# Integer Performance Evaluation of the Dynamically Trace Scheduled VLIW Architecture

*Alberto Ferreira de Souza*

Thesis submitted in partial fulfilment of the requirements for the degree of

**Doctor of Philosophy**
of the
**University of London**

Department of Computer Science
University College London
University of London

September 1999



ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 – 1346

# Abstract

Very long instruction word (VLIW) machines potentially provide the most direct way to exploit Instruction-Level Parallelism (ILP), but cannot be used to emulate current general-purpose instruction set architectures. In addition, programs scheduled for a particular implementation of a VLIW model cannot be guaranteed to be binary compatible with other implementations of the same model either with a different number of functional units or functional units with different latencies. This problem is known as the VLIW object code compatibility problem. The Dynamic Instruction Formatting (DIF) concept, however, can be used to implement machines that execute code in a VLIW fashion and that are capable of overcoming the VLIW object code compatibility problem. A DIF machine schedules instructions into blocks of VLIW instructions while executing them on a simple engine and caches these blocks for repeated execution on a VLIW engine.

This thesis presents an architecture, named Dynamically Trace Scheduled VLIW (DTSVLIW), which follows the DIF concept. The DTSVLIW architecture was conceived independently of DIF and its implementation is significantly different from the DIF implementation suggested by the proponents of DIF. A DTSVLIW machine differs in the instruction-scheduling algorithm, register renaming mechanism, register access mechanism, and VLIW cache organisation.

To evaluate the DTSVLIW, a trace-driven simulator has been implemented and experiments using SPEC benchmark programs have been performed. The effect of various architectural parameters on the DTSVLIW integer performance has been studied and the effectiveness of the DTSVLIW instruction-scheduling algorithm has been evaluated. In addition, comparisons between the DTSVLIW performance and that of DIF and Superscalar implementations have been made. The results show that the DTSVLIW achieves significant ILP with feasible machine configurations and that, although simpler, the DTSVLIW instruction-scheduling algorithm is as effective as the DIF's. The results also show that the DTSVLIW performs better than the DIF and Superscalar architectures for representative machine configurations while using less hardware resources and in a way that should not produce a longer clock cycle than these architectures.

The principal scientific contributions of this thesis are: (i) conception of a VLIW-based architecture – the DTSVLIW – that uses a pipelined instruction-scheduling algorithm, which effectively produces VLIW instructions dynamically; (ii) proof that the core of the DTSVLIW instruction-scheduling algorithm has complexity comparable to that of an adder, and as such can be implemented in hardware without impacting the DTSVLIW clock cycle time; (iii) evaluation of the effect of important DTSVLIW architectural parameters on its performance; (iv) evaluation of the effectiveness of the DTSVLIW instruction-scheduling algorithm; (v) comparison of the DTSVLIW performance with that of the DIF and Superscalar architectures.

# Acknowledgements

I would like to give a very special thanks to my supervisor Peter Rounce for his friendship, encouragement, attention, support, kindness, and invaluable advice throughout these three years.

I would also like to give a special thanks to Antonio Liotta. His company and friendship in all the critical moments of this research work has been invaluable.

My thanks to Jorge Ortega-Arjona, Lee Braine, Tom Quick, Artur d'Avila Garcez (from Imperial College London), Rafael Bordini, Nadav Zin, Adil Qureshi, and all other friends of the Department of Computer Science at UCL for their encouragement, advice, and criticism.

I am indebted to Eliseu Monteiro Chaves Filho for allowing me to use the code of his Sparc scalar simulator.

I owe a huge debt of gratitude to my colleagues of the *Departamento de Informática* at *Universidade Federal do Espírito Santo* (UFES) for carrying out my teaching duties while I have been here.

I gratefully acknowledge the financial support provided by the Brazilian people through the *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior* (CAPES) and UFES.

Finally, but by no means the least, I wish to thank my wife, and my family and friends in Brazil whose love and support nurtured me all this time.

*To Tina*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In the standard model of computation, a computer program coded in machine language can be viewed as a vector in which each element is an instruction that commands a small set of simple operations. The computer starts executing the program at an instruction and proceeds executing instructions in ascending order. This order can be broken, however, by conditional or unconditional branch instructions, which can move the execution flow, forward or backward, a specified number of instructions. The program semantics is guaranteed to be correct if the instructions are executed atomically, and in the sequence specified by the program ordering and the taken branches.

A simple sequential computer processor executes each instruction completely before starting the execution of the next instruction. However, many operations are common to the execution of all instructions, which makes it possible to implement a processor in a pipelined way, as in a factory, to improve its performance (its ability to execute the program faster). In addition, sometimes instructions can be executed in parallel or even out of the order specified in the program without changing the program semantics. This can be exploited by properly designed processors containing many functional units capable of operating in parallel. Furthermore, the pipeline technique can be combined with parallel and out-of-order execution capabilities in a single processor for improving performance even further. Such processors are said to follow the *Superscalar* architecture [Johnson91].

An alternative approach to improve the performance is to generate machine

code in a parallel form. In this case, each element of the vector representing the program can hold a certain number of independent operations that can be executed in parallel. Properly designed processors execute such a program by fetching one element of the vector at a time and by issuing the operations in parallel. Such processors are said to follow the *Very Long Instruction Word* (VLIW) architecture, because their instructions have hundreds or even thousands of bits to accommodate several operations [Fisher84]. VLIW processors can also be implemented in a pipelined fashion.

In VLIW systems, the compiler has complete responsibility for creating a package of operations that can be simultaneously issued. VLIW processors do not dynamically make any decisions about multiple operation issue, and thus they are simple and fast. In addition, because the VLIW compiler can look for parallelism between instructions in the whole program, VLIW architectures are potentially the most direct way of exploiting the *instruction-level parallelism* (ILP). However, the assumptions built into the code by the VLIW compiler about the hardware prevent code compatibility between different implementations of the same VLIW *instruction set architecture* (ISA). VLIW processors with different levels of hardware parallelism require recompilation of the source code. For instance, the code generated for a VLIW processor with four operations per VLIW instruction could not run on another VLIW processor with five operations per VLIW instruction without recompilation. This problem, known as *the VLIW object-code compatibility problem*, has limited the commercial interest in VLIW architectures [Rau93b].

Recently, a new architectural concept named *Dynamic Instruction Formatting* (DIF) has been proposed [Nair97]. A machine implementing this concept can overcome the VLIW object-code compatibility problem by executing the program in two distinct phases. First, when a fragment of code is initially encountered, a simple processor in the machine, not aggressive in exploiting parallelism, executes it. At the same time, this fragment of code is scheduled by a special functional unit of the machine into VLIW instructions. These VLIW instructions are saved in a special cache memory. If the same fragment of code is encountered again, another processor of the machine, this one a VLIW parallel processor, executes the scheduled version instead of the previous one.

In this thesis, we present a new architecture that follows the DIF concept. This architecture, named *Dynamically Trace Scheduled Very Long Instruction Word* (DTSVLIW) architecture [deSouza98a], has been conceived independently of DIF, which has permitted an implementation significantly different from that suggested by the proponents of DIF. We have shown that the DTSVLIW is easier to implement than DIF and delivers equivalent performance with less hardware resources [deSouza99a].

Figure 1.1 shows a block diagram of the DTSVLIW architecture. In the DTSVLIW architecture, the Scheduler Engine fetches instructions from the Instruction Cache and executes them first using a simple pipelined processor – the Primary Processor. In addition, its Scheduler Unit dynamically schedules the trace produced during this execution into VLIW instructions, placing them as blocks of VLIW instructions in the VLIW Cache. If the same code is executed again, it is then fetched by the VLIW Engine from this cache and executed in a VLIW fashion. In the DTSVLIW architecture, the Scheduler Engine provides object-code compatibility, and the VLIW Engine provides VLIW performance and simplicity.

To execute code in two distinct modes, one sequential and one parallel, results in four positive characteristics:

1. Code compatibility between different machine generations is facilitated – DTSVLIW machines with different levels of hardware parallelism can easily share the same ISA.
2. Complex instructions can be dealt with in sequential mode – During sequential execution, complex instructions can be decomposed into several simpler operations, which can later be executed in parallel mode.
3. The task of finding parallelism is simplified – The DTSVLIW's Scheduler Unit receives no more than one instruction per cycle and, therefore, can have a simple and fast hardware implementation.
4. Instruction exceptions can be dealt with in sequential mode – In case of an instruction exception during parallel execution, a DTSVLIW machine can switch to sequential mode and deal with the exception in this mode.

In order to take advantage of these characteristics, however, a DTSVLIW machine has to operate in parallel mode most of the time. This means that it has to

reuse the blocks of VLIW instructions saved in the VLIW Cache many times. A DTSVLIW machine, then, relies on code temporal execution locality to improve program execution performance.

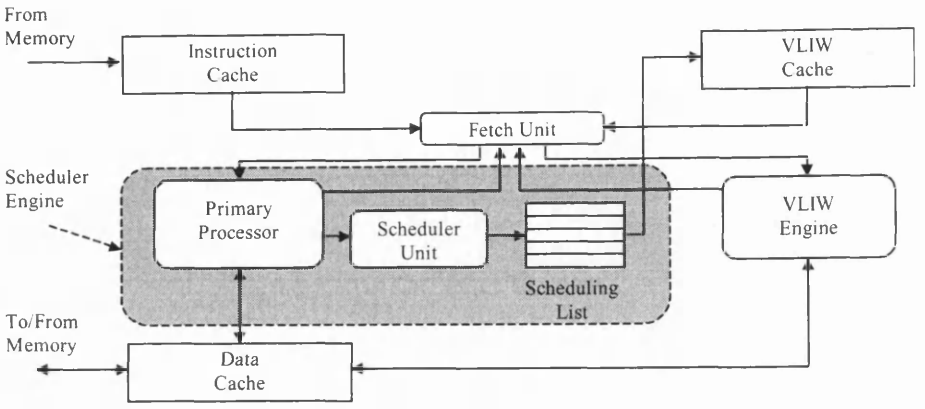



**Figure 1.1: The Dynamically Trace Scheduled VLIW (DTSVLIW) architecture**

## 1.1 Research Motivation

The main motivation for this research came from the observation that even small instruction caches (16-Kbyte or 4098 instructions) can achieve average hit rates higher than 99% with the SPEC92 and SPEC95 benchmark suites [SPEC, Gee93, Charney97]. This shows that there is strong temporal execution locality in programs. The DTSVLIW exploits temporal execution locality by converting the code into blocks of VLIW instructions in the first execution encounter and by executing it in the VLIW Engine in subsequent encounters. To make a mechanism like this work and take advantage of the DTSVLIW architecture characteristics, the conversion algorithm has to be effective in producing VLIW code. In addition, it has to be simple enough not to render the clock cycle time longer than that determined by the VLIW Engine design. The results achieved with this research so far supports the view that this can be achieved [deSouza99a, deSouza99c].

## 1.2 Goal of this Thesis

The aim of the research work presented in this thesis is to examine the following hypotheses:

- *The DTSVLIW architecture can overcome the VLIW object code compatibility problem while preserving the VLIW architecture simplicity and high clock rate.*
- *The DTSVLIW can achieve higher performance than the DIF and Superscalar architectures using equivalent hardware.*

With this aim, we describe the DTSVLIW in detail and show that it can be implemented without taking the clock cycle far from that determined by a pure VLIW design. In addition, we show that the DTSVLIW can execute legacy code in VLIW mode. To better understand the DTSVLIW architecture, we investigate the effect of various architectural parameters on its performance via experiments. We then compare the DTSVLIW performance with that of DIF and Superscalar architectures.

## 1.3 Overview of the Contributions

The main contributions of this research work are:

- Conception of a VLIW-based architecture – the DTSVLIW – capable of taking advantage of the code temporal locality for achieving program execution performance. The DTSVLIW uses a pipelined code-scheduling algorithm that effectively produces VLIW instructions dynamically [deSouza98a].
- Proof that the core operation of the DTSVLIW scheduling algorithm has complexity comparable to that of an integer adder and, as such, can be implemented in hardware level without impacting the DTSVLIW clock cycle time [deSouza99a].
- Evaluation of the effect of important DTSVLIW architectural parameters on its performance [deSouza98b, deSouza99a, deSouza99b].
- Evaluation of the effectiveness of the DTSVLIW scheduling algorithm [deSouza99c].
- Comparison of the DTSVLIW performance with that of the DIF, pure VLIW, and Superscalar architectures [deSouza99a, deSouza99c].

## 1.4 Organisation of this Thesis

This thesis is organised in eight chapters. After this introduction, Chapter 2 presents the background for this research. Chapter 3 contains discussions of previous research relevant to this thesis on microcode scheduling and VLIW architectures. In addition, in this chapter we discuss research that either tackles the VLIW object code compatibility problem or proposes mechanisms to exploit code execution locality. Chapter 4 describes the DTSVLIW architecture, showing how it can be implemented with a fast clock cycle and how it solves the VLIW object code compatibility problem. Chapter 5 presents the methodology used to carry out the experiments to evaluate the DTSVLIW architecture and the metrics used in the evaluation. The trace-driven simulator of the DTSVLIW and the benchmark programs used are also presented. In Chapter 6, we describe the experiments used for investigating the effect of various architectural parameters in the DTSVLIW performance, and for comparing the DTSVLIW performance with that of DIF and Superscalar architectures. Chapter 7 puts the DTSVLIW in perspective by comparing its characteristics with those of other machine architecture proposals. In Chapter 7, we make a critical assessment of this research work by examining the limits of validity of the experimental results and the possible difficulties in implementing a DTSVLIW machine. Chapter 8 presents a summary of this thesis, its conclusions, and suggests future directions for improving the DTSVLIW architecture performance.

# Chapter 2

# Background

The ultimate goal of the computer designer is to minimise the execution time of any given program. We can express this execution time, $T$, as:

$$T = N \times C \times S$$

where $N$ is the number of instructions that need to be executed, $C$ is the average number of processor clock cycles per instruction, and $S$ is the number of seconds per cycle. To a first approximation, $N$, $C$, and $S$ are affected primarily by the compiler technology, the ISA, and the implementation technology, respectively [Rau89].

An approach to reduce $N$ at the expense of a smaller increase in $C$ is to exploit the parallelism available in horizontally microcoded machines [Salisbury76]. This can be done by defining complex instructions that exploit the internal micro-parallelism of these machines in the hope that $N$ would decrease more sharply than $C$ would increase. This is the general idea behind *Complex Instruction Set Computers* (CISC).

Microprogramming can be used to implement powerful CISC ISAs, with complex instructions capable of commanding many operations involving many operands in registers or main memory. A typical CISC instruction, for example, can read a value from memory, add this value to the content of an internal register, and store the result in another position in the memory.

By contrast, the *Reduced Instruction Set Computer* (RISC) approach to reduce

$T$ focuses on the use of very simple instructions, which would reduce $C$ and, due to its simplicity, $S$ as well. The resulting increase in $N$ can be minimised by fine-tuning the compiler technology and the RISC ISA's implementation [Patterson85].

RISC instructions usually command a single operation involving no more than one access to memory. Due to their simplicity, RISC instructions are not implemented using microprogramming. RISC machines are, therefore, equivalent to a vertical microprogrammable microarchitecture exposed to the programmer of the ISA level.

Improvements in the implementation technology alone can also reduce $T$ by reducing $S$. However, assuming the use of the fastest technology, any further increase in performance would require the exploitation of the parallelism available in programs.

## 2.1 Architectures for Exploiting ILP

Machines capable of exploiting parallelism in the ISA level realise ILP. In this section, we discuss briefly some important machine architectures for exploiting ILP.

### 2.1.1 Pipelined Architectures

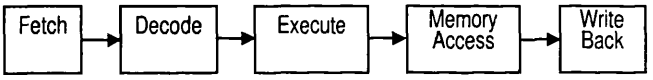
There are two basic types of ILP: temporal and spatial. *Pipelining* realises temporal ILP. It refers to the segmentation of an instruction's execution into several sub-processes that are executed by dedicated autonomous units (pipe stages). Successive instructions can be executed in a mode analogous to car assembly in a factory. Using pipelining, several instructions can be executed in parallel, each one in a different pipe stage, and in a different phase of its execution (Figure 2.1). Because the operations performed by each pipe stage are simple, the pipe stages can be implemented with simple hardware, which results in a high machine-clock frequency. Theoretically, the deeper (larger number of pipe stages) the pipeline, the faster the machine is, but there are of course practical limitations to this rule. Kunkel and Smith [Kunkel86] have studied the relationship between the theoretical linear speedup that pipelining offers and its practical limitations.

The first general-purpose pipelined machine is considered to be the IBM 7030

Stretch [Bloch59]. After IBM Stretch, the majority of high-end machines have used some form of pipelining. Surveys on pipelining include Keller [Keller75], Ramamoorthy and Li [Ramamoorthy77], and Kogge's book [Kogge81], entirely devoted to pipelining.

Spatial ILP is that present in processors with multiple functional units. It refers to the execution of more than one instruction simultaneously in different functional units of the processor. Temporal and spatial parallelism can be present at the same time. Indeed, a few years after the IBM 7030 Stretch had been constructed, the CDC6600 was produced with pipelining and multiple functional units that can operate in parallel [Thornton70].

The arithmetic portion of CDC6600 has 10 functional units, and many instructions can be issued to its functional units and executed in parallel. To do so, the CDC6600 *looks-ahead* in the sequence of instructions originally specified in the program in order to determine those that can be executed at the same time.




*(a)*

| Clock Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 7 |
|---|---|---|---|---|---|---|---|
| Fetch Stage | I1 | I2 | I3 | I4 | I5 | I6 | I7 |
| Decode Stage | | I1 | I2 | I3 | I4 | I5 | I6 |
| Execute Stage | | | I1 | I2 | I3 | I4 | I5 |
| M. Access Stage | | | | I1 | I2 | I3 | I4 |
| W. Back Stage | | | | | I1 | I2 | I3 |

*(b)*

**Figure 2.1: Pipelining. *(a)* The classic five-stage computer pipeline. *(b)* After filling all pipeline stages, one instruction can be completed per clock cycle. Up to five instructions, each in a different pipe stage, can be executed in parallel in this pipeline.**

## 2.1.2 Superscalar Architectures

The term *look-ahead* derives from a class of schemes in which programs are specified in a conventional serial manner, but the processor can look ahead during execution and execute instructions either in parallel or out of order, provided no logical inconsistencies arise as a result of doing so. If the processor has sufficient capabilities, several instructions can be executed concurrently using look-ahead

[Keller75]. The look-ahead scheme used in CDC6600 is called Thornton's algorithm and is described in [Thornton64] and [Thornton70].

The IBM 360/91 introduced a powerful look-ahead scheme named Tomasulo's algorithm [Anderson67, Tomasulo67]. A variant of Tomasulo's algorithm was used in the IBM RISC System/6000 design [Grohoski90] and, since then, various variants of Thornton and Tomasulo's algorithms have been used in many machines known as *Superscalar* machines [Johnson91].

The core of a Superscalar machine, shown in Figure 2.2, is the look-ahead hardware, also called *dynamic scheduling* hardware. The role of the dynamic scheduling hardware is to dispatch decoded instructions to the *instruction window*, and to select instructions from this window to issue to the functional units each clock cycle. The instruction window is central to the dynamic scheduling hardware and may be organised as a large structure with many entries or as several small structures, one for each functional unit. In the latter case, each structure has few entries and each entry is called *reservation station*.

During the dispatch process, dependency information is added to the instructions, their outputs are renamed (see Subsection 2.2.1), and they are stored into the instruction window. The dependency information comprises dependencies between the instructions being dispatched themselves and between these and other instructions previously dispatched, such as which instruction is going to produce a needed operand. Available input operands are also added to the instructions and saved with them into the instruction window during dispatch.

During the issue process, the issue hardware collects results from the result buses (Figure 2.2), updates the instruction window with these results, selects groups of instructions ready to execute, and issues these groups to the functional units. The results collected from the result buses are stored into the instruction window in the entries that have instructions that were dispatched with an incomplete set of input operands. When all input operands of an instruction are available, the issue hardware marks it as ready to execute. Groups of ready instructions are selected by the issue hardware and issued to the functional units to execute.

One of the main problems of highly parallel Superscalar machines is the issue hardware. In order to set the instructions as ready to execute, the issue logic has to

compare the output of all functional units with the inputs of all instructions in the instruction window. This requires wires to connect all functional units to all instructions in the instruction window, and a number of comparators and associated logic proportional to the number of functional units times the number of instructions in the window. Therefore, Superscalar processors with many functional units and large instruction windows have a complex issue hardware, and a complex issue hardware impacts negatively on the clock cycle time of these machines. A possible solution to this problem is to organise the Superscalar core in clusters of functional units and associated small instruction windows, and to add special buses for communicating results between these clusters [Palacharla97, Vajapeyam97].

The term superscalar was coined by Agerwala and Cocke [Agerwala87] to name machines that dispatch multiple independent instructions per clock cycle (in their original form, the Thornton and Tomasulo's algorithms can dispatch one instruction per cycle only). In fact, Agerwala and Cocke proposed this approach as an extension of the RISC ideas [Hennessy86]. A machine that follows the RISC philosophy has a fixed instruction length, load-store instruction set (all instructions operate on registers and two specific instructions operate with memory: load and store), limited addressing modes, and a small number of possible operations. These characteristics result in easier pipelined and superscalar implementation. Nevertheless, CISC ISAs can also be interpreted by machines that follow the Superscalar architecture such as, for example, the Intel Pentium [Saini93] and the Intel Pentium Pro [Bhandarkar97] processors with the Intel IA-32 ISA. The Intel Pentium, less powerful than the Intel Pentium Pro, can dispatch up to two *simple instructions* (RISC-like instructions present in the IA-32 ISA repertoire) to two parallel execution pipelines of the machine. The Intel Pentium Pro, on the other hand, has three parallel decoders that convert up to three IA-32 ISA instructions per clock cycle into multiple RISC-like micro-operations which can be dispatched to reservation stations of five different functional units.

Computer systems that employ Superscalar techniques leave to the processor the obligation of finding and exploiting the parallelism that exists in programs. Although the compiler can help the Superscalar processor to find the parallelism [Hwu93], the Superscalar hardware still has the main responsibility for finding it.

However, the compiler can be given the primary responsibility for finding the parallelism and, in addition, for expressing it in the form of parallel code. From the different forms of parallel code that can be produced, two have been particularly successful: *Vector* code and VLIW code.

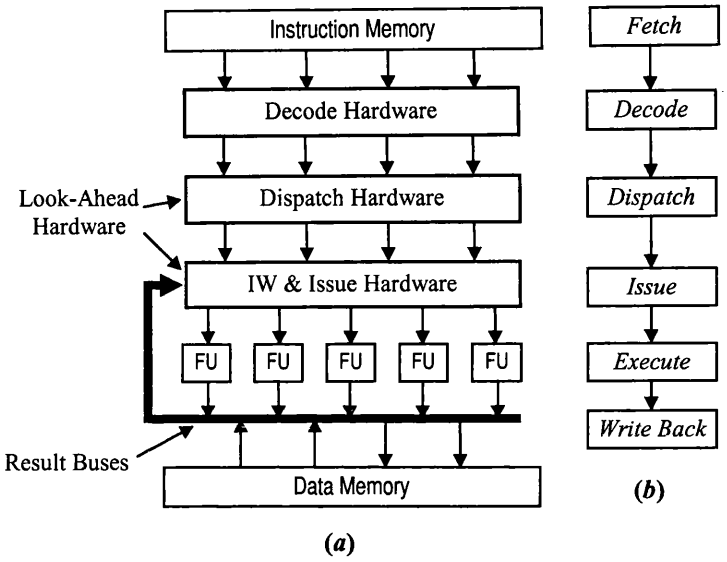



**Figure 2.2: Superscalar machine. IW stands for instruction window and FU for functional unit. (*a*) A simple Superscalar data path. (*b*) A simple Superscalar pipeline.**

### 2.1.3 Vector Architectures

To produce Vector code, the compiler looks for loops in the program and, by using vectorization techniques [Padua86], generates vector instructions that implement, or partially implement, the loops. The Vector code produced is executed by Vector machines. These machines have specialised vector functional units that compute individual element operations of a vector operation in parallel [Patterson96 (Appendix B)].

The main advantage of Vector architectures is that their hardware is simple, which results in fast machine implementations. Their hardware is simple because the compiler generates code in the explicit parallel form of vector instructions, whose implementation in hardware involves a small set of elementary operations only. A typical vector instruction might add two 64-element floating-point vectors to obtain a single 64-element vector as result. The Vector processor interprets this instruction by reading two elements from the input vector registers, adding these elements, and

writing the result into the output vector register repeatedly, from the first until the last vector element of each vector register. These operations are executed sequentially; however, in a very deep pipelined functional unit. Some Vector processors do not have vector registers and operate directly with memory.

A vector functional unit can be implemented with a very deep and fast pipeline because each result is independent of the previous in a vector operation. This independence is decided at compile time, when the vector instruction is selected to perform a sequence of operations. Because the parallelism is decided at compile time, Vector processors do not need the complex look-ahead hardware of Superscalars.

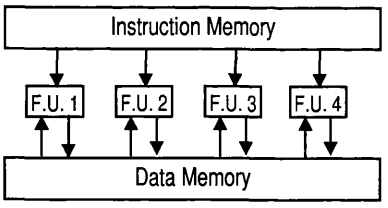
Vector operations can execute in parallel in different functional units if they are independent. Using *chaining*, a technique that allows a vector operation to start as soon as the individual elements of its vector source operand become available, even some dependent operations can execute in parallel [Ramamoorthy77, Patterson96 (page B-24)]. Scalar and vector operations can also be executed in parallel if they do not write into each other's inputs. Nevertheless, in these cases, look-ahead hardware is required.

Many Vector machines became successful commercially; however, by the end of the 1980s their importance started fading. This was caused by improvements in Superscalar architectures, their compilers, and their implementation techniques. At the beginning of the 1990s, these improvements made Superscalar machines, which were more than 10 times cheaper than contemporary Vector machines, faster than Vector machines in the execution of the very important class of *scalar programs* [Patterson96 (page B-38)]. Scalar programs are characterised by a lack of long loops that can be vectorized. Compilers, database manipulation programs, and system utilities are examples of scalar code. Nowadays, Vector computers are demanded only by specialised users that require vector performance.

Other parallel processing paradigms such as Multiprocessors, Workstation Clusters, and Massively Parallel Processors [Fernandes89, deSouza95, Patterson96 (Chapter 8)] have also challenged vector parallel processing. In these paradigms, not the compiler but the programmer has the primary responsibility for specifying the program parallelism, which is a drawback and has also limited the use of machines that follow such paradigms to specialised users.

### 2.1.4 VLIW Architectures

The other successful form of parallel code that can be produced by compilers, VLIW code, is executed by VLIW machines [Fisher84]. VLIW machines can execute several scalar operations in a single clock cycle. They have *long instructions* (hundreds to thousands of bits), with fields to control each of their many functional units. These long instructions (the term used in the rest of this thesis to refer to VLIW instructions) are fetched from memory, one per processor clock cycle, and issued to functional units that operate in parallel. In VLIW machines, the compiler has complete responsibility for creating a package of operations that can be simultaneously issued. The hardware does not dynamically make any decisions about multiple operation issue, and thus the VLIW hardware is simple and fast (Figure 2.3). However, the assumptions built into the object code by the compiler about this hardware prevent object code compatibility between different implementations of the same VLIW ISA. VLIW processors with different levels of parallelism require specific object-code. This problem is known as the VLIW object code compatibility problem [Rau93b]. Furthermore, it is not possible to implement current RISC or CISC general-purpose ISAs using the standard VLIW architecture because their existing legacy code has been produced with a sequential machine in mind. All this has made VLIW machines of limited commercial interest. Nevertheless, some VLIW machines have been produced commercially [Cowell88, Rau89], and the research efforts on their VLIW compilers has proved very useful in improving the performance of compilers targeting modern pipelined and Superscalar machines [Rau93a].




*(a)*

| Clock Cycle | 0 | 1 | 2 | 3 | 4 | 5 | 7 |
|---|---|---|---|---|---|---|---|
| Functional Unit 1 | I1 | I5 | | I11 | | I14 | I16 |
| Functional Unit 2 | I2 | | I8 | | | I15 | I17 |
| Functional Unit 3 | I3 | I6 | I9 | | I12 | | I18 |
| Functional Unit 4 | I4 | I7 | I10 | | I13 | | I19 |

*(b)*

**Figure 2.3: VLIW machines. *(a)* A hypothetical VLIW machine. *(b)* The compiler can put up to four instructions into each long instruction of this VLIW machine. If one instruction slot cannot be filled, a *no-operation* (nop) instruction is used.**

### 2.1.5 Multithreaded Architectures

Like pipelining, *multithreading* realises temporal parallelism, but in a different way. A Multithreaded architecture can execute many threads (processes from different programs or the same program) in parallel with the same hardware. To do so, a multithreaded processor has multiple sets of storage positions for recording the state (also called context) of different threads. The multithreaded processor executes one or more instructions of one thread and then switches to another thread ready for execution, repeating this process continuously in a round robin or prioritised way. Thread switching can be triggered either by the clock or by dynamic events such as cache misses. By using multithreading, the utilisation of the processor hardware is increased because latencies due to cache misses, inter-instruction dependencies, etc, can be hidden by the execution of other threads.

Multithreading and multiprogramming (also called time-sharing) are similar in the way they allow different programs to execute in the same hardware. They differ, however, in the kind of latency they hide. While multiprogramming hides long latencies, such as *input/output* (I/O) latencies, multithreading hides short latencies related to instruction execution. In multiprogramming, the operating system controls process switching. This involves saving one process context in memory and bringing another from memory into the processor, which typically requires tens to thousands

of clock cycles. In multithreading, the processor has storage space for several threads and its hardware is typically able to switch between threads in one clock cycle. Nevertheless, Multithreading can be used to implement multiprogramming.

To my knowledge, multithreading was first employed in the peripheral subsystem of the CDC6600 [Thornton70], in which a single processor is connected to 10 independent ferrite-magnetic-core memories and 10 independent register files. On each clock cycle, the processor is connected to one particular register file and one associated memory. The processor executes one instruction in this clock cycle and switches to the subsequent register file-memory pair on a circular basis. Because the CDC6600's peripheral processor hardware is 10 times faster than its ferrite-magnetic-core memories, it is able to perform as if it were 10 processors instead of one, by sharing its hardware between 10 different programs. When 10 I/O programs are running simultaneously, the Multithreaded architecture of the CDC6600's peripheral processor hides the memory latency completely, making the most of the available hardware.

The basic architectural principle behind multithreading – sharing hardware resources between threads – can be associated with other parallel processing paradigms. Some researchers have proposed, for example, multithreaded multiprocessors [Smith_BJ81], while others have proposed multithreaded multiprocessors where the processors are VLIW machines [Alverson90]. Tullsen and his colleagues have proposed an advanced Superscalar architecture capable of issuing several instructions from different threads to the machine's functional units in the same clock cycle [Tullsen95]. They called the technique embodied in this architecture *simultaneous multithreading*. In Sohi's *Multiscalar* architecture [Sohi95], fine-grain tasks from a single program are defined automatically at compile time, as opposed to threads defined in the source code by the programmer or threads of different programs. These tasks are dynamically assigned to different execution contexts of the Multiscalar processor.

An intrinsic problem of Multithreaded architectures is the need for several execution contexts, which include register files, pipe registers, and sometimes caches and memories. The implementation of these contexts increases the machine design complexity substantially, which impacts on the clock cycle time, reducing the

multithreading advantage. Nevertheless, as the gap between processor and main memory speed increases, multithreading may become more important due to its latency-hiding ability.

## 2.2 ISA and Hardware Support for Exploiting ILP

Pipelined, Superscalar, Vector, and VLIW architectures exploit the ILP available in programs. The amount of ILP that can be exploited is limited by data and control dependencies between instructions (Figure 2.4). These dependencies impede:

- execution of consecutive instructions in pipelined architectures
- parallel instruction issue in Superscalar architectures
- loop vectorization in Vector architectures
- placement of more instructions into the long instructions of VLIW architectures by VLIW compilers

All this leads to poor utilisation of the available hardware – while one hardware unit is busy others may be idle, waiting for the results from the busy unit.

The ISA itself can give support to overcome dependencies by allowing direct expression of ILP, as in Vector and VLIW ISAs. However, there are other forms of ISA support for exploiting ILP, such as bits in the instructions encoding to help *branch prediction* or to express *predication*. Hardware can support ILP exploitation with, for example, *register renaming* or *dynamic branch prediction*. In the rest of this section, we discuss ISA and hardware support for ILP exploitation.

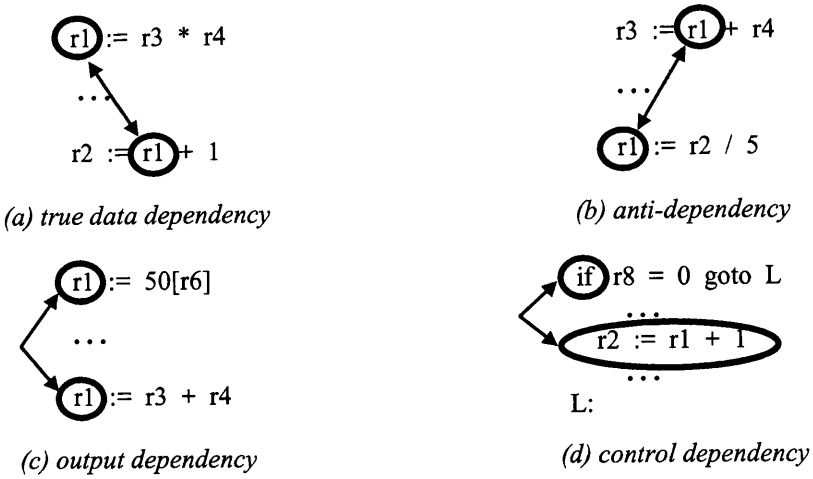



**Figure 2.4: Data and control dependencies. *(a)* Two instructions have a *true data dependency* if the result of the first is an operand of the second. *(b)* Two instructions have an *anti-dependency* if the first uses the old value in some location and the second sets that location to a new value. *(c)* Two instructions have an *output dependency* if they both assign a value to the same location. *(d)* There is a *control dependency* between a branch and an instruction whose execution is conditional on it.**

### 2.2.1 Register Renaming

Anti-dependencies and output dependencies (Figure 2.4) can be eliminated by renaming. They are called *name dependencies* because, as opposed to true data dependencies, there is no value being transmitted between the instructions. In anti-dependencies and output dependencies, it is the use of a name that causes the dependency, not the need for a previously computed value. For example, in Figure 2.4b, if the result of the second operation, "*r2 / 5*", is written into another register, say "*r5*", the anti-dependency disappears and the two statements can then be executed either in parallel or in reverse order. The same is true for Figure 2.4c if we change "*r1*" to "*r2*" in the second statement, for example. The names used for renaming cannot be in use at the point of renaming; that is, the names used cannot specify valid values that might be needed for subsequent instructions.

Although renaming can be applied to both memory and register operands it can be more easily applied to register operands, in which case it is called register renaming. Register renaming can be done either statically by the compiler or dynamically by the hardware. Superscalar machines employing the Tomasulo's dynamic scheduling algorithm [Tomasulo67] for example, perform register renaming

dynamically. In fact, the main difference between the Thornton and Tomasulo's dynamic scheduling algorithms is the ability to perform renaming – the Thornton's algorithm does not support renaming [Patterson96 (pages 240-261)].

### 2.2.2 Branch Prediction

Branch prediction [Lee_JFK84, Fisher92, Pan92, McFarling93, Nair95] can reduce the impact of control dependencies (Figure 2.4d) on machine performance. Control dependencies impact on performance because instructions that depend on a conditional branch can only be executed after the branch outcome is known. For example, if in Figure 2.1 (page 22) the instruction *I5* is a conditional branch that targets instruction *I32* at that point of program execution, the processor fetches and decodes instruction *I6* in vain and fetches *I7* in vain. This happens because the processor only learns the branch outcome when *I5* is in the execute pipeline-stage.

Branches can be predicted statically by the compiler or dynamically by the hardware. They can be predicted at compile time using knowledge about the program itself. A conditional branch at the end of a loop that is used to branch back to the beginning for another loop interaction may be predicted as taken, for example. Another alternative for branch prediction at compile time is *profiling* [Fisher92]. In this case, a version of the program is executed one or more times with typical inputs and the branches' behaviour is saved. The branch behaviour information is then used to statically predict the program branches as taken or not taken in an eventual program recompilation. Whether or not profiling is used, the compiler establishes its prediction by setting a taken-bit in the branch instruction to inform the hardware whether the branch is predicted as taken or not taken. Therefore, compiler oriented static branch prediction has to be supported by the ISA.

Usually, the hardware checks the taken-bit and moves the control-flow to the appropriate branch-target instruction when the branch instruction leaves the decode-stage of the processor's pipeline. In the example mentioned above, if *I5* has a taken-bit and it is set at true, the processor can redirect the fetch under the control of *I5* when it is in the decode pipeline-stage. In such case, *I6* is still fetched but not *I7*; *I32* is fetched instead. This saves one cycle, improving the overall machine performance. Experimental results have shown that profiling achieves prediction accuracy of 85%

on average for the integer programs of the SPEC89 benchmark suite [Patterson96 (page 176)].

Certain ISAs go beyond the static branch prediction mechanism described and use the concept of *delayed branches* to try to avoid the useless fetch of *I6* in the last example. In such ISAs, one or more instructions that follow a branch (usually just one) are unconditionally executed. These instructions are said to be in the *branch delay slots*. The compiler can use branch delay slots to accommodate useful instructions. An even more elaborate scheme adds an extra *annul-bit* to the branches encoding to say whether the instruction in the branch slot should be annulled when the branch is not taken. Using these facilities, the compiler could put *I32* after the *I5* branch in the example above, and set the annul-bit and the taken-bit of *I5* at true. This would favour a possibly more frequent case when *I5* is taken – no processor cycles would be lost in such case, provided that the *I5* target can be computed in the decode pipeline-stage. The Sparc Version 9 ISA [Patterson96 (page C-25)], for example, incorporates the annul-bit and the taken-bit in some of its conditional branches. Experiments have shown that the compiler is able to fill 79% of branch delayed slots and that delayed slots that can be annulled are not annulled 66% of the time on average in the SPEC89 integer programs [Patterson96 (page 171)].

Branches can be predicted dynamically using pure hardware mechanisms. The *Branch Target Buffer* (BTB) is an example of such a mechanism [Lee_JFK84]. The BTB is a special cache memory in the processor that stores the target address of taken branches. When a branch is executed and its outcome is taken, an entry in the BTB receives its target address; not-taken branches have any current entry in the BTB removed. The BTB entries are tagged with the branch instruction addresses. On every instruction fetch, the processor checks the BTB using the instruction fetch address and, if this address hits in the BTB, it means that the fetched instruction is a branch and that it was previously taken. The next instruction fetch, then, uses the branch target address previously stored in the BTB. If the processor fetches more than one instruction per cycle, as Superscalars do, the address stored in the BTB is the target of the first branch in the previously fetched group of instructions. The BTB address tag is the address of this group of instructions, in this case. The mentioned policies for adding, removing, and tagging BTB entries are typical; there are other policies

[Lee_JFK84].

An alternative implementation scheme for BTBs has been evaluated by Calder and Grunwald [Calder95]. In this scheme, called *next line* and *set* (NLS) predictor, the predicted next instruction cache line and set are stored either in a special cache or in the instruction cache directory (not the contents but the line and set values). This saves silicon space when compared with the BTB, because a branch target address specification requires more bits than a cache line and set, and no address tags are required, since they are already available for the instruction cache operation. Calder and Grunwald have shown that the NLS performs better than the BTB for the same silicon area in some important configurations [Calder95]. An example of processor that uses NLS is the UltraSparc-I [Tremblay96].

The BTB is a one-bit branch prediction scheme: it either contains or does not contain the target of a specific branch. A shortcoming of a one-bit predictor is that, even if a branch is almost always taken, this predictor will predict incorrectly twice rather than once when the branch is not taken and then taken. In a more elaborated branch prediction scheme, small registers record the history of previously executed branches. These registers are also saved in a cache, called the *Branch History Table* (BHT). In a predictor that follows this scheme and uses two-bit registers, a prediction may be set to miss twice before change, for example. The two-bit scheme is actually a specialisation of a more general scheme that has an $n$-bit register for each entry in the BHT. With an $n$-bit register it is possible to record the outcome of $n$ branches if we use the register as a shift register: the newest branch outcome is inserted in one side and the oldest removed from the other side. When a branch fetch address hits in the BHT, the corresponding register content is used to predict its outcome. When the branch outcome is later computed, this BHT register incorporates its outcome.

Lee and Smith [Lee_JFK84] have shown that a two-bit predictor predicts almost as well as a $n$-bit ($n > 2$) one, and thus most systems rely on two-bit predictors rather than the more general $n$-bit predictors. In these systems, the two-bit registers are used as *saturating counters* [Smith_JE81]. With a 2-bit saturating counter it is possible to record values between 0 and 3. When a branch address hits in the BHT, the corresponding counter is either incremented if the branch is taken or decremented if the branch is not taken, but saturates in 3 and 0, respectively. When the counter is

equal to 2 or 3 the branch is predicted as taken; otherwise, it is predicted as not taken. A branch predictor based in a BHT that is updated this way is also called *bimodal branch predictor* [McFarling93].

A BHT only says whether a branch is predicted as taken or not. Although very useful, this information can only produce positive results in the case of taken branches after they are decoded and their targets computed, which usually requires more than one pipeline stage. However, BHTs and BTBs can work together in a single processor: the BTB recording the target of taken branches and the BHT governing the general conjugate-predictor behaviour. This behaviour can follow several different schemes. As an example of such a scheme, instruction fetches that hit in the BTB take the branch target address stored in the BTB as the new fetch address. However, not taken branches only have their entry removed from the BTB if they are predicted as not taken by the BHT, and taken branches only have their target address stored in the BTB if they are predicted as taken by the BHT. Schemes like this are interesting because BHTs require less hardware per entry than BTBs, since they do not record branch target addresses, which allows larger BHTs than BTBs. An example of processor that uses such a scheme is the PowerPC620 [IBM94]. Other schemes may join the BTB and the BHT together to save silicon space with tags, or use different polices of replacement in the BTB, BHT, or both. As an example of BHT (or bimodal branch predictor) performance, a 4096-entry (1K-byte) direct-mapped two-bit saturating counter BHT achieves accuracy of 89% average in the SPEC89 integer benchmark suite [Pan92].

The BTB, the BHT, and their combinations are the most simple and well-known branch predictors. There are, however, many other predictors. Yeh [Yeh91, Yeh92, Yeh93a] has proposed *two-level adaptive branch predictors*, which use two tables (or caches) to predict if branches are taken or not. The first table contains shift-registers and records the history of the branches while the second contains saturating counters and determines the prediction; the second table is partially or fully addressed by the content of the first. Yeh proposed a taxonomy for two-level adaptive branch predictors to classify them according to the addressing of the two tables and the tables sizes. He and his colleagues studied the predictor accuracy with different configurations and found average accuracy for the complete (integer and

floating-point) SPEC89 benchmark suite of 96.5% with a 1-Kbyte per-address history two-level adaptive branch predictor (PAs). Pan [Pan92] studied a particular case of two-level adaptive branch predictor where a single shift-register is used to record the branch history. This shift-register is concatenated with the branch address to produce an index to the table of saturating counters. Pan has called this predictor *correlation-based branch predictor* and found average prediction accuracy of 94.1% on the SPEC89 integer benchmark suite. McFarling [McFarling93] proposed instead of concatenation, the performance of an exclusive-or (*xor*) between the shift-register and the branch address. He called this new predictor *gshare* and reported gshare prediction accuracy of 95.5% with the complete SPEC89 benchmark suite. McFarling also proposed, in the same paper, the combination of two branch predictors to exploit the different characteristics of each of them. In this case, an extra table of saturating counters is used to choose the most accurate predictor for a particular branch. He found average prediction accuracy of 96.5% with a bimodal/gshare combined branch predictor on the complete SPEC89 benchmark suite. Kaely and Emma [Kaeli91] have proposed the use of a small (less than 32 entries) *return-address-stack* specifically for predicting subroutine returns. Recent proposals of repair mechanisms of the return-address-stack contents have demonstrated accuracy of nearly 100% in subroutine return branches for the SPEC95 benchmark suite [Skadron98].

There is an enormous amount of research on dynamic branch prediction in the literature and the research interest in this field is still high. This interest exists because dynamic branch prediction has a fundamental role when used together with *speculative execution* in wide and deep pipelined Superscalar architectures [Theobald92, Wall94, Uht97].

### 2.2.3 Speculative Execution and Interrupt Handling

Instructions are said to execute speculatively when they are dependent for execution on the outcome of a conditional branch and yet are executed before the branch result is available, but do not have their output values committed to the ISA state until the outcome of the branch instruction is known. This is done to allow the machine hardware to be utilised while the branch outcomes are still not computed.

Although more frequent, branch mispredictions are similar to *interrupts* (also called exceptions or traps). Interrupts are caused by the execution of some instructions, such as load/store (page faults, access violations) or divide (divide by zero), or by events external to the processor (timer, I/O, etc). When a branch misprediction occurs, the machine must be restored to a state such that all instructions that precede the misprediction have updated the machine state, while none of those following it have. The same is true for interrupts: after an interrupt is handled, the machine must be restored to a state such that all instructions that precede the interrupt have updated the machine state, while none of those following it have. Speculative execution makes it difficult to establish the precise ISA state to return to in these cases because instructions are allowed to initiate before knowing the outcome of branches that precede them. This problem is known in the literature as the *precise interrupt* problem.

Smith and Pleszkun [Smith_JE85] described several mechanisms to implement precise interrupts. The simplest one is *in order completion.* In a machine that implements this mechanism, instructions modify the ISA state only when all previously issued instructions are known to be free of exception conditions and all previously issued branches have confirmed their targets. Another mechanism proposed by Smith and Pleszkun is the *history buffer.* The history buffer allows a machine to undo the effects of instructions executed speculatively. The history buffer is a circular list, with pointers to the top and bottom. Every time the machine issues an instruction to the functional units, it allocates a slot for the instruction at the bottom of the history buffer. When the instruction executes, the values it overwrites in the machine state (the old values) are stored into the instruction's slot in the buffer. Instructions at the top of the history buffer are removed as they complete execution. An instruction can only interrupt the machine when it arrives at the top of the buffer. On an interrupt, the machine restores the precise ISA state before the instruction execution using the old values preserved in the history buffer.

Hwu and Patt have proposed the *checkpoint repair* mechanism to recover from mispredicted branches and exceptions [Hwu87]. A processor employing checkpoint repair has a set of logical spaces, which consists of a full set of ISA registers and extra registers for recording values which result from store instruction execution. Of

all logic spaces, only one is used for current execution. The other spaces contain backup copies of previous states of the processor. At various times during execution (at every branch instruction for example), a checkpoint is made by copying the contents of the processor registers to one of these logic spaces. The logic spaces are managed as a stack; therefore, a checkpoint operation discards the oldest checkpointed state. In case of an exception or misprediction and depending on the implementation, a suitable previously saved machine state either becomes the active state or is copied to the machine state. Store instructions are managed using one of two possible mechanisms. In the first, they have their store values saved in special registers in the logical space. After each new checkpoint, the values previously saved are stored into memory. In the second mechanism, they have the values that they overwrite in the cache saved in the special registers. In case of interrupt or misprediction, the values saved are restored into the cache.

Most schemes for dealing with interrupts and branch mispredictions in the presence of speculative execution described in the literature incorporate the idea of keeping multiple copies of any overwritten ISA register or memory position. These mechanisms recover from interrupts or branch mispredictions by discarding all values produced after these events took place and restoring the ISA state to the previous values. They differ in the impact on the machine performance and in the implementation cost. Wang and Emnett [Wang_CJ93] have examined these trade-offs for the in-order completion, history buffer, reorder buffer [Smith_JE85], and future file [Smith_JE85] mechanisms. Butler and Patt [Butler93] compare the performance implications of using checkpointing with that of using either the history buffer or the reorder buffer.

### 2.2.4 Memory Disambiguation

When load/store instructions are executed out of the order specified by the programmer, special mechanisms must be used to avoid *memory aliasing*. Memory aliasing occurs when loads read from or stores write to the same memory positions that are going to be written by stores originally assigned to execute before them. The mechanisms employed to avoid memory aliasing are known as *memory disambiguation* mechanisms or *memory aliasing detection* mechanisms.

In VLIW machines, memory disambiguation is usually done at compile time, although some hardware support can be added to the VLIW machine for detecting and resolving memory aliasing between near long instructions at execution time [Colwell88]. Johnson proposed the use of a *store buffer* for dynamic memory disambiguation in Superscalar machines that execute load and store instructions speculatively [Johnson91]. The store buffer records store instructions, the store data, and the store addresses as they are computed. When a store instruction is dispatched, it is also copied to the store buffer together with the store data. After being computed, the store address is placed into the appropriated entry of the store buffer. The store then waits in the store buffer until all previously issued instructions are completed. After that, the cache write is performed and the store instruction removed from the store buffer. Load disambiguation is performed by holding the execution of load instructions until the addresses of all stores in the store buffer are computed. If a load address matches an address in the store buffer, the load can either read the data directly from the buffer or wait until all matching store-buffer entries are removed from the store buffer. Store disambiguation is performed by removing the store instructions from the buffer and writing their data into the cache in execution order. Franklin and Sohi have proposed a more elaborate memory disambiguation scheme, called *Address Resolution Buffer* (ARB) [Franklin92]. In a machine with ARB, loads are allowed to read from the data cache even when there are previous store instructions not yet issued or whose addresses are not yet computed. This is possible because the ARB records the same information recorded by the store buffer plus load addresses and the relative order between loads and stores. When a store address is computed, the ARB is able to signal any existent memory aliasing. Special recovery actions are then taken when aliasing is detected.

### 2.2.5 Predication

*Predication* is a form of ISA support for exploiting ILP. Predicated, or guarded, execution refers to the conditional execution of instructions based on the value of a boolean source operand, referred to as predicate [Park91]. Predicates are associated with instructions at compile time. An instruction with a predicate value of true executes normally, while an instruction with a predicate value of false – although

issued – does not write its results into the ISA state. Additional compare and move instructions are used to set the predicates' value at execution time. Predication can be used to remove conditional branches, particularly the hard to predict ones, such as those associated with *if – then – else* statements. The process of eliminating conditional branches utilising predicated execution support is referred to as *if-conversion*, and it was initially proposed to assist automatic vectorization of loops with conditional branches [Allen83]. Predication was also extensively used in the Cydra 5 VLIW computer [Rau89], and is a key architectural feature of the Intel IA-64 ISA (first 64-bit Intel ISA, which will supersede the Pentium IA-32 ISA) [Dulong98].

Consider, for example, the following fragment of C code extracted from the eqntott program of the SPEC92 benchmark suite:

```
if (aa == 2)
  aa = 0;
if (bb == 2)
  bb = 0;
...
```

This code fragment can be translated to the following assembly code fragment (the destination register is the rightmost, branches are not delayed, and the suffix *cc* in the *sub_cc* instructions indicate that the machine's condition codes are modified by these instructions):

```
        ld      aa(sp), r1      # load the aa variable into r1 (sp is the stack pointer)
        sub_cc  r1, 2, r0       # write to r0 have no effect
        bnz     if_2            # branch if not zero
        st      r0, aa(sp)      # r0 always read as zero
if_2:   ld      bb(sp), r1      # load the bb variable into r1
        sub_cc  r1, 2, r0
        bnz     end
        st      r0, bb(sp)
end:    ...
```

In wide and deep pipelined Superscalars, when the instruction *bnz if_2* is detected as taken and was predicted as not taken, all subsequent instructions shown above might already have been issued. In such case, all work done is discarded only

to fetch the store possibly for a second time, since it might already have been fetched.

With predication, the assembly code fragment above can be modified to:

```
        ld      aa(sp), r1      # load the aa variable into r1
        sub     r1, 2, p        # p is a predicate
        ~p, st  r0, aa(sp)      # the store is only executed if p is zero
if_2:   ld      bb(sp), r1
        sub     r1, 2, p
        ~p, st  r0, bb(sp)
end:    ...
```

The assembly code is now shorter (this is not always the case with predication) because the branches have been removed. In addition, all instructions can be issued at the same time in a Superscalar employing the Tomasulo algorithm.

### 2.2.6 Instruction Hoisting

Mahlke et al. have proposed a form of ISA support for speculative execution of instructions that can cause exceptions, such as loads, stores, and floating-point instructions [Mahlke92a]. We use here the nomenclature used by Dulong [Dulong98], who calls this ISA support *instruction hoisting*. Instruction hoisting is used to advance exception-causing instructions past conditional branches in order to improve ILP exploitation. (Instruction hoisting is also one of the key architectural features of the Intel IA-64 ISA [Dulong98].)

Hoisted instructions receive a tag at compile time that indicates to the machine hardware that they have been hoisted. The machine executes these instructions normally, but does not deliver exceptions related to them. If a hoisted instruction performs any exception-causing operation, the information related to the exception is saved in bits associated with the destination register of this instruction, and the address of the offending instruction is saved into its destination register. In the case of store instructions, the address of the offending store is saved in a modified version of the store buffer [Mahlke92a], ARB, or equivalent architectural support for store instructions.

Special instructions, called *sentinel instructions*, are used to check the exception information saved. When sentinel instructions detect exceptions, fix-up code is invoked to deal with the exceptions.

Consider the following fragment of C code:

```
if (x > y)
  k = a[x + y];
...
```

This code fragment can be translated to the following assembly code fragment:

```
        ld      x(sp), r1       # load the x variable into r1
        ld      y(sp), r2       # load the y variable into r2
        ld      a(sp), r3       # load the a pointer into r3
        sub_cc  r1, r2, r0
        ble     else            # branch if x <= y
        add     r2, r1, r4      # r4 = x + y
        ld      (r3+r4), r4     # load a[x + y] into r4
        st      r4, k(sp)       # k = a[x + y]
else:   ...
```

With hoisting, the assembly code fragment above can be modified to:

```
        ld      x(sp), r1       # load the x variable into r1
        ld      y(sp), r2       # load the y variable into r2
        ld      a(sp), r3       # load the a pointer into r3
        sub_cc  r1, r2, r0
        add     r2, r1, r4      # r4 = x + y
        ld.hois (r3+r4), r4     # hoisted load a[x + y] into r4
        ble     else            # branch if x <= y
        check   r4              # special instruction for exception check
        st      r4, k(sp)       # k = a[x + y]
else:   ...
```

The assembly code fragment is now larger, but it is not always the case, since the *st r4, k(sp)* can perform the sentinel function of the instruction *check r4* when reading *r4*, ISA permitting. In the new code fragment there is always one instruction between a load and the instruction that consumes the loaded value. This reduces the impact of load latencies in the machine performance. If the sentinel instruction detects an exception such as a page-fault, the operating system is invoked and fix-up code is used to execute the *ld.hois (r4+r2), r4* instruction again. The program execution then restarts after the *check r4* instruction.

### 2.2.7 Data Dependence Collapsing

*Data dependence collapsing* can be used to alleviate the impact of true data dependencies on ILP. Consider the example of true data dependency shown in Figure 2.4 (*a*) (page 31) and repeated below:

```
r1 := r3 * r4
r2 := r1 + 1
```

The second statement has to wait the execution of the first due to a true data dependency on *r1*. The dependency between these two statements can be eliminated, however, by executing the second in a functional unit capable of reading three inputs, and performing one multiply operation and one add operation at the same time as shown below:

```
r2 := (r3 * r4) + 1
```

This is called data dependency collapsing. Data dependency collapsing functionality has been added to some processors, such as the Intel i860 [Intel89], IBM RS/6000 [Oehler90], and the PowerPC family [Diefendorff94] in the form of multiply-add floating-point instructions. Malik and his colleagues [Malik92] have proposed and evaluated an *Arithmetic and Logic Unit* (ALU) design that allows data dependency collapsing of integer instructions. Sazeides and Vassiliadis have studied the performance potential of data dependency collapsing for improving processor performance in [Sazeides96].

## 2.3 Compiler Support for Exploiting ILP

Superscalar architectures deal with instruction dependencies in hardware level by using the Thornton, Tomasulo, or another dynamic scheduling algorithm. However, due to hardware limitations, a significant part of the ILP may be lost. Nevertheless, the compiler can help the Superscalar hardware by generating code where instruction dependencies are spaced apart, and by using predication and hoisting when there is

ISA support. VLIW architectures differ from Superscalars in their exploitation of ILP by relying only on the compiler, which produces already parallel code in the form of long instructions. In fact, the appearance of VLIW architectures was a result of the development of a compiler technique called *trace scheduling*.

### 2.3.1 Trace Scheduling

Trace scheduling was first developed as a solution to the global *microcode compaction* problem [Fisher81]. Microcode compaction is the process of combining the microoperations that compose a microprogram into microinstructions in a way that reduces the space required by the microprogram and, hopefully, the time needed for the microprogram execution. Microcode compaction is used to produce code for horizontal microprogrammable machines.

There are two types of microcode compaction: local and global. Local microcode compaction is restricted to *basic blocks* of microcode [Davidson81]. A basic block is a straight-line sequence of instructions that are not targeted by branches except at the beginning and with no branches except at the end. Basic blocks are short – typically about 5-20 instructions on average – and, because of this, there is not much scope for finding parallelism in them. Therefore, in order to make highly parallel horizontally microcoded machines worthwhile, it was important to go beyond basic blocks by using global microcode compaction techniques. Global microcode compaction techniques compact large sections of microcode containing many basic blocks. The first technique capable of doing that effectively was the trace scheduling technique [Fisher81].

The success of trace scheduling in compacting microcode for horizontally microcoded machines triggered the development of the VLIW architecture, which is basically an architecture where horizontal microinstructions are exposed to the programmer of the ISA level as long instructions. A VLIW compiler employing trace scheduling is able to find the parallelism in the program source code and express it in the form of long instructions [Colwell88, Ellis86].

A trace-scheduling compiler analyses the program source as a whole. After applying a set of classical optimisations (loop invariant motion, common sub-expression elimination, induction variable simplification, etc [Aho86]), the compiler

generates the intermediate code and builds a graph of the program with operations represented by nodes and control flow represented by edges. Using estimates of branch directions produced through heuristics, the compiler selects a path, or trace, in the program graph, likely to be followed during execution. The instructions in this trace are dealt with as if they were a single basic block, with little attention paid to branch instructions except that they are constrained to remain in their original order. Instructions are allowed to move upward and downward and to cross their original basic block limits. The compiler schedules the trace into long instructions using the list-scheduling algorithm [Adam74, Fisher81], taking into consideration data dependencies between instructions, efficient use of functional units, registers, cache ports, etc. Of course, the scheduling done without considering branch instructions can lead to inconsistencies when branch instructions leave the trace or target instructions inside the trace. To avoid these inconsistencies, the compiler adds compensation code outside the trace in all points leading to or leaving the trace, connecting it to the rest of the program. This activity, termed *bookkeeping*, repairs the inconsistencies, restoring the program semantics. After scheduling the most likely trace, the compiler selects and schedules the second most likely trace, then the third, and so on, until there are no traces left.

Trace scheduling suffers from two problems: inaccurate trace selection and code explosion. Traces are selected statically at compile time by guessing the direction of branches, which is good for some programs but overall is not very efficient [Patterson96 (page 175)]. Nevertheless, trace selection can be improved with profiling [Fisher92, Fisher93]. Code explosion is caused by bookkeeping and by code replication during motion of code downward across branches and upward across merges. Researchers have addressed this problem and proposed heuristics to circumvent it [Su84, Gross90, Freudenberger92].

### 2.3.2 Superblock Scheduling

An important issue for trace scheduling is the compiler implementation complexity incurred by the need to perform bookkeeping to maintain correct program execution after moving instructions across basic blocks. *Superblock scheduling* is a technique

derived from trace scheduling that avoids bookkeeping via *superblock* formation [Hwu93].

A superblock is a trace that has no side entrances (execution paths that lead to instructions inside the trace); that is, control can only enter from the top but may leave at one or more exit points. Superblocks are formed in two steps. First, traces are selected using execution profile information. Second, *tail duplication* is performed to eliminate any side entrances into the trace. Tail duplication is performed by copying the tail portion of the trace from the first side entrance to the end of the trace. All side entrances into the trace are then moved to the corresponding tail copy. This trace without side entrances is a superblock, also known as an extended basic block in the compiler literature. After formation, the compiler schedules superblocks using the list-scheduling algorithm.

A structure similar to the superblock is the *hyperblock* [Mahlke92b]. A hyperblock is also a block of instructions where control can only enter at the top, but may exit from one or more locations. However, unlike the superblock, predicate instructions are used within hyperblocks. Thus, a hyperblock may contain instructions from more than a single path of control. Therefore, for programs without heavily biased branches, hyperblocks provide a more flexible framework for compile-time ILP expression.

Other forms of ISA support, such as hoisting, can also be used for improving trace scheduling and its variants, superblock and hyperblock scheduling [Mahlke92a].

### 2.3.3 Percolation Scheduling

Percolation scheduling was designed as a general scheduling technique for any parallel machine, but it is particularly appropriate for VLIW machines. Percolation scheduling modifies a program little by little using four semantics-preserving primitive transformations: move-op, move-cj, delete, and unify [Nicolau85]. The input code is presented to the percolation scheduler as a graph, with nodes containing instructions and edges representing the control flow. The primitive transformations are applied to this graph, grouping independent instructions in single nodes. Move-op and move-cj move simple operations and conditional branches between adjacent

nodes, respectively. These transformations may cause replication of instructions and empty nodes. The unify transformation unifies identical operations in neighbour nodes, and the delete operation deletes empty nodes resulting from previous transformations. The use of these transformations is guided by heuristics.

The original percolation scheduling has several drawbacks. First, it assumes unbounded resources. Second, the possibility of moving an instruction in a target path of a conditional branch above this conditional branch is limited to the case in which the destination of the instruction is dead in the other path of the branch. This restricts the achievable parallelism. Third, there is no control over code explosion. Finally, it cannot schedule instructions that take more than one cycle to complete (latency larger than one cycle).

Ebcioglu and Nakatani developed a new technique, called *enhanced pipeline percolation scheduling*, which builds on the original ideas of Nicolau, tackling the percolation scheduling drawbacks [Ebcioglu89, Nakatani89, Nakatani93]. Enhanced pipeline percolation scheduling schedules code into *tree instructions*. A tree instruction implements a small binary decision tree. At each internal node of the tree, there is a conditional branch. At the external nodes of the tree, there are labels to which the tree instruction can branch. On each directed edge of the tree, there can be zero, one, or more instructions, which have no data dependencies with each other and can thus execute simultaneously. Tree instructions allow operations to be conditionally executed, depending on to where its branches are branching. During execution, only those instructions in the followed path through the tree instruction are executed. This reduces the critical path length to less than what was possible with the original percolation scheduling. The resources that a tree instruction can hold are limited in enhanced pipeline percolation scheduling.

Enhanced pipeline percolation scheduling adds renaming to move-op and move-cj. With renaming, an instruction in a target path of a conditional branch can move above the conditional branch even if the instruction destination is live in the other target path, since the instruction destination can be renamed.

Code explosion is avoided in enhanced pipeline percolation scheduling by adopting move-op and move-cj conditionally and by limiting the code area that is scheduled at any given time to a look-ahead window computed using heuristics.

Multicycle instructions are accomplished by adding $n$ - 1 "dummy" instructions with the semantic "$z \leftarrow$ *delay(z)*" after each $n$-stage pipelined "$z \leftarrow x$ *operation* $y$" instruction. Dummy instructions do not take resources during scheduling and are not saved in the final tree instructions but are only used to guide the scheduling.

### 2.3.4 Loop-unrolling

Trace scheduling, superblock scheduling, and percolation scheduling are, in their original formulation, different forms of global *acyclic* scheduling. They try to expose the ILP available in acyclic code by overlapping the execution of multiple basic blocks.

Although dynamically executed many times, cyclic code – or loops – are usually statically short and contain few basic blocks. Nevertheless, as with global acyclic scheduling, ILP in cyclic code can be obtained by overlapping the execution of multiple basic blocks. In this case, however, these basic blocks are the result of multiple iterations of the same piece of code. The most natural way of exposing these multiple *dynamic* basic blocks, that result from the execution of loops, to the compiler's scheduler is to *unroll* the body of the loop some number of times [Rau93a].

Consider the following fragment of C code, which calculates the sum of all elements of a vector:

```
sum = 0;
for (i = 0; i < a_size; i++)
    sum = a[i] + sum;
```

This code fragment can be translated to the following assembly code fragment (the compiler knows that *a_size* is larger than zero):

```
        mov     0, r2           # r2 represents the sum variable at this point
        mov     0, r1           # r1 represents the i variable at this point
        ld      a(sp), r3       # load the a pointer into r3 (sp is the stack pointer)
        ld      a_size(sp), r5  # load the a_size into r5
1.loop: ld      (r3+r1), r4     # r4 = a[i]
2.      add     r1, 1, r1       # i++
3.      sub_cc  r1, r5, r0      # write to r0 have no effect
4.      add     r4, r2, r2      # sum = a[i] + sum
5.      bnz     loop            # branch if not zero
        st      r1, i(sp)       # update i in memory
        st      r2, sum(sp)     # update sum in memory
        ...
```

The loop can be unrolled once at source level as shown below, if *a_size* is known to be multiple of 2:

```
sum = 0;
for (i = 0; i < a_size; i = i+2)
{
    sum = a[i] + sum;
    sum = a[i+1] + sum;
}
```

This unrolled loop can be translated to the following assembly code fragment:

```
        mov     0, r2           # r2 represents the sum variable at this point
        mov     0, r1           # r1 represents the i variable at this point
        ld      a(sp), r3       # load the a pointer into r3 (sp is the stack pointer)
        ld      a_size(sp), r5  # load the a_size into r5
1.loop: add     r3, r1, r6      # r6 points to a[i]
2.      ld      0(r6), r4       # r4 = a[i]
3.      ld      1(r6), r7       # r7 = a[i+1]
4.      add     r1, 2, r1       # i = i + 2
5.      sub_cc  r1, r5, r0      # write to r0 have no effect
6.      add     r4, r2, r2      # sum = a[i] + sum
7.      add     r7, r2, r2      # sum = a[i+1] + sum
8.      bnz     loop            # branch if not zero
        st      r1, i(sp)       # update i in memory
        st      r2, sum(sp)     # update sum in memory
        ...
```

The unrolled loop uses a single conditional branch and a single index computation for two iterations of original loop, and the *i+1* in the loop body

statement is a constant computed at compile time and attached to the load instruction used to load *a[i+1]*. The original loop requires 10 instructions to compute the sum of two vector elements, while the unrolled loop requires 8. In addition, if the original version is allowed to run in a parallel machine, the minimum number of cycles for two iterations is 6, as shown below:

```
1.loop: ld      (r3+r1), r4   |     add    r1, 1, r1
2.      sub_cc  r1, r5, r0    |     add    r4, r2, r2
3.      bnz     loop          |
```

While the unrolled version can complete two iterations in 4 cycles when running in a parallel machine, which shows that loop unrolling can expose a significant amount of the ILP available in loops:

```
1.loop: add     r3, r1, r6    |     add    r1, 2, r1
2.      ld      0(r6), r4     |     ld     1(r6), r7
3.      sub_cc  r1, r5, r0    |     add    r4, r2, r2
4.      add     r7, r2, r2    |     bnz    loop
```

Compilers can unroll loops automatically and, after unrolling, they can perform trace scheduling or some other form of global acyclic scheduling.

### 2.3.5 Software Pipelining

Software pipelining is another technique for exposing the ILP available in loops. Software pipelining is the software equivalent of a hardware pipeline. In a pipelined machine, several instructions in different phases of their execution can occupy different stages of the machine pipeline, while in a software-pipelined loop, several instructions belonging to different iterations can coexist in the same loop body. Thus, a compiler that employs software-pipelining scheduling interleaves instructions from different loop iterations in such way that ILP is exposed.

The fragment of assembly code shown below is a software-pipelined version of the assembly code of the original fragment of C code that computes the sum of all elements of a vector shown in Subsection 2.3.4.

```
        mov     0, r2           # r2 represents the sum variable at this point
        mov     0, r1           # r1 represents the i variable at this point
        ld      a(sp), r3       # load the a pointer into r3 (sp is the stack pointer)
        ld      a_size(sp), r5  # load the a_size into r5
        ld      (r3+r1), r4     # r4 = a[0]                 (iteration 0)
        add     r1, 1, r1       # i++                       (iteration 0)
        sub_cc  r1, r5, r0      # test a_size               (iteration 0)
1.      add     r4, r2, r2      # sum = a[i] + sum          (iteration i)
2.loop: bz      end             # branch if zero            (iteration i)
3.      ld      (r3+r1), r4     # r4 = a[i+1]               (iteration i+1)
4.      add     r1, 1, r1       # i++                       (iteration i+1)
5.      sub_cc  r1, r5, r0      #                           (iteration i+1)
6.      b       loop            # unconditional branch
end:    st      r1, i(sp)       # update i in memory
        st      r2, sum(sp)     # update sum in memory
...
```

As can be seen in the code above, some amount of start-up code has been added to guarantee the original semantics, and the loop body now has 6 instructions as opposed to 5 in the original version. However, more parallelism can be exploited; that is, more instructions can be executed in a cycle. As shown below, now the parallel version can execute two iterations in only 4 cycles (instructions to the right of the *bz end* instruction are not executed if this branch is taken):

```
1.loop: add     r4, r2, r2  |  bz    end     |  ld   (r3+r1), r4  |  add   r1, 1, r1
2.      sub_cc  r1, r5, r0  |  b     loop    |                    |
```

This is possible because, as can be seen in the sequential version of the software-pipelined loop, instructions belonging to different iterations are now part of the same loop body and can execute in parallel. This shows that software pipelining can expose a significant amount of the ILP available in loops

To implement software pipelining or loop unrolling in the compiler, several techniques must be used for generating start-up and clean-up code, for optimising the number of different iterations present in the loop body, or the number of times the loop is unrolled. Rau and Fisher [Rau93a] present a comprehensive survey of such techniques.

## 2.4 ILP Available in Programs

Several studies have been made to measure the amount of ILP available in programs [Tjaden70, Foster72, Riseman72, Nicolau84, Jouppi89, Smith_MD89, Wall91, Wall93]. Two of these studies were focused on the parallelism available inside basic blocks [Tjaden70, Foster72]. These studies have shown that parallelism within basic blocks, measured as the number of instructions that can be executed in parallel per cycle on average, rarely exceeds 3. This is unsurprising. Basic blocks are typically no more than 10 instructions long, which leaves little scope for more parallelism (measurements have shown that the average size of a basic block in the SPEC92 integer programs is 6 instructions [Patterson96 (page 171)]).

Results presented in these early studies on the ILP available inside basic blocks diminished the appeal of architectures that exploit ILP for almost ten years. However, at the end of the 70s and beginning of the 80s, research in the field of microprogramming showed that it is possible to exploit parallelism beyond basic blocks [Tokoro78, Fisher81, Tokoro81]. A study by Nicolau and Fisher [Nicolau84] found that an average ILP of 90 can be extracted from numerical programs if one has a scheduler that knows the outcome of all branches and uses a VLIW machine with unlimited resources. The good results and methodology used by Nicolau and Fisher were not new though. Riseman and Foster found an average ILP of 50 with equivalent assumptions in the 70s [Riseman72]. However, Nicolau and Fisher's results came at a time when researchers were aware that branches could be predicted with reasonable accuracy using affordable hardware [Lee_JFK84], which would pave the way for the exploitation of the ILP available beyond basic blocks.

One of the most detailed studies on the ILP inside and beyond basic blocks has been done by Wall [Wall93]. Wall studied the effect of various machine parameters on the amount of ILP that can be extracted from programs, including:

- Instruction window size – the instruction window holds a certain number of instructions that are made available for issuing.
- Number of instructions issued per cycle – the instructions in the instruction window that can be issued in parallel to the functional units of the machine each clock cycle.
- Number of renaming registers.

- Type of memory alias analysis.
- Type of branch prediction.
- Branch fanout – the number of paths generated by conditional branches that can be executed speculatively.

Wall has shown that all these parameters have a strong effect on the amount of ILP that can be extracted from programs. He has also shown that none of these parameters should be considered singly – the overall performance of a machine is determined by how these parameters are chosen as a group.

Wall's results confirm Nicolau and Fisher, and Riseman and Foster's results – average ILP of 90 and as high as 500 are reported in [Wall93]. However, according to Wall's results, machines that can harness this amount of parallelism are impossible to build. They would require perfect branch prediction, perfect memory alias analysis, and an immense amount of hardware resources. Nevertheless, ILP in the range of 4.3 – 8.7, average of 6, are reported in [Wall93] with a machine that may be built in the near future. This machine would fetch up to 64 instructions each cycle from memory to an instruction window of 64 instructions. Up to 64 instructions could be issued to 64 fully pipelined functional units each cycle, and 256 renaming registers could be used to remove output or anti-dependencies between them. Memory alias analysis would be perfect (this can be achieved with an instruction window of limited size) and the branch prediction hardware large but possible to implement. Up to four different paths can be executed speculatively. An average ILP of 6 may still be optimistic for the machine described, since, in the experiments that produced this ILP, all instructions were able to complete execution in a single clock cycle.

Although important for the understanding of the nature of the performance bottlenecks in the exploitation of ILP available in state-of-the-art code, experiments that attempt to measure the maximum ILP available in programs should be examined with care. This is because these experiments cannot consider the effect of yet-unknown processor architectures and compiler optimisation techniques.

## 2.5 Historical Perspective of ILP Exploitation

In this section, we revisit a discussion presented in [Rau93a], in order to place the architecture of this thesis in the context of processor technology development from the 70s to the present moment.

### 2.5.1 Late 70s

The late 70s witnessed the emergence of VLIW architecture. In many ways, VLIWs were a natural outgrowth of horizontal microarchitectures [Salisbury76], the first ILP technology. Their appearance was triggered by the same changes in semiconductor technology that had such a profound impact upon the entire computer industry in the 80s – the development of fast and cheap *Very Large Scale Integrated* (VLSI) circuit technologies. For sequential processors, as the speed gap between writeable and read-only memory narrowed, the advantages of a small, dedicated, read-only microcode memory began to disappear. One natural effect of this was to diminish the advantage of microcode – it no longer made as much sense to define a complex ISA as a compiler target and then interpret this with a very fast read-only microcode. Instead, the vertical microcode interface was presented as clean, simple compiler target – the RISC ISA.

### 2.5.2 Beginning of the 80s

In the 80s, due to the development of VLSI technologies, the general movement of microprocessor products was towards the RISC concept. The availability of VLSI technologies was having a somewhat different effect upon horizontally microprogrammed processors though.

During the 70s, a large market had grown in specialised signal processing computers. Not aimed at general-purpose use, these machines hardwired fast fourrier transform and other important algorithms directly into read-only horizontal microcode, gaining tremendous advantage from the microinstruction-level parallelism available there. When fast, writeable memory became available, some of these manufacturers, most notably Floating-point Systems [Charlesworth81], replaced the read-only microcode memory with writeable memory, giving users

access to ILP in far greater amounts than early Superscalar processors had. These machines were extremely fast, the fastest processors by far in their price range, for important classes of scientific applications. However, despite attempts on the part of several manufacturers to market their products for more general, everyday use, they were almost always restricted to a narrow class of applications. This was caused by two factors. First, by the lack of good system software which, in turn, was caused by the idiosyncratic architecture of processors built for a single application. Second, by the lack at that time of good code generation algorithms for ILP machines with that much parallelism. As with RISC, the crucial step for these architectures was to present a simple, clean interface to the compiler. However, in this case, the clean interface was horizontal, not vertical microcode. This style of architecture, which exposes a horizontal microarchitecture to the ISA level, was denominated VLIW. Code generation techniques, some of which had been developed for generating horizontal microcode, were extended to these general-purpose VLIW machines so that the compiler could specify the parallelism directly [Fisher81].

### 2.5.3 Late 80s

In the second half of the 80s, VLIW machines were offered commercially in the form of capable, general-purpose machines. Some start-up companies – Multiflow and Cydrome for example – built VLIWs with varying degrees of parallelism [Colwell88, Rau89]. These companies demonstrated that it was possible to build practical machines that achieve large amounts of ILP on scientific and engineering codes. Although for various reasons none was a lasting business success, several major computer manufacturers acquired access to the technologies developed by these start-ups. Furthermore, many of the compiler techniques developed with VLIWs in mind started to be used in compilers for Superscalar machines as well.

### 2.5.4 Beginning of the 90s

Processor designers in the 90s have more silicon space on a single chip available to them than a RISC processor requires. Because of this, virtually all manufacturers began to add some degree of Superscalar capability to processors and some started to

investigate VLIW processors as well. By the middle of the 90s, virtually all new processors of major manufacturers embodied some degree of ILP.

### 2.5.5 Late 90s

The computer industry saw a fantastic growth in the 90s due to widespread use of personal computers. Consumer demand forced processor manufacturers to deliver one minor processor version half yearly and one major every two years. Today, it is common to see two or more hardware generations interpreting the same ISA in the same room. Although ILP had never been pursued as much as in the late 90s, due to the object code compatibility problem, there was no place for VLIW architectures except in specialised segments where the code is kept in read-only memories or seldom changed [Wolfe97]. Backward code compatibility became fundamental for survival in the processor market – the personal computer user wanted new, faster processors but capable of running already available software.

The availability of enormous amounts of transistors and the continuous search for new architectures for exploiting ILP created a nurturing environment for radically new architectural concepts [Burger97]. The DTSVLIW is a result of these historical conditions. It consumes extra silicon area with its VLIW Cache and wide VLIW Engine, but, as we shall see in Chapter 6, it can harness a significant amount of ILP with feasible machine configurations, while allowing for backward object code compatibility.

# Chapter 3

# Related Work

The DTSVLIW architecture exploits the ILP available in programs by executing VLIW code on its VLIW Engine. This VLIW code is dynamically generated by hardware from original sequential code that is fetched as usual from the memory. This process solves the VLIW object code compatibility problem and allows the VLIW Engine parallelism to be utilised most of the time. All this is possible due to the code execution locality that exists in ordinary programs, which is capitalised upon by the DTSVLIW.

In this chapter, we discuss aspects of the DTSVLIW architecture that are related to previous research on microcode scheduling and VLIW architectures. In addition, we discuss research that either tackles the VLIW object code compatibility problem or proposes mechanisms to exploit code execution locality.

## 3.1 Related Work on Microcode Scheduling

The DTSVLIW schedules instructions using a simplified version of a single-pass algorithm that has historically been used for microcode compaction – the *First Come First Served* (FCFS) algorithm [Davidson81]. Microcode compaction is the process of scheduling microoperations into microinstructions in a way that minimises the space required by the microprogram and, hopefully, the time needed for the microprogram execution. DeWitt [DeWitt76] has shown that microcode compaction is an NP-complete problem; therefore, a single-pass algorithm such as the FCFS is a

cost-effective solution only. Nevertheless, the FCFS algorithm can achieve optimum execution-time scheduling, as shown by Davidson et al. [Davidson81].

The FCFS algorithm was first proposed by Dasgupta and Tartar [Dasgupta76]; however, the description presented here is based on that of Davidson et al. The original algorithm operates over a list of microoperations coming from a *straight-line microcode segment*, which is a sequence of microoperations containing no branch microoperation except perhaps one at the end, and no entry point except at the beginning (a straight-line microcode segment is a basic block of microoperations). The algorithm takes microoperations from the straight-line microcode segment and groups them to form microinstructions with multiple microoperations. In the DTSVLIW, however, we use the FCFS algorithm to schedule instructions coming from a trace (produced dynamically during program execution) into long instructions (equivalent to microinstructions). Because we are using a trace, where the direction of each branch is fixed, we are able to schedule instructions past conditional and indirect branches by renaming these instructions. The details of the FCFS algorithm, bounded to work within a list of long instructions of size LIST_SIZE, are as follows.

1. Take one instruction from the trace and, if there is no dependency, add it to the last long instruction of an initially empty list of long instructions. If there is any dependency, add one empty long instruction to the end of the list and add the instruction to this long instruction. If this makes the list longer than the LIST_SIZE, save the previous list's contents and start a new list with a single long instruction containing the instruction.
2. Search the list of long instructions and find the earliest long instruction where flow dependencies and resource dependencies allow the added instruction to be placed. Rename the instruction if appropriate, and put it in the long instruction found.
3. If the added instruction cannot move up due to the lack of a suitable slot in any long instruction above the one in the tail and the list is smaller than LIST_SIZE – 1, add one long instruction at the top of the list and put the new instruction there. (A new long instruction is added to the top to allow any subsequent instruction that may be data dependent on the just added

instruction to be added to an already existing long instruction, instead of forming a new long instruction at the bottom of the list.)

4. Go to step 1.

There are two differences between the DTSVLIW and the FCFS algorithms. First, the DTSVLIW does not implement step 3 of FCFS; i.e., the DTSVLIW algorithm never adds long instructions at the top of the scheduling list but only at the bottom. Second, the DTSVLIW algorithm only moves up an instruction if there is a slot available in the next long instruction on the list. This can cause premature installing of instructions that could be moved to a long instruction two or more entries up in the list, limiting the code density and the achievable parallelism. However, it facilitates a pipelined implementation of the DTSVLIW algorithm. The DTSVLIW scheduling algorithm is detailed in Subsection 4.1.1, while its pipelined implementation is described in Section 4.6.

## 3.2 Related Work on VLIW Architectures

A trace-scheduling compiler selects traces in a program and schedules these traces statically, using heuristics or profiling (Subsection 2.3.1). Different from a VLIW compiler, a DTSVLIW machine performs dynamic trace scheduling instead of static trace scheduling.

In a DTSVLIW machine, the execution trace produced by the Primary Processor feeds the Scheduler Unit, which schedules the instructions into blocks of long instructions and saves these blocks into the VLIW Cache. Each block of long instructions may encompass many basic blocks. Scheduling is performed in a way that allows any branch inside any block to exit without side effects. The unique entry point of each block is its first instruction. Therefore, if a path in the program leads to an instruction inside an existent block, this path will cause the scheduling of a new block. This is equivalent to tail duplication, which is performed during superblock formation by a superblock-scheduling compiler (see Subsection 2.3.2). In fact, what the DTSVLIW's Scheduler Unit really does is superblock scheduling. However, different from the compiler, which performs static superblock scheduling, the DTSVLIW hardware performs dynamic superblock scheduling. This allows the

DTSVLIW to use dynamic information about the branches behaviour not available to the compiler. The compiler selects traces statically and these traces must be suitable for all input data sets of the program. In contrast, a DTSVLIW machine performs dynamic trace selection and as such can achieve good performance for all input data sets.

A core scheduling operation performed by the DTSVLIW is the *move up* operation, which moves instructions through a list of long instructions inside the machine to compact long instructions (see Subsection 4.1.1). This operation is similar to the *move-op with renaming* operation of the *enhanced pipeline percolation scheduling* technique [Nakatani93]. However, its application is different, reflecting its different purposes: move-op was designed for scheduling during compile time, whereas move up was designed for scheduling during execution time. The application of the move-op operation requires the evaluation of all execution paths that transverse the instruction being moved; on the other hand, the application of the move up operation requires the evaluation of the current trace only. The move-op operation is applied in a sequential fashion by the compiler, while the move up operation is applied in a pipelined parallel fashion by the DTSVLIW hardware. The application of the move-op operation can cause the generation of a new long instruction, whereas the move up operation never causes this.

The DTSVLIW schedules the code trace observed during execution into long instructions, all with the same format (Subsection 4.2). This format is similar to the tree instruction described by Ebcioglu [Ebcioglu88].

The tree instruction was proposed to hold VLIW code scheduled by VLIW compilers. Although suitable as a target for VLIW compilers, the tree instruction has some drawbacks. First, it has many possible next long instruction targets. For this reason, a VLIW machine using it has to compute all branch outcomes and only after that select the next long instruction. This can pose difficulties to the production of a short cycle time in the circuitry responsible for long instruction fetch. Second, the tree instruction needs physical space to accommodate the two target addresses of all branches it holds. Finally, the tree instruction holds code for both paths determined by a branch, but while only one path is actually followed during execution, the other path is still executed with its results discarded. This can cause ineffective

exploitation of machine parallelism.

In contrast to the tree instruction, the long instruction format of the DTSVLIW has a default direction for each branch, which can be the taken or not taken path. Its long instructions hold instructions in the default path only, and each long instruction has a default next long instruction target, which simplifies the implementation of fast fetch hardware. The long instruction format of the DTSVLIW has only one branch target for each branch it holds, which is used when the branch does not determine the default direction. The long instruction format of the DTSVLIW can be viewed as a special case of the tree instruction, although it has not been derived from this approach. The DTSVLIW methodology is particularly suitable for a VLIW machine that executes scheduled trace code.

## 3.3 Related Work on Tackling the VLIW Object Code Compatibility Problem

VLIW machines potentially provide the most direct way to exploit ILP [Nicolau84, deSouza93, deSouza97]; however, for widespread use as general-purpose processors, the VLIW object code compatibility problem has to be overcome. This problem can be understood by examining the following example.

The fragment of code below is the same software pipelined loop shown in Subsection 2.3.5.

```
1.loop: add     r4, r2, r2  |  bz    end     |  ld  (r3+r1), r4  |  add  r1, 1, r1
2.      sub_cc  r1, r5, r0  |  b     loop    |                   |
```

This fragment of code can run in a VLIW machine where loads complete execution in one or two cycles and all other instructions complete execution in one cycle. However, if this code were to execute in a machine where loads have latency of three cycles, the *add r4,r2,r2* instruction above will use the value loaded two iterations behind instead of one iteration behind. In this case, the result of the loop execution will be incorrect. In addition, this code does not fit in VLIW machines with 3-instruction long instructions.

Software [Sites93, Conte95b, Hookway97, Ebciouglu97] and hardware

[Rau93b, Franklin94, Nair97, Banerjia98] techniques have been proposed to get over this problem.

### 3.3.1 Software Approaches

The simplest software technique to overcome the VLIW object code compatibility problem is off-line recompilation of the programs' source code. Recompilation yields appreciable performance because the compiler can expose the ILP in the source code to different VLIW hardware implementations. The drawback of this approach is that it is awkward to use – machine upgrades require either recompilation of all installed software, whose source code may not always be available, or reinstallation of a complete set of new binaries. Binary translation [Sites93] is a variant of this technique that can be performed without the source code, but machine upgrades still require either translation or reinstallation of the binaries. Alternatively, interpreters can be used to emulate different architectures at run-time; however, this approach usually suffers from poor performance. Binary translation and emulation can be combined, however. For example, the Digital FX!32 is a software system that performs emulation and incremental binary translation, enabling Win32/Intel IA-32 ISA applications to run on Windows NT/Alpha ISA platforms [Hookway97, Chernoff98]. In the FX!32, the first time the code is run, it is emulated. The binary translation is done incrementally in the background after the program execution, and only executed parts of the code are translated. To implement this, a database of translated binaries is maintained by the system: when a significant number of new fragments of code are executed, translations are generated and added to the database. The FX!32 system requires no user interaction.

The FX!32, like perhaps any other systems that perform emulation and posterior binary translation, has some drawbacks. First, the fact that any new code is initially emulated creates two patterns of system performance easily perceived by the user: one slow when emulating and one fast after code translation. In the FX!32, emulation is 10 times slower than translated-code execution [Hookway97]. Second, the operating system binaries used in the Alpha system are not the same as those in an Intel system: the Windows NT operating system must be recompiled for the Alpha platform. This is also true for any device driver. Therefore, either their source code

must be available or their interface must be precisely documented, which may not always be the case (the FX!32 has many dependencies with undocumented Windows NT features [Chernoff98]). Third, the profiling information cannot contain conditional branch directions since it would generate large profile files and slow down the emulation. This results in poor scheduling of the translated code for parallel machines such as VLIWs. Fourth, the disk space required for binaries is almost or even more than doubled. Finally, system development for the Intel ISA cannot be performed in the Alpha system, since debugging is too much impaired due to its frequent need of the Intel ISA's state [Hookway97].

Emulation and binary translation are powerful tools for migrating applications from one ISA to another. They have been successfully used in the past to migrate VAX ISA applications to Alpha ISA [Sites93], for example. However, we believe they are not competitive concepts to make families of backward incompatible VLIW systems viable due to the mentioned drawbacks.

*Dynamic Rescheduling*, proposed by Conte and Sathaye [Conte95b], is another software technique which can be used to overcome the VLIW object code compatibility problem. When a binary incompatible VLIW program is invoked in a system that implements dynamic rescheduling, the operating system translates its first page to a new page, which is binary compatible with the system's hardware. This process is repeated each time a new page fault occurs, and provides correct execution over different VLIW machine generations.

In Conte and Sathye's proposal, the pages of the original program are rescheduled to reflect the new latencies and parallelism of the hardware but the translated pages must be the same size as the original ones. However, the new schedule may grow larger due to the insertion of empty cycles, or may have its size reduced due to the deletion of empty cycles from the old schedule. To avoid this, Conte and Sathye have suggested a special encoding which hides the nop instructions, ensuring that code rescheduling within basic blocks does not trigger any code size changes. Scheduling beyond basic blocks – or speculative code motions – introduces two problems, however. The first problem is incorrect translation due to moving code above instructions that are the targets of instructions from other pages. To avoid this, Conte and Sathye suggested the use of superblock or hyperblock

scheduling across all families of dynamic-scheduling compatible VLIW systems. Because superblocks and hyperblocks have a unique entry point at the top of the block, speculative code motions are allowed within them; however, code generation must be guaranteed not to make any superblock or hyperblock span page boundaries. A second problem is caused by the addition of new code that might be necessary to undo effects of speculative code motion. To circumvent this problem, code motion is limited to those that do not require code expansion. However, if the rescheduler does this blindly, its opportunity to expose more ILP to a wider VLIW hardware is restricted. Thus, in a dynamic rescheduling system implementation, the compiler saves the live-out set information for each branch (storage positions written within the superblock/hyperblock that are read afterwards) in the program object file [Conte95b]. This information is used during rescheduling to improve the scheduled-code parallelism in the face of the no-code-increasing limitation.

Dynamic rescheduling has some drawbacks. First, the time spent rescheduling pages on each page fault is significant. Conte and Sathye have found that around 50000 VLIW-machine cycles are necessary for rescheduling a page, which represents about 20% of the cost of a page fault. To reduce the impact of the rescheduling time on performance, they suggest the use of *a persistent rescheduled-page cache* for saving previous page translations across execution of different programs [Conte96]. This reduces the impact of the rescheduling-time on performance but adds cost in the form of disk space. Second, the special encoding suggested in [Conte95b] may prohibit the use of new instructions or interesting new hardware features, such as more registers, because more bits might be needed for encoding, which can make the code larger. Third, software pipelined loops pose difficulties to the rescheduler, to which no solutions were presented in [Conte95b] or [Conte96]. Fourth, the inability to use speculative code motions that increase code size is a strong scheduling limitation if VLIW machines with large differences in the degree of parallelism are considered. Fifth, due to the large number of branches in programs, to add the live-out sets for each branch in the object code may increase code size substantially. Sixth, the operating system and device driver binaries used in one VLIW system should not be the same as those in another system for performance reasons. Therefore, the operating system should be recompiled for each VLIW platform, as in

the FX!32 system. Finally, and perhaps most important, the rescheduling (and the original scheduling performed by the compiler) is done statically within page boundaries, not taking into consideration the dynamics of branches during execution. This limits the amount of ILP that can be achieved with this technique even if profiling is used.

Ebcioglu and Altman [Ebcioglu97], with their *Dynamically Architected Instruction Set from Yorktown* (DAISY) machine, have extended the concept of dynamic rescheduling to *dynamic compilation*, allowing VLIW hardware to emulate a generic ISA. In a DAISY machine, every time an instruction page fault occurs, a *Virtual Machine Monitor* (VMM) is invoked. The VMM reads the original ISA instructions from memory and translates then to simpler operations (if required). These operations are scheduled into long instructions, and the resulting VLIW machine code is saved as a new page in a portion of main memory not visible to the original ISA.

Different to dynamic rescheduling, in dynamic compilation translated pages are $N$ times larger than the original ISA pages, where $N$ should be a power of 2 (Ebcioglu and Altman have suggested $N = 4$ [Ebcioglu97]). This allows the code size to increase as a result of the dynamic compilation. The virtual address space of DAISY is divided in three parts: the first for the emulated ISA, the second for the VMM, and the third for the translated code. An original ISA code page at address $n$ has its translation at DAISY's virtual address $n \times N + VLIW\_BASE$, where VLIW_BASE is the address of the beginning of the translated code part of DAISY's vitual address space.

The VLIW machine uses special branches to move the control flow from one translated page to another translated page. However, not all long instructions in translated pages are valid targets for such branches. A long instruction fetch, resulting from a cross-page branch, has to check a valid entry bit in the long instruction fetched. Nevertheless, a VLIW cross-page branch can branch to an invalid entry point in another translated page, in which case an invalid entry point exception occurs and the VMM retranslates the target page.

In the Ebcioglu and Altman experiments with DAISY reported in [Ebcioglu97], they found that their DAISY implementation required an average of

4315 operations to compile each original ISA instruction. If the DAISY's VMM executes an average of 4 operations per cycle during dynamic compilation and each page of code of the original ISA can hold 1024 instructions, more than one million cycles are required for each page compilation ($\frac{4315 \times 1024}{4}$). (1024 instructions per page is a typical value for a RISC ISA, such as the PowerPC ISA used in [Ebcioglu97], i.e. 4096-byte pages and 32-bit instructions.) This is much more than that required per page translation in dynamic rescheduling (50000 cycles per page), but this is to be expected, since in this case dynamic compilation is performed. However, according to Ebcioglu and Altman, the DAISY dynamic compiler has performance 20% inferior to a traditional static VLIW compiler. Our results show that the DTSVLIW performs better than a pure VLIW executing code produced by a static VLIW compiler and as such should perform significantly better than DAISY (see Section 6.4).

Kelly et al. have been granted a patent for a memory controller which gives support for dynamic compilation [Kelly98]. In their invention, instructions from a generic ISA are translated by a software translator into long instructions and saved in a translation buffer, which may be implemented as a data structure in memory or as hardware cache. As in DAISY, the translator and the translated code run on a VLIW machine. Translations are built incrementally, starting and finishing at basic block boundaries. However, because the translator does not schedule instructions across basic blocks and does not consider the dynamics of branches, the performance of a system implemented according to the patent should be inferior to that of a DAISY machine.

Dynamic rescheduling and dynamic compilation rely on the ability of a software system to translate code rapidly and on the reusability of this code. However, since they are implemented in software, the cost of the translation is high. In addition, these approaches do not take into consideration the dynamic behaviour of branches and perform static-code scheduling only, although done during the execution of the program. The DTSVLIW performs dynamic trace scheduling or, more specifically, dynamic superblock scheduling, learning the dynamic branch behaviour and taking advantage of it to achieve performance. Our results show that the DTSVLIW performance is better than that achieved with a pure VLIW executing code produced by a VLIW compiler (see Section 6.4), and we believe that the

DTSVLIW can achieve even better performance if compile-time scheduling specially designed for it is employed.

### 3.3.2 Hardware Approaches

Rau [Rau93b] proposed a new type of VLIW machine, named *Dynamically Scheduled VLIW* (DSVLIW), which tackles the VLIW software compatibility problem at the hardware level via *split-issue*. The DSVLIW compiler generates VLIW code scheduled for a specific VLIW ISA with fixed long instruction width and functional units' latency. During program execution, after the decoding of each long instruction, the DSVLIW machine splits each instruction member of a long instruction in two components: *phase1* and *phase2*. The phase1 component is the original instruction with its destination renamed, while phase2 is a copy instruction copying the phase2 result to the original instruction destination. Both these components can be dispatched simultaneously to reservation stations of functional units. Once execution of phase1 finishes, the reservation station with phase2 receives the result. The execution of each original instruction is completed after the execution of phase2, which can be done in just one more cycle. This mechanism allows a DSVLIW machine with functional unit latencies different from those assumed by the compiler to execute the VLIW ISA code.

The DSVLIW architecture allows fetch and split-issue of more than one long instruction per cycle. Therefore, this architecture solves the VLIW object code compatibility problem since machines with different functional unit latencies and different degrees of parallelism can execute the same VLIW ISA code. However, the DSVLIW architecture cannot be used to implement an existent sequential ISA. In addition, it requires dynamic scheduling hardware in the main data path of the machine, which can have a negative effect on the clock cycle time.

Franklin and Smotherman [Franklin94] proposed the use of a *Fill Unit* [Melvin88] to compact a dynamic stream of scalar instructions into long instructions. Their Fill Unit accepts decoded instructions from the machine decoder, compacts them into a long instruction, and saves the long instruction in the *shadow cache*. At the same time, the Fill Unit sends this long instruction to the functional units for execution. Fetch accesses hitting the shadow cache provide long instructions directly

to the functional units.

The Franklin and Smotherman's Fill Unit does not rename registers and works within a window of one long instruction only. For these reasons, it cannot exploit ILP extensively.

Banerjia and his colleges presented a processor architecture similar to the Franklin and Smotherman proposal, called *Miss Path Scheduling* (MPS) architecture [Banerjia98]. The main difference between the two proposals is that MPS schedules blocks of long instructions as opposed to a single long instruction. In MPS, scheduling hardware is placed between the instruction cache and the next level of memory. Instruction scheduling is performed at instruction cache misses and the blocks of long instructions formed are saved in a special instruction cache [Banerjia98]. Once in the cache, long instructions are fetched and issued to a VLIW core.

In a machine implementing MPS, scheduling is performed using two tables: the register def-use table, which records reads and writes in registers; and the reservation table, which records the availability of functional units. Renaming is not performed; therefore, anti and output dependencies impede parallel and out-of-order execution of instructions. Banerjia and his colleges have estimated that four cycles are required to schedule each instruction, as the register def-use table must be read, a starting point in the reservation table computed, new table values computed, and the tables must be updated with these values. They have shown a way in which the whole process can be pipelined, although dependencies between pipe stage can happen and may reduce the scheduling pipeline performance. Nevertheless, more than one instruction each four cycles can be scheduled with pipelining.

Speculation is allowed in MPS. However, a reorder buffer with a future file must be used to prevent incorrectly speculated instructions from retiring their results. In order to decide which instruction to speculate, branch prediction (dynamic, static, or profiling) is used. To calculate the target address of branches, the instructions are decoded in the scheduling pipeline. Indirect branches cannot be resolved without reading ISA registers contents. Because of this, when an indirect branch is found, scheduling stops.

MPS has three main drawbacks. First, instruction cache miss penalty is

increased in a MPS machine, since instruction scheduling takes at least one cycle per instruction and no useful execution is performed during scheduling. Second, MPS machines do not rename registers, which can have a severe impact on scheduled-code parallelism. Third, MPS machines perform static scheduling only. Although dynamic branch prediction can be used during scheduling, instructions are scheduled at instruction cache misses and are not likely to have been executed before. Therefore, dynamic branch behaviour information is not likely to be available at scheduling time.

Nair and Hopkins [Nair97] suggested a VLIW based machine organisation named *Dynamic Instruction Formatting* (DIF), which also follows the Franklin and Smotherman proposal. The DIF machine incorporates two engines: the VLIW Engine and the Primary Engine. The latter is a simple processor, less aggressive in exploiting parallelism, which executes instructions of a generic ISA when first fetched. Simultaneously with the execution of a code sequence, this engine reformats (schedules) the code, generating groups of long instructions as opposed to a single long instruction. These groups, which can encompass many basic blocks, are saved in a special cache – the DIF Cache. Following accesses to the same sequence will hit the DIF Cache, and the long instructions fetched will be executed by the VLIW Engine.

In a DIF machine, the instructions are executed during scheduling, therefore useful execution occurs during scheduling time. Register renaming is performed and the dynamic branch behaviour is recorded into the scheduled blocks. This allows for more parallelism than the MPS and all previously mentioned proposals. Table 3.1 compares the DIF with some of the approaches to circumvent the VLIW object code compatibility problem discussed in this section.

The DTSVLIW architecture is a variant of the DIF architecture. We show in this thesis similar or better performance than the DIF implementation proposed by Nair and Hopkins, but with a simpler architecture that should be much easier to implement (see Section 6.3). The DTSVLIW differs from the DIF in its register renaming mechanism, VLIW engine register access, communication between the cache that stores the long instructions and the VLIW Engine, and most importantly in the instruction scheduling algorithm. The DTSVLIW was first presented in

[deSouza98a]. Preliminary DTSVLIW SPECint95 performance measurements have been shown in [deSouza98b]. A detailed description of the DTSVLIW and the differences between DTSVLIW and DIF was first presented in [deSouza99a]. The impact of multicycle instructions on the performance of the DTSVLIW was first presented in [deSouza99b], and the effectiveness of the DTSVLIW instruction-scheduling algorithm in [deSouza99c].

**Table 3.1: Features implemented by different approaches for tackling the VLIW object code compatibility problem**

| | Emulation plus Binary Translation | Dynamic Rescheduling | Dynamic Compilation | DSVLIW | Fill Unit | MPS | DIF & DTSVLIW |
|---|---|---|---|---|---|---|---|
| Execution of sequential ISA code | ✔ | | ✔ | | ✔ | ✔ | ✔ |
| Scheduling across basic blocks | ✔ | ✔ | ✔ | ✔ | | ✔ | ✔ |
| Scheduling considering dynamic branch behaviour | | | | | | | ✔ |
| Register renaming | ✔ | ✔ | ✔ | ✔ | | | ✔ |
| Useful execution during (re)scheduling | | | | ✔ | ✔ | | ✔ |

## 3.4 Related Work on Exploiting Code Locality

The DTSVLIW architecture takes advantage of code execution locality for exploiting the ILP available in programs. However, other architectures described in the literature take advantage of similar code characteristics with the same purpose. In this section, some of these architectures are described and compared with the DTSVLIW.

### 3.4.1 Trace Cache

In the integer programs of the SPEC92 benchmark suite, on average 19% of the executed instructions are branches [Patterson96 (page 105)] and on average 62% of them change the control flow [Patterson96 (page 166)]. This means that about 12% of the instructions executed in these programs change the control flow – almost one in eight. Current Superscalar machines fetch up to four instructions each clock cycle. For the next generation, it is going to be possible to fetch eight or more instructions each cycle. This means that almost every fetch will contain a branch that will change the control flow. Since these branches are distributed evenly throughout the address space, many of these fetch cycles (almost half for an 8-wide fetch) will be only partially effective. In addition, instead of incrementing the program counter to the

next fetch address, Superscalar machines capable of fetching eight instructions per cycle will have to find the target address of a branch (possible more than one) almost every cycle.

Several high bandwidth fetch mechanisms based on the conventional instruction cache have been proposed [Conte95a, Seznec96, Yeh93b]. In such mechanisms, on every cycle instructions from non-contiguous locations in the instruction cache are fetched and assembled into dynamic sequences using information collected by dynamic branch predictors. To do this, branch target tables are inspected and pointers are generated to all non-contiguous instruction blocks. A moderately to highly interleaved instruction cache is accessed and provides multiple lines simultaneously. These lines are aligned by an alignment network, which then sends the instructions to the decode stage of the Superscalar processor.

The disadvantage of these high bandwidth fetch mechanisms is their complexity. Sophisticated dynamic branch predictors, interleaved multiport instruction caches, and complex alignment networks are required to make them work. The *Trace Cache* architecture, on the other hand, avoids this complexity by caching dynamic instruction sequences, rather than only the information for constructing them [Rotenberg96]. A diagram of the Trace Cache architecture is shown in Figure 3.1.

Machines employing the Trace Cache architecture take advantage of code execution locality to achieve performance. A machine that follows this architecture fetches instructions from the instruction cache and attempts to schedule them across multiple functional units using, for example, the Tomasulo's algorithm [Tomasulo67]. These instructions are then grouped by a Fill Unit [Melvin88] and placed in a trace cache, which stores them in execution order, as opposed to the static order determined by the compiler. On an instruction fetch, the trace cache will provide a line of instructions if available. This line can encompass more than one line from the instruction cache through merging of lines affected by partial fetches caused by taken branches: this increases instruction bandwidth and throughput.

The Trace Cache architecture has attracted significant research interest due to its potential for supplying enough instructions to make aggressively parallel Superscalar machines viable, and several aspects of it have been studied recently

[Friendly97, Jacobson97, Patel97, Patt97, Rotenberg97, Smith_JE97, Vajapeyam97, Friendly98, Patel98, Patel99, Rotenberg99].

The Trace Cache architecture is an enhanced Superscalar architecture. Therefore, it has the same dynamic scheduling overheads of Superscalars (Subsection 2.1.2). These dynamic scheduling overheads can be particularly severe in aggressively parallel Superscalars, and may substantially lengthen their clock cycle time.

According to Hara et al. [Hara96], logic fan-out and wire delays are the most important scheduling overheads of aggressively parallel Superscalar-like machines. The main fan-out overheads are caused by the logic that forwards the functional units' results to all instructions in the instruction window or reservation stations of the machine, the bypass logic. The main wire delay overheads are caused by the long wires necessary to connect these functional units to the various instructions in instruction window (or reservation stations), or bypass wire delay. In the near future, wire delays are expected to dominate the clock cycle time of Superscalar-like machines [Matzke97]. VLIW and DTSVLIW machines do not need hardware mechanisms equivalent to instruction windows or reservation stations in their main data path and do not suffer from their characteristic bypass logic and bypass wire delay overheads. Bypass logic and bypass wires are of course necessary in VLIW and DTSVLIW machines. However, they connect functional units' outputs to functional units' inputs only and not functional units' outputs to several reservation stations at the input of each functional unit, or to all instructions of a large instruction window. Therefore, they can have a faster clock than Superscalar-like machines even considering wire delays [Hara96].

Palacharla, Jouppi, and Smith have studied the impact of the complexity of the instruction dispatch and instruction issue hardware, and the impact of the bypass logic (fan-out) and wire delay in the performance of future Superscalars [Palacharla97]. Like Hara and his colleges [Hara96], they have concluded that wire delays are going to dominate the clock cycle time of Superscalar machines. They have also concluded that the delays incurred by the *wakeup logic* and *selection logic* may also impact the clock cycle time of Superscalars. The wakeup logic is responsible for matching the results produced by functional units with the source

operands of instructions waiting in the instruction window or reservation stations and for setting the instructions as ready. The selection logic is responsible for selecting instructions for execution from the pool of ready instructions. To reduce the impact of wire delays and wakeup and selection logic overheads, Palacharla, Jouppi, and Smith have suggested dividing the Superscalar core into several smaller clusters of functional units (A similar proposal specifically tailored to Trace Cache architectures is reported in [Vajapeyam97].) The resulting architecture has been named the *Dependence-Based* architecture. This architecture groups dependent instructions and sends them to the same cluster. This grouping of dependent instructions in clusters simplifies the wakeup and selection logic and helps mitigate the wire delays to some extent by using short local connections more frequently than long inter-cluster connections. The Dependence-Based architecture has inferior performance than non-clustered Superscalar, however. Therefore, the DTSVLIW can compete in performance with this variant of Superscalar as well. Moreover, we believe that clustering can also be employed in the DTSVLIW, although we do not examine clustered DTSVLIWs in this thesis.

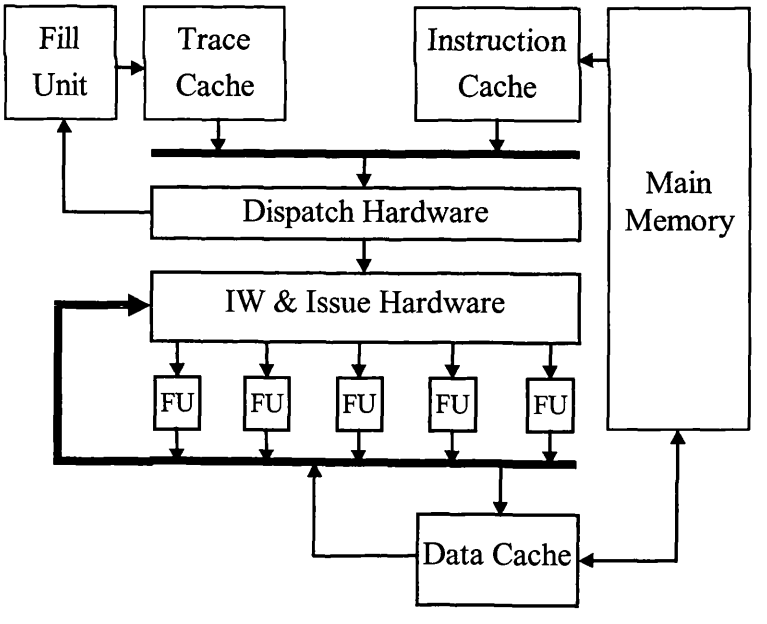



**Figure 3.1: Trace Cache Architecture. IW stands for instruction window and FU for functional unit.**

### 3.4.2 Value Prediction and Instruction Reuse

*Value prediction* and *instruction reuse* are architectural concepts that take advantage of the *value locality* that exist in programs and can be incorporated into Superscalar machines. Value prediction has been proposed by Lipasti and Chen initially for predicting the values of load instructions [Lipasti96a] and later extended for predicting the value of registers as well [Lipasti96b]. A machine employing value prediction uses a special hardware table for reducing the effect of data dependencies by predicting registers and memory contents from current information held in this table. The instruction reuse technique, on the other hand, takes advantage of value locality by using a hardware table where the results of instructions previously executed are stored [Sodani97]. Experiments have shown that many instructions and groups of instructions having the same inputs are executed repeatedly producing, of course, the same results [Sodani97]. However, such instructions do not have to be executed repeatedly – their results can be provided by this hardware table, avoiding subsequent executions and improving overall processor performance. The key difference between value prediction and instruction reuse techniques is the way they verify the validity of the values read from the tables. In the value prediction technique, the values are used speculatively and validated later, while in the instruction reuse technique the values are validated before being used [Sodani98].

By using value prediction and instruction reuse, a Superscalar machine can exploit more ILP and, in this sense, value prediction and instruction reuse are improvements on the Superscalar architecture [Lipasti97, Sodani98]. These two architectural concepts have also attracted significant research interest recently [Sazeides97, Gabbay97, Wang_K97, Citron98, Fu98, Gabbay98, Reinman98, Nakra99].

Superscalar machines enhanced with trace cache, value prediction, and instruction reuse may become a very effective way of exploiting ILP. This might transform such machines in the general-purpose processors of the future. However, Superscalar machines employing trace cache, value prediction, and instruction reuse are very complex devices and the impact of such complexity on the design cost and clock cycle time can be severe. For this reason, the DTSVLIW stands as an alternative general-purpose machine due to its simplicity and performance.

Moreover, we believe that the value prediction and instruction reuse concepts can also be incorporated to the DTSVLIW architecture [Fu98, Nakra99], although we do not examine this possibility in this thesis.

# Chapter 4

# The DTSVLIW Architecture

The symbolic diagram of a DTSVLIW machine is shown in Figure 4.1 (page 78). It has two caches for instructions and two processing engines. The Instruction Cache stores fragments of the original compiled code while the VLIW Cache stores *blocks* of long instructions. The original code is executed first by the Primary Processor. The code trace produced during this execution is scheduled by the Scheduler Unit into blocks of long instructions that are saved in the VLIW Cache. The VLIW Engine executes these long instructions if an already scheduled code fragment has to be executed again.

In a DTSVLIW machine, the VLIW Engine and the Primary Processor never operate at the same time and no machine state has to be transferred between them, as they share the DTSVLIW machine state. This simplifies the design of both, even allowing the VLIW Engine to share ports of the register file and Data Cache with the Primary Processor. The cost in cycles of swapping the execution between the VLIW Engine and the Primary Processor is equal to the sum of a number of pipeline stages of each one only (the pipeline stages discarded in one plus the pipeline stages refilled in the other).

While the Primary Processor is executing the code, the Fetch Unit (Figure 4.1) issues different addresses to the Instruction Cache and the VLIW Cache. To the Instruction Cache is issued the *program counter* (PC) content. To the VLIW Cache is issued the address of the instruction in the execute stage of the Primary Processor (dashed arrow in Figure 4.1). If this instruction has been executed before, there may

be a block with its address in the VLIW Cache. On a VLIW Cache hit, the VLIW Engine takes over execution. The block being constructed by the Scheduler Unit is flushed to the VLIW Cache – this block is made to point at the hit block. The contents of all but the write back pipeline stage of the Primary Processor are annulled and the PC receives the memory address that hit the VLIW Cache. In subsequent cycles, the VLIW Engine controls the PC.

On a VLIW Cache miss, the Primary Processor takes over execution, fetching from the last PC value computed by the VLIW Engine. The Fetch Unit does not issue fetches to the VLIW Cache again until an instruction arrives at the execute stage of the Primary Processor. At this point, the Scheduler Unit restarts to schedule a new block, the address of which will be the last address produced by the VLIW Engine when executing the previous block. This connects these blocks forming a block chain. In steady state, the VLIW Cache contains all most frequently executed traces.

The key issues to be resolved in the DTSVLIW architecture are the scheduling of the instruction trace into long instructions and the addressing within these long instructions. The Primary Processor and the VLIW Engine themselves are not a challenge. In this chapter, the Scheduler Engine operation and the VLIW Engine long instruction addressing are examined in detail together with other relevant aspects of the DTSVLIW architecture.

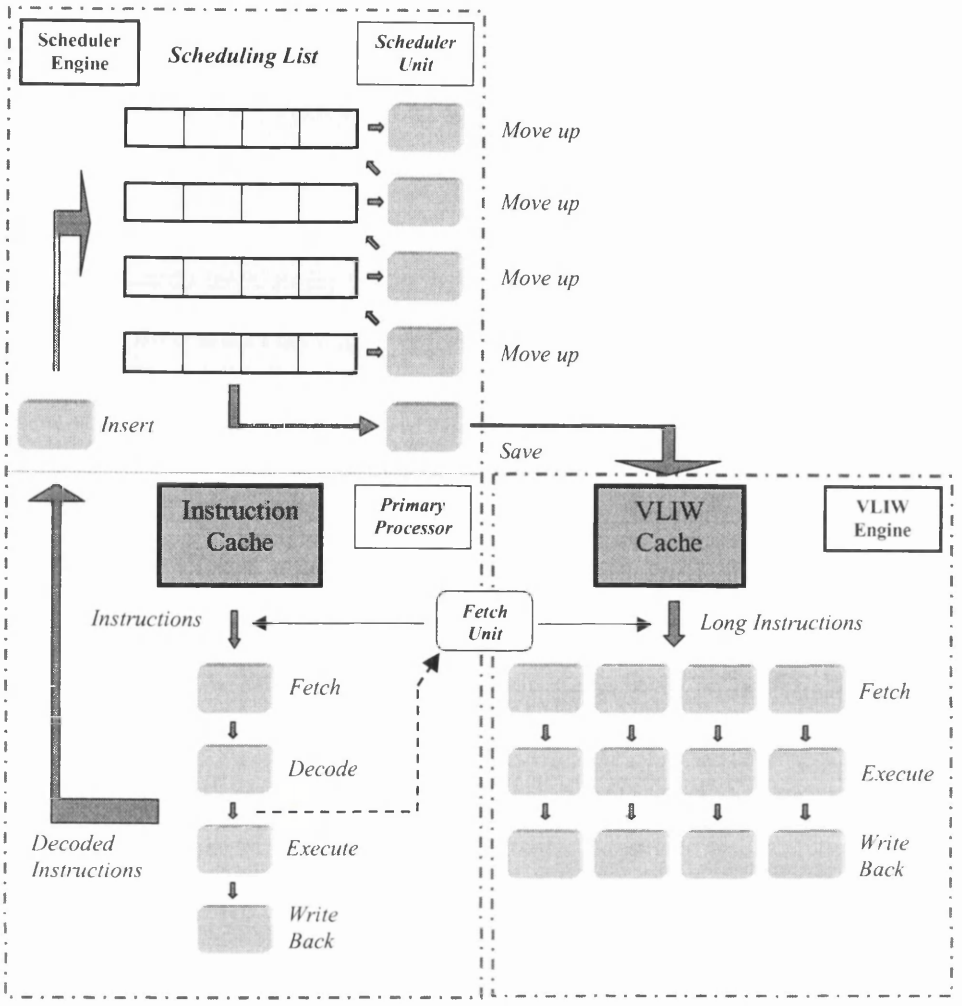



**Figure 4.1: A DTSVLIW machine.**

## 4.1 The Scheduler Engine

The Scheduler Engine is composed of the Primary Processor plus the Scheduler Unit (Figure 4.1). The Primary Processor is a simple pipelined processor capable of executing all instructions of the Sparc Version 7 ISA [Sun87] – the ISA we have chosen for the implementation of the DTSVLIW machine described in this thesis. When a valid instruction moves from the decode pipeline stage to the execute pipeline stage, the Primary Processor sends it to the Scheduler Unit. The Scheduler Unit implements in hardware a simplified version of the FCFS algorithm, which historically has been used to statically schedule microcode (Section 3.1). We have chosen this algorithm for three reasons. First, it operates with one instruction at a

time and considers instructions in the strict order that they appear during program execution, which perfectly fits the DTSVLIW mode of operation. Second, the FCFS algorithm produces optimum or near-optimum scheduling [Davidson81]. Finally, the FCFS algorithm is easy to implement in hardware in a pipelined fashion in the form it is presented here (see Section 4.5).

### 4.1.1 The Scheduling Algorithm

The implemented version of the FCFS algorithm acts on a list, the *scheduling list.* This list has a number of elements equal to *BLOCK_SIZE* (a hardware constant) and each element contains one long instruction and a *candidate instruction*, which holds an instruction for scheduling into the long instruction. A broad overview of the algorithm is that an instruction coming from the Primary Processor in one clock cycle is placed at the end of the scheduling list on the next clock cycle. On each subsequent cycle, this instruction can *move up* to the next higher element in the list if:

- it has not reached the head of the list
- and there is space for it in the next element
- and there is not a dependency with instructions in the next element

Figure 4.2 shows an example of the algorithm scheduling a fragment of code that adds all elements of a vector. In Figure 4.2, slh and slt stand for scheduling list head and tail, respectively, and the destination register of the instructions is the rightmost. The scheduling algorithm ignores the nop instruction. The details of the algorithm's operation follow.

An instruction arriving in the execute pipeline stage of the Primary Processor in one cycle is *inserted* into a suitable slot in the scheduling list in the next cycle. If there are no data, control, or resource dependencies on any instruction in the list's tail element, the incoming instruction is inserted into the list's tail element; otherwise, it is inserted into a new tail element added to the list. In Figure 4.2, instructions 1 and 2 are inserted by the first method, while instruction 3 is inserted by the second method due to a true data dependency on r8.

An instruction inserted with the first method is a candidate for moving up the list on subsequent clock cycles. There can only ever be a single candidate instruction in a long instruction, but each long instruction in the list may have a candidate for

promotion. Thus, candidate instructions of different long instructions can be moved up in parallel in a clock cycle. After an instruction has been inserted into the end of the list, the next step is to move this instruction up as far as it can go in the list of long instructions. An instruction can move up from long instruction *i* to long instruction *i* - 1 if it has no true data dependency on any instruction in the long instruction *i* - 1 and there is a suitable slot available. If the instruction cannot move up, it is *installed* in long instruction *i*. In Figure 4.2, instruction 3 is installed in the fourth cycle, while instruction 8 is moved up in the ninth cycle.

The candidate instruction in *i* can be placed in long instruction *i* - 1 even if:

- there is an output dependency on any instruction in $i-1$
- or there is an anti dependency on any instruction in *i*
- or there is a control dependency on any instruction in *i* (there is a conditional branch or indirect branch in *i*)

However, in such cases, the candidate instruction has to be *split*. The split is done by renaming either the candidate instruction's output that has caused the output/anti dependency or all outputs if there is a control dependency, and by inserting a *copy instruction* permanently in the current slot in long instruction *i*. This copy instruction performs the copy of the renaming register (or the renaming registers) content to the instruction's original output (or instruction's original outputs). In Figure 4.2, instruction 7 is split in the ninth cycle.

Conditional and indirect branches do not move up. They are installed when inserted and establish a *tag* for their long instruction. All instructions subsequently placed in this long instruction receive the last established tag. During VLIW execution, the VLIW Engine evaluates the conditional and indirect branches and validates their tags if they follow the same direction observed during scheduling. Only instructions with valid tags have their results written in the machine state. In Figure 4.2, the second instance of instruction 5 receives the tag established by instruction 9 in cycle eleven.

When there is no free element for an incoming instruction, the scheduling list content is sent to the VLIW Cache as a *block* and the incoming instruction is inserted into the list as the first instruction of a new block. All renaming registers previously used are restored to the pool of renaming registers and can be reused. The list is

saved as a block, but on a one long instruction per cycle basis; nevertheless, instructions can be continuously inserted into the new block at the same time as the old block is being saved. This is achieved by making the scheduling list circular, and by using three registers to handle it: the *scheduling list head* register, the *scheduling list tail* register (Figure 4.2), and the *output long instruction pointer* register.

The scheduling list tail register together with the scheduling list head register delimits the active elements of the scheduling list. The output long instruction pointer register is used to flush the list to the VLIW Cache. All three are zeroed after the DTSVLIW is reset. The scheduling list tail register is incremented when new entries are added to the scheduling list. If the number of valid elements in the list exceeds the BLOCK_SIZE, the list is full. When the list is found full on going to insert a new instruction, the content of the scheduling list tail register is copied to the scheduling list head register. This makes the latter different from the output long instruction pointer register and the list empty. When the output long instruction pointer and the scheduling list head registers are different, the long instruction that is pointed at by the output long instruction pointer is sent to the VLIW Cache and the output long instruction pointer is incremented. These repeat every clock cycle until the output long instruction pointer register becomes equal to the scheduling list head register again and the block has been flushed. As instructions are inserted into the list at the maximum rate of one instruction per clock cycle, the same rate as long instructions are written into the VLIW Cache, one action does not interfere with the other.

```
for (sum = 0,  i = 0; i < x; i++)
{
        sum = a[ i] + sum;
}
```

*(a)*

```
        1: or      r0, 0, r9          # r9 = sum
        2: sethi   hi(56), r8         # r8 = temp
        3: or      r8, 8, r11         # r11 = *a
        4: or      r0, 0, r10         # r10 = 4*i
loop:   5: ld      [r10+r11], r8
        6: add     r9, r8, r9
        7: add     r10, 4, r10
        8: subcc   r10, 4*x-1, r0
        9: ble     loop
       10: or      r0, 0, r0          # nop
```

*(b)*

| | | | | | | |
|---|---|---|---|---|---|---|
| slh-> | or | r0, 0, r9 | sethi | hi(56), r8 | | |
| slt-> | or | r8, 8, r11 | | | | |
| | | | | | | |
| | | | | | | |

after 3 cycles

| | | | | | | |
|---|---|---|---|---|---|---|
| slh -> | or | r0, 0, r9 | sethi | hi(56), r8 | or | r0, 0, r10 |
| | or | r8, 8, r11 | | | | |
| | ld | [r10+r11], r8 | add | r10, 4, r10 | | |
| slt -> | add | r9, r8, r9 | subcc | r10, 4*x-1, r0 | | |

after 8 cycles

| | | | | | | |
|---|---|---|---|---|---|---|
| slh -> | or | r0, 0, r9 | sethi | hi(56), r8 | or | r0, 0, r10 |
| | or | r8, 8, r11 | add | r10, 4, r32 | | |
| | ld | [r10+r11], r8 | COPY | r32, r10 | subcc | r32, 4*x-1, r0 |
| slt -> | add | r9, r8, r9 | ble | loop | | |

after 9 cycles

| | | | | | | |
|---|---|---|---|---|---|---|
| slh -> | or | r0, 0, r9 | sethi | hi(56), r8 | or | r0, 0, r10 |
| | or | r8, 8, r11 | add | r10, 4, r32 | | |
| | ld | [r10+r11], r8 | COPY | r32, r10 | subcc | r32, 4*x-1, r0 |
| slt -> | add | r9, r8, r9 | ble | loop | ld | [r10+r11], r8 |

after 11 cycles

*(c)*

**Figure 4.2: Scheduling algorithm running example. *(a)* C code fragment. *(b)* Assembly language version of the C code *(c)* Four snapshots of a three instructions wide and four long instructions deep scheduling list, filled with instructions coming from the Primary Processor after 3, 8, 9, and 11 cycles of the execution of the first instruction. The shaded instructions in each snapshot are also candidate instructions.**

## 4.1.2 Copy Instructions Handling

Copy instructions do not cause data dependencies and they can be overwritten by other instructions during scheduling. If an instruction in long instruction $i$ reads from a register written by a copy instruction in long instruction $i - 1$, this instruction can be moved to long instruction $i - 1$ during scheduling. In this case, the mentioned input register is renamed to the same name of the copy instruction input register. On the other hand, if an instruction in long instruction $i$ writes into the register written by a copy instruction in long instruction $i - 1$, this instruction may be moved to long instruction $i - 1$ without splitting. This instruction can be moved to long instruction $i - 1$ without splitting if the tag of the copy instruction in $i - 1$ is equal to the current

tag of the long instruction $i$ - 1. In this case, the copy instruction is overwritten by the instruction. If the tag of the copy instruction is different from the tag of the long instruction, it means that a conditional or indirect branch has been installed in the long instruction after the copy instruction had been generated. To overwrite this copy instruction would be equivalent to crossing a conditional or indirect branch without renaming, which would be an invalid scheduling operation. Therefore, if the tag of the copy instruction is different from the tag of the long instruction $i - 1$, the instruction in long instruction $i$ has to be split in order to be moved up to long instruction $i - 1$.

### 4.1.3 Control-Transfer Instructions Handling

During scheduling, one or more control-transfer instructions can be placed in a single long instruction, but they cannot move up (their order is preserved). Control dependencies are caused only by conditional and indirect branches (subroutine return is a special case of this) and they do not impede scheduling beyond basic blocks. Instructions can cross basic block limits imposed by conditional and indirect branches and execute speculatively.

Speculative execution is implemented by splitting instructions and moving up their first part past conditional or indirect branches, leaving the copy part behind. If a conditional or indirect branch does not follow the same direction during execution, the copy part of the split instruction is not executed, not committing the corresponding instruction.

Conditional branch instructions read the conditional code register (the *flags*), which is written by many different instructions. Output and anti data dependencies caused by this register are tackled as other dependencies of these kinds. The VLIW Engine has many conditional code registers; therefore, instruction splitting can be used to avoid these dependencies.

Because the Sparc ISA implements delayed branches, an extra pipeline stage between the Primary Processor and the Scheduler Unit is necessary for the insertion of delayed slot instructions (see Figure 4.1). When a delayed branch is detected inside this pipeline stage, it is held on it for one cycle. In this cycle, instead of the branch, the delayed slot instruction in the Primary Processor's decode pipeline stage

is inserted into the scheduling list, bypassing the insert pipeline stage. In the following cycle, the delayed branch is inserted normally. This guarantees the correct assignment of tags to the delayed branch and delayed slot instruction.

### 4.1.4 No-operation Instructions Handling

No-operation (nop) instructions are ignored and not placed in the scheduling list.

### 4.1.5 Load and Store Instructions Handling

Load and store instructions can be split and moved up by the scheduling algorithm without restrictions. For dependency test, their data addresses are compared with the data address of other load/store instructions, while the registers they use (including those used to compute data addresses) are compared with registers of other instructions (including load/store). *Memory renaming registers* provide for the renaming of memory positions. Load/store address aliasing is discussed in Section 4.7.

### 4.1.6 Save and Restore Instructions Handling

Save and restore instructions, which deal with the register windows of the Sparc ISA [Sun87], are scheduled as any other integer instruction. To make it possible, the value of the *cwp* (current window pointer) register of the Sparc ISA, which is used for computing the address of the physical integer registers, accompanies the instructions to the scheduling list and VLIW Cache.

### 4.1.7 Non-schedulable Instructions Handling

Non-schedulable instructions are a number of instructions of the Sparc ISA that are not executable by the VLIW Engine, but must always be executed by the Primary Processor because they are too complex for the VLIW Engine to handle. When such instructions are sent to the Scheduling Unit, they flush the scheduling list to the VLIW Cache. Thus *trap*, *return from trap*, and *co-processor handling* instructions are non-schedulable. In addition, *load/store* instructions are non-schedulable when they are *non-cacheable*, or when they *perform I/O operations*, or when they *provide support for cache coherence and multiprocessing*.

### 4.1.8 Multicycle instructions Handling

Multicyle instructions, such as integer divide, floating-point multiply, or (sometimes) loads and stores, impact upon both the operation and performance of the architecture. Their scheduling requires special care to respect dependencies in any of their cycles. The DTSVLIW scheduling of multicycle instructions has been implemented as follows [deSouza99b].

To schedule multicycle instructions, extra features were added to the scheduling list. These are an extra candidate instruction for each element of the list, the *candidate instruction B*, and an extra slot in each long instruction for each multicycle functional unit, the *slot B*. Two instances, A and B, of a multicycle instruction are inserted into the scheduling list. These are just copies of the original instruction, and have cross-references to each other's position in the scheduling list. The purpose of the A and B instructions is to delimit the scheduling list elements in which the instruction is active to prevent instructions with dependencies being scheduled in these elements. The primary role of the B instruction is for dependency checking against instructions moving up.

Instance A is inserted into the tail of the scheduling list. If the current block size plus the instruction latency minus one is larger than BLOCK_SIZE, a new long instruction is added to the scheduling list for inserting the instance A of the multicycle instruction. In such cases, a new block starts at the scheduling list position where the instance A is inserted. Instance B, on the other hand, is inserted into the *scheduling list tail* + (*instruction latency* – 1) position of the scheduling list in the candidate instruction B and slot B; the scheduling list tail register is made to point at this element. After insertion, the Scheduler Unit handles these two instances as other instructions, except that:

- Instance B does not suffer or cause resource dependencies, because it does not use a normal slot but a B slot.
- Instance B does not suffer or cause data dependencies related to its inputs.
- Instance A does not suffer or cause data dependencies related to its outputs but only control dependencies, in which case A's output is renamed and instance B is split. The copy instruction generated is not placed in a B slot, but in a normal one.

- Instances A and B move up together; if A cannot move up both are permanently placed in their current long instructions (B can always move up if A can move up).
- If instance B suffers an output or anti data dependency it is split and A is also renamed.
- Instance B is not saved in the VLIW Cache and is only used for scheduling purposes.

Multiple scheduling list elements need to be added to a block for a single multicycle instruction. Because of this, the Primary Processor is required to have only one instruction in the execute pipeline stage at any time. Therefore, the Scheduler Unit only has to add one extra scheduling list element per cycle, since the Primary Processor is expected to hold its pipeline to complete the multicycle instruction. This reduces its performance, but the overall performance of the architecture is dependent on the VLIW Engine performance, not that of the Primary Processor.

Scheduling a multicycle instruction lengthens a block by the latency of the instruction minus one. This impacts on efficiency since the longer block is more difficult to fill, reducing parallelism and wasting space in the VLIW Cache. Some instructions have very long latencies, and in certain programs, are too frequent to be left to the Primary Processor to execute (one option for dealing with them): floating-point divide is one example. These instructions are scheduled as multicycle instructions, but the latency used by the Scheduler Unit is not the same as the instruction latency. The latency used by the Scheduler Unit is set to a maximum (4 for example) and, under VLIW execution, the VLIW Engine holds the execution of the long instruction containing these instructions for the number of extra cycles necessary for its proper execution. This saves on VLIW Cache space but does not affect the reduction in parallelism.

## 4.2 DTSVLIW Long Instruction Format

The format of the long instructions of the DTSVLIW can be appreciated with the help of Figure 4.3. In this figure, two long instructions are shown, and each of them has five instructions. The individual instruction tags are represented by the shaded fields. These two long instructions do not have any particular programming meaning

and are only examples. However, they could be seen as two neighbour instructions of a block.

The first long instruction does not have conditional or indirect branches; therefore, all instructions have tags with the value zero. Instructions with tag equal zero are executed unconditionally. The second long instruction has two conditional branches. In this long instruction, only the branch *ble (0) loop* is executed unconditionally. The *(0)* in this branch indicates which conditional code register the branch is testing – the DTSVLIW has many conditional code registers and instructions that read from or write into them can be renamed. An example of this renaming is shown in Figure 4.3.

The instruction *subcc* of the Sparc ISA writes into conditional code registers. An instance of *subcc* is shown in the first long instruction of Figure 4.3: the instruction *subcc r1, r3, r0: (1)*. This instruction writes into the conditional code register 1, as indicated by the *(1)* in the instruction. The copy instruction *COPY cc1, cc0* copies the content of the conditional code register 1 to the conditional code register 0: the original *subcc* was split and moved up by the Scheduler Unit, and left this copy instruction behind.

When installed in the second long instruction of Figure 4.3, the branch *ble (0) loop* changes this long instruction's tag value to 1, and three instructions receive this new tag value, including the second branch, *bz (1) exit*. When installed, this second branch changes the tag again, and the instruction *add r8, r2, r3* receives the new tag value, 2. When executing this long instruction, the VLIW Engine validates the tags 1 and 2 by checking if the branches follow the same direction observed during scheduling. Only instructions with valid tags, including branches, have their results written to the machine state. Therefore, if the branch *ble (0) loop* follows a direction different from that observed during scheduling, no other instruction in its long instruction has its results stored. On the other hand, if this branch does follow the same direction observed during scheduling and the branch *bz (1) exit* does not, only the instruction *add r8, r2, r3* does not have its results stored.

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | ld [r10+r11], r8 | 0 | subcc r10, 50, r0: (0) | 0 | add r10, 4, r32 | 0 | subcc r1, r3, r0: (1) | 0 | nop |
| 0 | ble (0) loop | 1 | COPY 32, r10 | 1 | COPY cc1, cc0 | 1 | bz (1) exit | 2 | add r8, r2, r3 |

**Figure 4.3: DTSVLIW long instruction format examples.**

## 4.3 DTSVLIW Long Instruction Addressing

Once instructions are scheduled into blocks of long instructions, the VLIW Engine instruction addressing has to be different from the Primary Processor instruction addressing. In the DTSVLIW, a block of long instructions is stored as a VLIW Cache line. Since the only entry point of a block is the first instruction scheduled in the block, there is a single address for the whole block, and this is the address of the first instruction scheduled in the block. For fetching a long instruction from the VLIW Cache, the VLIW Engine uses a fetch address with two fields: the *address* field and the *line index* field. The address field is a Sparc ISA address and specifies the block, while the line index field specifies a long instruction in the block. This *long instruction address* is produced by concatenating the PC with a *line index register* maintained by the VLIW Engine and incremented from zero.

The number of valid long instructions in a block is stored into the VLIW Cache with the block. The line index register content is compared with this number to determine the fetch of the last valid long instruction in a block. When they have the same value, the next fetch is made using the address of the instruction that follows the block, which is also stored into the VLIW Cache line. This mechanism requires only two instruction addresses to be stored in a cache line: the address of the first instruction of the block and that of the following block. Individual instruction addresses are not required, since the block will execute as a whole unless a branch is made out of the block, in which case the information needed to build the target address is stored as part of each branch instruction.

We have developed a mechanism to generate and save in the VLIW Cache the number of valid long instructions of each block and the address of the following block. This mechanism allows several blocks to coexist in the scheduling list, in addition to allowing the scheduling of one block and the saving of another block in parallel. To implement this mechanism, we have added to each element of the scheduling list two stores to hold the current and the next long instruction addresses: the *long instruction address* store, and the *next long instruction address* store. Each of these has an *address* field to hold a Sparc ISA address and a *line index* field to hold the position of the element of the list. Figure 4.4 shows examples of how our mechanism operates, and the following paragraphs give the details of its operation.

When instruction insertion causes a new long instruction to be added to a block, the long instruction address store of this element receives a copy of the long instruction address store of the previous tail element with the line index field incremented. In addition, every time a new long instruction is added to a block, the next long instruction address store of the previous scheduling list's tail element receives the long instruction address of the new long instruction. In Figure 4.4, the first snapshot shows a new long instruction being added to a block as explained.

When the insertion of an instruction causes the creation of a new block in an element of the scheduling list, this instruction's address is copied to the address field of the long instruction address store of the element. At the same time, the line index field of the long instruction address store of the element is zeroed. The second snapshot of Figure 4.4 shows the creation of a new block in this way.

Because the next long instruction address store of the previous scheduling list's tail element always receives the long instruction address of an added long instruction, the next long instruction address store of the last long instruction of a block always point to the first long instruction of the fall-through block. The only exception are when a non-schedulable instruction is sent to the Scheduling Unit and when there is a fetch hit on the VLIW Cache. In these cases, the block being scheduled is made to point to the non-schedulable instruction or to the hit block, and start being flushed to the VLIW Cache. This is accomplished by adding a new empty long instruction to the scheduling list, as show in the third snapshot of Figure 4.4. This creates a new empty block. The insertion of an instruction into an empty block gives this block the instruction's address – see the fourth snapshot of Figure 4.4.

The contents of the long instruction address store and next long instruction address store are not saved into the VLIW Cache; instead, a unique *next block address* (**nba**) is saved with each block. The **nba** has the same format as the long instruction address and contains the address of the following block and the number of valid long instructions of the block. The address field of **nba** holds the Sparc ISA address of the following block, while the line index field of **nba** holds the number of valid long instructions of the block. The **nba** value is computed as follows.

When a long instruction is being saved, the address field of its next long instruction address store is copied into the address field of the **nba** of the long

instruction's block in the VLIW Cache. At the same time, the line index field of this long instruction's long instruction address store is copied into the line index field of the **nba**. Thus, the **nba** of each block ends up with the address of the fall-through block (the address field of the next long instruction address store of the last long instruction of each block) and with the number of valid long instructions of the block (the line index field of the long instruction address store of the last long instruction of each block). Figure 4.4 shows four **nba** values generated for a single block. The value of the **nba** in the last snapshot is the final value written into the VLIW Cache.

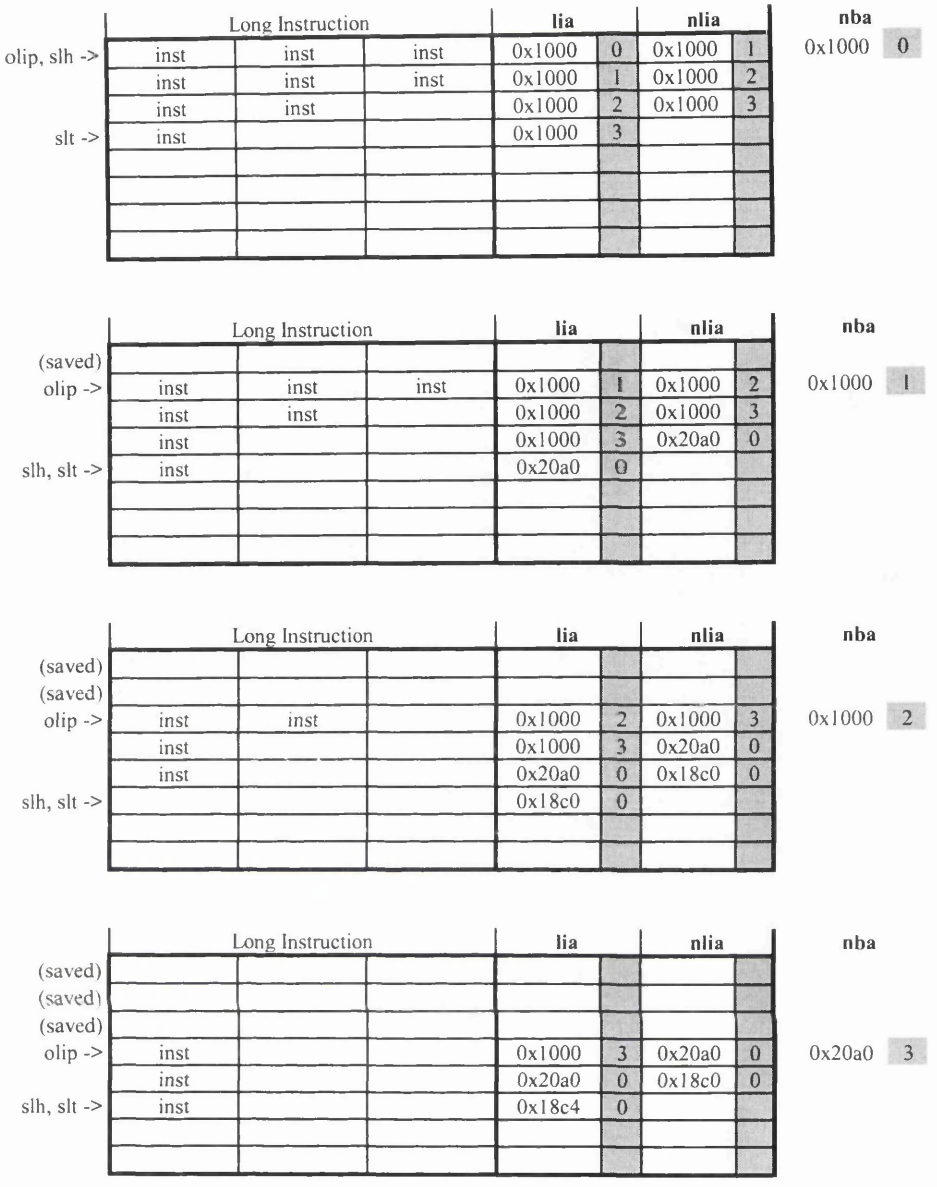



**Figure 4.4: VLIW Engine instruction addressing. Four snapshots of a three instructions wide and eight long instructions deep scheduling list are shown. lia, nlia, and nba stand for *long instruction address*, *next long instruction address*, and *next block address*, respectively. olip, slh, and slt stand for *output long instruction pointer*, *scheduling list head*, and *scheduling list tail*, respectvely. The shaded fields represent the index field of each address.**

## 4.4 The VLIW Cache

The VLIW Cache is an ordinary set associative cache with line size equal to one block of long instructions. It is tagged with the Sparc ISA address of the first instruction placed in the blocks by the Scheduler Unit. In the VLIW Cache, each long instruction can be accessed directly by using addresses in the long instruction address format, with the line index field of the address choosing the specific long instruction

in the block. The single additional feature of the VLIW Cache is the **nba** store associated with each cache line. The **nba** value is used in the VLIW Engine fetch, as described next.

## 4.5 The VLIW Engine

The VLIW Engine of the DTSVLIW has a simple fetch-execute-write back pipeline for each functional unit. Multicycle instructions execute in pipelined functional units with more than one execute stage. A decode stage is not necessary as decoded instructions are saved in the VLIW Cache. To access the VLIW Cache, the VLIW Engine concatenates the PC with the contents of the line index register. This register is incremented after each VLIW Engine fetch, while the PC is left unchanged. On a long instruction fetch, the **nba** value associated with the cache line is fetched as well. If the line index register content is equal to the line index of **nba**, then, at the end of the cycle, the content of the address field of **nba** is copied to the PC and the line index register is zeroed. These actions cause the fetch of the first long instruction of the fall-through block in the following cycle without causing pipeline bubbles.

When blocks are sequentially executed no bubbles occur in the VLIW Engine pipeline, and only a single bubble occurs when a branch is made out of a block and another block is hit in the VLIW Cache.

All conditional and indirect branches are resolved in the execute stage of the VLIW Engine. The direction taken by them during the scheduling, recorded in the VLIW Cache, is used during the execution to determine if any of them are following a different direction. If the branch direction of a branch with a valid tag is different from that recorded, the current VLIW fetch is annulled and the PC receives the new branch target, while the line index register is zeroed. This causes a 1-cycle bubble in the VLIW Engine pipeline.

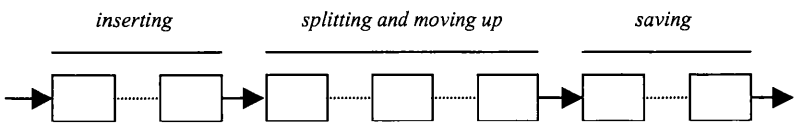



**Figure 4.5: Scheduler Unit pipeline.**

## 4.6 Scheduler Unit Implementation

The Scheduler Unit can be implemented in a pipelined fashion as depicted in Figure 4.5. One or more pipeline stages can be used for inserting instructions into the scheduling list, each scheduling list entry can be made a pipeline stage, and none, one or more pipeline stages can be used for saving the scheduled long instructions into the VLIW Cache.

The checking operations required on the scheduling list on each clock cycle are just comparison operations between each candidate instruction and the instructions in the current and next element of the list. Each check operation is independent. However, the decision to install, split, or move up a candidate instruction may depend on a chain of decisions as long as the scheduling list. Nevertheless, the information necessary to each one can be gathered in a way similar to carry propagation in carry-lookahead integer adders [Patterson96 (Appendix A)], and the logic required can be made as fast as an *and-or* gate delay. This can be proved with the help of Figure 4.6.

In Figure 4.6, the value of CRd(i), CTd(i), COd(i), Rd(i), Td(i), Od(i), Ad(i), and Cd(i) for each element $i$ of the list ($0 < i <$ block size $- 1$) is available at the beginning of each clock cycle after the comparators delay (*xor* gate delay). Invalid candidate instructions never produce CRd(i), CTd(i), or COd(i) signals. Valid candidate instructions could influence the Rd(i), Td(i), Od(i), and Ad(i) signal values; for this reason, their companion position is used for disabling the comparators associated with the slot where the companion instruction is. CRd(i) is also disabled if there is more than one slot available in $i$ - 1 for candidate instruction $i$.

Let us analyse the installing case first. A valid candidate instruction must be installed on true dependencies or resource dependencies. So, if Td(i) is true there is an instruction already installed in long instruction $i$ - 1 causing a true dependency on the candidate instruction $i$. In this case, the candidate instruction in $i$ must be installed. If only CTd(i) is true one cannot tell whether or not the candidate instruction should be installed, because the candidate instruction in $i - 1$ might move up in this cycle. The same can be said about Rd(i) and CRd(i) signals. Nevertheless, using the position of the candidate instruction in the list, which is recorded in the line

index field of the long instruction address store of the long instruction, an *install signal* can be computed for each candidate instruction in the scheduling list as follows:

$$
\begin{aligned}
&\text{install signal} = \\
&(i \otimes 0) + \\
&(i \otimes 1).(Td(1) + Rd(1) + CTd(1) + CRd(1)) + \\
&(i \otimes 2).(Td(2) + Rd(2) + \{CTd(2) + CRd(2)\}.\{Td(1) + Rd(1) + CTd(1) + CRd(1)\}) + \\
&(i \otimes 3).(Td(3) + Rd(3) + \{CTd(3) + CRd(3)\}. \\
&\qquad \{Td(2) + Rd(2) + [CTd(2) + CRd(2)].[Td(1) + Rd(1) + CTd(1) + CRd(1)]\})
\end{aligned}
\tag{1}
$$

The equation above represents the logic necessary to compute the install signal for a DTSVLIW machine with a block size equal to 4. The rule to produce equations for larger blocks is easily deduced by visual inspection. The operator "⊗" means binary vector comparison: (i⊗x) evaluates to true if $i$ is equal to $x$. The operator "+" means logic *or*, and the operator "." means logic *and*.

When the line index field of the list element containing the candidate instruction $i$ is equal to zero, the first line of the equation evaluates to true and, consequently, the install signal becomes true. This implements the first rule for installing a candidate instruction, i.e., if the candidate instruction is at the head of the scheduling list it is installed. If $i$ is equal to 1, only the second line of the equation can evaluate as true. In this case, the candidate instruction $i$ will be installed if there is a true dependency on any instruction installed in long instruction $i$ - 1 (the head of the list), or there is not a slot available in this long instruction, or there is a true dependency or resource dependency on a valid candidate instruction in this long instruction. For $i$ greater than 1, the information from lower order list elements is added to each equation line as shown.

A *split signal* can be computed for each candidate instruction in the scheduling list of a DTSVLIW machine with a block size equal to 4 as follows:

split signal =

$$
\begin{aligned}
&(i\otimes 1).(Od(1) + Ad(1) + Cd(1) + COd(1)) + \\
&(i\otimes 2).(Od(2) + Ad(2) + Cd(2) + COd(2).\{Td(1) + Rd(1) + CTd(1) + CRd(1)\}) + \\
&(i\otimes 3).(Od(3) + Ad(3) + Cd(3) + COd(3).\{Td(2) + Rd(2) + [CTd(2) + CRd(2)]. \\
&\qquad\qquad [Td(1) + Rd(1) + CTd(1) + CRd(1)]\})
\end{aligned}
\tag{2}
$$

Again, the rule to produce equations for larger blocks is easily deduced by visual inspection. It is important to observe that part of this equation comes from the previous one. This is so because an output dependency caused by COd(i) generates a split signal only if the candidate instruction in element $i$ - 1 of the scheduling list is going to be installed.

If the split signal is true, the respective candidate instruction is split. If the install signal is true, the candidate instruction is installed. If the install and the split signals are both true the respective candidate instruction is only installed. If the candidate instruction is not going to be installed or split, it is moved up.

The install and split signal generation is the most complex operation performed by the Scheduler Unit, and its complexity is governed by the block size. Since a block of 32 long instructions is a large block, the Scheduler Unit design does not pose constraints on the cycle time of DTSVLIW machines. This is because the complexity of the logic necessary for generating these signals is equivalent to that of an integer adder and DTSVLIW machines with data words of 32-bit or more have to perform integer add operations in one cycle.

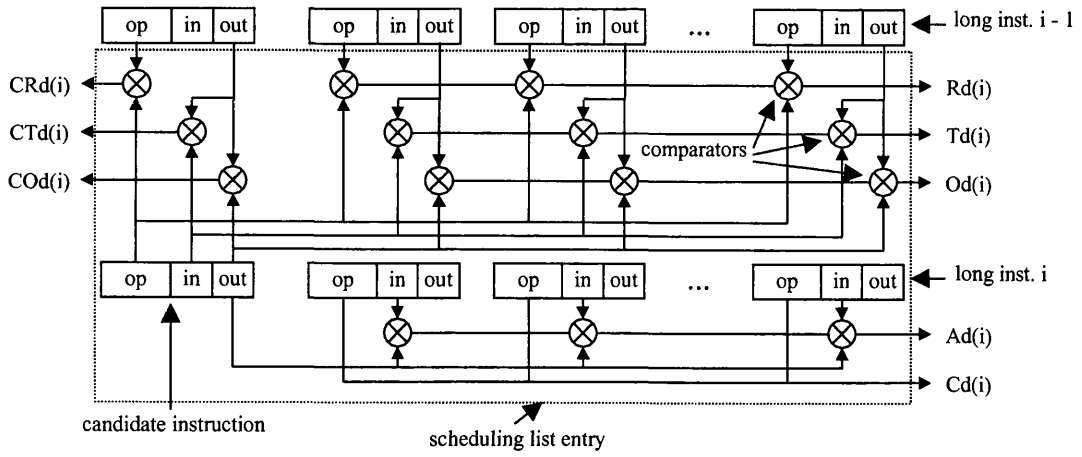



**Figure 4.6: Scheduling list. Rd(i), Td(i), Od(i), Ad(i), and Cd(i) stand for resource dependency, true data dependency, output data dependency, anti data dependency, and control dependency on candidate instruction *i*, respectively. CRd(i), CTd(i) and COd(i) stand for resource dependency, true data dependency and output data dependency on candidate instruction *i* caused only by the candidate instruction in long instruction *i* - 1, respectively.**

## 4.7 Memory Aliasing Detection

Memory aliasing (see Subsection 2.2.4) can occur, as the memory address observed during scheduling is not necessarily the same during VLIW execution. To detect memory aliasing and generate memory aliasing exceptions during VLIW execution, load and store instructions receive two extra fields when they are scheduled: the *order* and the *cross bit* fields. The order field receives the load/store insertion order, which is copied from the *load/store order counter*. This counter is zeroed every time the scheduling list is found empty and is incremented every time a load/store is inserted into the scheduling list. The cross bit field is set in the load/store when it is placed in a long instruction containing a store or a memory copy instruction generated from a store split.

The VLIW Engine keeps a *store list* and a *load list*, which are emptied every time a block starts execution. During VLIW execution, loads and stores with the cross bit set have their addresses and order fields stored in these lists as they execute. Load instructions executed in VLIW mode have their addresses associatively compared with the store addresses in their long instruction and all store addresses in the store list. On an address match, if the order field of the load is smaller than the order field of the corresponding store (which means that a late store to the same

address has been executed), an aliasing exception is signalled. The store instructions executed in VLIW mode have their addresses associatively compared with the load and store addresses in the same long instruction and all load and store addresses in the load and store lists. On an address match, if the order field of the store is smaller than the order field of the corresponding load/store, an aliasing exception is signalled. DTSVLIW exception handling is discussed next.

## 4.8 Exception Handling

Exceptions (interrupts) may be generated by the execution of some instructions, such as load/store (page faults, access violations) or divide (divide by zero). However, split instructions should not signal exceptions until their copy part is executed. To avoid uncommitted instructions generating exceptions while allowing true exceptions to be handled, an *exception bit* is added to each renaming register of the DTSVLIW.

When a split instruction generates an exception, the exception bit of its renamed destination register is set and execution proceeds normally. If this register is read by any other instruction, the exception bit is propagated to the destination registers of this instruction. This is accomplished by performing logic *or* of the exception bits of all input registers of this instruction and by writing the result of this operation to the exception bits of the instruction's destination registers. When an instruction that has an input register with an exception bit set is executed and its destination register is part of Sparc ISA state, an exception is signalled. The DTSVLIW exception handling mechanism operates as follows.

The DTSVLIW uses the *Checkpointing* exception handling mechanism, proposed by Hwu and Patt [Hwu87] (see Subsection 2.2.3). Checkpointing occurs at the beginning of the execution of each block of long instructions, when all registers that make up the Sparc ISA state are saved in shadow registers. Store instructions executed in the block cause the data they overwrite in the Data Cache to be saved in the *checkpoint recovery store list*. This list contains the address, data overwritten, and data type.

If the VLIW Engine detects an exception during the execution of a block, the Scheduler Engine enters a *recovery mode* of execution. In this mode, registers receive the values stored in the shadow registers, each entry of the checkpoint recovery store

list is written back into the Data Cache, and the load and store lists are emptied. If the exception detected is an aliasing exception, the VLIW Cache entry containing the block that caused the exception is invalidated. Execution is then resumed.

For an aliasing exception, execution resumes in normal trace mode and the block that has caused it is scheduled in a way that prevents new aliasing exceptions: data dependencies keep load/stores in a new order inside the block, different from before. For other exceptions, execution resumes in *exception mode* until the exception repeats, from which point the operating system handles the exception. In exception mode only the Primary Processor operates.

The scheme described for dealing with store instructions is not the only one that would work with the DTSVLIW. An alternative scheme make the stores write into a *data store list* as oppose to the Data Cache, and the checkpoint recovery store list is not used. The data store list contains the address, data, data type, and the order field of store instructions. This list works as a queue for incoming store data. Nevertheless, the order field can be used to transfer this data to the Data Cache in order, which can be useful when using the DTSVLIW for applications requiring intensive in order memory or I/O writing – in the former scheme the Primary Processor has to handle in order data store. Data is only transferred from the data store list after the block containing the respective store instructions has finished without exceptions. In case of an exception, data generated in the block where the exception is detected is annulled. Load instructions read from the Data Cache and from the data store list at the same time, and use the last data stored in the list on a list hit. This scheme has not been used as it is much harder to implement in a simulator, and its advantages need to be identified through further research.

## 4.9 Object Code Compatibility Issues

There are some important issues related to backward code compatibility that have not yet been discussed. These are: self-modifying code, programs that read their own code as data, load instructions side effects, and the implementation of CISC ISAs with the DTSVLIW architecture.

Self-modifying code is a problem even for simple pipelined processors, since a store instruction can change an instruction in memory that is already in some stage of

the processor pipeline. Modern processors use the virtual memory supporting hardware for detecting attempts to modify code, and they generate exceptions when this happens [Patterson96 (page 449)]. The DTSVLIW can use the same mechanism. If the execution of self-modifying code is necessary, the exception handler just needs to flush the entire instruction and VLIW caches. If self-modification is frequent in such code, the performance is going to be seriously affected, however. Nevertheless, this is an uncommon programming practice usually present in old programs only. Therefore, support for the execution of self-modifying code is limited in the DTSVLIW architecture and, if performance is required, specially designed architectures should be used for the execution of this kind of code.

Certain programs need to read their own code, such as programs that compute their own checksum for security reasons for example. The DTSVLIW architecture does not impose any restrictions on the execution of such programs since the scheduled code in the VLIW Cache is not accessible by load/store instructions.

In some systems, load instructions may have side effects due to memory mapped I/O. In these systems, a load may reset an I/O register for example. Since the DTSVLIW can execute loads speculatively, programs could have incorrect behaviour due to a detected but not committed exception prior to the execution of such a speculative load that may cause side effects. This does not happen, however, because the DTSVLIW architecture includes a data cache and must already be aware that some types of data cannot be stored in the cache. Load accesses that change system state are a subset of *non-cacheable* memory accesses and as such are non-schedulable instructions (see Subsection 4.1.7).

The description of the DTSVLIW until here assumes the implementation of an RISC ISA; however, the DTSVLIW architecture can also be used to implement CISC ISAs. The most direct way of doing this is to implement the Primary Processor with a vertical microprogrammed machine. The RISC like microinstructions executed by this Primary Processor could then be dealt with in the same way as has been described for a RISC ISA implementation.

## 4.10 Differences Between DTSVLIW and DIF

The DTSVLIW architecture differs from the DIF architecture in the organisation of the instruction cache used by the VLIW Engine, in its scheduling algorithm, in its register renaming, and in the VLIW Engine register access mechanism.

The unit of communication between the DIF cache and its VLIW Engine is an entire block of long instructions, whereas the DTSVLIW machine accesses one long instruction per VLIW Cache access. We believe that this should make the DTSVLIW VLIW Cache implementation simpler than a DIF VLIW cache implementation.

A DIF machine schedules instructions using a hardware table, which has as many entries as resources in the machine and records the earliest long instruction in which each resource is available. Its proposed scheduler implements the Greedy algorithm, by checking all resources necessary for each new instruction against this table, and scheduling the instruction in the earliest long instruction possible. The DIF scheduling scheme is similar to the MPS scheme described in Subsection 3.3.2, although more complex, and suffer from the same pipelining problems of MPS. Nair and Hopkins have not addressed these pipelining problems in [Nair97]. The DTSVLIW, on the other hand, uses a simplified pipelined version of the FCFS algorithm, which operates over a list of long instructions. An instruction has only to be checked for dependencies against other instructions in its current and next position in the list, as opposed to all resources available in the machine.

Instead of using copy instructions to implement register renaming, a DIF machine has a number of instances of each ISA register and extra bits are added to each register specifier to specify the register being used during VLIW execution. A register-mapping table is used to access the current ISA register set. Renaming is performed by specifying the extra bits during scheduling and by copying the new register mapping – the *exit map* – to the register-mapping table every time the execution leaves a block. Each exit point of a block (all branches and the final long instruction) has to carry its own exit map. This mechanism may not be practical for machines with a large number of physical registers, however. The Sparc ISA, for example, allows processor implementations with as many as 520 integer registers due to its register windows. Although most Sparc processors have only 128 integer registers, a single exit map for such a processor, with four instances of each register,

would require 256 bits only for integer registers. The DTSVLIW splits instructions with the purpose of renaming registers to overcome data and control dependencies and the copy instructions generated are simpler to handle than mapping tables.

The DIF VLIW Engine accesses its register file differently to the DTSVLIW. It has to translate each register specifier to access the register file during VLIW execution because of its renaming mechanism – this translation is in the data path of the DIF VLIW Engine. A DTSVLIW machine, on the other hand, accesses its register file directly.

# Chapter 5

# Experimental Methodology

We have used experiments to evaluate the DTSVLIW architecture. In order to perform these experiments, we have implemented a parameterised and instrumented simulator of the DTSVLIW. This simulator is capable of executing ordinary programs for the Sparc Version 7 ISA compiled with standard compilers. Integer programs from the SPEC92 and SPEC95 benchmark suites have been compiled and used as input for the simulator. The data gathered is presented and discussed in the chapters that follow this.

In the rest of this chapter, we describe the DTSVLIW simulator, present the DTSVLIW parameters held constant and those we have varied during the simulations, the benchmark programs, and the metrics we have used to gauge the DTSVLIW performance.

## 5.1 The DTSVLIW Simulator

The DTSVLIW simulator has been implemented in C (23K lines of code), it is parametric, allowing the specification of various parameters of the DTSVLIW architecture, and it performs *execution-driven* simulation.

In execution-driven simulation, the simulator that models the machine under study fully executes the test programs used in the study. This is in contrast with *trace-driven* simulation, where instruction traces of test programs are produced in a simple simulator and then used to feed a program that models the machine under

study. Trace-driven simulation is usually faster but often less precise than execution-driven simulation because assumptions have to be made about instructions that are not in the trace but might have been fetched or even speculatively executed in a real machine. In order to avoid the need for such assumptions, we have decided to implement our simulator as an execution-driven simulator.

To guarantee correct simulation results, all results have been produced with the simulator running in a special mode called *test mode*. The test mode puts two machines to run together: the DTSVLIW and a *test machine* with the same characteristics of the Primary Processor of the DTSVLIW. The DTSVLIW starts first, and every time an instruction or a block of long instructions is completed, the simulator switches to the test machine, which runs until its PC becomes equal to the DTSVLIW PC. The Sparc ISA state of both machines is compared and, if not equal, an error is signalled and the simulation interrupted. The test mode has been very useful not only to validate the execution but also because in this mode it is possible to measure the precise number of instructions necessary for the execution of a program, which the test machine can provide. A DTSVLIW simulator alone cannot provide this number due to copy instructions and instructions executed speculatively.

The simulator fully executes all user-level instructions, including instructions that are part of linked libraries; however, it does not execute operating system instructions. When a system call occurs during simulation, a module of the simulator intercepts it. This module decodes the system call, copies its arguments, makes the corresponding system call to the host's operating system, copies the results of the system call into the simulated program's memory, and then restarts the execution of the simulated program.

The simulated DTSVLIW's Primary Processor fully implements the Sparc Version 7 ISA [Sun87]. The simulator receives as input any SunOS 4.1.3 (Sun operating system) executable for this ISA and faithfully models the DTSVLIW architecture operation.

## 5.2 Simulation Parameters

Except when stated otherwise, each program was run for 50 million or more instructions each experiment, as counted by the test machine. We have chosen to run

this number of instructions because this is optimistically the number of instructions that a DTSVLIW machine is capable of executing between operating system context switches. (Supposing that the DTSVLIW can execute 5 instructions per cycle, a clock rate of 1 GHz, and one context switch every 10ms.)

We cannot always specify the exact number of executed instructions because, when the simulator is executing code in VLIW-mode, the exact number of executed instructions is only known when the execution leaves a block. That is the reason why we say that "each program was run for 50 million *or more* instructions each experiment," and not exactly 50 million instructions.

The DTSVLIW parameters that are invariant for all simulations are presented in Table 5.1, while the parameters that we have varied to appreciate their influence on the DTSVLIW performance are shown in Table 5.2 together with their default values. Except when stated otherwise, the default values were used in the simulations.

**Table 5.1: Fixed parameters**

| | |
|---|---|
| Trace Processor | • four-stage (fetch, decode, execute, and write back) pipeline<br>• not taken branches cause a 2-cycle bubble in the pipeline (the Sparc ISA's delayed branches allow for zero-bubble taken branches)<br>• instructions following a load, requiring the data loaded cause a one-cycle bubble in the pipeline |
| Decoded Instruction Size | 6 bytes |
| VLIW Engine List Sizes | load = store = checkpoint recovery store = unlimited |
| Scheduler Unit Pipeline | inserting = 1 stage<br>splitting and moving up = block-size stages<br>saving = 1 stage |

**Table 5.2: Variable parameters**

| **Parameter** | **Default Value** |
|---|---|
| Number of VLIW Engine functional units | equal to the long instruction size |
| VLIW Engine functional units type | untyped (can execute any instruction) |
| Number of renaming registers | unlimited |
| Next long instruction miss penalty | no penalty (0-cycle) |
| Instructions latency | 1-cycle |
| VLIW Cache size | 3072-Kbyte |
| VLIW Cache associativity | 4-way |
| Instruction Cache | perfect (no miss penalty) |
| Data Cache | perfect (no miss penalty) |

## 5.3 Benchmark Programs

In 1988, the *Standard Performance Evaluation Corporation* (SPEC) was established as a non-profit corporation devoted to "establishing, maintaining and endorsing a standardized set of relevant benchmarks that can be applied to the newest generation of high-performance computers" [SPEC]. SPEC is supported by leading companies in the processor market, such as Intel and IBM.

SPEC's first product was the SPEC Benchmark Release 1 suite, currently identified as SPEC89. Since then, two other releases have been developed: SPEC92 and SPEC95 benchmark suites. These new releases became necessary due to advances in processors, systems, and compiler technology, and the improvements from one version to another are in aspects such as execution time, application size, application type, etc.

The SPEC benchmark programs and their respective input data used in the experiments reported in this thesis are shown in Table 2. We have used integer programs from the SPEC95 and SPEC92 benchmark suites, although the experiments with SPEC92 programs have only been included for comparison with previously published results. All programs have been compiled with the gcc 2.7.2 compiler, using optimisation flag -O. In this level of optimisation, the gcc compiler performs several optimisations such as automatic register allocation, common sub-expression elimination, invariant code motion from loops, induction variable optimisations, constant propagation and copy propagation, filling of delay slots, etc. We could have used the higher levels of optimisation -O2 or -O3; however, these levels include optimisations such as loop-unrolling and function inlining whose effect in the DTSVLIW performance would require a careful study in isolation. We have left the study of the compiler-DTSVLIW architecture interaction for future work.

**Table 5.3: Benchmark programs and Input Data**

| SPEC95 Benchmarks | Description | Inputs |
|---|---|---|
| compress | Unix utility. Compresses data using the adaptive Limpel-Ziv coding. The data is generated randomly. | 20000 q 2131 |
| gcc | C compiler. Compiles pre-processed source into optimised Sparc assembly code. | -O3 jump.i |
| go | Game. Plays the game Go against itself. | 40 19 null.in |
| ijpeg | Image processing tool. Performs jpeg image compression. | vigo.ppm –GO |
| m88ksim | Simulator. Simulates the Motorola 88100 processor running a program. | dhry.big |
| perl | Shell Interpreter. Performs text and numeric manipulations. | primes.pl |
| vortex | Database. Builds and manipulates three interrelated databases. | vortex.in |
| xlisp | Lisp interpreter. Interprets lisp programs. | queens 7 |
| **SPEC92 Benchmarks** | **Description** | **Inputs** |
| compress | Unix utility. Compresses files using the adaptive Limpel-Ziv coding. | in |
| eqntott | Logic design tool. Translates a logical representation of a boolean equation to a truth table. | int_pri_3.eqn |
| espresso | Logic design tool. Minimise boolean functions. | cps.in |
| gcc | C compiler. Compiles pre-processed source into optimised Sparc assembly code. | -O jump.i |
| xlisp | Lisp interpreter. Interprets lisp programs. | queens 7 |

## 5.4 Metrics

The *instructions per cycle* (IPC) index is the main performance measurement index used in this thesis. It has been produced by dividing the number of instructions necessary to execute the program, as counted by the test machine, by the number of cycles taken for DTSVLIW execution.

We refrain from averaging benchmark performances most of the time and show performance measurements for each individual benchmark. However, sometimes averages are useful. Jacob and Mudge [Jacob95], and Giladi and Ahituv [Giladi95] have discussed which average should be used when dealing with computer performance indices and have suggested the use of the harmonic mean for indices like IPC. Therefore, when appropriate, we use the harmonic mean. Nevertheless, we

also show, for extra clarity, the arithmetic mean between parenthesis at the side of the harmonic mean in the form (4.1 u.a.m.), where u.a.m. means *using arithmetic mean*.

# Chapter 6

# Experiments

In this chapter we present and discuss the experiments carried out to evaluate the integer performance of the DTSVLIW architecture. We start by examining the effect of some architecture parameters on the DTSVLIW performance. We then evaluate the effectiveness of the DTSVLIW instruction-scheduling algorithm by comparing it with the FCFS algorithm, used in microcode compaction, and the Greedy algorithm, used in the DIF architecture. Finally, we present comparisons between the DTSVLIW and the DIF, pure VLIW, and Superscalar architectures.

## 6.1 Effect of Some Architectural Parameters on the DTSVLIW Performance

In this section, the effect of several parameters of the DTSVLIW architecture upon its performance is examined.

### 6.1.1 VLIW Fetch Starting Point

When a DTSVLIW machine is scheduling code, every time a valid instruction moves from the Primary Processor's decode pipeline stage to the Primary Processor's execute pipeline stage, a VLIW fetch can be attempted with its address. On a VLIW Cache hit, the VLIW Engine takes over execution and the block being scheduled is saved into the VLIW Cache. If no special action is taken, the blocks produced this way can have any number of long instructions from 1 to block size, but small blocks

(1 or 2 long instructions) are likely not to have much parallelism. However, instead of always allowing VLIW fetches, we can easily allow VLIW fetches only when the size of the block being scheduled is near its maximum. That is, we can establish a starting point for VLIW fetches associated with the size of the current block being scheduled, forcing the production of larger and, hopefully, more compact (parallel) blocks.

Figure 6.1 shows the impact of the VLIW fetch starting point on the DTSVLIW performance. We have used DTSVLIW machines with 8 instructions per long instruction and 8 long instructions per block (8x8-block geometry) in this experiment. To ensure the absence of extraneous effects, the experiments leading to the results in this figure were performed with perfect instruction and data caches (no miss penalty), large VLIW Cache (3072-Kbyte), and no next long instruction miss penalty. The legend of Figure 6.1 shows the different VLIW fetch starting points used: at any block size (Start at 0), at half of the maximum block size (Start at 1/2), at three quarters of the maximum block size (Start at 3/4), and at full block (Start at 1/1).

As the graph in Figure 6.1 shows, the strategy of discarding some possible VLIW fetch opportunities to favour the production of larger blocks is worthwhile. The machine configuration with VLIW fetch starting point at 1/2 of the maximum block size performs better than the configuration with starting point at any block size in seven of the eight benchmarks. The configuration with starting point at 3/4 of the maximum block size has equivalent or better performance than the two previous configurations in all benchmarks.

The configuration that starts issuing VLIW fetches only at full block has an anomalous behaviour, however. It has equivalent or better performance than the previous three configurations discussed in all but the compress benchmark. With compress, this configuration has performance inferior than the other three. This happens because, if VLIW fetches are allowed only when the block is full, the DTSVLIW sometimes does not have many opportunities to perform VLIW fetches and spends too much time executing code in the Primary Processor. This is more apparent when larger blocks are used, as shown in Figure 6.2. In Figure 6.2, we have used DTSVLIW machines with 16 instructions per long instruction and 16 long

instructions per block (16x16-block geometry). With the 16x16-block geometry, the performance is overall better with starting point at 1/2 than with starting point at 3/4 of the of the maximum block size. In addition, in Figure 6.2, the benchmarks compress, ijpeg, and xlisp show clearly that there is a performance maximum when the VLIW fetch starting point is near 1/2 of the block size.

We have performed several experiments to try to find a single rule for computing the VLIW fetch starting point that could be used for all block geometries without causing significant performance loss in cases where the rule does not perfectly fit. We have found that a starting point at 1/2 + 1 of the block size offers a good trade off for large and small block sizes. Figure 6.3 shows the performance of DTSVLIWs with 8x8-block geometry and 16x16-block geometry and two different starting point rules for each block geometry: the new 1/2 + 1 rule and the best rules previously used in the experiments of Figure 6.1 and Figure 6.2. As the graph in Figure 6.3 shows, the 1/2 + 1 rule produces slightly inferior performance (1.8%) than the 3/4 rule for the 8x8-block geometry – harmonic mean of 3.22 (3.31 u.a.m. (see Section 5.4)) IPC versus harmonic mean of 3.28 (3.24 u.a.m.) IPC. On the other hand, for the 16x16-block geometry, the 1/2 + 1 rule produces significantly better performance (4.6%) than the 3/4 rule – 4.29 (4.53 u.a.m.) IPC versus 4.10 (4.28 u.a.m.) IPC. Thus, in the face of these results, we have chosen to configure the DTSVLIW to start fetching from the VLIW Cache at 1/2 + 1 of the maximum block size in the remained experiments discussed in this thesis. It does not mean that we believe that 1/2 + 1 rule should be used for all DTSVLIW implementations, but only that it is an approximation of the optimal rule. In a silicon implementation, a larger set of experiments must be made, with block geometry and other machine parameters set, in order to determine the adequate VLIW fetch stating point.

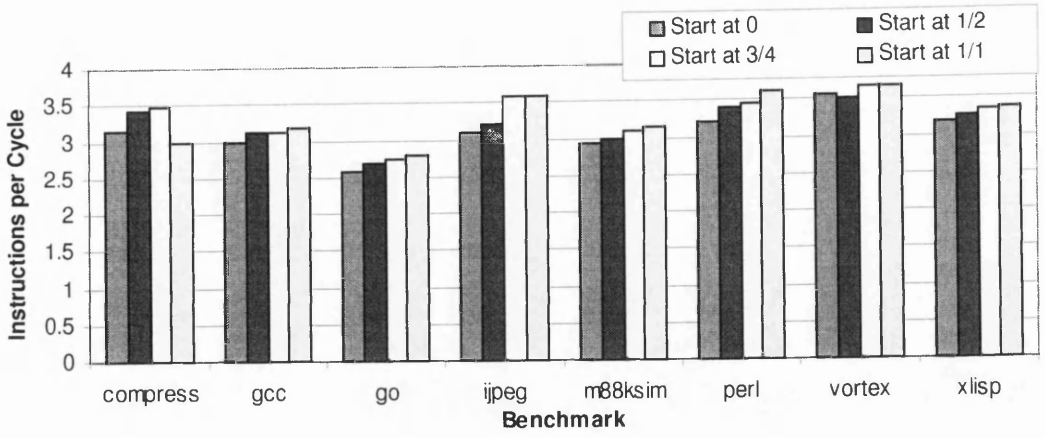



**Figure 6.1: Variation of parallelism with VLIW fetch starting point: 8x8-block**

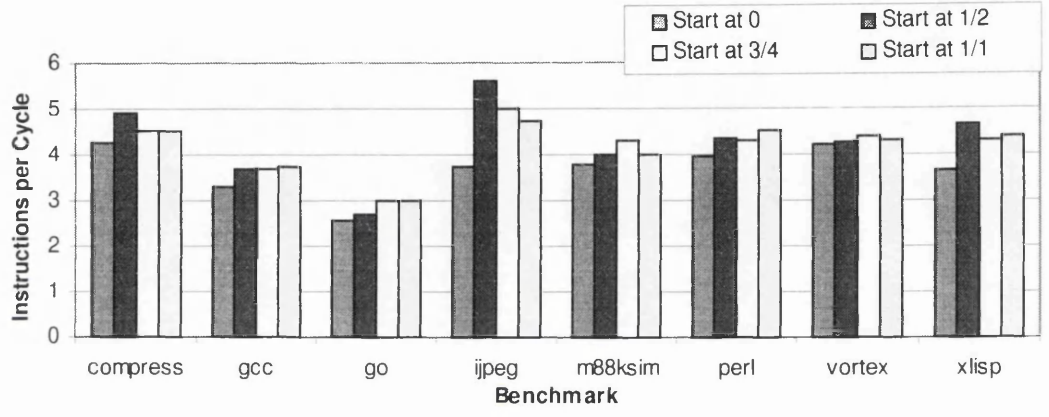



**Figure 6.2: Variation of parallelism with VLIW fetch starting point: 16x16-block**

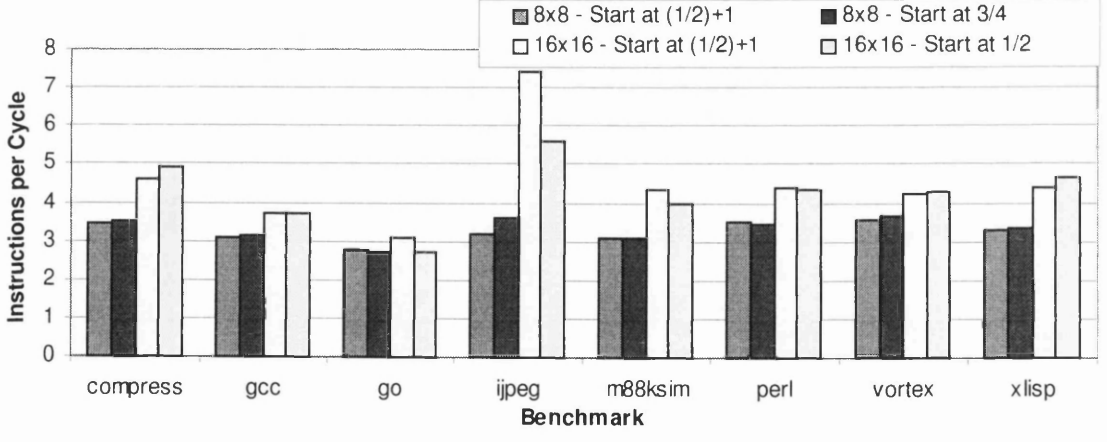



**Figure 6.3: Variation of parallelism with VLIW fetch starting point: 8x8-block and 16x16-block**

### 6.1.2 Block Size and Geometry

Figure 6.4 shows the effect of the block size (in number of instructions) and block geometry (instructions per long instruction (width) versus long instructions per block (height)) on the DTSVLIW performance. To ensure the absence of extraneous effects, we have used the same experimental set-up of the previous section to produce the results shown in Figure 6.4: perfect instruction and data caches (no miss penalty), large VLIW Cache (3072-Kbyte), and no next long instruction miss penalty. The numbers in the figure's legend are instructions per long instruction and long instructions per block, respectively.

As the graph in Figure 6.4 shows, the performance of machines with the same block sizes and different geometries is significantly different. For example, the performance of the machine with 4x8-block geometry is lower than the machine with 8x4-block geometry for all benchmark programs. The block width and height affect the cost of implementing a DTSVLIW machine in different ways. Large long instructions imply many functional units, Data Cache ports, and register file ports. A large number of long instructions in a block implies many renaming registers, and long load/store and checkpoint recovery store lists (see Section 4.7 and Section 4.8). To increase just the width or just the height of the block does not appear to be the best approach to achieve cost/effective performance – a DTSVLIW with 8x8-block geometry performs better than machines with 4x16-block geometry and 16x4-block geometry in the majority of the SPECint95 benchmarks. The DTSVLIW benefits from large block sizes but not linearly. A 16-fold increase in the number of instructions of a block (from 4x4 to 16x16) does not quite double its performance.

The performance of the 16x16 configuration on the ijpeg benchmark is extraordinary and has been investigated. This benchmark spends most of its execution in one loop. With a large enough block size, more than one iteration of the loop can be scheduled into a single block, allowing instructions from these iterations to be overlapped, extracting much greater parallelism. (In Figure 4.2, instruction 5 of the second loop iteration overlaps with instructions of the first.)

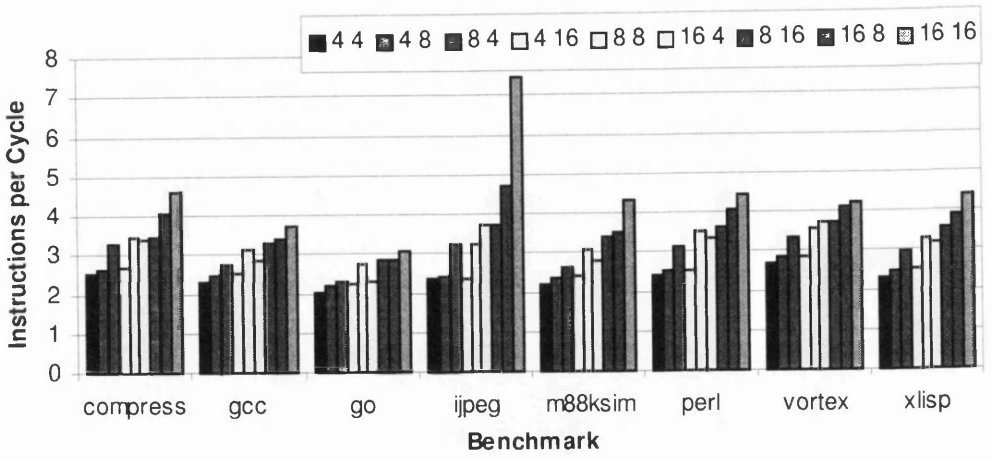



**Figure 6.4: Variation of parallelism with block size and geometry**

### 6.1.3 VLIW Cache Size

The results of Figure 6.4 represent the highest achievable SPECint95 performance of the DTSVLIW for the block sizes shown. However, when the VLIW Cache is smaller, the performance is expected to be lower due to premature flushing of useful scheduled blocks by replacement blocks, leading to the need to rebuild the blocks flushed. This requires the Primary Processor to run, reducing parallelism.

Figure 6.5 shows the impact of different VLIW Cache sizes (in Kbytes, the directory information is not included) on the performance of a DTSVLIW machine with 8x8-block geometry. The associativity is the same for all sizes and equal to 4. As the graph shows, some benchmark programs do not demand a large VLIW Cache in order to exploit the performance of the DTSVLIW. The benchmarks compress, ijpeg, and xlisp have small instruction working set [Charney97] and are insensitive to the VLIW Cache size, achieving the same performance for the range of sizes used. However, go, which has a large working set [Charney97], would appear to benefit from a VLIW Cache larger than 3072-Kbyte.

Some benchmarks sometimes show better performances with smaller VLIW Caches, as for example compress for the 48-Kbyte and 96-Kbyte run, and xlisp for the 96-Kbyte run. This happens because, with a small VLIW Cache, sometimes blocks have to be replaced and later rescheduled, and the newer block versions contain traces that are more frequently executed to the end than the replaced blocks.

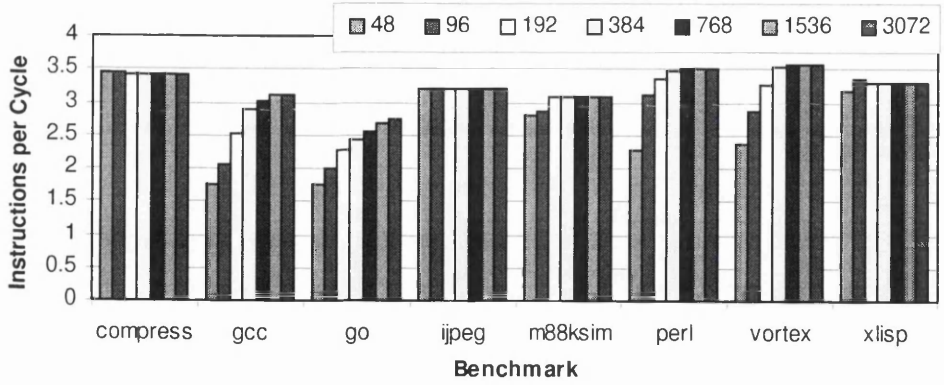



**Figure 6.5: Variation of the parallelism with VLIW Cache size**

### 6.1.4 VLIW Cache Associativity

Figure 6.6 shows the effect of the VLIW Cache associativity on the performance of the DTSVLIW. Two cache sizes are presented: 96-Kbyte and 384-Kbyte, and the associativity is varied from 1 to 8. The figure shows that ijpeg is insensitive to the VLIW Cache associativity in this range; however, m88ksim, perl, xlisp, and compress (for the 96-Kbyte cache) benefit from extra associativity. From Figure 6.5 and Figure 6.6 it is possible to infer that a two- or four-way set-associative 384-Kbyte VLIW Cache offers a cost-effective solution for a DTSVLIW machine with 8x8-block geometry.

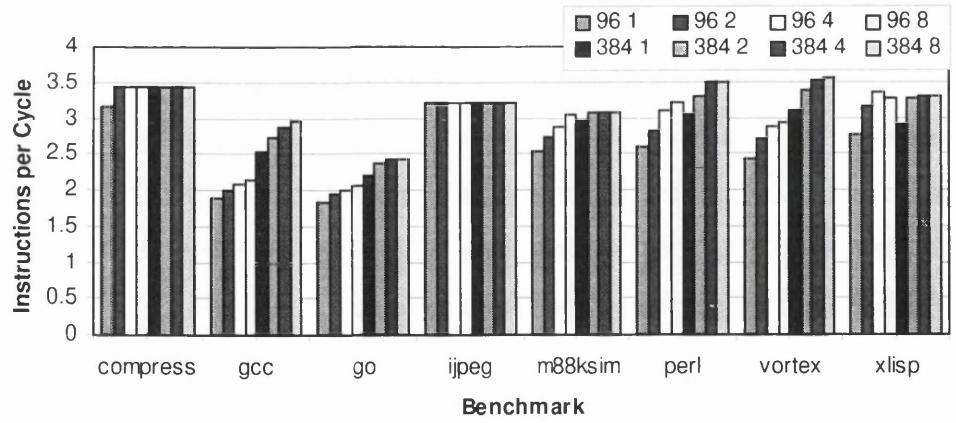



**Figure 6.6: Variation of parallelism with VLIW Cache associativity**

### 6.1.5 Instruction Cache Size

Figure 6.7 shows the effect of the Instruction Cache on the performance of a DTSVLIW machine with a 3072-Kbyte VLIW Cache. Five Instruction Cache

configurations with increasing implementation costs and access times are presented: 256-byte fully associative, 1-Kbyte 2-way set associative, 4-Kbyte direct mapped, 16-Kbyte direct mapped, and perfect (no miss penalty). The line size for all configurations is 64-byte (16 instructions) and the miss penalty is 8 cycles (except for the perfect configuration, of course). The 256-byte fully associative configuration has only four lines (256/64); therefore, it should not have an access time longer than the 1-Kbyte configuration.

As the graph in Figure 6.7 shows, the impact of Instruction Cache misses on the performance of a DTSVLIW machine with a large VLIW Cache is negligible. The performance loss, when compared with a perfect Instruction Cache, of the smallest configuration is only more than 1% (1.54%) for gcc. However, a 3072-Kbyte VLIW Cache is hardly implementable in an efficient way with current technology and, with small VLIW Caches, the Instruction Cache may be more important for the overall DTSVLIW performance.

Figure 6.8 shows the effect of the Instruction Cache on the performance of a DTSVLIW machine with a 192-Kbyte VLIW Cache. Five Instruction Cache configurations, similar to those used in previous experiment, are presented. These are: 512-byte fully associative, 1-Kbyte 4-way set associative, 8-Kbyte 2-way set associative, 32-Kbyte direct mapped, and perfect (no miss penalty). The line size and the miss penalty for all configurations are the same as in the previous experiment: 64-byte and 8 cycles, respectively.

As shown in Figure 6.8, the same benchmarks that are sensitive to the VLIW Cache capacity (see Figure 6.5 and Figure 6.6) are also sensitive to the Instruction Cache capacity. These benchmarks, namely gcc, go, perl, and vortex, have an instruction working set of significant size [Charney97] and, therefore, require large caches for instructions.

From Figure 6.8 is possible to see that, if the VLIW Cache is not large enough, a significant number of blocks may have to be rescheduled. If the instructions that are necessary to build these blocks are not in the Instruction Cache due to lack of capacity, misses will occur and the overall machine performance will be affected. On the other hand, Figure 6.7 shows that, if the VLIW Cache is large enough, the Instruction Cache can be substituted for a simple instruction buffer.

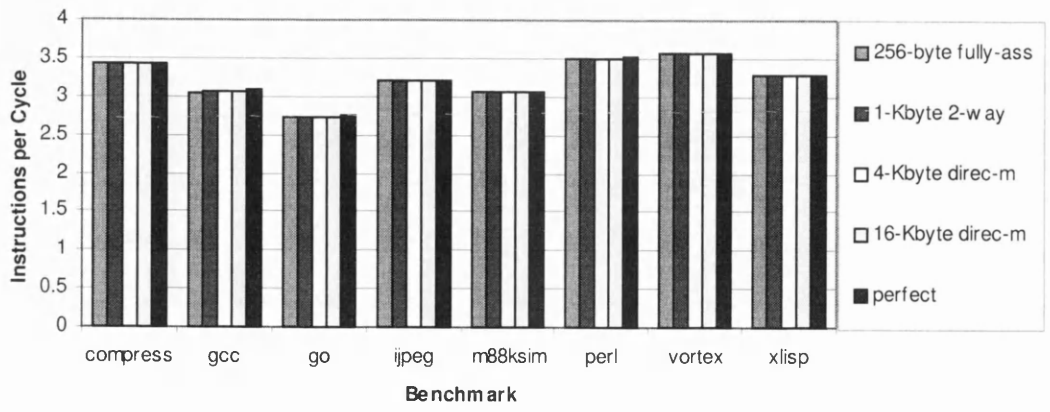



**Figure 6.7: Variation of parallelism with different Instruction Cache configurations: 3072-Kbyte VLIW Cache**

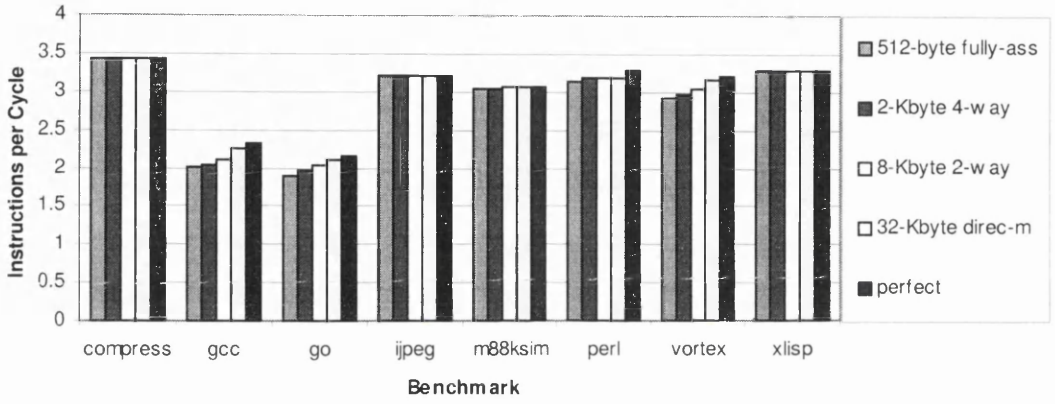



**Figure 6.8: Variation of parallelism with different Instruction Cache configurations: 192-Kbyte VLIW Cache**

### 6.1.6 Multicycle Instructions Latency

The Sparc 7 ISA does not have integer divide or multiply instructions, but only a multiply-step instruction that executes in a single cycle [Sun87]. Therefore, from the set of integer instructions, only loads and stores require more than one cycle to execute in this ISA.

The graph in Figure 6.9 shows the effect of the load/store instructions latency on the performance of the DTSVLIW with 8x8-block geometry. In the figure's legend, LxSy stands for load instructions with latency of $x$ and store instructions with latency of $y$. In these results, we express the latency as the number of cycles necessary for the load/store execution and the functional units of the VLIW Engine are fully pipelined.

As the graph in the figure shows, the latency of load instructions has a severe

impact in the DTSVLIW performance – 25.9% (25.3% u.a.m.) performance loss with 2-cycle and 50.7% (50.2% u.a.m.) with 3-cycle load latency. This occurs because load instructions are frequent in integer code and loaded data is usually required shortly after the load instructions. On the other hand, the latency of store instructions does not have a strong impact on the DTSVLIW performance. Store instructions are also frequent in integer code; however, the stored data is often not required again for a relatively large number of instructions.

The graph in Figure 6.10 shows the impact of load/store latency on the performance of a DTSVLIW machine with 8x16-block geometry. The impact is smaller with this geometry – 20.6% (20.4% u.a.m.) performance loss with 2-cycle load latency and 43.1% (42.7% u.a.m.) with 3-cycle load latency on average. With a longer block, the Scheduler Unit has more opportunities to accommodate instructions in the empty long instructions created by the scheduling of multicycle loads. This results in better scheduling and better performance, but the latency impact is still high and there are costs for using long blocks (see Subsection 6.1.2).

As mentioned in Subsection 4.1.8, the Primary Processor does not pipeline multicycle instructions but retains them on its execute pipeline stage until they complete execution. Table 6.1 presents the percentage of cycles the DTSVLIW spends waiting for these multicycle instructions to complete in the Primary Processor. As the table shows, the cost of waiting in the Primary Processor is very small and it is not an issue in the impact of the load/store latency on the DTSVLIW performance. Table 6.1 also presents the percentage of VLIW execution cycles for the DTSVLIW with 8x16-block geometry. This machine configuration executes 98.56% (98.57% u.a.m.) of the cycles in VLIW mode on average. This strongly suggests that the DTSVLIW architecture is effective in taking advantage of its VLIW Engine.

There are two simple approaches for reducing the impact of the load instruction latency on the DTSVLIW performance. The first is always to implement loads with 1-cycle latency. This would, in some cases, almost double the DTSVLIW clock cycle length and, therefore, it may not be a cost-effective alternative. The second is to implement loads with 1-cycle latency and to use a small and fast Data Cache (8-Kbyte direct mapped, for example). This would increase the Data Cache miss rate,

but it may be a more cost-effective alternative. Another approach to further reduce the impact of loads on the DTSVLIW performance is to use hardware- or software-implemented data prefetching [VanderWiel97]. This approach can also be used together with one of the former approaches.

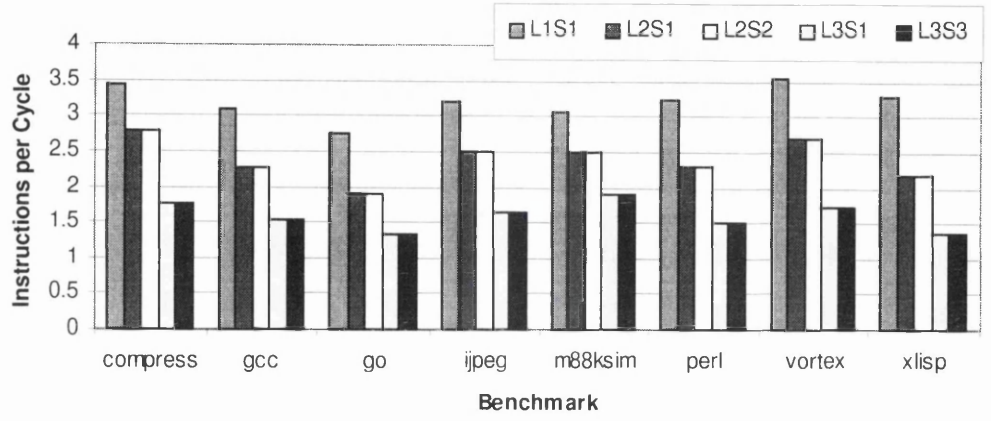



**Figure 6.9: Variation of the parallelism with load/store instructions latency – 8x8-block**

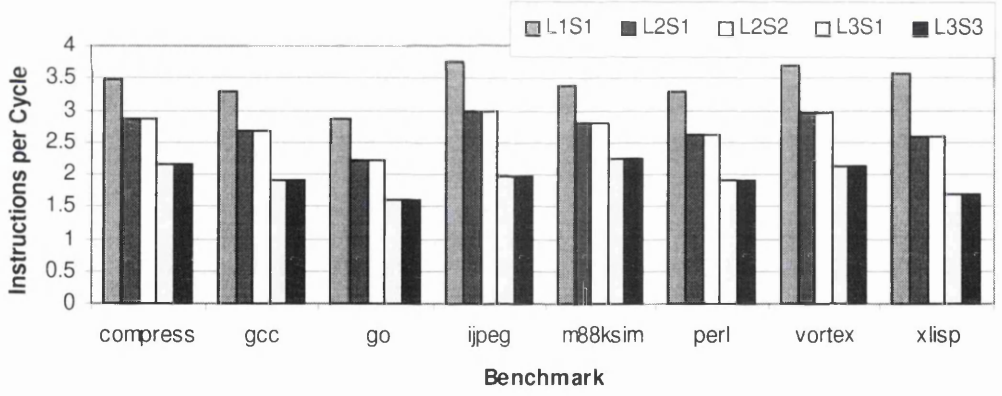



**Figure 6.10: Variation of the parallelism with load/store instructions latency – 8x16-block**

**Table 6.1: Percentage of cycles waiting load/store latency in the Primary Processor and percentage of VLIW execution cycles**

| | compress | gcc | go | ijpeg | m88ksim | perl | vortex | xlisp | H. Mean | A. Mean |
|---|---|---|---|---|---|---|---|---|---|---|
| Waiting Cycles (8x8) | 0.14% | 0.11% | 0.08% | 0.00% | 0.01% | 0.01% | 0.03% | 0.01% | 0.00% | 0.05% |
| Waiting Cycles (8x16) | 0.11% | 0.18% | 0.21% | 0.00% | 0.01% | 0.02% | 0.05% | 0.02% | 0.01% | 0.08% |
| VLIW Cycles (8x16) | 97.97% | 98.20% | 97.49% | 99.98% | 99.87% | 96.37% | 98.89% | 99.80% | 98.56% | 98.57% |

### 6.1.7 A Feasible DTSVLIW Machine Configuration

So far, the results presented have been produced under ideal assumptions to allow appreciation of individual architecture parameters. However, the DTSVLIW architecture permits straightforward implementation using current VLSI technology if reasonable design parameters are used. The graph in Figure 6.11 presents the performance of a DTSVLIW machine with a set of parameters that permits implementation using available technology. These parameters are:

- Blocks with 16 long instructions and 8 instructions per long instruction.
- 12-wide VLIW Engine, with typed (specialised) functional units and 2-cycle next long instruction miss penalty. The functional units used are: 5 integer, 3 load/store, 2 floating-point, and 2 branch functional units. Although this VLIW Engine has twelve functional units, the VLIW fetch is 8-instruction wide because the block is 8-instruction wide. This is the main reason why we are using 2-cycle next long instruction miss penalty. One extra cycle has been added to the VLIW Engine pipeline to allow the unpacking of 8-instruction wide long instructions, which are fetched from the VLIW Cache, into the 12-instruction wide long instructions required by VLIW Engine. The Scheduler Unit is conscious of the number of functional units available and schedules the 8-instruction wide long instructions respecting their availability per cycle.
- 2-cycle latency load instructions and 1-cycle latency store, integer (the Sparc 7 ISA does not have integer divide or multiply but only multiply-step, which can execute in one cycle), branch, and floating-point instructions. Latency of one cycle is a low latency for floating-point instructions; nevertheless, this latency has been used because the benchmarks are integer and, therefore, the number of floating-point instructions executed is zero or negligible.
- 192-Kbyte 4-way set-associative VLIW Cache with 2-cycle fully pipelined access.
- 8-Kbyte 2-way set-associative Instruction Cache with 1-cycle access and 8-cycle miss penalty.
- 32-Kbyte 2-way set-associative Data Cache with 2-cycle fully pipelined access and 8-cycle miss penalty.
- Perfect (pre-initialised with all instructions and data) unified second level cache.

The number of entries of the VLIW Engine lists (load, store, and checkpoint recovery store) and the number of renaming registers has been left unlimited. However, the maximum number of entries required for these lists and the maximum number of renaming registers used during the simulation have been measured and are shown in Table 6.3.

Figure 6.11 puts together, in form of stacked bars, the result of various simulations to allow appreciation of the impact of various architectural parameters in the performance of this DTSVLIW machine. The first bar is the performance of the DTSVLIW with the parameters presented. The following bars represent the extra performance that would be added if the corresponding cost (shown in the legend) was removed. Table 6.2 shows these costs as percentages of the maximum performance, together with other relevant information. Some items in this table do not have harmonic mean because it cannot be computed when the list of values contains zeros.

As the graph in Figure 6.11 and in Table 6.2 show, the load instruction latency is the principal contributor to the reduction of this DTSVLIW machine performance, and its cost is significant for all benchmark programs used. On the other hand, the second most important parameter that affects the machine performance – the VLIW Cache size – has a significant impact only on the gcc, go, and vortex benchmarks. It is important to note, however, that, although large, this VLIW Cache can hold only 256 blocks and only 35% (36% u.a.m.) of the instructions saved in these blocks during the simulations have been valid (Table 6.2, last row). Therefore, if nop instructions had not been saved in the VLIW Cache, its capacity would have been better used and the performance of gcc, go, and vortex would have been significantly better. In addition, if the VLIW Cache capacity had been better used, the Instruction Cache could have been even smaller than described. Instruction Cache misses cause significant performance losses only for gcc, go, and vortex and, as discussed in Subsection 6.1.5, if the VLIW Cache capacity is large enough, the size of the Instruction Cache can be smaller than the size used.

Next long instruction misses have a small, although significant, impact on the machine performance. If nop instructions were not saved in the VLIW Cache, a more elaborated VLIW fetch would be required, which would result in the need for the next long instruction miss penalty to be even higher than 2-cycle. However, the next

long instruction address can be predicted and the number of misses reduced using techniques similar to those used in dynamic branch prediction.

The impact of using typed (specialised) as opposed to untyped functional units (capable of executing all instructions) is also small. This means that, the combination of specialised functional units used is in good balance with the ILP available in the benchmarks.

Table 6.3 shows that the number of renaming registers required for execution are within a range that does not cause significant cycle time increase due to register file size. The VLIW Engine lists (load, store, and checkpoint recovery store) do not reach unacceptable sizes either, and they can be implemented without imposing extra penalty on the cycle time. However, since the number of aliasing exceptions is low (Table 6.2), a cheaper aliasing exception detection and recovery mechanism is advisable.

A performance of 1.89 (2.01 u.a.m.) instructions per cycle in a machine with 12 functional units seems to be low. However, experiments with the PowerPC620, an aggressive Superscalar machine with 6 functional units, have shown an average (arithmetic mean) of 1.2 instructions per cycle only [Patterson96 (page 341)]. Taking into consideration that DTSVLIW machines can be implemented with clock speed higher than equivalent Superscalar machines such as the PowerPC620, it appears to be worth implementing DTSVLIW machines with current technology. Simple machines with fast clocks have proved to be more powerful than their more complex counterparts [Smith_JE94]. In addition, DTSVLIW machines using equivalent hardware can perform better than Superscalars (see Section 6.5).

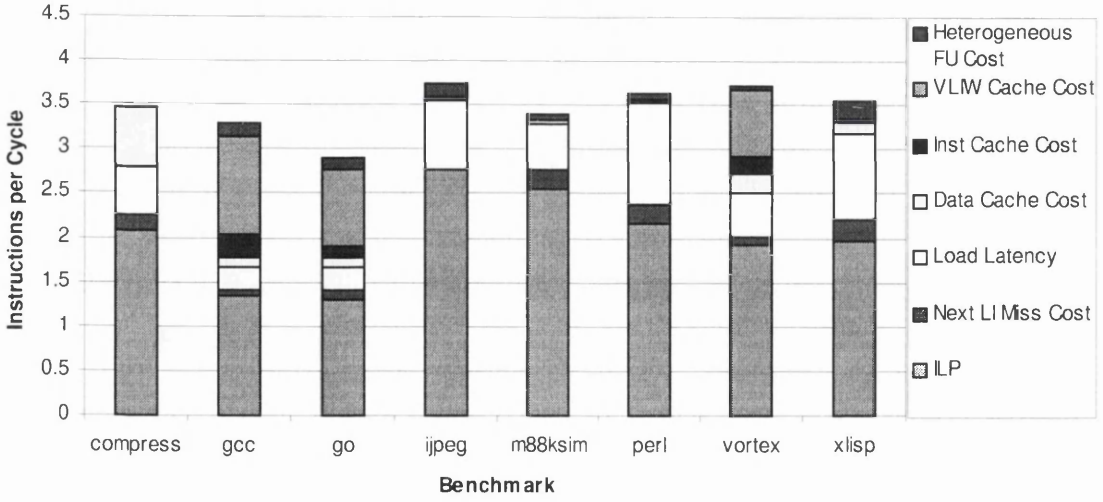



**Figure 6.11: Performance of a feasible DTSVLIW machine**

**Table 6.2: Performance data of a feasible DTSVLIW machine**

| | compress | gcc | go | ijpeg | m88ksim | perl | vortex | xlisp | H.Mean | A.Mean |
|---|---|---|---|---|---|---|---|---|---|---|
| Instructions per Cycle | 2.07 | 1.33 | 1.31 | 2.77 | 2.56 | 2.17 | 1.92 | 1.98 | 1.89 | 2.01 |
| Load Latency Cost | 15.64% | 8.16% | 9.22% | 20.47% | 15.53% | 31.67% | 13.26% | 27.32% | 14.50% | 17.66% |
| VLIW Cache Cost | 0.00% | 33.27% | 29.64% | 0.00% | 0.01% | 0.00% | 20.26% | 1.31% | - | 10.56% |
| Data Cache Cost | 19.24% | 3.39% | 3.30% | 1.38% | 0.98% | 0.18% | 5.69% | 3.58% | 0.95% | 4.72% |
| Next LI Miss Cost | 5.37% | 2.23% | 3.38% | 0.03% | 5.84% | 5.49% | 2.74% | 6.72% | 0.22% | 3.97% |
| Typed F.U. Cost | 0.00% | 4.48% | 4.34% | 3.98% | 1.96% | 1.97% | 1.51% | 5.43% | - | 2.96% |
| Instruction Cache Cost | 0.01% | 7.81% | 4.92% | 0.01% | 0.23% | 0.95% | 5.17% | 0.05% | 0.02% | 2.39% |
| Aliasing Exceptions | 0 | 4 | 39 | 0 | 1 | 0 | 1 | 0 | - | 5.63 |
| VLIW Engine Execution Cycles | 96.43% | 53.42% | 60.47% | 99.97% | 98.47% | 86.56% | 73.58% | 99.09% | 79.19% | 83.50% |
| Valid Instructions per Block | 25.18% | 39.25% | 35.15% | 38.95% | 36.44% | 41.05% | 44.96% | 30.35% | 35.35% | 36.42% |

**Table 6.3: Resource consumption of a feasible DTSVLIW machine**

| | compress | gcc | go | ijpeg | m88ksim | perl | vortex | xlisp | H.Mean | A.Mean |
|---|---|---|---|---|---|---|---|---|---|---|
| Integer Renaming Registers | 33 | 49 | 44 | 24 | 38 | 33 | 39 | 28 | 34.32 | 36.00 |
| Flag Renaming Registers | 17 | 20 | 17 | 15 | 18 | 21 | 18 | 17 | 17.70 | 17.88 |
| Memory Renaming Registers | 14 | 14 | 8 | 7 | 11 | 8 | 15 | 8 | 9.78 | 10.63 |
| Load List Size | 10 | 15 | 16 | 7 | 10 | 11 | 12 | 10 | 10.72 | 11.38 |
| Store List Size | 32 | 15 | 32 | 5 | 13 | 12 | 23 | 14 | 13.24 | 18.25 |
| Checkpoint Rec. Store List Size | 48 | 34 | 48 | 11 | 20 | 31 | 39 | 25 | 25.82 | 32.00 |

## 6.2 Effectiveness of the DTSVLIW Scheduling Algorithm

The scheduling algorithm used in the DTSVLIW architecture and the Greedy algorithm used in the DIF [Nair97] architecture are subsets of the FCFS algorithm [Davidson81], described in Section 3.1; i.e., they are simplifications of the FCFS algorithm. The difference between the Greedy and the FCFS algorithms is that the Greedy algorithm does not implement step 3 of FCFS (see Page 58). That is, the Greedy algorithm never adds long instructions at the top of the scheduling list but only at the bottom. The DTSVLIW algorithm does not add instructions at the top

either and, in addition, only moves up an instruction if there is a slot available in the next long instruction in the list. The FCFS and the Greedy algorithms, on the other hand, can move up an instruction to any available slot in the list, from the next long instruction to the top.

In this section, we evaluate the effectiveness of the DTSVLIW scheduling algorithm by comparing its performance with that of FCFS and Greedy algorithms. To perform the comparisons, we have modified our DTSVLIW simulator to make it able to use the FCFS and Greedy algorithms and have performed experiments using the SPECint95 benchmark suite.

### 6.2.1 Evaluation of the Performance of the DTSVLIW, Greedy, and FCFS Scheduling Algorithms – Untyped Functional Units

In order to compare the three scheduling algorithms, we have chosen to use three different block geometries: 4x4, 8x8, and 16x16. To ensure the absence of extraneous effects and identical conditions, the experiments were performed with perfect instruction and data caches (no miss penalty), large VLIW Cache (3072-Kbyte), instruction latencies of 1 cycle, and no next long instruction miss penalty. The DTSVLIW renaming mechanism was used in all algorithms.

As shown in Figure 6.12, all three scheduling algorithms perform very similarly, although the DTSVLIW algorithm achieves marginally inferior results in most cases. This is to be expected as it is possible for instructions to be blocked from moving up the scheduling list by full long instructions at some interior position of the list. This prevents empty instruction slots at higher list positions from being filled, which reduces the code density in the block and limits the achievable parallelism. Blocking in this fashion does not occur for the other two algorithms. However, the DTSVLIW algorithm is expected to provide a much more feasible and faster implementation, and the results in Figure 6.12 demonstrate that its use should not significantly prejudice the architecture. In some cases, our simplified algorithm does as well as and even outperforms the other algorithms. This is markedly so for the 16x16-ijpeg run, but it is also seen in the 16x16-m88ksim run.

The full FCFS is as good as or better than the Greedy algorithm for the 16x16 runs, but is outperformed by the Greedy algorithm for the smaller geometries, particularly for the 4x4 runs. This happens because, in some cases, the extra long

instruction added at the top of the block by the FCFS algorithm cannot be filled by subsequent instructions as these have dependencies with instructions in the middle of the block. These instructions when added to the end of the block cause the block to be filled and flushed to the VLIW Cache with the first long instruction only partially filled, reducing the code density and the achievable parallelism. The Greedy algorithm does not add the new long instruction at the top of the block allowing for another one at the end of the block that must be more effectively filled despite the dependencies caused by instructions added to it. Increasing the number of long instructions in the block and the width of these long instructions reduces the resource blocking of instructions and allows more instructions to be added by the FCFS algorithm after resource blocking occurs. This gives more opportunity for the added front long instruction to be filled.

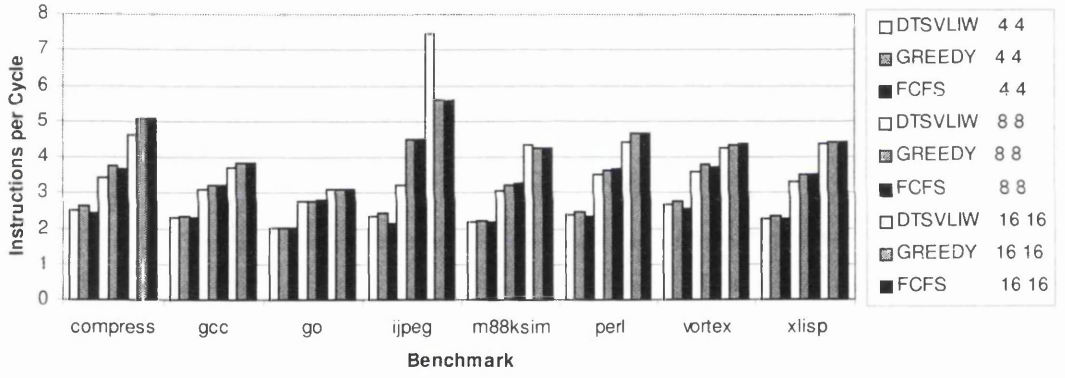



**Figure 6.12: Performance of the DTSVLIW, Greedy, and FCFS algorithms – untyped F.U.**

### 6.2.2 Evaluation of the Performance of the DTSVLIW, Greedy, and FCFS Scheduling Algorithms – Typed Functional Units

The simulations described in this section so far assume that all functional units can execute all instructions; i.e., the functional units are untyped. However, machines using typed functional units are more likely scenarios in real implementations. In order to evaluate the impact of typed functional units in the performance of the scheduling algorithms, we have performed experiments using three machine configurations:

1. 5x4 – with 2 integer, 1 load/store, 1 floating-point, and 1 branch functional units in the VLIW Engine, and a 4 long instructions long block

2. 10x8 – with twice the number of functional units and twice the number of long instructions of the previous configuration
3. 20x16 – with four times the number of functional units and four times the number of long instructions of the 5x4 configuration

The results we have obtained with these three machine configurations are presented in graph form in Figure 6.13 and summarised in Table 6.4, Table 6.5, and Table 6.6.

As a comparison between the graphs in Figure 6.12 and Figure 6.13 shows, the use of typed functional units causes significant performance loss, even though the typed configurations have 25% more instruction slots in each long instruction than the similar untyped ones. However, a small increase in the number of functional units might significantly reduce this performance loss. For example, in the DTSVLIW machine configuration described in Subsection 6.1.7, the performance loss due to the use of typed functional units is small, even though that configuration has only 2 functional units more than the 10x8 configuration. Nevertheless, we have chosen the number of functional units in the typed configurations described in this subsection with the specific purpose of stressing any existing differences between the algorithms' performance. As summarised in Table 6.4, Table 6.5, and Table 6.6, the 5x4 configuration achieves from 77% (77% u.a.m.) to 79% (79% u.a.m.) of the untyped 4x4 average performance for the three algorithms, while 10x8 configuration achieves from 81% (81% u.a.m.) to 85% (85% u.a.m.) of the untyped 8x8 performance, and the 20x16 achieves from 85% (83% u.a.m.) to 90% (89% u.a.m.) of the untyped 16x16 performance.

The relative performance of the three algorithms did not change much from machines with untyped to machines with typed functional units. In Table 6.4, Table 6.5, and Table 6.6, the last two columns contain the average performance of each algorithm as a percentage of that of the FCFS algorithm. As the tables show, the performances of the DTSVLIW and Greedy algorithms as percentage of the FCFS algorithm vary from 93% (92% u.a.m.) to 105% (106% u.a.m.) percent for configurations with untyped functional units, and from 95% (96% u.a.m.) to 102% (102% u.a.m.) for typed configurations. That is, the DTSVLIW and Greedy algorithms have presented performances closer to the FCFS with typed functional units. This shows that, for the range of configurations used, the DTSVLIW algorithm performs almost as well as the more complex Greedy and FCFS algorithms.

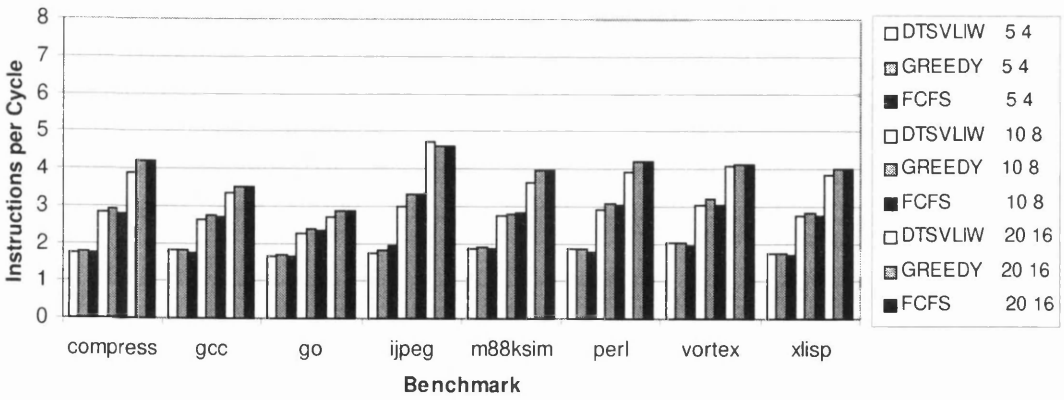



**Figure 6.13: Performance of the DTSVLIW, Greedy, and FCFS algorithms – typed F.U.**

**Table 6.4: Summary of the results – 4x4 & 5x4 machine configurations**

| | | | *Relative Performance – H.Mean (A.Mean)* | | |
|---|---|---|---|---|---|
| | *IPC – H.Mean (A.Mean)* | | | *Algorithm / FCFS* | |
| | *4x4* | *5x4* | *5x4 / 4x4* | *4x4* | *5x4* |
| DTSVLIW | 2.33 (2.35) | 1.80 (1.80) | 77% (77%) | 103% (103%) | 100% (100%) |
| GREEDY | 2.39 (2.41) | 1.82 (1.83) | 76% (76%) | 105% (106%) | 102% (102%) |
| FCFS | 2.27 (2.28) | 1.79 (1.80) | 79% (79%) | 100% (100%) | 100% (100%) |

**Table 6.5: Summary of the results – 8x8 & 10x8 machine configurations**

| | | | *Relative Performance – H.Mean (A.Mean)* | | |
|---|---|---|---|---|---|
| | *IPC – H.Mean (A.Mean))* | | | *Algorithm / FCFS* | |
| | *8x8* | *10x8* | *10x8 / 8x8* | *8x8* | *10x8* |
| DTSVLIW | 3.22 (3.24) | 2.75 (2.77) | 85% (85%) | 93% (92%) | 97% (97%) |
| GREEDY | 3.48 (3.55) | 2.88 (2.91) | 83% (82%) | 100% (100%) | 102% (102%) |
| FCFS | 3.47 (3.53) | 2.82 (2.85) | 81% (81%) | 100% (100%) | 100% (100%) |

**Table 6.6: Summary of the results – 16x16 & 20x16 machine configurations**

| | | | *Relative Performance – H.Mean (A.Mean)* | | |
|---|---|---|---|---|---|
| | *IPC – H.Mean (A.Mean))* | | | *Algorithm / FCFS* | |
| | *16x16* | *20x16* | *20x16 / 16x16* | *16x16* | *20x16* |
| DTSVLIW | 4.29 (4.53) | 3.68 (3.77) | 86% (83%) | 100% (103%) | 95% (96%) |
| GREEDY | 4.28 (4.40) | 3.87 (3.94) | 90% (89%) | 100% (100%) | 100% (100%) |
| FCFS | 4.29 (4.41) | 3.86 (3.93) | 90% (89%) | 100% (100%) | 100% (100%) |

## 6.3 DTSVLIW versus DIF

An important difference between the DTSVLIW and the DIF scheduling algorithms is the register renaming mechanism. The DTSVLIW uses copy instructions for renaming, with the disadvantage that the slots used by them cannot be used for other instructions. The DIF, on the other hand, has several extra instances of each ISA

register, with the disadvantage of requiring large register files if the renaming of the same register is to be allowed many times within the same block.

In the experiments described so far, we have used the DTSVLIW renaming mechanism, since we wanted to compare the algorithms under identical conditions. In order to compare the DTSVLIW with the DIF, we have fully implemented the DIF scheduling algorithm in our simulator and have run experiments with DIF machine configurations and equivalent DTSVLIW configurations. We have used four instances of each integer and floating-point register in the DIF simulations, which corresponds to 96 integer and 96 floating-point renaming registers, and 32 instances of each integer and floating-point Sparc condition register. The same number of renaming registers was given to the DTSVLIW, but it has never used any of them to the full during the simulations. We have set the DTSVLIW's VLIW Cache size at 3072-Kbyte, while the DIF's VLIW cache size has been set at a value that allows the same number of blocks of the equivalent DTSVLIW machine configuration. (For VLIW caches with equivalent number of blocks, the DIF's VLIW cache requires more bytes than the DTSVLIW's VLIW Cache does due to the DIF's exit maps. See Section 4.10.) The results are shown in Figure 6.14 and Figure 6.15.

As the graph in Figure 6.14 shows, the DIF outperforms the DTSVLIW in all benchmarks with the 4x4 configuration. However, with the 8x8 configuration, the DTSVLIW outperforms DIF in the compress, gcc, go, and mk88sim. With the 16x16 configuration, the DTSVLIW outperforms DIF in all benchmarks by a large margin. This happens because, for larger configurations, renaming is more frequent and four instances of each integer and floating-point register, as used in the DIF, is not enough. With typed functional units, the DTSVLIW advantage is smaller, as shown in Figure 6.15. In this case, the DTSVLIW is outperformed by DIF in all benchmarks for the 5x4 and 10x8 configurations, although not by a large margin. However, for the 20x16 configuration, the DTSVLIW outperforms DIF in all benchmarks by a significant margin. One can infer that a DIF implementation with sufficient instances of each register would always outperform the DTSVLIW. However, it is not practical to implement machines with more than a few instances per ISA register. The number of register read and write ports grows with the number of functional units, so a large multiported register file would render the machine clock rate too slow.

Figure 6.16 shows a comparison between a DTSVLIW and a DIF machine using more realistic parameters. The performance data of the DIF machine and the parameters used for both machines have been collected from [Nair97]. The parameters were: 2 branch units plus four untyped functional units; 2-way set-associative Instruction Cache with 128-byte lines, 16 lines per set (4-Kbyte), and 2 cycle miss penalty; direct-mapped Data Cache with 128 lines each of length 32 bytes (4-Kbyte), and a 2-cycle miss penalty; 2-way set associative VLIW Cache with 512x2 blocks; and a block size of 6 long instructions of 6 instructions each.

From this data and assuming an instruction size of 6 bytes for both machines, the DTSVLIW VLIW Cache size is 216-Kbyte and the DIF VLIW cache size 463-Kbyte. The DIF VLIW cache is larger due to the DIF register renaming system. For each block exit point, the DIF machine requires 19 bytes for the exit map [Nair97]. The number of renaming registers is different for the same reason. Four instances of each integer and floating-point register were required in the DIF simulation; i.e., 96 integer and 96 floating-point extra registers for renaming, while the maximum number of integer and floating-point renaming registers required in the DTSVLIW simulation was 18 and 6, respectively.

As can be seen in Figure 6.16, the average performance of the two machines is similar: 2.4 (2.4 u.a.m.) instructions per cycle for the DTSVLIW and 2.2 (2.2 u.a.m.) for the DIF, a difference of approximately 10% (10% u.a.m.) in favour of DTSVLIW. However, the DTSVLIW achieves this performance with fewer resources. DIF performs better in compress and xlisp, while DTSVLIW performs better in the remaining benchmarks.

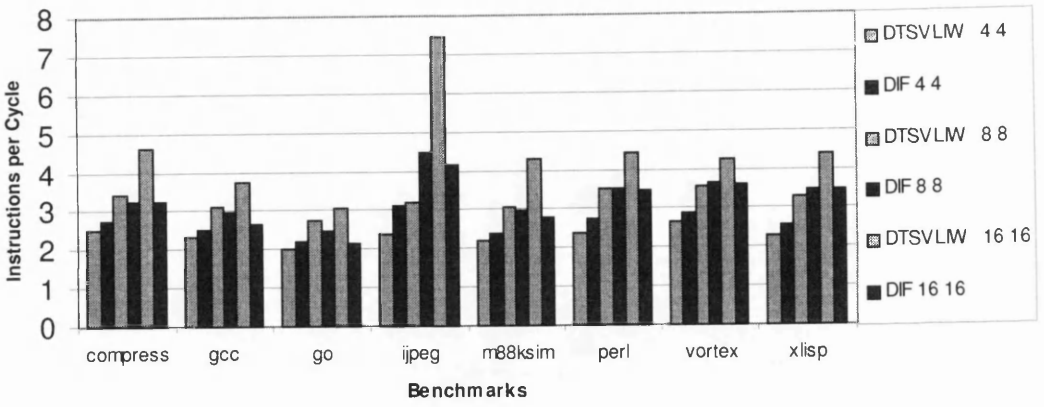



**Figure 6.14: DTSVLIW versus DIF: untyped functional units**

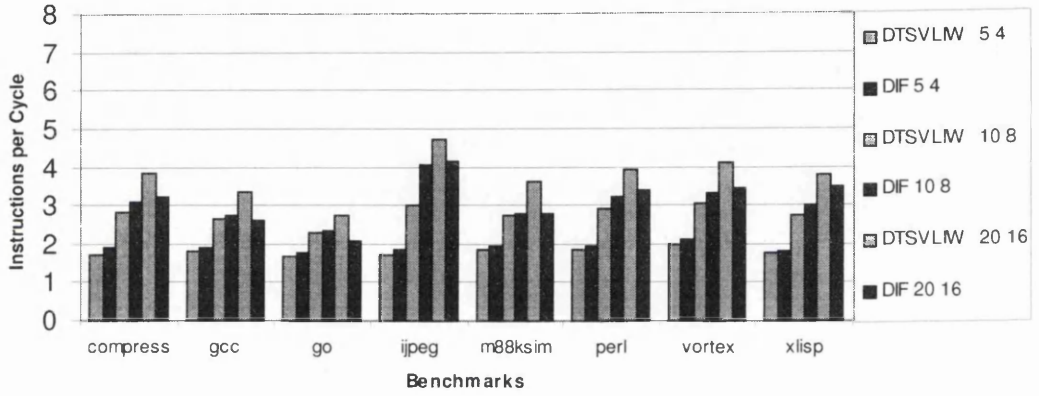



**Figure 6.15: DTSVLIW versus DIF: typed functional units**

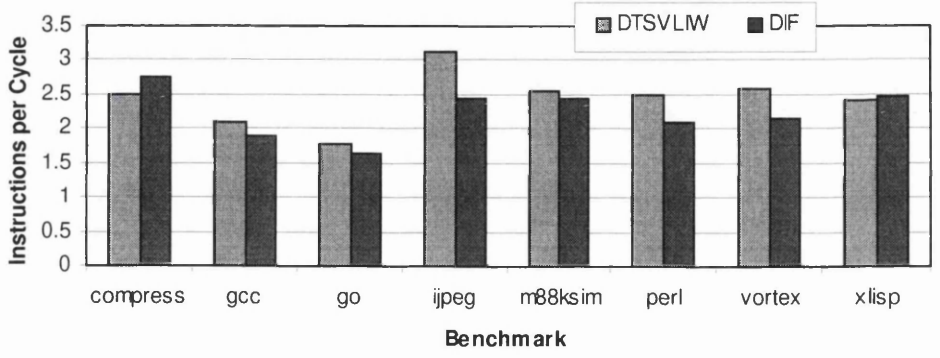



**Figure 6.16: DTSVLIW versus DIF: more realistic models**

## 6.4 DTSVLIW versus VLIW

The VLIW research group at IBM is a leading group on VLIW compiler and architecture technologies. In [Moreno97] they have presented experimental performance results of various VLIW configurations using a powerful VLIW compiler named *Chameleon*. Chameleon has an aggressive suite of optimisations, including a complete set of traditional optimisations plus several ILP-increasing optimisations such as loop unrolling, if-conversion (predication), and instruction hoisting.

In Figure 6.17, we show the performance figures for programs from the SPECint92 and SPECint95 running in two VLIW configurations described in [Moreno97] and [Moudgill96] executing code compiled by Chameleon, and in two DTSVLIW configurations. The benchmarks m88ksim and go are from SPECint95 while the others are from the SPECint92. One VLIW configuration used in the IBM's experiments has been set with 8 untyped functional units and the other has been set with 16. The instruction latencies have been set at 1-cycle for integer (including load/store), 3-cycle for integer multiply, 10-cycle for integer divide, and 3-cycle for floating-point instructions. The number of added registers for renaming has been 64-integer, 64-floating-point, and 16-condition in the 8-wide configuration, and 128-integer, 128-floating-point, and 32-condition in the 16-wide configuration. The experiments have been performed with perfect instruction and the data caches (no miss penalty).

We have configured the two DTSVLIW machines with parameters identical or equivalent to those used in the two IBM VLIW machines. One DTSVLIW configuration used in our experiments has been set with an 8x8-block and the other has been set with a 16x16-block, both with untyped functional units. The instruction latencies have been all set at 1-cycle, which is a value lower than that used in the IBM's experiments for integer multiply, integer divide, and floating-point instructions. However, the Sparc 7 ISA does not have integer multiply or divide instructions but only multiply-step, which can execute in one cycle. Since the benchmarks are all integers, the number of floating-point instructions executed is negligible; therefore, the different latencies used for floating-point instructions do not constitute a problem. The number of renaming registers used during the DTSVLIW

simulations has never exceeded the number used in the IBM's simulations in any combination of benchmark program and machine configuration. Our experiments have also been performed with perfect instruction and the data caches.

As the graph in Figure 6.17 shows, the VLIW outperforms the DTSVLIW in compress and eqntott by a large margin. However, for gcc, go, m88ksim, and xlisp the DTSVLIW has consistent better performance for both machine configurations shown. These results demonstrate that, in most cases, the DTSVLIW algorithm is able to find more parallelism than a state-of-the-art VLIW compiler under similar conditions. This is possible because the DTSVLIW scheduling algorithm has access to dynamic information not available to the VLIW compiler. We believe that, a conjunction of the DTSVLIW architecture and compiler technology such as loop unrolling, software pipeline, and predication (if added to the DTSVLW ISA) would perform even better than shown, in particular with compress and eqntott. Published results corroborate this view, showing that the use of such optimisations does significantly improve performance, in particular the use of predication in the eqntott and compress benchmarks [August98]. Other compiler techniques could also be developed specifically for the DTSVLIW architecture.

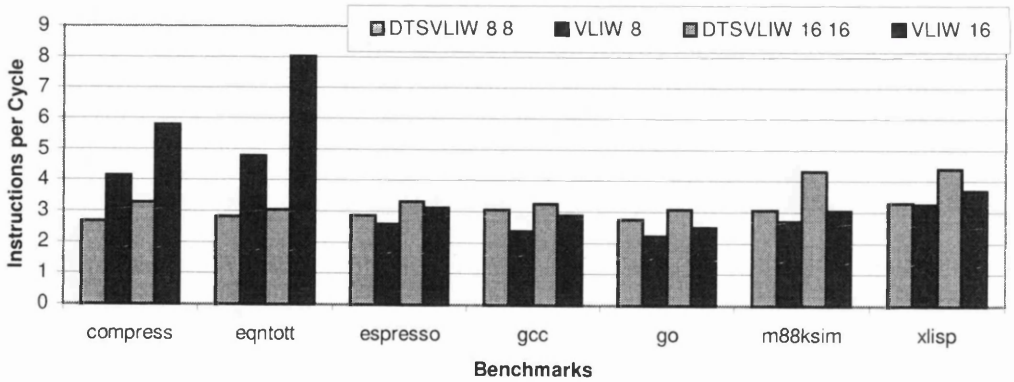



**Figure 6.17: DTSVLIW versus VLIW**

## 6.5 DTSVLIW versus Superscalar

The graph in Figure 6.18 shows a comparison between the performance of the Superscalar processor PowerPC620, as described in [Diep95], and a DTSVLIW

machine using equivalent hardware. The PowerPC620 performance figures have been taken from [Diep95], and the parameters used there are:

- 4-instruction wide fetch, dispatch, complete, and writeback pipeline stages
- 1-cycle integer, 2-cycle load, and 3-cycle floating-point instruction latency
- 3 integer, 1 load/store, 1 floating-point, and 1 branch functional units, with 2, 3, 2, and 4 reservation stations for the functional units of each kind (a total of 15 reservation stations), respectively
- 32-Kbyte, 8-way set associative instruction and data L1 caches, with 8-cycle miss penalty (a perfect unified L2 cache was assumed)
- branch predictor with a 256-entry 2-way BTB and a 2048-entry (2-bit counters) direct mapped BHT

The DTSVLIW has been configured with a 4x8 block and functional units of the same type, in the same number, and with same latency of the PowerPC620. The only exception has been the latency of the floating-point functional unit, whose latency has been set to 1-cycle. This does not constitute a problem because the benchmarks are integer and, therefore, the number of floating-point instructions executed is zero or negligible. Although six functional units are available in the VLIW Engine with this machine configuration, we have used 4-instruction wide long instructions in the DTSVLIW to allow the same dispatch width for both machines. An extra dispatch pipeline stage was added to the VLIW Engine pipeline to account for the logic necessary to unpack the long instructions coming from the VLIW Cache and to issue them to the appropriate functional units. A branch predictor with the same characteristics of the PowerPC620's has been used to try to reduce the extra cost added to next long instruction misses by this dispatch stage. An 8-Kbyte, 8-way set associative instruction cache, and a 24-Kbyte, 8-way set associative VLIW Cache have been used. These sizes have been chosen to make this pair equivalent to the instruction cache of the PowerPC620. The DTSVLIW's Data Cache has been configured with the same characteristics of the PowerPC620's. In this simulation, the DTSVLIW has been allowed to execute approximately the same number of instructions executed in the PowerPC620 simulation described in [Diep95]. The number of instructions executed is shown in Table 6.7.

As Figure 6.18 shows, the performance of the two machines is comparable,

although the DTSVLIW performance is better overall. This is so because the scheduling list of the DTSVLIW is larger than the instruction window (all reservation stations) of the PowerPC620, which allows more opportunities for finding ILP. Although larger than instruction window of the PowerPC620 (15 instructions), the scheduling list of the DTSVLIW (32 instructions) is simpler. The complexity of the instruction window of the PowerPC620 is proportional to the number of reservation stations times the number of functional units (15 * 6 = 80, see Subsection 2.1.2), while the complexity of the scheduling list of the DTSVLIW is proportional to the number of candidate instructions times the number of instructions per long instruction (8 * 4 = 32, see Section 4.6). Note that a DTSVLIW implementation is likely to have a significantly higher clock rate than that of the PowerPC620, because, different from the PowerPC620, the DTSVLIW scheduling hardware is not in its main data path. In addition, due to the instruction fetch bandwidth problem of Superscalar machines, described in the Subsection 3.4.1, the PowerPC620 performance in terms of ILP can be seen as a high-end performance for standard Superscalar machines. The DTSVLIW described in this section, on the other hand, is a low-end DTSVLIW and larger DTSVLIW configurations, such as that described in Subsection 6.1.7, can be implemented with increasing performance returns.

**Table 6.7: Number of instructions executed**

| Benchmark | compress | eqntott | espresso | xlisp |
|---|---|---|---|---|
| Instructions Executed | 6,884,257 | 3,147,235 | 4,615,093 | 3,376,416 |

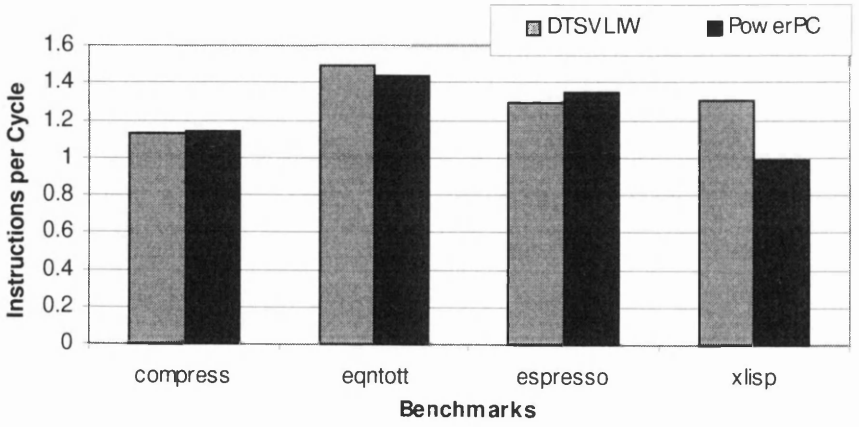



**Figure 6.18: DTSVLIW versus PowerPC620**

# Chapter 7

# Discussion

An ISA is a contract between a class of programs and a set of processor implementations [Rau93a]. Usually, this contract is concerned with the instructions format and the interpretation of the bits that constitute each instruction. However, in the case of systems that exploit ILP, this contract extends to information embedded in the programs regarding the available parallelism between the instructions of the programs. Special functions are performed by ILP exploiting systems in order to find and take advantage of ILP, and different forms of ISA contract divide these functions between the compiler and the hardware differently. These functions can be summarised as follows:

- To determine dependencies between instructions.
- To determine independencies between instructions; i.e., to find out the instructions that are independent of any instruction that has already been assigned to execute but may have not yet completed.
- To bind resources; i.e., to schedule the independent instructions to execute at some particular time on some specific functional unit, and to assign registers into which the results of these instructions may be written.

Figure 7.1 (page 136) shows some forms of ISA contract for ILP exploiting systems.

Superscalar machines execute sequential ISA code and exploit ILP; therefore, their hardware has to determine dependencies and independencies between several instructions, and bind several instructions to resources at the same time, dynamically.

The hardware for doing this is in the main data path of Superscalar machines. Because of this, Superscalar machines with elaborate instruction scheduling hardware have slower clocks than simpler Superscalar machines, and the latter may have a better performance than the former due to their fast clocks [Smith_JE94].

Superscalar and VLIW machines are at opposite extremes regarding ILP exploitation. In standard VLIW systems, the compiler is responsible for all functions that have to be performed in order to exploit ILP. This allows the VLIW hardware to be simple and fast, but makes the VLIW ISA contract very restrictive. Developments in compiler or hardware technology following a VLIW ISA specification may allow for greater parallelism than that which can be expressed within this VLIW ISA specification. To take advantage of these developments, the VLIW ISA may have to be changed; this creates the VLIW object code compatibility problem.

DTSVLIW machines execute sequential code, and their hardware, similar to that of Superscalars, has to determine dependencies and independencies between instructions and to bind instructions to resources dynamically. However, different from Superscalars, DTSVLIW machines perform the functions related to ILP exploitation with one instruction at a time, producing VLIW code that is cached and thereafter executed many times. We have shown that a DTSVLIW machine performs better than a Superscalar machine with equivalent hardware. We have also shown that DTSVLIWs perform better than VLIWs with the same degree of parallelism.

In the rest of this chapter, we compare the DTSVLIW architecture with other architectures that have sequential ISA contract and with promising architectures that do not have sequential ISA contract.

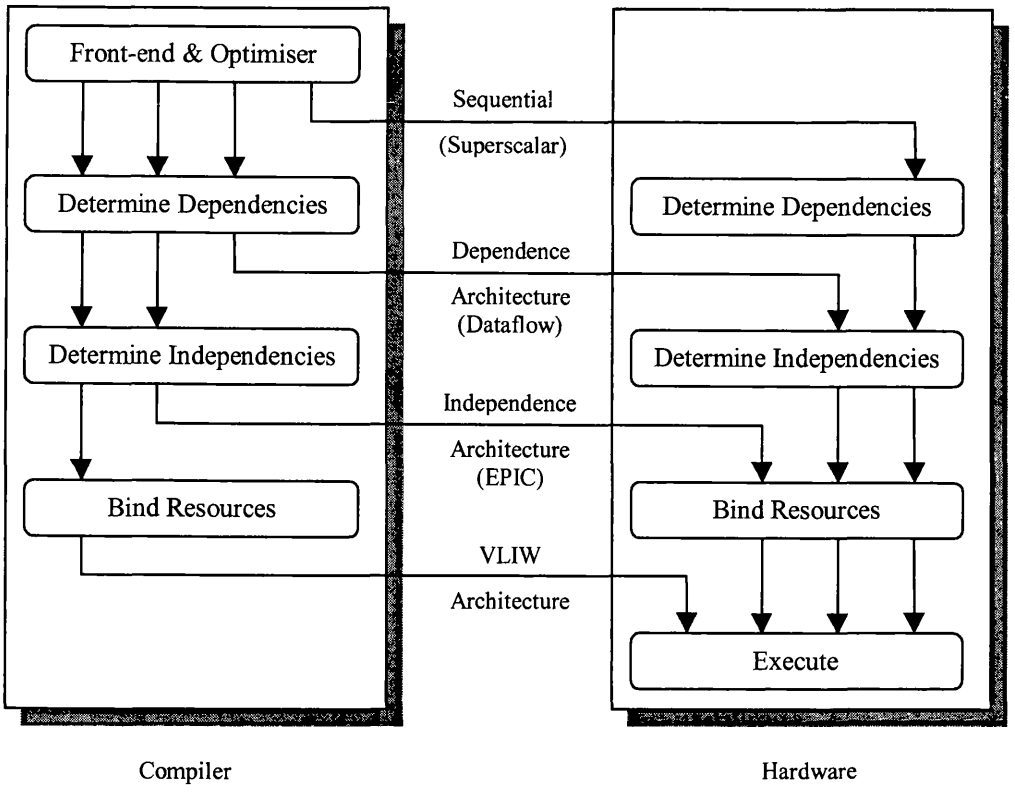



**Figure 7.1: Different forms of ISA contract.**

## 7.1 DTSVLIW versus Enhanced Superscalar Architectures

The Fill Unit of Superscalar machines enhanced with a trace cache (Subsection 3.4.1) is the equivalent to the Scheduler Unit of DTSVLIW machines. However, the Scheduler Unit performs all functions necessary to ILP exploitation on a DTSVLIW machine. In contrast, Superscalar machines implemented according to the standard Trace Cache architecture use their Fill Unit only to ameliorate the fetch bottleneck (Subsection 3.4.1). Nevertheless, the DTSVLIW and Trace Cache architectures can be seen as members of the same family of architectures. They can be put in a scale of how many ILP extraction functions are performed by their Scheduler Unit/Fill Unit and how many of these functions are performed by their parallel execution-core. Figure 7.2 shows such a scale and the positions of the DTSVLIW and Trace Cache architectures.

Like standard VLIW and standard Superscalar, the DTSVLIW and Trace Cache

architectures are at opposite extremes. As shown in Figure 7.2, the Trace Cache architecture's execution-core performs all ILP related functions, while the DTSVLIW's execution-core performs none. On the other hand, the DTSVLIW's Scheduler Unit performs all ILP related functions and the Trace Cache architecture's Fill Unit performs none.

One might argue that the DTSVLIW sequential execution during the scheduling phase would strongly affect its performance. However, in the DTSVLIW or in any architecture that always operate in parallel such as the Trace Cache, the first time a fragment of code is executed it is likely to cause data and instruction cache misses that will determine the performance during its execution. In addition, no machine has dynamic branch behaviour information when fresh code fragments are encountered, and the more parallel the machine the larger the effect of branch mispredictions. Furthermore, our results show that most of the time the DTSVLIW is executing code in VLIW-mode (Table 6.1) if the VLIW Cache is large enough to hold pre-scheduled code. Results with the Trace Cache architecture also show that the code can be found in the trace cache most of the time [Rotenberg97].

Real implementations usually distance themselves from standard architecture definitions due to real world constraints. For example, in the DTSVLIW configuration described in Subsection 6.1.7 and in the DTSVLIW configuration compared with the PowerPC620 in Section 6.5, we have left to the execution-core the task of dispatching the instructions to specific functional units. Researchers working with the Trace Cache architecture have also suggested moving some ILP related functions from the execution-core to the Fill Unit in order to allow implementations with fast clock [Rotenberg97, Vajapeyam97, Friendly98]. Other Superscalar enhancements such as value prediction [Fu98, Nakra99] and instruction reuse (Subsection 3.4.2) may also be incorporated in both architectures.

Although future research may produce variants of the DTSVLIW and Trace Cache architectures that are more closely together in the scales shown in Figure 7.2, their pure definitions still represent useful models to understand the nature of the ILP that can be exploited dynamically. Ultimately, the available VLSI technology and design expertise will determine which of the two extremes represented by the DTSVLIW and Trace Cache architectures will reach the marketplace. We believe,

however, that a point close to the DTSVLIW's in the scales of the Figure 7.2 is a better option than one close to the Trace Cache's. This view is corroborated by the evolution of Alpha microprocessors. They implement the simplest RISC ISA (which means simple execution-core) in the mass microprocessor market and are "performance leaders since their introduction in 1992" [Kessler99] mainly, we believe, due to their simple high-clock-speed-tailored implementation.

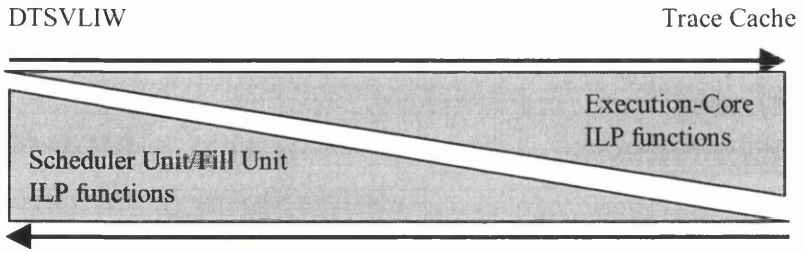



**Figure 7.2: Distribution of ILP related functions.**

## 7.2 DTSVLIW versus Explicitly Parallel Instruction Computing Architectures

The term *Explicitly Parallel Instruction Computing* (EPIC) was coined recently by Intel and Hewlett Packard in their joint announcement of the IA-64 ISA [Gwennap97]. EPIC ISAs require the compiler to determine the dependencies and the independencies between instructions (Figure 7.1). However, the hardware interpreter of an EPIC ISA is responsible for binding the independent instructions specified by the compiler to the machine resources. This is in contrast with VLIW ISAs where the compiler is responsible for determining dependencies and independencies, and also for binding resources.

To my knowledge, *Horizon* [Thistle88] is the first EPIC architecture proposed. Horizon is a multiprocessor where each processor executes EPIC code. The architecture of each Horizon processor is VLIW and Multithreaded (the processors switches execution context each clock cycle). Horizon long instructions are 3-instructions long. Each of them has a look-ahead field that controls long instruction execution overlap. The compiler specified look-ahead value for this field of each long instruction, *la*, indicates the number of subsequent long instructions that may be

issued without the completion of this long instruction. The compiler guarantees that the next *la* long instructions in the long instruction stream are data-independent of the current long instruction. In order to execute the EPIC program correctly, all the hardware has to do to issue a long instruction is to assure itself that no more than *la* subsequent long instructions are issued before this instruction has completed. The hardware is still responsible for scheduling functions but this is restricted to sending long instructions to the respective functional units: no data dependence check is required.

Similar to the Horizon EPIC ISA, in the Intel/Hewlett Packard EPIC ISA, or IA-64 for short, each three instructions are grouped together into 128-bit sized and aligned containers called *bundles*. Each bundle contains three 41-bit *instruction slots* and a 5-bit *template field* [Intel99]. The template field specifies two properties: the mapping of instruction slots to execution unit types, and stops within the current bundle. The stops define *instruction groups*. An instruction group is a sequence of instructions starting at a given bundle address and slot number and including all instructions at sequentially increasing slot numbers and bundle addresses up to the first stop or taken branch. Therefore, instruction groups can encompass several bundles.

The IA-64 specifies dependency restrictions that allow the processor to execute all (or any subset) of the instructions within a legal instruction group in parallel or serially with the end result being identical. That is, two or more instructions within a group should not write to the same register (output dependency), or read from and write to the same register (true and anti dependencies). Memory reads and writes to the same address are allowed within groups, however. The IA-64 compiler has to generate code according to these dependency restrictions. To facilitate this task, the IA-64 exposes to the compiler a large number of registers: 128 integer, 128 floating-point, 64 predicate, and a large number of other registers.

Conditional branch instructions use the predicate registers to decide their outcome. In addition, these registers are used to implement predicated execution (see Subsection 2.2.5). Instruction hoisting (Subsection 2.2.6) is also supported by IA-64.

Because the dependencies and independencies between instructions are specified by the compiler, the IA-64 processor only has to fetch one or more bundles,

to identify the stops, and to send the independent instructions to the respective functional units. Memory dependencies can be dealt with via compiler assisted memory disambiguation, compiler and ISA assisted speculative memory access [Intel99], and the use of an ARB (Subsection 2.2.4). The new IA-64 processor still executes legacy 8086 ISA code and all is upgrades [Intel99] but it does so using special modes that may not take full advantage of the new EPIC processor core. Intel and Hewlett Packard have not yet disclosed how the microarchitecture of IA-64 processors executes legacy code.

The Intel/Hewlett Packard EPIC architecture relies on compiler technologies such as predication, instruction hoisting, loop unrolling, software pipelining, etc, to exploit the ILP available in programs. The DTSVLIW architecture, on the other hand, does not rely on the compiler to exploit ILP and can achieve performance executing legacy sequential code. In addition, the DTSVLIW architecture can be employed to emulate legacy ISA code in IA-64 processors, taking advantage of the EPIC core. It can also be used to execute EPIC code directly, collecting dynamic branch behaviour information and organising the code during the scheduling phase of program execution to increase the processor performance. Due to the availability of predication and hoisting ISA support, we expect that the DTSVLIW will achieve performance levels even higher than those shown in this thesis when executing EPIC code.

## 7.3 Research Architectures for ILP Exploitation

In this section, we discuss some important machine architectures that have been targeted by the computer architecture research community's research efforts, and try to compare them with the DTSVLIW.

### 7.3.1 IRAM Architectures

Recently, researchers have suggested the integration of high-performance processors and *dynamic random access memory* (DRAM) on the same chip, forming an architecture called *Intelligent RAM* (IRAM) [Kozyrakis97]. This suggestion comes out of the fact that, in the near future, billions of transistors are going to be available

for computer architects [Burger97], and this would be a good way to make use of them. They suggested the use of Vector architectures to provide high performance. We believe that the same principles that motivate the use of Vector architectures in an IRAM also apply to DTSVLIW architectures.

### 7.3.2 Simultaneous Multithreaded Architectures

Simultaneous multithreaded processors [Tullsen95, Eggers97] are Superscalar processors that support multiple machine contexts and execute multiple instruction streams simultaneously (see Subsection 2.1.5). They do so to reduce the latency of multithreaded programs (programs where threads are specified by the programmer or compiler).

The performance of multithreaded machines depends on finding enough thread parallelism, a task left to the software level. However, to develop multithreaded applications is challenging due to the extreme difficulty of debugging multithreaded programs and the lack of automatic thread-partitioning compilers. Because of this, we share the view of other researchers that simultaneous multithreading is primarily a technique for improving throughput in multiprogrammed workloads [Lipasti97]. In addition, the implementation of several hardware contexts is costly and may have a negative impact on the clock cycle time.

The Multithreaded architecture addresses issues that are orthogonal to those addressed by the DTSVLIW architecture. Nevertheless, multithreading could be added to the DTSVLIW, although it might be expensive in terms of silicon area. We have left the assessment of Multithreaded DTSVLIW architectures for future work.

### 7.3.3 Dataflow Architectures

Dataflow machines are examples of the class of dependence architectures (Figure 7.1). Programs for Dataflow ISAs explicitly indicate the dependencies that exist between instructions [Veen86, Rau93a]. This is typically done by including in each instruction a list of successor instructions. An instruction is a successor of another instruction if it uses as one of its input operands the result produced by that other instruction. As soon as all of the input operands of an instruction already inside the Dataflow machine are available, its hardware executes the instruction and updates

tables that will trigger the fetch or execution of other instructions. Therefore, the availability of operands (data) rather than the PC update is responsible for triggering instruction execution in Dataflow machines and this is what gives rise to the name of this type of processor.

Studies from past dataflow projects revealed inefficiencies in Dataflow computing [Veen86]. Compared to its control-flow counterpart, the fine-grain approach to ILP exploitation of the Dataflow model of parallel execution incurs more overheads. The overheads involved in detecting enabled instructions and using their outputs to enable other instructions generally result in poor performance in applications with low degrees of parallelism, such as scalar programs [Lee_B94].

In a dataflow machine, instructions become enabled when their input data become available. Thus, the machine has to keep many (hundreds to thousands) not yet enabled instructions in tables inside the processor in order to have a large enough instruction window in which to find ILP. To detect many enabled instructions in this large window simultaneously is a costly operation to perform in hardware. The machine also has to use the many results produced during execution to update this large instruction window simultaneously, which is also a costly operation to do in hardware.

It is interesting to note that the core of superscalar machines operates in a dataflow manner: The results produced by the functional units propagate back to the instruction window or reservation stations and trigger the execution of other instructions. As discussed in Subsection 3.4.1, the overheads incurred by this restricted dataflow mode of execution are precisely those that are exacerbated in the general dataflow machine architecture.

Experimental dataflow machines have now been around for more than twenty years [Davis79], but still there is no consensus as to whether the data-driven model of execution exposed to the ISA level is a viable means to exploit ILP. In addition, dataflow architectures obviously cannot be directly used to execute sequential code. Because of this, they do not offer backward code compatibility for legacy sequential code as the DTSVLIW does. The realisation of the pure dataflow potential performance in a feasible cost/effective machine running large scalar programs (such as those of the SPEC benchmark suite) is yet to be shown. This thesis shows that the

DTSVLIW can achieve ILP with cost/effective machine configurations.

### 7.3.4 Multiscalar Architectures

The Multiscalar architecture uses fine-grain and coarse-grain dataflow to exploit ILP. In a Multiscalar system, sequential programs are partitioned into sequential tasks by the compiler [Sohi95]. These tasks are dynamically assigned to *processing units* for execution. These processing units, like small superscalar processors, use fine-grain dataflow to exploit intratask ILP. In addition to task partitioning, the compiler specifies the intertask communication [Vijaykumar99]. When a processing unit reaches a point in its task execution where a value produced by another processing unit needs to be consumed, it waits until the value is produced in order to proceed executing its task code. This realises coarse-grain dataflow.

In a Multiscalar system, the compiler specifies a list of successors for each task. The Multiscalar hardware predicts a task successor using techniques similar to branch prediction, and assigns this new task to a new processing unit dynamically. Apart from the successors of the tasks, the Multiscalar hardware needs to know where the tasks end, so that it may terminate the tasks' execution and start the execution of new ones. To make it possible, the compiler tags the instructions at the boundaries of the tasks with extra bits to indicate where the tasks end.

Multiple tasks are speculatively executed simultaneously on the multiple processing units of a Multiscalar machine; however, they are retired in program order. All intertask register dependencies are honoured by communication and synchronisation, as specified by the compiler. To do this, the compiler determines the set of register values that may be consumed by each task. If a task consumes a data value produced by another task, then the consumer task waits until the value is received. The compiler determines the last update of each register within tasks and tags the instructions that modify them with extra bits to indicate that these registers have to be forwarded. A network is used to communicate register values between tasks. Intertask memory dependencies are honoured by speculation and validation in hardware. This is achieved using an ARB (see Subsection 2.2.4). Intratask dependencies are handled by the processing units' hardware in a way similar to that of Superscalar processors.

To achieve performance, Multiscalar architectures rely on the compiler to partition the sequential program in tasks that exhibit ILP. They also rely on the compiler to orchestrate data and control communication between tasks within hardware support. Therefore, compatibility with legacy sequential code can only be achieved through recompilation or binary translation. The DTSVLIW, on the other hand, can execute legacy code directly and does not rely on the compiler to exploit ILP.

Multiscalar is an interesting architecture, but there are many potential problems in producing effective hardware and compiler implementations. We believe the DTSVLIW is a more straightforward approach to exploit ILP.

## 7.4 Critical Assessment of this Research Work

The goal of this thesis has been to examine the two hypotheses presented in Subsection 1.2 and repeated below.

- *The DTSVLIW architecture can overcome the VLIW object code compatibility problem while preserving the VLIW architecture simplicity and high clock rate.*
- *The DTSVLIW can achieve higher performance than the DIF and Superscalar architectures using equivalent hardware.*

In Chapter 4, we have described the DTSVLIW architecture in detail showing that it is able to overcome the VLIW object code compatibility problem while preserving the VLIW simplicity on its main processing engine and overall high clock rate. In Chapter 6, we have presented and discussed experimental results that show that, for representative machine configurations, the DTSVLIW achieves higher performance than the DIF and Superscalar architectures using equivalent hardware. In this section, we discuss the limits of the validity of the experimental results and the potential difficulties for the implementation of a DTSVLIW machine in silicon.

### 7.4.1 On the Limits of Validity of the Experimental Results

In this thesis, we have examined the integer performance of the DTSVLIW. Therefore, the results presented here are valid for the class of integer programs, but

not necessarily valid for other classes of programs. We have chosen to examine the DTSVLIW performance with this class of programs first, because experiments have shown that integer programs do not have much ILP and their ILP is hard to extract, if compared with floating-point programs for example [Wall93]. We have left the evaluation of the DTSVLIW performance with other classes of programs for future work.

Within the class of integer programs, we have chosen the integer programs of the SPEC95 and SPEC92 benchmark suites as test programs for the DTSVLIW in the experiments reported here. The number of test programs in the SPEC95 and SPEC92 is, however, small. Nevertheless, results obtained using SPEC programs are considered to be representative by researchers in the experimental computer architecture field and these programs are officially used by most computer manufacturers to compute and advertise performance indices.

We have used the first 50 million instructions executed in each program as sample for most of the experiments in this thesis. However, the first instructions executed in a program include start up code that is not part of the algorithms exercised in the program. In addition, 50 million instructions would represent a short execution time in a real DTSVLIW machine. Nevertheless, experiments reported in the literature have shown that the inputs of the SPEC integer programs can be chosen in a way that allows them to complete execution running from as little as 30 million to little more than 200 million instructions [Nair97, Rotenberg99]. This indicates that 50 million instructions is representative of the programs execution behaviour. In addition, a real machine is likely to switch the execution context to another program within the execution of 50 million instructions, as discussed in Section 5.2.

We have used our DTSVLIW simulator to produce the experimental results for most of the experiments described in Chapter 6. The only exceptions are experimental results for the Nair and Hopkins's DIF [Nair97], in Figure 6.16, for the IBM's VLIW [Moreno97, Moudgill96], in Figure 6.17, and for the Diep's PowerPC620 [Diep95], in Figure 6.18, which have been collected from the literature. The use of results previously published in literature is of course valid and common practice in research. However, the validity of the comparison of the results produced by our simulator and those mentioned above is limited by the following factors:

- The ISA we have used has been the Sparc 7 ISA, while the ISA used in the results we have collected from the literature was the PowerPC ISA.
- We have used execution-driven simulation in our experiments, while trace-driven simulation has been used to produce the results of some of the experiments we have collected from the literature.
- The input files used to produce the results we have collected from the literature are not shown in the literature, and are possibly different from those we have used.
- The compiler used to produce the object code of the benchmarks used in our experiments was different from the compilers used in the experiments reported in the literature.
- The number of instructions executed is significantly different in the comparisons between the DTSVLIW and the DIF (Figure 6.16), and DTSVLIW and VLIW (Figure 6.17).

RISC ISAs are very similar. According to Patterson and Hennessy [Paterson96 (page C-22)], "In the history of computing, there has never been such widespread agreement on computer architecture." The PowerPC ISA is, however, more powerful than the Sparc 7 ISA. The PowerPC ISA has features such as multiple conditional code fields and, load/store & update, branch on count register, load/store multiple, and string instructions that are not found in the Sparc 7 ISA [Patterson96 (Appendix C)]. Because of these features, the number of PowerPC instructions needed for executing a program may be smaller than the number of Sparc 7 instructions needed for executing the same program under the same conditions. This may result in an IPC index for the Sparc 7 ISA better than the IPC index for the PowerPC ISA under the same architectural conditions.

We have used execution-driven simulation in our experiments, while trace-driven simulation has been used to produce the results of the Nair and Hopkins's DIF [Nair97] and Diep's PowerPC620 [Diep95] experiments. Due to the characteristics of the DIF architecture, the use of trace-driven simulation is not likely to affect the DIF's results. However, the use of trace-driven simulation certainly affects the measured performance of Superscalar machines. This happens because instructions in the mispredicted paths of branches are fetched and use resources of the Superscalar

core in real machines (or execution-driven simulations). In trace-driven simulations, these resources are not consumed and, because of this, Superscalar machines are likely to perform better when simulated with trace-driven simulators than when simulated with execution-driven simulators.

The input data may affect the amount of ILP exhibited by programs. However, the input data used to produce the results we have collected from the literature has not been made available in the literature. Therefore, we may have used different input files and this may have a significant impact on the relative performances presented in the comparisons between the DTSVLIW and the DIF, VLIW, and PowerPC620. We have used a compiler different from those used in the results collected from the literature, and the number of instructions simulated in the DIF and VLIW simulations have also been different. All these limit the validity of the mentioned comparisons.

To isolate the effect of each of these factors (and possibly others not mentioned) for each comparison would be very difficult if at all possible. One might argue that it would have been better to implement our own VLIW simulator/compiler and Superscalar simulator and use these for the comparison with the DTSVLIW. (We have implemented our own DIF simulator and compared its performance with DTSVLIW's in Subsection 6.3.) However, this amount of work is beyond the scope of a research work of the magnitude of this thesis. In addition, the results we would obtain are unlikely to be significantly better than those we have collected from the literature.

### 7.4.2 On the DTSVLIW Implementation Difficulties

The hardware of a DTSVLIW machine can be divided into five main blocks (see Figure 1.1): three caches – the Data Cache, the Instruction Cache, and the VLIW Cache; and two engines – the VLIW Scheduler Engine and the VLIW Engine. The Data Cache is a standard multiported cache and the Instruction Cache is a small standard single-ported cache. The VLIW Cache is also an ordinary cache, with one read and one write ports; its single additional feature is an extra field per directory entry to store the **nba** of the blocks saved (see Section 4.4). We believe there are no particular difficulties in the implementation of these caches, except for the extra silicon area that may be required by the VLIW Cache. Nevertheless, a significant part

of the silicon area for the VLIW Cache, if not all, can come from that which otherwise would be used for the Instruction Cache, as we have shown in Section 6.5.

The DTSVLIW Scheduler Engine can be divided into the Scheduler Unit and the Primary Processor. We have shown that the core operations performed by the Scheduler Unit are not complex (see Section 4.6), and the Primary Processor is a simple pipelined scalar processor; therefore, both offer no significant implementation difficulties.

In order to exploit the ILP available in programs, the VLIW Engine of a DTSVLIW machine has to employ many functional units that operate in parallel. A machine with a large number of functional units requires a large number of register file ports and data cache ports, but large multiported data cache and register files impact on the clock cycle time. However, any machine using an equivalent number of functional units would suffer the same problem. Moreover, techniques such as functional units clustering and data cache interleaving can be used in the DTSVLIW to ameliorate this problem. We have left the study of the impact of incorporating these techniques into the DTSVLIW architecture for future work.

We have not examined the implementation of the register renaming mechanism of the DTSVLIW in Chapter 4. This is because the logic necessary for register renaming in the DTSVLIW is likely to be simple. The DTSVLIW has to rename less than one register per cycle on average because not all instructions inserted in the scheduling list during scheduling have to be renamed. Nevertheless, in the worst case all candidate instructions in the DTSVLIW scheduling list may have to be renamed in the same clock cycle. If the number of long instructions in the scheduling list is large, 16 for example, this may complicate the register renaming logic. However, renaming in the DTSVLIW consists of simply getting a new name for a register from the pool of available registers, which may be implemented as a simple array of bits. Therefore, the logic necessary for implementing register renaming is not likely to affect the DTSVLIW implementation complexity or clock cycle time.

A simple register renaming scheme that takes advantage of the DTSVLIW register renaming characteristics would, for example, allocate the renaming registers necessary for each instruction during their insertion in the scheduling list. This could be done by using registers containing the renaming register names as pointers to the

array of bits representing the pool of available registers. Every cycle, the pointer for each type of renaming register would be moved to a bit in the respective array indicating a renaming register available. An instruction being inserted in the scheduling list receives the pointer value as the name of a renaming register. These renaming register names accompany each candidate instruction, and the Scheduler Unit uses them if a candidate instruction needs to be renamed. In case the candidate instruction is installed without being renamed, the renaming register name is used to change the bit in the renaming register pool, indicating that the renaming register is available again. This scheme only requires each cycle: updating the position of the pointers moved due to the insertion of one instruction, and updating the arrays representing the renaming registers due to return of unused registers. The maximum number of renaming registers consumed by a single instruction is two and the maximum number of array updates is equal to the number of long instructions in the scheduling list. Therefore, the logic necessary for implementing this scheme is not likely to affect the DTSVLIW clock cycle time.

In the scheme for renaming registers described, some registers are taken from the poll, held for some cycles, and then made available again. This appears to reduce the effective number of renaming registers available by almost the size of the scheduling list. However, in fact, instructions requiring different renaming register types are likely to alternate during insertion, which reduces the number of renaming registers of each type being held. Furthermore, there are potentially other register renaming schemes more effective than this. We have described this one only to show that is possible to implement register renaming in the DTSVLIW with simple hardware that should not affect the DTSVLIW clock cycle time.

# Chapter 8

# Summary, Conclusions and Future Work

This thesis presents the definition and the evaluation of the integer performance of the *Dynamically Trace Scheduled VLIW* (DTSVLIW) architecture. This architecture can be used to implement machines that execute code of current RISC or CISC ISAs in a VLIW fashion, delivering instruction-level parallelism with backward code compatibility. Our main motivation for the development of the DTSVLIW architecture came from the observation that small instruction caches (16Kbytes) can achieve high average hit rates (99%), which shows that there is strong code execution locality in programs. The DTSVLIW architecture takes advantage of the code execution locality by executing programs in two distinct modes. The first time a fragment of code is encountered, it is executed in sequential mode by a simple pipelined processor, the DTSVLIW's Primary Processor. At the same, this code fragment is scheduled by the DTSVLIW's Scheduler Unit into parallel long instructions (VLIW instructions) and saved in the DTSVLIW's VLIW Cache. In subsequent execution encounters, the DTSVLIW's VLIW Engine executes the scheduled version of the code in parallel mode.

The DTSVLIW architecture is similar to the DIF, proposed by Nair and Hopkins [Nair97]. We have conceived the DTSVLIW independently of the DIF, however, which has permitted an implementation significantly different from that suggested by the proponents of DIF. The DTSVLIW differs from the DIF in the organisation of the cache used by the VLIW Engine, in its scheduling algorithm, in its register renaming mechanism, and in the VLIW Engine register access

mechanism.

## 8.1 Thesis Summary

In this thesis, we present and evaluate the DTSVLIW architecture, a processor architecture that we have designed and developed to exploit ILP by using a dynamic code scheduler and a VLIW execution core. In order to evaluate the DTSVLIW, we have surveyed: several architectures for exploiting ILP; several ISA, hardware, and software support for exploiting ILP; and published research on the amount of ILP available in programs. We have then discussed aspects of the DTSVLIW architecture that are related to previous work on microcode scheduling and VLIW architectures. In addition, we have discussed systems that, like the DTSVLIW, can be used to solve the VLIW object code compatibility problem or take advantage of code locality to exploit the ILP available in programs.

The key features of the DTSVLIW are the scheduling of the instruction trace produced by the Primary Processor into long instructions and the VLIW Engine addressing of long instructions. The DTSVLIW schedules instructions using a simplified version of the FCFS algorithm, historically used for microcode compaction. This algorithm has been adapted to perform superblock scheduling and to operate in a pipelined fashion. The superblocks are generated in the form of a list of long instructions. A simple and effective long instruction addressing mechanism groups these long instructions in blocks, which are saved in the VLIW Cache for the VLIW Engine to execute. The DTSVLIW scheduling and addressing of long instructions allows support for precise interrupts and for the execution of: programs that read their own code as data, programs with self-modifying code, and programs with load instructions that cause side effects (memory mapped I/O).

We have implemented a parameterised and instrumented execution-driven simulator of the DTSVLIW. Using this simulator, we have examined the effect of various architectural parameters on the DTSVLIW performance when running the SPECint95 benchmark suite. Our results show that the DTSVLIW can achieve average ILP higher than 4 IPC with 16 untyped functional units and perfect caches, and 2 IPC with a feasible machine configuration with 12 typed functional units and caches of reasonable sizes. Our results also show that the DTSVLIW performs better

than DIF, VLIW, and Superscalar architectures for important machine configurations.

## 8.2 Conclusions

The design of the DTSVLIW architecture has been driven by the requirement to develop an architecture that can be effectively implemented to realise the fast clocking that can be achieved with VLIW designs. The Primary Processor and the VLIW Engine of the DTSVLIW do not restrict the achievable clock rate. It is the Scheduler Unit that is the key to an efficient and high clock rate implementation. As detailed in Section 4.6, the core logic of the Scheduler Unit is straightforward to implement, being comparable to an integer adder, and as such should not render the DTSVLIW cycle time longer than that determined by the VLIW Engine design. The remaining operations performed by the simplified version of the FCFS instruction scheduling algorithm used by the DTSVLIW have a complexity that is readily implementable.

Our experimental results confirm that the DTSVLIW takes advantage of code execution locality. The experiments discussed in Subsection 6.1.5 show that the DTSVLIW's Instruction Cache can be substituted for a simple instruction buffer if its VLIW Cache is large enough. That is, with a large enough VLIW Cache, the DTSVLIW spends most of its time executing parallel code. The experiments in Subsection 6.1.5 further confirm this, showing that the fact that the Primary Processor retains multicycle instructions at its execute pipeline stage until they complete execution does not impact on the DTSVLIW performance.

Our results demonstrate, however, that multicycle load instructions impose a severe performance penalty on the DTSVLIW architecture and it is clear that it is important to get the load latency as close to one cycle as possible. Single cycle store instructions are not so important. Nevertheless, reasonably low load latency (3 cycles) is achievable with high clock rate Superscalars, as demonstrated by the Alpha 21264 processor [Kessler99], and the same or lower load latency can be expected in DTSVLIW implementations.

Our results demonstrate the effectiveness of the DTSVLIW scheduling algorithm. Experiments in Section 6.2 show that there is no significant reduction in performance over other candidate scheduling algorithms, even though these

algorithms are expected to be more difficult to implement. Even more so if the high clock rate of the VLIW Engine is to be achieved.

The DTSVLIW scheduling algorithm requires far fewer resources than the Greedy algorithm used by the DIF architecture in order to achieve equivalent performance, as have been shown in our previous paper [deSouza99c] and in Section 6.3. In this section, our experiments show that the DTSVLIW achieves performance similar to the DIF. However, while the DTSVLIW uses 18 integer and 6 FP renaming registers and a 216-Kbyte VLIW Cache, the DIF requires 96 integer and 96 FP renaming registers and a 463-Kbyte DIF Cache.

In many areas that have great needs for processing power, the behaviour of algorithms is irregular and dependent on the input data, making it necessary to perform instruction scheduling at run time to achieve performance. Architectures such as the VLIW and EPIC rely solely on the compiler to exploit the ILP available in programs. The DTSVLIW, on the other hand, does not rely on the compiler to exploit ILP. The experiments presented in Section 6.4 demonstrate that the DTSVLIW algorithm is able to find more parallelism than a state-of-the-art VLIW compiler under similar conditions. This is possible because the DTSVLIW scheduler algorithm has access to dynamic information not available to the VLIW compiler. Nevertheless, we believe that a conjunction of the DTSVLIW architecture and compiler technology such as loop unrolling, software pipeline, and predication (if added to the DTSVLW ISA) would produce performance even better than that shown in Section 6.4.

In Section 6.5, we have compared the performance of a powerful Superscalar processor with the performance of an equivalent DTSVLIW machine configuration. Our results have shown that the DTSVLIW performance is overall better than the Superscalar. This is so because the scheduling list of the DTSVLIW is larger than the instruction window that the Superscalar uses for scheduling. However, even though larger, the DTSVLIW's scheduling list is likely to be simpler to implement than the Superscalar's instruction window. The DTSVLIW's scheduling list is potentially simpler to implement because only one instruction can be inserted into the DTSVLIW's scheduling list each cycle, while the instruction window of a Superscalar processor has to be able to receive many instructions per cycle. This

difference in complexity of the scheduling hardware, aggravated by the fact that the Superscalar scheduling hardware is in its main data-path, is likely to make the Superscalar clock cycle slower than the DTSVLIW's. Therefore, in real implementations, the DTSVLIW performance is likely to be significant better than the Superscalar performance for implementations using equivalent technologies.

Due to the Superscalar fetch bottleneck (discussed in Subsection 3.4.1), which is going to limit the performance of future more parallel Superscalar machines, several new architectures have being proposed and some classical architectures revisited recently. We believe that two of these new architectures are particularly promising: Trace Cache and EPIC.

The Trace Cache architecture, like the DTSVLIW, takes advantage of code execution locality to achieve ILP. In fact, the DTSVLIW and the Trace Cache architecture can be seen as members of the same family of architectures. The main difference between them is in how they divide the responsibility for performing ILP extraction functions within their functional units. The Trace Cache architecture's execution-core performs all ILP related functions, while the DTSVLIW's execution-core performs none. On the other hand, the DTSVLIW's Scheduler Unit performs all ILP related functions and the Trace Cache architecture's Fill Unit performs none. We believe that machines following the DTSVLIW philosophy of simplicity in the main execution-core are likely to perform better than machines in which the execution-core is the main responsible for instruction scheduling tasks. This view is corroborated by the evolution of the Alpha family of microprocessors. With their simple ISA and implementation tailored to high clock rate, Alpha microprocessors are "performance leaders since their introduction in 1992" [Kessler99].

The new generation of Intel processors, which is going to substitute the current Pentium processors (x86 or IA-32 ISA), is going to have EPIC ISA. EPIC architectures are a result of two decades of research in VLIW compilers and architectures. They are similar to pure VLIW but their ISA incorporate features that resolve most of the VLIW disadvantages. EPIC architectures, however, still rely solely on the compiler to expose the ILP to their hardware. The DTSVLIW architecture, on the other hand, does not rely on the compiler to exploit ILP. In addition, the DTSVLIW architecture can be employed to emulate legacy ISA code in

new EPIC processors, taking advantage of the EPIC core. It can also be used to execute EPIC code directly, collecting dynamic branch behaviour information and organising the code during the scheduling phase of program execution to increase the EPIC processor performance. Due to the availability of predication and hoisting ISA support in EPIC architectures, we expect that the DTSVLIW will achieve performance levels even higher than those shown in this thesis when executing EPIC code.

## 8.3 Future Work

The DTSVLIW architecture opens several new avenues of research. In this section, we discuss some of them that we will investigate in future work.

### 8.3.1 Mechanisms for Reducing the Impact of Load Latency

The experiments in Subsection 6.1.6 and Subsection 6.1.7 show that the load instruction latency strongly affects the DTSVLIW performance. However, techniques such as *fast address calculation* [Austin96], *zero-cycle loads* [Austin96], or *load value prediction* [Lipasti96a] can be adapted and incorporated into the DTSVLIW for reducing the impact of the load latency. Another promising research topic that we intend to investigate is the incorporation of mechanisms for data prefetching in the Scheduling Unit or VLIW Engine of the DTSVLIW.

### 8.3.2 New VLIW Cache Organisations

The results presented in Subsection 6.1.7 show that the DTSVLIW does not use its VLIW Cache efficiently. A large number of nop instructions are saved into the VLIW Cache. However, the Scheduling Unit could compact the blocks of long instructions before saving them in the VLIW Cache. This would allow good performance with small VLIW Caches. On the other hand, this would increase the number of pipeline stages of the VLIW Engine and consequently the next long instruction miss penalty. Nevertheless, next long instruction prediction hardware might be used to minimise the next long instruction miss penalty.

### 8.3.3 Next Long Instruction Prediction

When the VLIW Engine misses in the VLIW Cache, at least one DTSVLIW cycle is lost. However, we believe that available techniques for hardware branch prediction can be adapted to be used for next long instruction prediction in the DTSVLIW.

### 8.3.4 Clustered DTSVLIW

Larger DTSVLIW machines requires large number of register file and data cache ports. This may affect the DTSVLIW clock cycle time negatively. However, the VLIW Engine core can be divided into several clusters of functional units, each cluster with its private register file. Register values that are required by other clusters can be forward, paying a delay cost, through a network connecting the clusters. The data cache can be interleaved, allowing multiple fast accesses via the association of different clusters to different ranges of data addresses.

### 8.3.5 DTSVLIW Performance with Other Classes of Program

This thesis investigates the DTSVLIW integer performance. We expect the DTSVLIW performance to be higher when execution other classes of program such as floating-point and multimedia code due to their higher available ILP. However, the latency of multicycle instructions may have a significant impact on the performance of the DTSVLIW when executing these classes of code.

### 8.3.6 DTSVLIW-Tailored Compilers

Although the DTSVLIW does not rely on the compiler to exploit the ILP available in programs, it can benefit from compiler optimisations tailored to improve the performance of its scheduling algorithm. In fact, the DTSVLIW architecture is complementary to compiler techniques for extracting ILP, in that the latter does static scheduling while the former does dynamic scheduling. While dynamic scheduling is better, it is constrained by the amount of code that the hardware can analyse and by the characteristics of the code presented for analysis. The compiler can assist the DTSVLIW by producing code that allows easier implementation of the scheduling hardware, e. g. EPIC code, and code that, when dynamically scheduled by the DTSVLIW, results in high ILP. Hence, new compiler techniques, such as predication

and instruction hoisting, can be used to produce code tailored for the DTSVLIW. The impact of these compiler techniques on the DTSVLIW performance is another topic that we intend to investigate in future work.

# Glossary

| | |
|---|---|
| ALU | *Arithmetic and Logic Unit* |
| ARB | *Address Resolution Buffer* |
| BHT | *Branch History Table* |
| BTB | *Branch Target Buffer* |
| CISC | *Complex Instruction Set Computers* |
| DAISY | *Dynamically Architected Instruction Set from Yorktown* |
| DIF | *Dynamic Instruction Formatting* |
| DSVLIW | *Dynamically Scheduled VLIW* |
| DTSVLIW | *Dynamically Trace Scheduled Very Long Instruction Word* |
| EPIC | *Explicitly Parallel Instruction Computing* |
| FCFS | *First Come First Served* |
| I/O | *input/output* |
| ILP | *instruction-level parallelism* |
| IPC | *instructions per cycle* |
| IRAM | *Intelligent Random Access Memory* |
| ISA | *instruction set architecture* |
| MPS | *Miss Path Scheduling* |
| nba | *next block address* |
| NLS | *next line* and *set* |

| | |
|---|---|
| nop | *no-operation* |
| PC | *program counter* |
| RISC | *Reduced Instruction Set Computer* |
| SPEC | *Standard Performance Evaluation Corporation* |
| u.a.m. | *using arithmetic mean* |
| VLIW | *Very Long Instruction Word* |
| VLSI | *Very Large Scale Integrated* |
| VMM | *Virtual Machine Monitor* |

# Bibliography


[Adam74] T. L. Adam, K. M. Chandy, and J. R. Dickson, " A Comparison of List Schedules for Parallel Processing Systems", Communications of the Association for Computing Machinery, Vol. 17, pp. 685-690, December 1974.

[Agerwala87] T. Agerwala and J. Cocke, "High Performance Reduced Instruction Set Processors", Technical Report RC12434, IBM Thomas J. Watson Research Center, March 1987.

[Aho86] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers – Principles Techniques and Tools", Addison-Wesley Publishing Company, USA, 1986.

[Allen83] J. R. Allen, K. Kennedy, C. Porterfield, and J. Warren, "Conversion of Control Dependence to Data Dependence", Proceedings of the 10th Annual ACM Symposium on Principles of Programming Languages, pp. 177-189, 1983.

[Alverson90] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The Tera Computer System", Proceedings of the International Conference on Supercomputing, pp. 1-6, 1990.

[Anderson67] D. W. Anderson, F. J. Sparacio, R. M. Tomasulo, "The IBM 360 Model 91: Machine Philosophy and Instruction Handling", IBM Journal of Research and Development, Vol.11, No. 1, pp. 8-24, January 1967.

[August98] D. I. August, D. A. Connors, S. A. Mahlke, J. W. Sias, K, M. Crozier, B.-C. Cheng, P. R. Eaton, Q. B. Olaniran, and W. W. Hwu, "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture", Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 227-237, 1998.

[Austin96] T. M. Austin, "Hardware and Software Mechanisms for Reducing Load Latency", PhD Thesis, University of Wisconsin, USA, 1996.

[Banerjia98] S. Banerjia, S. W. Sathaye, K. N. Menezes, and T. Conte, "MPS: Miss-Path Scheduling for Multiple-Issue Processors", IEEE Transactions on Computers, Vol. 47, No. 12, pp. 1382-1397, December 1998.

[Bhandarkar97] D. Bhandarkar, "RISC versus CISC: A Tale of Two Chips", Computer Architecture News, Vol. 25, No. 1, pp. 1-12, March 1997.

[Bloch59] E. Bloch, "The Engineering Design of the Stretch Computer", Proceedings of the Fall Joint Computer Conference, pp. 48-59, 1959.

[Burger97] D. Burger and J. R. Goodman, "Billion-Transistor Architectures", IEEE Computer, pp. 46-48, September 1997.

[Butler93] M. Butler and Y. Patt, "A Comparative Performance Evaluation of Various State Maintenance Mechanisms", Proceedings of the 26th Annual International Symposium on Microarchitecture, pp. 70-79, 1993.

[Calder95] B. Calder and D. Grunwald, "Next Cache Line and Set Prediction", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 287-296, 1995.

[Charlesworth81] A. E. Charlesworth, "An Approach to Scientific Array Processing: The Architectural Design of the AP-120B/FPS-164 Family", IEEE Computer, pp. 18-27, September 1981.

[Charney97] M. J. Charney and T. R. Puzak, "Prefetching and Memory System Behaviour of the SPEC95 Benchmark Suite", IBM Journal of Research and Development, Vol. 41, No. 3, pp. 265-285, May 1997.

[Chernoff98] A. Chernoff, M. Herdeg, R. Hookway, C. Reeve, N. Rubin, T. Tye, S. B. Yadavalli, and J. Yates, "FX!32: A Profile-Directed Binary Translator", IEEE Micro, pp. 56-64, March/April 1998.

[Citron98] D. Citron, D. Feitelson, and L. Rudolph, "Accelerating Multi-Media Processing by Implementing Memoing in Multiplication and Division Units", Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 252-261, 1998.

[Colwell88] R. P. Cowell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. Rodman, "A VLIW Architecture for a Trace Scheduling Compiler", IEEE Transactions on Computers, Vol. C-37, No. 8, pp. 967-979, August 1988.

[Conte95a] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 333-344, 1995.

[Conte95b] T. M. Conte and S. W. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures", Proceedings of the 28th Annual International Symposium on Microarchitecture, pp. 208-218, 1995.

[Conte96] T. M. Conte, S. W. Sathaye, and S. Banerjia, "A Persistent Rescheduled-Page Cache for Low Overhead Object Code Compatibility in VLIW Architectures", Proceedings of the 29th Annual International Symposium on Microarchitecture, pp. 4-13, 1996.

[Dasgupta76] S. Dasgupta and J. Tartar, "The Identification of Maximal Parallelism in Straight Line Microprograms", IEEE Transaction on Computers, Vol. C-25, pp. 986-991, October 1976.

[Davidson81] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machines", IEEE Transactions on Computers, Vol. C-30, No. 7, pp. 460-477, July 1981.

[Davis79] A. L. Davis, "A Data Flow Evaluation System Based on the Concept of Recursive Locality", Proceedings of the National Computing Conference, AFIPS Press, Reston, Va., pp. 1079-1086, 1979.

[deSouza93] A. F. de Souza, "Evaluating the Parameters of a VLIW Architecture", MSc Thesis, COPPE/Sistemas, Federal University of Rio de Janeiro, Brazil, 1993 (in Portuguese).

[deSouza95] A. F. de Souza, F. D. de Freitas, "Supercomputer Performance with a Workstation Cluster on Dense Matrix Multiplication", Proceedings of the 7th IASTED International Conference on Parallel and Distributed Computing and Systems, pp. 257-261, 1995.

[deSouza97] A. F. de Souza, E. S. T. Fernandes, and A. Wolfe, "On the Balance of VLIW Architectures", Journal of Systems Architecture, No. 43, pp. 14-22, 1997.

[deSouza98a] A. F. de Souza and P. Rounce, "Dynamically Trace Scheduled VLIW Architectures", Proceedings of the High-Performance Computing and Networking' 98 – HPCN'98, on Lecture Notes in Computer Science, Vol. 1401, pp. 993-995, April 1998.

[deSouza98b] A. F. de Souza and P. Rounce, "SPECint95 Performance of an Implementation of the Dynamically Trace Scheduled VLIW Architecture", Proceedings of the 10th Brazilian Symposium on Computer Architecture and High Performance Computing, pp. 185-188, 1998.

[deSouza99a] A. F. de Souza and P. Rounce, "Dynamically Scheduling the Trace Produced during Program Execution into VLIW Instructions", Proceedings of the Second Merged Symposium IPPS/SPDP'1999: 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing, pp. 248-257, April 1999.

[deSouza99b] A. F. de Souza and P. Rounce, "Effect of Multicycle Instructions on the Integer Performance of the Dynamically Trace Scheduled VLIW Architecture," Proceedings of the High-Performance Computing and Networking' 99 – HPCN'99, on Lecture Notes in Computer Science, Vol. 1593, pp. 1203-1206, 1999.

[deSouza99c] A. F. de Souza and P. Rounce, "On the Effectiveness of the Scheduling Algorithm of the Dynamically Trace Scheduled VLIW Architecture," Proceedings of the 11th Brazilian Symposium on Computer Architecture and High Performance Computing, pp. 167-174, 1999.

[DeWitt76] D. J. DeWitt, "A Machine Independent Approach to the Production of Optimal Horizontal Microcode", PhD Thesis, University of Michigan, USA, August 1976.

[Diefendorff94] K. Diefendorff and E. Silha, "The PowerPC User Instruction Set Architecture", IEEE Micro, pp. 30-41, October 1994.

[Diep95] T. A. Diep, C. Nelson, and J. P. Shen, "Performance Evaluation of the PowerPC620 Microarchitecture", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 163-174, 1995.

[Dulong98] C. Dulong, "The IA-64 Architecture at Work", IEEE Computer, pp. 24-31, July 1998.

[Ebcioglu88] K. Ebcioglu, "Some Design Ideas for a VLIW Architecture for Sequential-Natured Software", Parallel Processing (Proceedings of the IFIP WG 10.3 Working Conference on Parallel Processing, M. Cosnard, M. H. Barton, and M. Vanneschi, editors), Elsevier Science Publishers, pp. 3-21, 1988.

[Ebcioglu89] K. Ebcioglu and T. Nakatani, "A New Compilation Technique for Parallelizing Loops with Unpredictable Branches on a VLIW Architecture", Languages and Compilers for Parallel Computing (D. Gelernter, A. Nicolau, and D. Padua, editors), pp. 213-229, 1989.

[Ebcioglu97] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 26-37, 1997.

[Eggers97] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, D. M. Tullsen, "Simultaneous Multithreading: A Platform for the Next-Generation Processors", IEEE Micro, pp. 12-19, September/October 1997.

[Ellis86] J. R. Ellis, "Bulldog: A Compiler for VLIW Architectures", MIT Press, 1986.

[Fernandes89] E. S. T. Fernandes, C. L. Amorim, V. C. Barbosa, F. M. G. França, A. F. de Souza, "MPH - A Hybrid Parallel Machine", Microprocessing and Microprogramming, North-Holland, Vol. 25, pp. 229-232, 1989.

[Fisher81] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", IEEE Transactions on Computers, Vol. C-30, No. 7, pp. 478-490, July 1981.

[Fisher84] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", IEEE Computer, pp. 45-53, July 1984.

[Fisher92] J. A. Fisher and S. M. Freudenberger, "Predicting Conditional Branch Directions from Previous Runs of a Program", Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 85-95, 1992.

[Fisher93] J. A. Fisher, "Global Code Generation for Instruction-Level Parallelism", Technical Report HPL-93-43, Computer Research Center, Hewlett Packard, 1993.

[Foster72] C. C. Foster and E. M. Riseman, "Percolation of Code to Enhance Parallel Dispatching and Execution", IEEE Transactions on Computers, Vol. C-21, No. 12, pp. 1411-1415, December 1972.

[Franklin92] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism", Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 58-67, 1992.

[Franklin94] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue", Proceedings of the 27th Annual International Symposium on Microarchitecture, pp. 162-171, 1994.

[Freudenberger92] S. M. Freudenberger and J. C. Ruttenberg, "Phase Ordering of Register Allocation and Instructions Scheduling", Code Generation – Concepts, Tools, Techniques (Proceedings of the International Workshop on Code Generation, R. Giegerich and S. L. Graham, editors), Spring-Verlag, pp. 146-172, May 1992.

[Friendly97] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Alternative Fetch and Issue Policies for the Trace Cache Fetch Mechanism", Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 24-33, 1997.

[Friendly98] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", Proceedings of the 31st Annual International Symposium on Microarchitecture, pp. 173-181, 1998.

[Fu98] C.-H. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value Speculation Scheduling for High Performance Processors", Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 262-271, 1998.

[Gabbay97] F. Gabbay and A. Mendelson, "Can Program Profiling Support Value Prediction?", Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 270-280, 1997.

[Gabbay98] F. Gabbay and A. Mendelson, "The Effect of Instruction Fetch Bandwidth on Value Prediction", Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 272-281, 1998.

[Gee93] J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, "Cache Performance of the SPEC92 Benchmark Suite", IEEE Micro, pp. 17-27, August 1993.

[Giladi95] R. Giladi and N. Ahituv, "SPEC as a Performance Evaluation Measure", IEEE Computer, pp. 33-42, August 1995.

[Grohoski90] G. F. Grohoski, "Machine Organization of the IBM RISC System/6000 Processor", IBM Journal of Research and Development, Vol. 34, No. 1, pp. 37-58, January 1990.

[Gross90] T. Gross and M. Ward, "The Supression of Compensation Code", Advances in Languages and Compilers for Parallel Computing (A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors), MIT Press, pp. 260-273, 1990.

[Gwennap97] L. Gwennap, "Intel, HP make EPIC Disclosure", Microprocessor Report, Vol. 11, No. 14, pp. 1-9, October 27, 1997.

[Hara96] T. Hara, H. Ando, C. Nakanishi, and M. Nakaya, "Performance Comparison of ILP Machines with Cycle Time Evaluation", Proceedings of the 23rd Annual International Symposium on Computer Architecture, pp. 213-224, 1996.

[Hennessy86] J. L. Hennessy, "RISC-Based Processors: Concepts and Prospects", New Frontiers in Computer Architecture Conference Proceedings, pp. 95-103, 1986.

[Hookway97] R. J. Hookway and M. A. Herdeg, "DIGITAL FX!32: Combining Emulation and Binary Translation", Digital Technical Journal, Vol. 9, No.1, pp. 3-11, 1997.

[Hwu87] W. W. Hwu and Y. N. Patt, "Checkpoint Repair for Out-of-order Execution Machines," Proceedings of the 14th Annual International Symposium on Computer Architecture, pp. 18-26, 1987.

[Hwu93] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective Technique for VLIW and Superscalar Compilation", The Journal of Supercomputing, Vol. 7, pp. 229-248, 1993.

[IBM94] IBM, "PowerPC620$^{tm}$ RISC Microprocessor Technical Summary", IBM Technical Summary, IBM Order Number MPR620TSU-01, 1994.

[Intel89] Intel, "i860 64-Bit Microprocessor Hardware Manual – Preliminary", Intel, Order Number: 240296-003, October 1989.

[Intel99] Intel, "IA-64 Application Developer's Architecture Guide", Intel, Order Number: 245188-001, May 1999.

[Jacob95] B. Jacob and T. Mudge, "Notes on Calculating Computer Performance", Technical Report CSE-TR-231-95, Department of Electrical Engineering and Computer Science, University of Michigan, USA, March 1995.

[Jacobson97] Q. Jacobson, E. Rotenberg, and J. E. Smith, "Path-Based Next Trace Prediction", Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 14-23, 1997.

[Johnson91] M. Johnson, "Superscalar Microprocessor Design", Prentice-Hall, 1991.

[Jouppi89] N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 272-282, 1989.

[Kaeli91] D. R. Kaeli and P. G. Emma, "Branch History Table Prediction of Moving Target Branches Due to Subroutines Returns", Proceedings of the 18th Annual International Symposium on Computer Architecture, pp. 34-41, 1991.

[Keller75] R. M. Keller, "Look-Ahead Processors", ACM Computer Surveys, Vol. 7, No. 8, pp. 177-195, December 1975.

[Kelly98] E. J. Kelly, R. F. Cmelik, and M. J. Wing, "Memory Controller for a Microprocessor for Detecting a Failure of Speculation on the Physical Nature of a Component Being Addressed", Transmeta Corporation, United States Patent No. 5,832,205, November 1998.

[Kessler99] R. E. Kessler, "The Alpha 21264 Microprocessor", IEEE Micro, pp. 24-36, March-April 1999.

[Kogge81] P. M. Kogge, "The Architecture of Pipelined Computers", McGraw-Hill, 1981.

[Kozyrakis97] C. E. Kozyrakis, S. Perissakis, D. A. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, "Scalable Processors in the Billion-Transistor Era: IRAM", IEEE Computer, pp. 75-78, September 1997.

[Kunkel86] S. R. Kunkel and J. E. Smith, "Optimal Pipelining in Supercomputers", Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 404-414, 1986.

[Lee_B94] B. Lee and A. R. Hurson, "Dataflow Architectures and Multithreading", IEEE Computer, pp. 27-39, August 1994.

[Lee_JFK84] J. K. F. Lee and A. J. Smith, "Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer, pp. 6-22, January 1984.

[Lipasti96a] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value Locality and Load Value Prediction", Proceedings of the 7th International Conference on Architecture Support for Programming Languages and Operating Systems, pp. 138-147, October 1996.

[Lipasti96b] M. H. Lipasti and J. P. Shen, "Exceeding the Data-Flow Limit Via Value Prediction", Proceedings of the 29th Annual International Symposium on Microarchitecture, pp. 226-237, December 1996.

[Lipasti97] M. H. Lipasti and J. P Shen, "Superspeculative Microarchitecture for Beyond AD 2000", IEEE Computer, pp. 59-66, September 1997.

[Mahlke92a] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors", Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 238-247, 1992.

[Mahlke92b] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective Compiler Support for Predicated Execution Using the Hyperblock", Proceedings of the 25th Annual International Symposium on Microarchitecture, pp. 45-54, 1992.

[Malik92] N. Malik, R. J. Eickemeyer, and S. Vassiliadis, "Interlock Collapsing ALU for Increased Instruction-Level Parallelism", Proceedings of the 25th Annual International Symposium on Microarchitecture, pp. 149-157, 1992.

[Matzke97] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?", IEEE Computer, pp. 37-39, September 1997.

[McFarling93] S. McFarling, "Combining Branch Predictors", Digital Western Research Laboratory – WRL Technical Note TN-36, June 1993.

[Melvin88] S. Melvin, M. Shebanow, and Y. Patt, "Hardware Support for Large Atomic Units in Dynamic Scheduled Machines", Proceedings of the 21st Annual International Symposium on Microarchitecture, pp. 60-66, December 1988.

[Moreno97] J. H. Moreno, M. Moudgill, K, Ebcioglu, E. Altman, C. B. Hall, R. Miranda, S.-K. Chen, and A. Polyak, "Simulation/Evaluation Environment for a VLIW Processor Architecture", IBM Journal of Research and Development, Vol. 41, No. 3, pp. 287-302, May 1997.

[Moudgill96] M. Moudgill, J. H. Moreno, K. Ebcioglu, E. Altman, S.-K. Chen, and A. Polyak, "Compiler/Architecture Interaction in a Tree-Based VLIW Processor", IBM Research Report RC20694, November 1996.

[Nair95] R. Nair, "Dynamic Path-Based Branch Correlation", Proceedings of the 28th Annual International Symposium on Microarchitecture, pp. 15-23, 1995.

[Nair97] R. Nair and M. E. Hopkins, "Exploiting Instructions Level Parallelism in Processors by Caching Scheduled Groups", Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 13-25, 1997.

[Nakatani89] T. Nakatani and K. Ebcioglu, "Combining as a Compilation Technique for a VLIW Architecture", Proceedings of the 22nd Annual International Symposium on Microprogramming and Microarchitecture, pp. 43-55, 1989.

[Nakatani93] T. Nakatani and K. Ebcioglu, "Making Compaction-Based Parallelization Affordable", IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 9, pp. 1014-1029, September 1993.

[Nakra99] T. Nakra, R. Gupta, and M. L. Soffa, "Value Prediction in VLIW Machines", Proceedings of the 26th Annual International Symposium on Computer Architecture, pp. 258-269, 1999.

[Nicolau84] A. Nicolau and J. A. Fisher, "Measuring the Parallelism Available for Very Long Instruction Word Architectures", IEEE Transactions on Computers, Vol. C-33, No. 11, pp. 968-976, November 1984.

[Nicolau85] A. Nicolau, "Percolation Scheduling: A Parallel Compilation Technique", Technical Report TR 85-678, Department of Computer Science, Cornell University, USA, May 1985.

[Oehler90] R. R. Oehler and R. D. Groves, "IBM RISC System/6000 Processor Architecture", IBM Journal of Research and Development, Vol. 34, No. 1, pp. 23-36, January 1990.

[Padua86] D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers", Communications of the ACM, Vol. 12, No. 29, pp. 1184-1201, December 1986.

[Palacharla97] S. Palacharla, N. Jouppi, and J. E. Smith, "Complexity-Effective Superscalar Processors", Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 206-218, 1997.

[Pan92] S. Pan, K. So, and J. T. Rahmeh, "Improving the Accuracy of Dynamic Branch Prediction Using Branch Correlation", Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 76-84, 1992.

[Park91] J. R. H. Park and M. S. Schlansker, "On Predicated Execution", Technical Report HPL-91-58, HP Laboratories, Palo Alto, CA, May 1991.

[Patel97] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Critical Issues Regarding the Trace Cache Fetch Mechanism", Technical Report CSE-TR-335-97, Department of Electrical Engineering and Computer Science, University of Michigan, USA, 1997.

[Patel98] S. J. Patel, M. Evers, and Y. N. Patt, "Improving Trace Cache Effectiveness with Branch Promotion and Trace Packing", Proceedings of the 25th Annual International Symposium on Computer Architecture, pp. 262-271, 1998.

[Patel99] S. J. Patel, D. H. Friendly, and Y. N. Patt, "Evaluation of Design Options for the Trace Cache Fetch Mechanism", IEEE Transactions on Computers, Vol. C-48, No. 2, pp.193-204, 1999.

[Patt97] Y. N. Patt, S. J. Patel, M. Evers, D. H. Friendly, and J. Stark, "One Billion Transistors, One Uniprocessor, One Chip", IEEE Computer, pp. 51-57, September 1997.

[Patterson85] D. A. Patterson, "Reduced Instruction Set Computers", Communications of the ACM, Vol. 28, No. 1, pp. 8-21, January 1985.

[Patterson96] D. A. Patterson and J. L Hennessy, "Computer Architecture: A Quantitative Approach, Second Edition", Morgan Kaufmann Publishers, Inc., 1996.

[Ramamoorthy77] C. C. Ramamoorthy and H. F. Li, "Pipeline Architecture", ACM Computer Surveys, Vol. 9, No. 1, pp. 61-102, March 1977.

[Rau89] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle, "The Cydra 5 Departmental Supercomputer: Design Philosophies, Decisions, and Trade-offs", IEEE Computer, pp. 12-35, January 1989.

[Rau93a] B. R. Rau and J. A. Fisher, "Instruction-Level Parallelism: History, Overview, and Perspective", The Journal of Supercomputing, Vol. 7, pp. 9-50, 1993.

[Rau93b] B. R. Rau, "Dynamically Scheduled VLIW Processors", Proceedings of the 26th Annual International Symposium on Microarchitecture, pp. 80-92, 1993.

[Reinman98] G. Reinman and B. Calder, "Predictive Techniques for Aggressive Load Speculation", Proceedings of the 31st Annual International Symposium on Microarchitecture, pp. 127-137, 1998.

[Riseman72] E. M. Riseman and C. C. Foster, "The Inhibition of Potential Parallelism by Conditional Jumps", IEEE Transactions on Computers, Vol. C-21, No. 12, pp. 1405-1411, December 1972.

[Rotenberg96] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching", Proceedings of the 29th Annual International Symposium on Microarchitecture, pp. 24-34, 1996.

[Rotenberg97] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace Processors", Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 138-148, 1997.

[Rotenberg99] E. Rotenberg, S. Bennett, and J. E. Smith, "A Trace Cache Microarchitecture and Evaluation", IEEE Transactions on Computers, Vol. C-48, No. 2, pp. 111-120,1999.

[Saini93] A. Saini, "An Overview of the Intel Pentium Processor", Proceedings of the Compcon Spring'93, pp. 63-72, 1993.

[Salisbury76] A. B. Salisbury, "Microprogrammable Computer Architectures", Elsevier, 1976.

[Sazeides96] Y. Sazeides and S. Vassiliadis, "The Performance Potential of Data Dependency Speculation & Collapsing", Proceedings of the 29th Annual International Symposium on Microarchitecture, pp. 238-247, 1996.

[Sazeides97] Y. Sazeides and J. E. Smith, "The Predictability of Data Values", Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 248-258, 1997.

[Seznec96] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-Block Ahead Branch Predictors", Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 116-127, 1996.

[Sites93] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary Translation", Communications of ACM, Vol. 36, pp. 69-81, February 1993.

[Skadron98] K. Skadron, P. S. Ahuja, M. Martonosi, and D. W. Clark, "Improving Prediction for Procedure Return with Return-Address-Stack Repair Mechanisms", Proceedings of the 31st Annual International Symposium on Microarchitecture, pp. 259-271, 1998.

[Smith_BJ81] B. J. Smith, "Architecture and Applications of the HEP Multiprocessor Computer System", Proceedings of the SPIE – Real Time Signal Processing IV, pp. 241-248, 1981.

[Smith_JE81] J. E. Smith, "A Study of Branch Prediction Strategies", Proceedings of the 8th Annual International Symposium on Computer Architecture, pp. 135-148, 1981.

[Smith_JE85] J. E. Smith and A. R. Pleszkun, "Implementation of Precise Interrupts in Pipelined Processors", Proceedings of the 12th Annual International Symposium on Computer Architecture, pp. 36-44, 1985.

[Smith_JE94] J. E. Smith and S. Weiss, "PowerPC601 and Alpha 21064: A Tale of Two RISCs," IEEE Computer, pp. 46-58, June 1994.

[Smith_JE97] J. E. Smith and S. Vajapeyam, "Trace Processors: Moving to Fourth-Generation Microarchitectures", IEEE Computer, pp. 68-74, September 1997.

[Smith_MD89] M. D. Smith, M. Johnson, and M. A. Horowitz, "Limits on Multiple Instructions Issue", Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 290-302, 1989.

[Sodani97] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse", Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 194-205, 1997.

[Sodani98] A. Sodani and G. S. Sohi, "Understanding the Differences Between Value Prediction and Instruction Reuse", Proceedings of the 31st Annual International Symposium on Microarchitecture, pp. 205-215, 1998.

[Sohi95] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar Processors", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 414-425, 1995.

[SPEC] SPEC, Standard Performance Evaluation Corporation, http://www.specbench.org.

[Su84] B. Su, S. Ding, and L. Jin, "An Improvement of Trace Scheduling for Global Microcode Compaction", Proceedings of the 17th Annual Workshop on Microprogramming, pp. 78-85, 1984.

[Sun87] Sun Microsystems, "The Sparc Architecture Manual – Version 7", Sun Microsystems Inc., 1987.

[Theobald92] K. B. Theobald, G. R. Gao, and L. J. Hendren, "On the Limits of Program Parallelism and its Smoothability", Proceedings of the 25th Annual International Symposium on Microarchitecture, pp. 10-19, 1992.

[Thistle88] M. R. Thistle and B. J. Smith, "A Processor Architecture for Horizon", Proceedings of Supercomputing' 88, pp. 35-41, November 1988.

[Thornton64] J. E. Thornton, "Parallel Operation in the Control Data 6600", Proceedings of the AFIPS Fall Joint Computer Conference, Vol. 26, part 2, pp. 33-40, 1964.

[Thornton70] J. E. Thornton, "Design of a Computer: The Control Data 6600", Scott, Foresman and Company, 1970.

[Tjaden70] G. S. Tjaden and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions", IEEE Transactions on Computers. Vol. C-19, No. 10, pp. 889-895, October 1970.

[Tokoro78] M. Tokoro, T. Takizuka, E. Tamura, and I. Yamaura, "A Technique of Global Optimization of Microprograms", Proceedings of the 11th Annual Workshop on Microprogramming, pp. 41-50, 1978.

[Tokoro81] M. Tokoro, E. Tamura, and T. Takizuka, "Optimization of Microprograms", IEEE Transactions on Computers, Vol. C-30, No. 7, pp. 491-504, July 1981.

[Tomasulo67] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units", IBM Journal of Research and Development, Vol. 11, No. 1, pp. 25-33, January 1967.

[Tremblay96] M. Tremblay and J. M. O'Connor, "UltraSparc I: A Four-Issue Processor Supporting Multimedia", IEEE Micro, pp. 42-50, April 1996.

[Tullsen95] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous Multithreading: Maximazing On-Chip Parallelism", Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp. 392-403, 1995.

[Uht97] A. K. Uht, V. Sindagi, and S. Somanathan, "Branch Effect Reduction Techniques", IEEE Computer, pp. 71-81, May 1997.

[Vajapeyam97] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences", Proceedings of the 24th Annual International Symposium on Computer Architecture, pp. 1-12, 1997.

[VanderWiel97] S. P. VanderWiel and D. J. Lilja, "When Caches Aren't Enough: Data Prefetching Techniques", IEEE Computer, pp. 23-30, July 1997.

[Veen86] A. H. Veen, "Dataflow Machine Architecture", ACM Computing Surveys, Vol. 18, No. 4, pp. 335-396, December 1986.

[Vijaykumar99] T. N. Vijaykumar and G. S. Sohi, "Task Selection for the Multiscalar Architecture", Journal of Parallel and Distributed Computing, No. 58, pp. 132-158, 1999.

[Wall91] D. W. Wall, "Limits of Instruction-Level Parallelism", Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 176-188, 1991.

[Wall93] D. W. Wall, "Limits of Instruction-Level Parallelism", Digital Western Research Laboratory – WRL Research Report 93/6, November 1993.

[Wall94] D. W. Wall, "Speculative Execution and Instruction-Level Parallelism", Digital Western Research Laboratory – WRL Technical Note TN-42, March 1994.

[Wang_CJ93] C.-J. Wang and F. Emnett, "Implementing Precise Interruptions in Pipelined RISC Processors", IEEE Micro, pp. 36-43, August 1993.

[Wang_K97] K. Wang and M. Franklin, "Highly Accurate Data Value Prediction using Hybrid Predictors", Proceedings of the 30th Annual International Symposium on Microarchitecture, pp. 281-290, 1997.

[Wolfe97] A. Wolfe, J. Fritts, S. Dutta, and E. S. T. Fernandes, "Datapath Design for a VLIW Video Signal Processor", Proceedings of the 3rd International Symposium on High-Performance Computer Architectures, pp 24-35, 1997.

[Yeh91] T.-Y. Yeh and Y. N. Patt, "Two-Level Adaptive Branch Prediction", Proceedings of the 24th Annual International Symposium on Microarchitecture, pp. 51-61, 1991.

[Yeh92] T.-Y. Yeh and Y. N. Patt, "Alternative Implementations of Two-Level Adaptive Branch Prediction", Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 124-134, 1992.

[Yeh93a] T.-Y. Yeh and Y. N. Patt, "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History", Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 257-266, 1993.

[Yeh93b] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache", Proceedings of the 7th International Conference on Supercomputing, pp. 67-76, 1993.