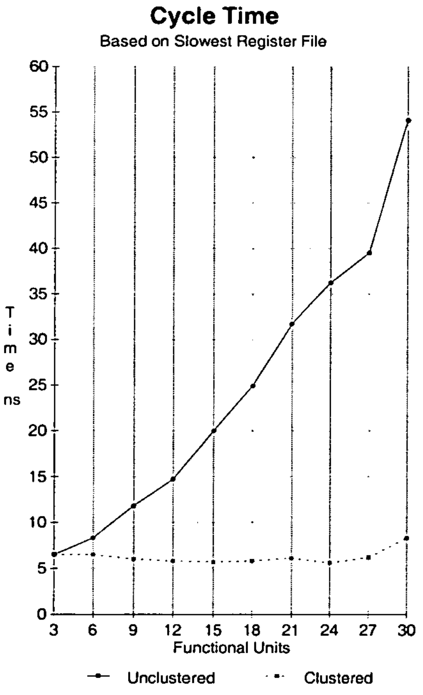


Figure 7.13: Machine cycle time

The performance results reported in Section 7.3.2 assume fixed cycle times for all machine configurations. However the actual cycle time should be taken into account to have an insight on the real machine performance.

The cycle time calculated for each configuration can be used to weight the results presented in Section 7.3.2. Actual performance improvement occurs for clustered machines up to 24 FUs (8 clusters), as seen in Figure 7.14. Similar results were observed if only loops without recurrences are considered (Figure 7.15), but a higher level of improvement is achieved in this case. Unclustered machines allow improvements up to 6 or 12 FUs, depending on the scheduled set of loops. All data presented are normalized using as baseline the execution time of a URF03 machine, using the values shown in Section 6.4.3.2.

We have confirmed previous findings, concluding that the performance of a clustered machine is always better than the corresponding unclustered organization. The results presented in this section shows that DMS further extends the advantages resulting from the use of the scheme proposed in Section 6.4.3.2. In that case, real performance gains were obtained up to 15 FUs (5 clusters). Using DMS, significant improvements can be achieved up to 24 FUs (8 clusters).

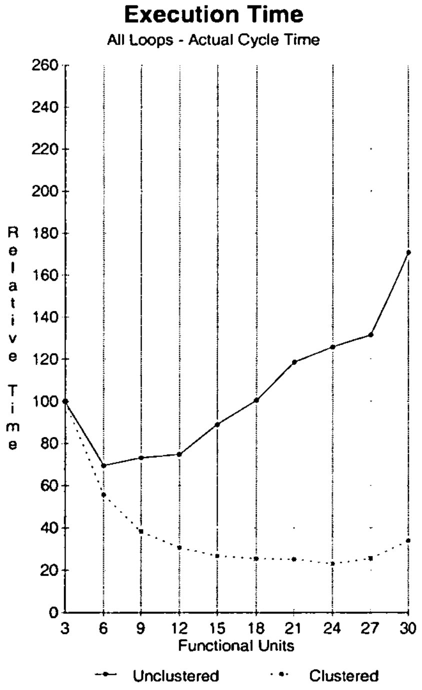


Figure 7.14: Execution time-Class 1

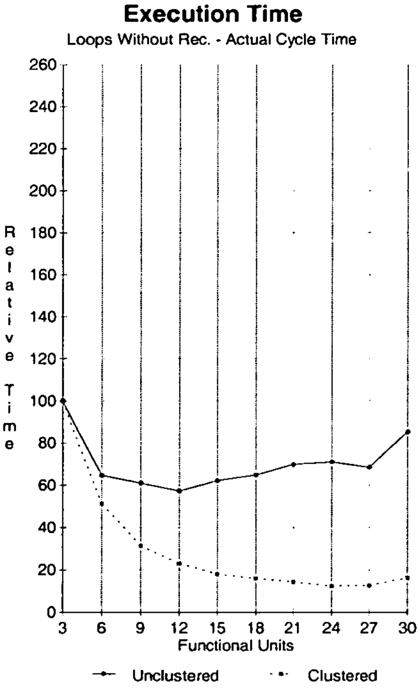


Figure 7.15: Execution time-Class 3

7.3.4 Summary of Results and Conclusions

The main conclusions obtained from the experimental analysis presented in this section are summarized in the following list:

- The proposed DMS algorithm is effective for machine configurations up to 8 clusters, resulting in low overhead due to partitioning. A larger overhead was observed for wider-issue machines, although that could be minimized by using additional FUs to schedule move operations. We suggest the use of existing standard FUs for this task. This alternative can partially address the problem without requiring extra access ports to the register files.

- DMS further extends the benefits of using a clustered architecture with the other partitioning heuristics, without requiring additional machine resources. In many cases, the use of a few move operations is enough to avoid dead-end states otherwise reached using the simplest partitioning algorithm. Thus, the silicon area and machine cycle time is kept lower than for the equivalent unclustered machine. Aggressive ILP scheduling using DMS translates into actual performance gains for configurations up to 24 FUs (8 clusters).

- The results presented in this chapter have confirmed the feasibility of a VLIW clustered architecture. The key advantage of this organization is the possibility of keeping the machine cycle time almost constant across a wide range of configurations. We have shown that the scalability of such architectures can be heavily constrained by the effectiveness of the scheduling/partitioning algorithm. The DMS algorithm can produce high quality schedules for clustered architectures comprising a number of clusters not previously considered in other works, to the best of our knowledge. Hence, it can significantly extend the potential for ILP exploitation in this kind of machine. Furthermore, an architecture employing an alternative inter-cluster communication mechanism or topology may allow extra flexibility to the scheduler, so a higher degree of improvement might be achieved.

Chapter 8

Scalability of Performance and Cost for Clustered VLIW

Future increases in transistor densities will make highly-parallel VLIW processors a realistic prospect. The scalability of VLIW processors is therefore a key issue. In this chapter we define scalability in terms of the relationship between processor area and actual performance, and focus on the scalability of clustered architectures.

A fundamental issue in microarchitecture is how to apply increasing transistor densities in ways that are most cost effective. Of course there is no single answer, but it is widely accepted that a greater exploitation of parallelism is a key requirement. However, it is equally important that the performance evaluation of candidate microarchitectures takes account of silicon area and logic delays as well as parallelism. In essence, the ability to apply increasing transistor densities effectively requires microarchitectures that are scalable.

It is axiomatic that increases in transistor count cannot translate indefinitely into processors with ever increasing logical complexity. Greater complexity requires greater verification effort, so adding new features to an already complex out-of-order superscalar processor is certain to increase design cost and may push design time beyond a viable limit. In contrast, an architecture which scales to larger configurations through replication of fixed-cost building blocks will be attractive provided it can also yield scalable performance.

In this chapter we focus on what it means to be scalable. We target specifically the novel clustered VLIW architecture model defined in Chapter 6, although it could be extended to similar architectures. We assess the scalability of our approach by considering the rate at which execution time of our benchmarks reduces when more transistors (*i.e.*, more functional units) are used.

8.1 Scalability and Technology Trends

The possibility of a billion-transistor chip is a strong motivating force in microarchitecture, but also one which presents many challenges [12]. Arguably the two most critical challenges for scaling performance with increasing chip area are:

- Exposing parallelism in applications
- Defining architectures in which time and space complexities scale linearly with increasing parallelism

As previously discussed in this thesis, the size and number of ports of a register file can seriously compromise the performance of ILP machines. The results presented in Sections 6.4 and 7.3 show clearly that *net* performance does not necessarily track *gross* ILP in very wide VLIW configurations. One must also take account of cycle time. We therefore define **scalability** as the rate at which net performance increases as the transistor budget increases.

The scalability of our clustered VLIW architecture depends upon the relationship between chip area and performance. Scalability should also imply an ability to exploit *future* advances in silicon technology. To address these issues we correlate the area estimates of the candidate configurations with predictions of future device characteristics from the 1997 *Semiconductor Industry Association* Roadmap for Semiconductors [88]. The data in Table 8.1 show certain key characteristics of the five generations expected to span 1997 to 2009.

λ (μm)	Year	Die Area (mm^2)	λ^2/die ($\times 10^6$)
0.25	1997	300	4,800
0.18	1999	340	10,494
0.13	2003	430	25,443
0.10	2006	520	52,000
0.07	2009	620	126,530

Table 8.1: SIA predictions of device capabilities (1997)

The area occupied by functional units can be estimated by reference to existing designs. For example, the FPU of the MIPS R10000 contains a multiplier, an adder, and a divider. In a 0.25 μm CMOS technology, this occupies an area of 12 mm^2 [74], or 1.92 $\times$ $10^8\lambda^2$. In the machine models we have considered so far in this thesis, one cluster group is similar to the R10000FPU, but also contains one L/S and one Copy functional units. However, both of these are integer units with low complexity. We therefore assume the MIPS R10000 FPU area to be a *reasonable approximation* to the area of one cluster group. For comparative purposes we assume that each group of three functional units in a unclustered machine is assigned the same estimated area. The total area of a given configuration therefore includes the area of LRFs, CQRFs, and the above approximation for the area of the FUs. As previously done, the area and cycle time of register files is estimated using the analytic model [62] presented in Section 4.5.

8.2 Experimental Framework Update

The features of the experimental framework last defined in Section 7.2 already present the capabilities to target a clustered VLIW machine. A key element in the process is the use of DMS, a single step scheduling/partitioning algorithm.

So far we have considered only one type of clustered configuration, consisting of 3 standard and 1 Copy FUs. We define this basic set as a FU group. To extend the analysis on scalability issues, we have updated the experimental framework in order to consider alternative cluster configurations. Hence, from now on cluster configurations can consist of 1, 2, 3, or 4 FU groups, which correspond to 3, 6, 9, and 12 standard FUs, respectively. These configuration are denoted G1, G2, G3, and G4. If necessary the suffix GX will be appended to the existing notation (CQFnn) to indicate the number of FU groups in each cluster. As an example, a machine model denoted by CQF12G2 comprises a total of 12 standard FUs, organized in 2 clusters of 2 FU groups each. Distinct cluster configurations also imply distinct access port requirements to the register files, defined according to the number of functional units.

8.3 Experimental Results

In this section we present some results regarding the scalability of the proposed VLIW architecture. The results are subdivided into two main topics: performance analysis, and scalability.

8.3.1 Performance Analysis

A range of new machine configurations is being considered in this analysis. Thus, machine configurations with the same issue-width may have distinct number of clusters, implying in the following trade-off when performance optimization is the main objective:

- Using a larger cluster configuration means that less clusters are required for a given issue-width. This reduces communication conflicts, possibly improving performance.
- A larger cluster increases register file requirements, which might increase the machine cycle time.

We have performed some experiments to investigate the effect of varying those parameters. As was done in previous chapters, results regarding execution time are normalized using as a baseline the execution time in a URF03 machine.

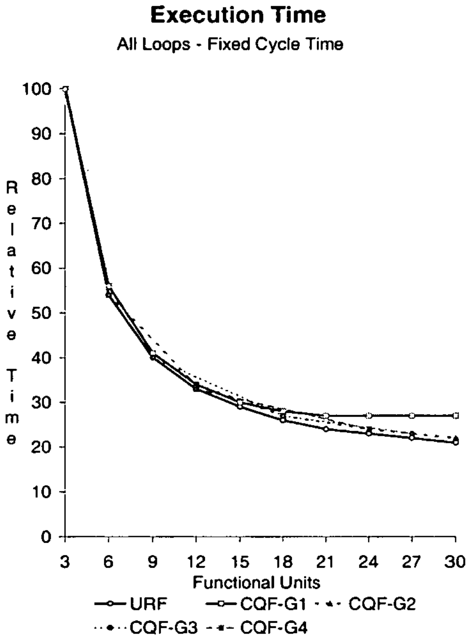

Execution Time
All Loops - Fixed Cycle Time

Figure 8.1: Number of cycles-Class 1

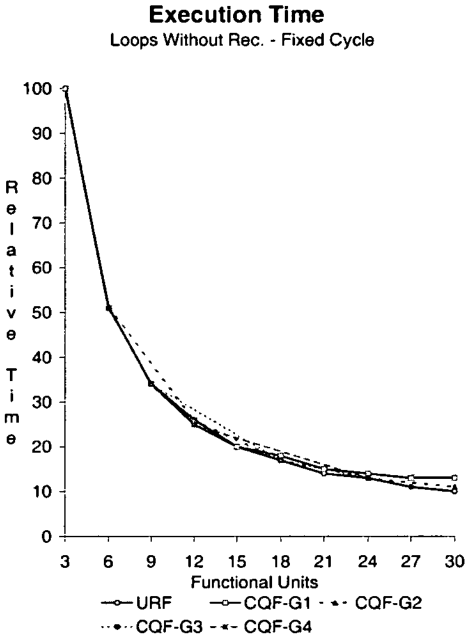

Execution Time
Loops Without Rec. - Fixed Cycle

Figure 8.2: Number of cycles-Class 3

The first set of results assumes a fixed cycle time for all configurations. Although the differences are not large, the charts in Figures 8.1 and 8.2 show that a larger cluster configuration in general results in shorter execution time. This results from the relatively smaller number of clusters required to organize the same number of FUs, which facilitates the partitioning process. It should be noticed that the performance loss is more significant for G1 configurations of 24 or more functional units. In those cases the partitioning algorithm is less effective because it works with 8 or more clusters. All other machine models comprise at most 4 clusters, resulting in less communication conflicts.

We have estimated the cycle time of all machine configurations using the same approach discussed in Section 6.4.3.2. Figure 8.3 shows how the cycle time of the register structures in those clustered and unclustered processors varies with the number of functional units, based on 0.8 micron CMOS technology parameters [62]. As previously found, the cycle time of a globally-shared register file is clearly a problem for all but the smallest configurations. However the cycle time of all clustered configurations remains essentially constant, although each one in a distinct level. As expected, the smaller the cluster configuration, the shorter the cycle time. Hence, G1 configurations allow the shortest cycle. It should be noticed that the cycle time of G1 and G2 configurations do not differ by a large factor. Furthermore, for 30 FUs G2 is better than G1. This is due to the difference between the register requirements of machines with 5 and 10 clusters, respectively. This is a clear example of the trade-off involved to define the best cluster configuration.

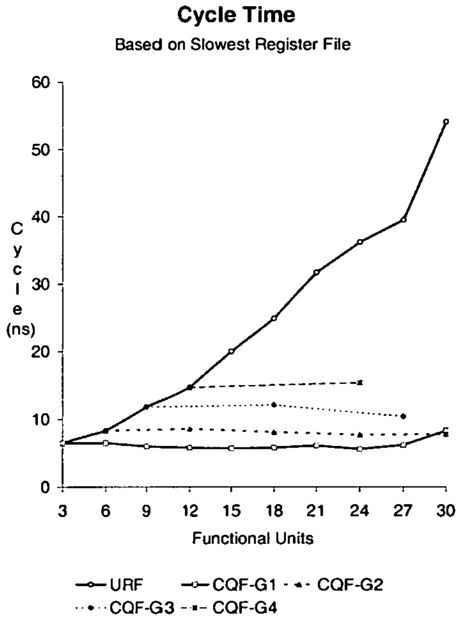


Figure 8.3: Machine cycle time

Once again we have used the cycle time calculated for each configuration to weight those results assuming a fixed cycle. As seen in Figures 8.4 and 8.5, the smallest cluster configuration generally results in the shortest execution time, for both loop sets considered. However, for 30 FUs the best results are obtained with configuration G2, which is due to lower pressure on the partitioning algorithm. The results suggest that further *effective* performance gains may be possible using configuration G2, which is not the case with G1 machine models (they peak at 24 FUs).

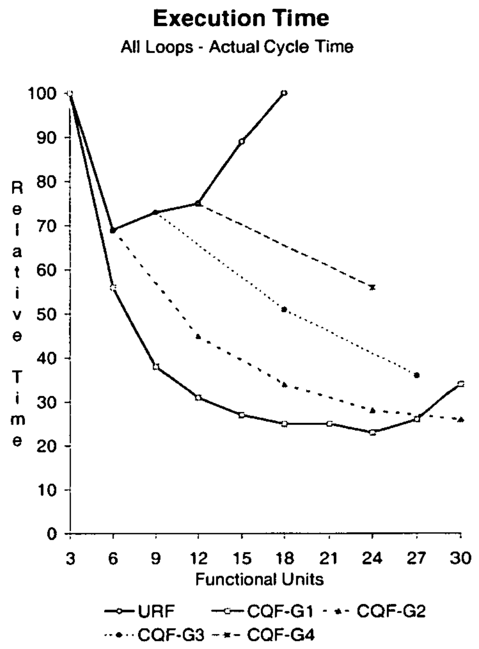


Figure 8.4: Execution time-Class 1

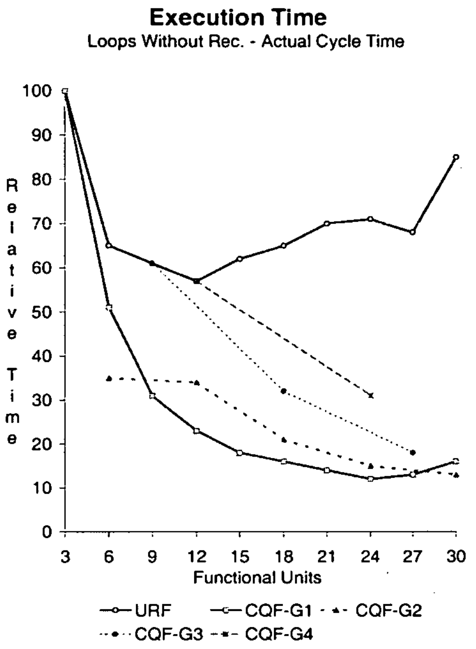


Figure 8.5: Execution time-Class 3

8.3.2 Scalability of Performance

This thesis focuses on single-chip implementations of an ILP processor. Therefore, the scaling characteristics must be viewed against expected future integration capabilities. There are four primary factors involved: the available chip area, the number of clusters and their issue-width, the expected cycle time of a configuration, and the effective IPC after scheduling/partitioning. All four factors are closely interlinked, and together determine the scalability of each configuration.

The relationship between $IPC_{dynamic}$ and chip area for the VLIW compute-engine is shown in Figures 8.6 and 8.7. We have seen in Section 7.3.2 that unclustered configurations yield the highest IPC. Here we see they have the worst IPC/area ratio. For device areas up to $4 \times 10^9 \lambda^2$ the G1 configuration has the best IPC/area ratio. Beyond that, the G2 configuration appears to be a promising candidate.

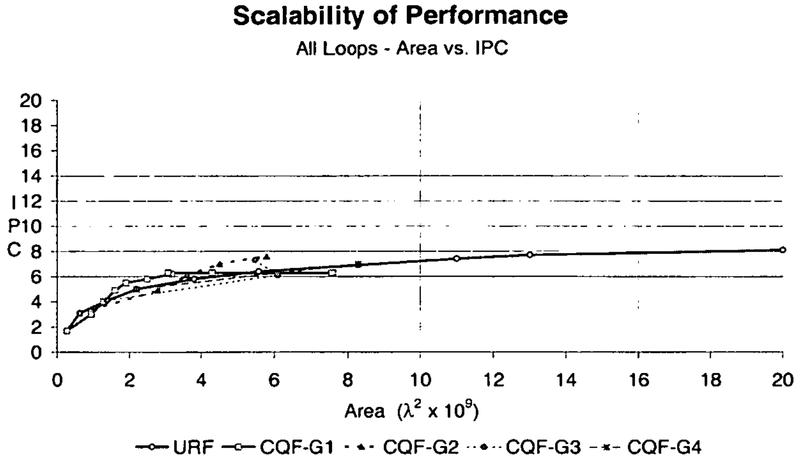


Figure 8.6: IPC vs Area-Class 1

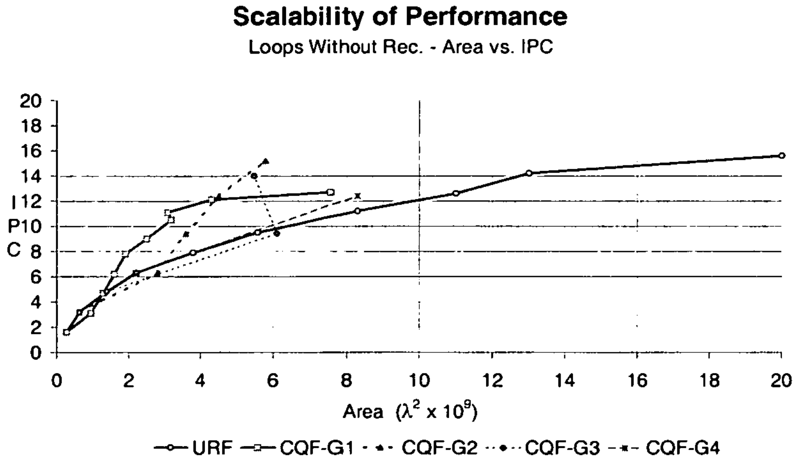


Figure 8.7: IPC vs Area-Class 3

Arguably the most fundamental metric of scalability for a single-chip ILP processor is the ratio of absolute performance to chip area. We have used the actual execution time calculated for each configuration (Section 8.3.1) as a measure of absolute performance. The graphs in Figures 8.8 and 8.9 show this relationship for all loops and recurrence-free loops, respectively. These graphs also delineate area values corresponding to intervals of 10 – 20% of the maximum chip area for the five technology generations outlined in Table 8.1. This gives an indication of the chronological scalability of each clustered configuration. Overall, these results show that statically-scheduled clustered VLIW processors scale very well up to 24 FUs, for G1 configurations. The G2 configuration scales well up to 30 FUs, and may even scale well beyond.

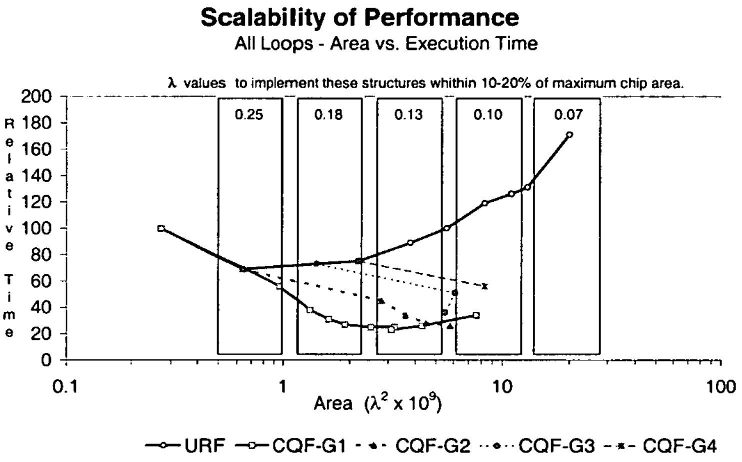


Figure 8.8: Performance vs Area-Class 1

8.3.3 Summary of Results and Conclusions

The conclusions obtained from the analysis presented in this section are summarized in the following list:

- In general, the smaller cluster configurations result in the most cost effective implementation. However the validity of this assumption depends on the effectiveness of the scheduling/partitioning algorithm.

- In this experimental framework, a cluster consisting of 1 FU group is the best option for up to 24 FUs, which corresponds to 8 clusters. Wider-issue

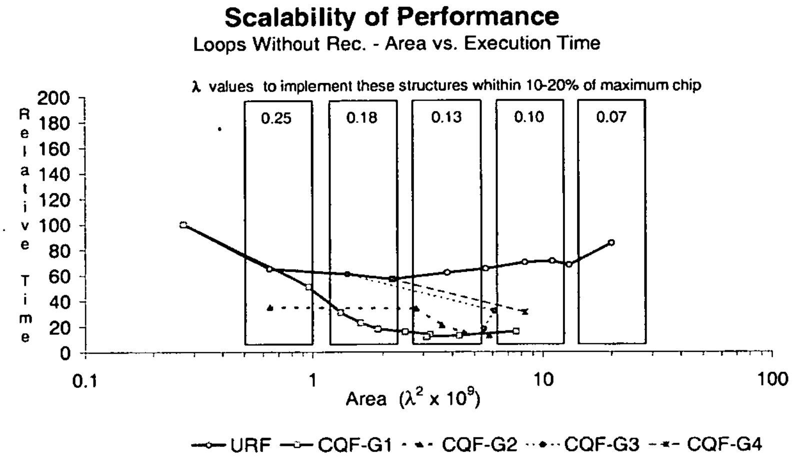


Figure 8.9: Performance vs Area-Class 3

machines can be better designed using 2 FU groups per cluster.

- Cluster configurations of more than 2 FU groups fail to deliver the expected performance, which is due to a large number of register file access ports. Actually, each of those clusters resembles an unclustered machine, with the corresponding advantages and disadvantages.

- The proposed VLIW architecture model scales well up to 24 FUs, possibly more. The scalability can be constrained by the number of clusters and register file access ports. We have found that the DMS algorithm performs well up to 8 clusters. Furthermore, it is possible to achieve a reasonable machine cycle time using up to 6 standard and 2 Copy FUs per cluster. These parameters should limit the *search space* for the most cost-effective VLIW design.

- The conditions required to employ more than 8 clusters would include a new partitioning strategy, and an alternative communication system among clusters. Employing a larger number of FUs in a single cluster, without increasing the cycle time, may only be possible with new hardware techniques.

Chapter 9

Conclusions

This final chapter summarizes all of the material presented thus far, focusing on the main aims and achievements of the thesis. This involves presenting the conclusions on our work to propose a scalable VLIW architecture model, and explicitly highlighting the original contribution to knowledge which has been made. Finally, a number of topics will be suggested for future research in this particular topic.

9.1 Main Findings

The main objective of this research work was to propose a scalable VLIW machine model targeting numeric applications, which may also be extended to some classes of DSP and multimedia applications. For this reason, software pipelining techniques were adopted to accelerate the execution of innermost loops. Our experiments have confirmed the conclusions of previous works, which indicated that an unclustered organization is inappropriate due to high register file requirements. Thus, we have proposed a clustered VLIW machine using queue register files to build a communication system between adjacent clusters. We have also proposed the integration into a single procedure of both, scheduling and code partitioning for a clustered machine. A novel algorithm performing software pipelining was developed within this strategy, called DMS. A final analysis was conducted to assess the scalability of this architecture model, in terms of performance and silicon area. The following list includes the main findings obtained throughout the development of this work:

- There is a significant amount of ILP in numeric applications to be exploited by wide-issue VLIW machines. This is particularly the case for innermost loops that are resource constrained. However, software pipelining techniques generate high register pressure, requiring large register files that may compromise the machine performance.

- Unclustered machine organizations having more than 6 functional units, and relying on a centralized register file, *fail* to deliver the expected performance. In those cases the number of access ports increases the machine cycle time to an extent that completely overshadows the performance gains obtained from ILP scheduling. Although a QRF organization in general is more efficient than a conventional RF in terms of silicon area and cycle time, it is also inappropriate to support a large number of functional units.

- Clustered machines scale well in terms of machine resources. Register requirements for both LRFs and CQRFs remain similar after successive inclusions of extra clusters in a given machine model. Thus, for a given cluster configuration, the machine cycle time shows very small variations, regardless of the number of clusters employed. This allows to increase the number of FUs without compromising the overall cycle time. As expected, a cluster configuration of 3 standard and 1 Copy FU results in the shortest cycle time. A small, possibly acceptable, increase in the cycle time results from using twice as many FUs. However, it rises sharply beyond this limit. Thus, we believe that 6 standard and 2 Copy FUs is the limit for an efficient cluster configuration, considering the architecture and corresponding hardware models we have adopted in this work.

- Good quality schedules can be produced for a clustered architecture, introducing only a small performance penalty due to code partitioning. The communication system among clusters and the efficiency of the algorithms employed determines the maximum number of clusters that allow actual performance gains. The DMS algorithm is effective for configurations of up to 8 clusters interconnected as a bi-directional ring.

- Using the architecture and compilation techniques proposed in this work, the most cost-effective design for up to 24 FUs comprises 8 clusters of 3 standard FUs each. If more functional units are used, an organization of 6 standard FUs per cluster should be the best option.

9.2 Thesis Contribution

In this section the main original contributions to knowledge made by this work are highlighted. To best of our knowledge, none of the following points have previously appeared in the literature, apart from the work of our own:

- Design of a *Queue Register File* specially targeting the execution of software pipelined loops. A key aspect of its functionality is the use of a *Q-Compatibility-Test* to perform allocation of loop variant lifetimes to queues.

- Design of a *Clustered VLIW Architecture* organized as a bi-directional ring of clusters, interconnected by queue register files.

- Propose an *integrated* approach to the problem of performing software pipelining and code partitioning for a clustered VLIW machine. Previous works have performed these tasks separately.

- Development of *DMS*, a novel modulo scheduling algorithm able to perform in a single step both scheduling and code partitioning for a clustered VLIW architecture. Experimental results showed the scheme is effective for configurations of at least 8 clusters, considering the architecture model and compilation techniques proposed in this thesis.

- Experimental *analysis* considering a large range of clustered and unclustered VLIW machines of up to 30 FUs. Scalability issues were investigated, taking into account parameters such as performance, silicon area, and cycle time.

9.3 Future Work

A few suggestions of further work are presented in this section. They can be either direct extensions of this thesis, or closely related topics in this field, as listed below:

- In terms of hardware, a major enhancement to the existing experimental framework would be the inclusion of a *memory* system connecting to the VLIW engine. This would allow us to quantify the performance degradation due to an imperfect cache system, as well as the extra hardware complexity involved in its implementation. Two issues are of particular concern regarding the organization of the memory system: the interconnection with FUs, and cache coherence. We believe that some sort of partitioning of the cache system is necessary to address these problems. In this case, the cache coherence would rely on a scheduler sensitive to the locality of dependent load and store operations. They should be assigned to the same cluster, and thus use the same cache partition. An alternative to caches is the use of local memory banks, which might be a better option according to the memory access pattern of the target applications. For a large number of FUs, the interconnection problem may be addressed with the advent of new hardware technologies.

- Improvements must be done to the QRF register allocator, in order to balance the distribution of lifetimes among queues, and thus optimize its utilization. That should reduce the required size of each queue. In addition, a scheme to introduce spill code would be necessary to work with limited machine resources.

- For comparison purposes it would be interesting to investigate another partitioning strategy. We suggest performing the partitioning of the *DDG prior* to modulo scheduling, using one of the available methods found in the literature. This would be followed by a scheduling algorithm, which should take into account the assignment of operations to clusters defined in the first step.

- This work has considered innermost loops typically found in numeric applications. Among them, resource constrained loops constitute the bulk of performance improvements achieved. On the other hand, recurrence constrained loops cannot fully benefit from the available functional units. It would be very useful to develop techniques able to reduce the latency of recurrence circuits found in those loops.

- We believe that the proposed architecture model is well suited for some classes of DSP and multimedia applications having the execution time dominated by highly parallelizable innermost loops. Hence, it would be interesting to use specific benchmarks to investigate the suitability of this architecture for those application domains.

- For commercial reasons, the execution of scalar code is increasingly important in microprocessor design. Finding enough ILP in those applications to sustain a high IPC rate still constitutes an open issue. Any wide-issue VLIW machine aiming for general purpose use should address this problem.

Bibliography

[1] A. Abnous and N. Bagherzadeh. Pipelining and bypassing in a VLIW processor. *IEEE Transactions on Parallel and Distributed Systems*, June 1994.

[2] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, December 1974.

[3] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[4] J. Allen, K. Kennedy, C. Porterfield, and J. Warren. Conversion of control dependence to data dependence. In *10th Annual Symposium on Principles of Programming Languages*, 1983.

[5] M. Aloqeely and C. Chen. A new technique for exploiting regularity in data path synthesis. In *EURO-DAC'94, European Design Automation Conference*, 1994.

[6] E. Ayguadé, C. Barrado, J. Labarta, J. Llosa, D. Lopez, S. Moreno, D. Padua, E. Riera, and M. Valero. ICTINEO: Una herramienta para la investigacion en paralelismo a nivel de instrucciones. In *VI Jornadas de Paralelismo*, 1995.

[7] D. Bacon, S. Graham, and O. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, pages 345–420, 1994.

[8] U. Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[9] I. Bennour and E. Aboulhamid. Register allocation using circular FIFOS. In *International Symposium on Circuits and Systems*, 1996.

[10] M. Berry, D. Chen, P. Koss, and D. Kuck. The perfect club benchmarks: Effective performance evaluation of supercomputers. Technical report, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign, 1988.

[11] P. Briggs, K. Cooper, and L. Torczon. Improvements to graph coloring register allocation. *ACM Transactions on Programming Languages and Systems*, 1994.

[12] D. Burger and J. Goodman. Billion-transistor architectures. *IEEE Micro*, September 1997.

[13] A. Capitanio, N. Dutt, and A. Nicolau. Design considerations for limited connectivity VLIW architectures. Technical Report TR59-92, University of California, Irvine, Department of Information and Computer Science, 1992.

[14] A. Capitanio, N. Dutt, and A. Nicolau. Partitioned register files for VLIWs: A preliminary analysis of trade-offs. In *Proceedings of the MICRO-25 - The 25th Annual International Symposium on Microarchitecture*, 1992.

[15] G. Chaitin. Register allocation and spilling via graph coloring. In *Proceedings ACM SIGPLAN Symp. on Compiler Construction*, 1982.

[16] A. Charlesworth. An approach to scientific array processing: The architectural design of the AP120B/FPS-164 family. *Computer*, 14(9), 1981.

[17] T. Conte et al. Instruction fetch mechanisms for VLIW architectures with compressed encodings. In *Proceedings of the MICRO-29 - The 29th Annual International Symposium on Microarchitecture*, 1996.

[18] J. Dehnert and R. Towle. Compiling for the Cydra 5. *The Journal of Supercomputing*, July 1993.

[19] G. Desoli. Instruction assignment for clustered VLIW DSP compilers: A new approach. Technical Report HPL-98-13, HP Laboratories Cambridge,Massachusetts, 1998.

[20] J. Dongarra and R.Hinds. Unrolling loops in Fortran. *Software-Practice and Experience*, pages 219–226, 1979.

[21] K. Ebcioglu. Some design ideas for a VLIW architecture for sequential-natured software. In *Proceedings of IFIP WG 10.3 Working Conference on Parallel Processing*, 1988.

[22] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* The MIT Press, Cambridge, Massachusetts, 1986.

[23] P. Faraboschi, G. Desoli, and J. Fisher. The latest word in digital and media processing. *IEEE Signal Processing Magazine*, March 1998.

[24] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. The multicluster architecture: Reducing cycle time through partitioning. In *Proceedings of the MICRO-30 - The 30th Annual International Symposium on Microarchitecture*, 1997.

[25] K. Farkas, N. Jouppi, and P. Chow. Register file design considerations in dynamically scheduled processors. Technical Report 95/10, Digital Western Research Laboratory, 1995.

[26] M. Fernandes, J. Llosa, and N. Topham. Allocating lifetimes to queues in software pipelined architectures. In *EURO-PAR'97, 3rd International Euro-Par Conference*, Passau, Germany, 1997.

[27] M. Fernandes, J. Llosa, and N. Topham. Extending a VLIW architecture model. Technical Report ECS-CSG-34-97, University of Edinburgh, Department of Computer Science, 1997.

[28] M. Fernandes, J. Llosa, and N. Topham. Using queues for register file organization in VLIW architectures. Technical Report ECS-CSG-29-97, University of Edinburgh, Department of Computer Science, 1997.

[29] M. Fernandes, J. Llosa, and N. Topham. Partitioned schedules for clustered VLIW architectures. In *IPPS'98, 12th IEEE/ACM International Parallel Processing Symposium*, Orlando, USA, 1998.

[30] M. Fernandes, Josep Llosa, and Nigel Topham. Distributed modulo scheduling. In *HPCA-5, 5th IEEE International Symposium on High Performance Computer Architecture*, 1999.

[31] C. Fidducia and R. Mattheyses. A linear time heuristic for improving network partitioning. In *19th Design Automation Conference*, 1982.

[32] J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, July 1981.

[33] J. Fisher. Very long instruction word architectures and the ELI-512. In *Proceedings of the 10th Annual International Symposium on Computer Architecture*, 1983.

[34] J. Fisher, P. Faraboschi, and G. Desoli. Custom-Fit processors: letting applications define architectures. In *Proceedings of the MICRO-29 - The 29th Annual International Symposium on Microarchitecture*, 1996.

[35] G. Gao, Q. Ning, and V. Van Dongen. Extending software pipelining for scheduling nested loops. In *6th International Workshop on Languages and Compilers for Parallel Computing*, 1993.

[36] S. Gerez and E. Woutersen. Assignment of storage values to sequential read-write memories. In *EURO-DAC'96, European Design Automation Conference*, 1996.

[37] B. Gieseke. A 600MHz superscalar RISC microprocessor with out-of-order execution. In *IEEE International Solid-State Circuits Conference*, 1997.

[38] J. Gonzalez and A. Gonzalez. The potential of data value speculation to boost ILP. In *12th ACM International Conference on Supercomputing*, 1998.

[39] R. Govindarajan, E. Altman, and G. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the MICRO-27 - The 27th Annual International Symposium on Microarchitecture*, 1994.

[40] J. Gray, A. Naylor, A. Abnous, and N. Bagherzadeh. VIPER: A VLIW integer microprocessor. *IEEE Journal of Solid State Circuits*, December 1993.

[41] L. Gwennap. Digital 21264 sets new standard. *Microprocessor Report*, 10(14), 1996.

[42] J. Gyllenhaal, W. Hwu, and B. Rau. Optimization of machine descriptions for efficient use. In *Proceedings of the MICRO-29 - The 29th Annual International Symposium on Microarchitecture*, 1996.

[43] T. Halfhill. Beyond pentium II. *Byte*, December 1997.

[44] R. Hank, W. Hwu, and B. Rau. Region-based compilation: Introduction, motivation, and initial experience. *International Journal of Parallel Programming*, 25:113–146, 1997.

[45] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., USA, 1996.

[46] A. Heubi, M. Ansorge, and F. Pellandini. A low power VLSI architecture with an application to adaptive algorithms for digital hearing aids. In *EUSIPCO-94, Seventh European Signal Processing Conference*, 1994.

[47] P. Hsu and E. Davidson. Highly concurrent scalar processing. In *13rd Annual International Symposium on Computer Architecture*, 1986.

[48] R. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the SIGPLAN'93 - Conference on Programming Language Design and Implementation*, 1993.

[49] W. Hwu, S. Mahlke, W. Chen, P. Chang, and N. Warter. The superblock: An effective technique for VLIW and superscalar compilation. *The Journal of Supercomputing*, July 1993.

[50] J. Janssen and H. Corporaal. Partitioned register file for TTAs. In *Proceedings of the MICRO-28 - The 28th Annual International Symposium on Microarchitecture*, 1995.

[51] M. Johnson. *Superscalar Microprocessor Design.* Prentice-Hall, Inc., Englewood Cliffs, N. Jersey, 1991.

[52] W. Karl. Some design aspects for VLIW architectures exploiting fine-grained parallelism. In *PARLE'93 - Parallel Architectures and Languages Europe*, 1993.

[53] C. Kozyrakis et al. Scalable processors in the billion-transistor era: IRAM. *Computer*, September 1997.

[54] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN'88 - Conference on Programming Language Design and Implementation*, 1988.

[55] M. Lam and R. Wilson. Limits of control flow on parallelism. In *19th International Symposium on Computer Architecture*, 1992.

[56] D. Lavery and W. Hwu. Unrolling-based optimizations for modulo scheduling. In *Proceedings of the MICRO-28 - The 28th Annual International Symposium on Microarchitecture*, 1995.

[57] D. Lavery and W. Hwu. Modulo scheduling of loops in control-intensive non-numeric programs. In *Proceedings of the MICRO-29 - The 29th Annual International Symposium on Microarchitecture*, 1996.

[58] C. Lee. *Code Optimizers and Register Organizations for Vector Architectures.* PhD thesis, University of California, Berkeley, 1992.

[59] D. Lilja. Exploiting the parallelism available in loops. *Computer*, February 1994.

[60] M. Lipasti, C. Wilkerson, and J Shen. Value locality and load value prediction. In *Proceedings of the ACM Conf. on Architectural Support for Programming Languages and Operating Systems*, 1996.

[61] J. Llosa. *Reducing the Impact of Register Pressure on Software Pipelined Loops.* PhD thesis, Universitat Politecnica de Catalunya-DAC, 1996.

[62] J. Llosa and K. Arazabal. Modelo de area y tiempo para bancos de registros multipuerto y bancos de colas. Technical Report UPC-DAC-1998-35, Universitat Politecnica de Catalunya-DAC, 1998.

[63] J. Llosa, A. Gonzalez, E. Ayguadé, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *PACT'96*, 1996.

[64] J. Llosa, M. Valero, and Ayguadé. Quantitative evaluation of register pressure on software pipelined loops. *International Journal of Parallel Programming*, 26(2):121–142, 1998.

[65] J. Llosa, M. Valero, and E. Ayguadé. Heuristics for register-constrained software pipelining. In *Proc. of the 29th Annual Int. Symp. on Microarchitecture (MICRO-29)*, pages 250–261, 1996.

[66] P. Lowney, S. Freudenberger, T. Karzes, W. Lichtenstein, and R. Nix. The multiflow trace scheduling compiler. *The Journal of Supercomputing*, July 1993.

[67] S. Mahlke. Design and implementation of a portable global code optimizers. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[68] S. Mahlke, D. Lin, W. Chen, , R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock. In *Proceedings of the MICRO-25 - The 25th Annual International Symposium on Microarchitecture*, 1992.

[69] W. Mangione-Smith, S. Abraham, and E. Davidson. Register requirements of pipelined processors. In *Int. Conference on Supercomputing*, 1992.

[70] K. Mehlhorn and S. Naher. LEDA: A platform of combinatorial and geometric computing. *Communications of the ACM*, January 1995.

[71] S. Moon and K. Ebċioglu. A study on the number of memory ports in multiple instruction issue machines. In *Proceedings of the MICRO-26 - The 26th Annual International Symposium on Microarchitecture*, 1993.

[72] J. Moreno et al. Architecture, compiler, and simulation of a tree-based VLIW processor. In *24th International Symposium on Computer Architecture*, 1997.

[73] C. Norris and L. Pollock. An experimental study of several cooperative register allocation and instruction scheduling strategies. In *Proceedings of the MICRO-28 - The 28th Annual International Symposium on Microarchitecture*, 1995.

[74] K. Olukotun et al. The case for a single-chip multiprocessor. In *ASPLOS-VII*, 1996.

[75] D. Papworth. Tuning the pentium pro microarchitecture. *IEEE Micro*, April 1996.

[76] S. Rathnam and G. Slavenburg. Processing the new world of interactive media. *IEEE Signal Processing Magazine*, March 1998.

[77] B. Rau. Dynamically scheduled VLIW processors. In *Proceedings of the MICRO-26 - The 26th Annual International Symposium on Microarchitecture*, 1993.

[78] B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the MICRO-27 - The 27th Annual International Symposium on Microarchitecture*, 1994.

[79] B. Rau. Iterative modulo scheduling. *The International Journal of Parallel Processing*, February 1996.

[80] B. Rau and C. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *14th Annual Workshop on Microprogramming*, 1981.

[81] B. Rau, M. Lee, P. Tirumalai, and M. Schlansker. Register allocation for software pipelined loops. In *Proceedings of the ACM SIGPLAN'92 - Conference on Programming Language Design and Implementation*, 1992.

[82] B. Rau and P. Tirumalai M. Schlansker. Code generation schema for modulo scheduled loops. In *Proceedings of the MICRO-25 - The 25th Annual International Symposium on Microarchitecture*, 1992.

[83] B. Rau, D. Yen, W. Yen, and R. Towle. The Cydra 5 departmental supercomputer. *Computer*, January 1989.

[84] R. Rau and J. Fisher. Instruction-level parallel processing: History, overview and perspective. *The Journal of Supercomputing*, July 1993.

[85] F. Sanchez. *Loop Pipelining with Resource and Timing Constraints.* PhD thesis, UPC - Universitat Politecnica de Catalunya, 1995.

[86] M. Schlansker et al. Compilers for instruction-level parallelism. *Computer*, December 1997.

[87] M. Schlansker and V. Kathail. Acceleration of first and higher order recurrences on processors with instruction level parallelism. In *6th International Workshop on Languages and Compilers for Parallel Computing*, 1993.

[88] Semiconductor Industry Association. *The National Technology Roadmap for Semiconductors*, 1997.

[89] N. Seshan. High VelociTi processing. *IEEE Signal Processing Magazine*, March 1998.

[90] A. Smith and J. Lee. Branch prediction strategies and branch-target buffer design. *Computer*, January 1984.

[91] M. Stoodley and C. Lee. Software pipelining loops with conditional branches. In *Proceedings of the MICRO-29 - The 29th Annual International Symposium on Microarchitecture*, 1996.

[92] J. Thornton. Parallel operation in the control data 6600. In *Proceedings of the Fall Joint Computer Conference*, 1964.

[93] R. Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of Research and Development*, January 1967.

[94] D. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 1991.

[95] D. Wall. Limits of instruction-level parallelism. Technical Report 93/6, Digital Western Research Laboratory, 1993.

[96] S. Wallace and N. Bagherzadeh. A scalable register file architecture for dynamically scheduled processors. In *PACT'96*, 1996.

[97] N. Warter-Perez and N. Partamian. Modulo scheduling with multiple initiation intervals. In *Proceedings of the MICRO-28 - The 28th Annual International Symposium on Microarchitecture*, 1995.

[98] S. Weiss and J. Smith. Instruction issue logic for pipelined supercomputers. In *Proceedings of the 11th Annual International Symposium on Computer Architecture*, 1984.

[99] S. Wilton and N. Jouppi. An enhanced access and cycle time model for on-chip caches. Technical Report 93/5, Digital Western Research Laboratory, 1994.