On the Effectiveness of the Scheduling Algorithm of the Dynamically Trace Scheduled VLIW Architecture

A. F. de Souza* and P. Rounce

Department of Computer Science
University College London
Gower Street, London WC1E 6BT - UK
a.souza@cs.ucl.ac.uk, p.rounce@cs.ucl.ac.uk

Abstract—

In a machine that follows the *dynamically trace scheduled VLIW* (DTSVLIW) architecture, VLIW instructions are built dynamically through a scheduling algorithm that can be implemented in hardware. These VLIW instructions are cached so that the machine can spend most of its time executing VLIW instructions without sacrificing any binary compatibility. This paper evaluates the effectiveness of the DTSVLIW instruction-scheduling algorithm by comparing it with the *first come first served* (FCFS) algorithm, used for microinstruction compaction, and the Greedy algorithm, used by the Dynamic Instruction Formatting architecture. In order to perform these comparisons, we have performed experiments using the SPECint95 benchmark suite.

Keywords—**VLIW, instruction scheduling, ILP**

I. INTRODUCTION

Recently, research on superscalars have incorporated into these architectures new features such as trace cache [ROT97], value prediction, and instruction reuse [SOD98], which allow them to exploit large amounts of *instruction-level parallelism* (ILP). These features, however, increase the implementation complexity, slowing down the clock of machines that use them. Moreover, simple superscalar machines with fast clocks have proved to be more powerful than their more complex counterparts [SMI94].

Very Long Instruction Word (VLIW) architectures [FIS84] are potentially the most simple and direct way of exploiting ILP, and have shown to perform better than superscalars using similar hardware [HAR96]. Different from superscalars, VLIW machines do not dynamically make any decision about multiple operation issue – the VLIW compiler is responsible to translate source code into long instructions – and thus their hardware is simple and fast. However, the assumptions built into the object code by the VLIW compiler about the VLIW hardware prevent object code compatibility between different implementations of the same VLIW *instruction set architecture* (ISA). VLIW processors with different levels of parallelism require recompilation of the source code. This problem is known as the *VLIW object code compatibility problem* and has limited the commercial interest in VLIW machines [RAU93].

To get over the VLIW object code compatibility problem, a *dynamically scheduled VLIW* (DSVLIW) was presented by Rau [RAU93]. However, this architecture cannot be used to implement an existent sequential ISA due to its VLIW ISA.

Ebcioglu and Altman [EBC97] with their DAISY machine can translate dynamically from the object code of a generic ISA to the object code of a VLIW using a Virtual Machine Monitor (VMM) implemented in software and running on a VLIW machine. The VMM operates on each page-fault, producing new pages of VLIW instructions from pages containing the existing ISA code. The DAISY machine concept relies on the ability of the VMM to translate code fast, and on the reusability of this code. However, since the VMM is implemented in software, the cost of the translation is necessarily high. In addition, because the VMM translates code on a page-fault basis, it does static scheduling only, which means that the VMM does not know much about the dynamic behaviour of branches, relaying on heuristics to determine their outcome. This can impose severe limitations on its performance.

A machine implementing the Dynamic Instruction Formatting (DIF) concept (Nair and Hopkins [NAI97]) performs code re-formatting in hardware. In a DIF machine, the original code is executed on a primary processor (a simple processor, less aggressive in exploiting parallelism) and, at the same time, re-formatted into blocks of VLIW instructions that are stored in a VLIW cache for subsequent execution on a VLIW engine. As with standard superscalar designs, code dependencies have to be handled, but this is only done when the code is reformatted, not each time it is fetched from the DIF's VLIW cache. This allows the extra speed of the VLIW engine to be fully utilised while allowing backward code compatibility.

* Sponsored by CAPES (Brazilian Government Agency)

A machine that follows the DIF concept schedules the trace observed during execution and as such records the dynamic branch behaviour related to this execution. This allows for more ILP than that achievable through techniques that perform static scheduling based on heuristics.

The architecture that forms the basis of this paper, the *dynamically trace scheduled VLIW architecture* (DTSVLIW) [DES98a], is a variant of the DIF architecture. Our earlier work [DES99a] demonstrates similar or better performance results to the DIF and superscalar, but a with simpler architecture that should be much easier to implement. The DTSVLIW schedules the instruction trace using an algorithm that can be implemented in hardware. To achieve performance, this algorithm has to be effective in producing VLIW code and has to be simple enough not to render the clock cycle time longer than that determined by the VLIW Engine design. In [DES99a] we have proved that the core operations of the DTSVLIW scheduling algorithm can be implemented with hardware as simple as an integer adder and, as such, should not impact the DTSVLIW clock cycle. However, the effectiveness of this algorithm has not yet been investigated. In this paper, the performance of the DTSVLIW scheduling algorithm is compared with that of two other algorithms: the First Come First Served (FCFS) algorithm, historically used for microcode compaction [DAV81]; and the Greedy algorithm, used in the DIF architecture. To perform these comparisons, we have modified our DTSVLIW execution-driven simulator to make it able to use the FCFS and Greedy algorithms, and have performed experiments using the SPECint95 benchmark suite.

This paper is organised as follows. Section II presents the DTSVLIW architecture and its scheduling algorithm, and Section III presents the DIF architecture and its scheduling algorithm. In Section IV, the FCFS algorithm is described and compared with the DTSVLIW and Greedy algorithms. Section V presents the experimental methodology, describes the experiments, and discusses the experimental results. Finally, in Section VI, our conclusions are presented together with future work proposals.

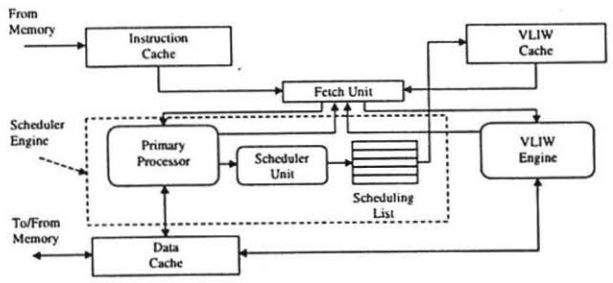

Fig. 1: The Dynamically Trace Scheduled VLIW Architecture.

II. The DTSVLIW Architecture and its Scheduling Algorithm

The DTSVLIW, in Figure 1, has two execution engines: the Scheduler Engine and the VLIW Engine; and two caches for instructions: the Instruction Cache and the VLIW Cache. The Scheduler Engine fetches instructions from the Instruction Cache and executes the original code for the first time using a simple pipelined processor, the Primary Processor. The instruction trace it produces is dynamically scheduled by the Scheduler Unit into VLIW instructions, which are saved as blocks of VLIW instructions in the VLIW Cache for the VLIW Engine to execute, if the same code needs to be re-executed. In a DTSVLIW machine, the Scheduler Engine provides for object-code compatibility, and the VLIW Engine provides VLIW performance and simplicity.

The Primary Processor executes Sparc-7 ISA [SUN87] code, while the VLIW Engine executes a sub-set. The VLIW Engine 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. The DTSVLIW implementation presented here uses the *checkpointing* exception handling mechanism proposed by Hwu and Patt [HWU87].

The key issues to be resolved in the DTSVLIW architecture are the scheduling of the instruction trace into *long instructions* (term used in the rest of this paper to refer to VLIW instructions) and the addressing within these long instructions. The Primary Processor and the VLIW Engine themselves are not a challenge. Multicyle instructions impact upon both the operation and performance of the architecture. Their scheduling requires special care to respect dependencies in any of their cycles. This can restrict the packing of instructions into long instructions limiting achievable parallelism. The DTSVLIW scheduling of multicycle instructions is described in [DES99b].

The Scheduler Engine of the DTSVLIW is composed of the Primary Processor plus the Scheduler Unit (Figure 1). When an instruction arrives in the execute pipeline stage of the Primary Processor, it is sent to the Scheduler Unit. The Scheduler Unit implements in hardware a simplified version of the *First Come First Served* (FCFS) algorithm. 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 [DAV81]. Finally, the simplified FCFS algorithm can be implemented in hardware in a pipelined fashion with a complexity comparable to that of an adder, as proved in [DES99a].

```
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		after 3 cycles
slt->	or r8, 8, r11			

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

slh ->	or r0, 0, r9	sethi hi(56), r8	or r0, 0, r10	after 9 cycles
	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		

slh ->	or r0, 0, r9	sethi hi(56), r8	or r0, 0, r10	after 11 cycles
	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	

(c)

Fig. 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 completion of the first instruction. The shaded instructions in each snapshot are also candidate instructions.

A. *The DTSVLIW Scheduling Algorithm*

The implemented version of the FCFS algorithm acts on a list, the *scheduling list*. This list has a fixed number of elements, each containing 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 completing execution by the Primary Processor is placed at the end of the scheduling list on the next clock cycle. On each subsequent cycle it 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 next element. Figure 2 shows an example of the algorithm scheduling a fragment of code that adds all elements of a vector. In Figure 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 nop instructions. The details of the algorithm's operation are as follows.

An instruction arriving in the execute pipeline stage of the Primary Processor in one cycle can be *inserted* into the scheduling list in the next, by *placing* a copy of it in a candidate instruction and also in a suitable slot of the corresponding long instruction. The copy in the long instruction slot is called the *companion instruction* and its position in the long instruction (the slot number) is recorded in the candidate instruction. If there are no data, control, or resource dependencies on any instruction in the list's tail element, the incoming instruction becomes a candidate instruction in the list's tail element; otherwise, the incoming instruction becomes a candidate instruction in a new tail element added to the list. In Figure 2b, instructions 1 and 2 are inserted by the first method, while instruction 3 is inserted by the second method due to a flow dependency on r8 (there is a flow dependency on instruction i if it reads from any position written by any instruction j before i).

After an instruction has been inserted into the list, the next step is to move this instruction and its companion up as far as they can go in the list of long instructions. An instruction can move up from long instruction i to long instruction i - 1 if it is not flow dependent 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 by invalidating the candidate instruction and leaving its companion in i. In Figure 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 (there is an instruction in i - 1 that writes in a storage position written by the candidate instruction in i), or an anti dependency on any instruction in i (there is an instruction in i that reads from a storage position written by the candidate instruction in i), or a control dependency on any instruction in i (there is an 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 anti/output dependency or all outputs if there is a control dependency, and by transforming the companion instruction into a *copy instruction* and leaving it permanently in the slot it occupies 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 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 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 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 is flushed to the VLIW Cache as a *block* and the incoming instruction is inserted into an empty list as the first instruction of a new block. 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 pointers to the head, tail, and current list element ready to be sent to the VLIW Cache. As instructions are inserted into the list at the maximum rate of one instruction per clock cycle, it is always possible to save the content of a list element before it is needed for an incoming instruction [DES99a].

B. *Addressing*

Instruction addressing has to change once instructions are scheduled into long instructions. A block of long instructions is stored as a VLIW Cache line. There is one address for the whole block, and this is the address of the first instruction scheduled in the block: the need to execute this instruction determines that the VLIW Engine can execute the block. For fetching long instructions from a block, the VLIW Engine maintains a line index that is incremented from zero. This is compared with a maximum value in the VLIW Cache line to determine the fetch of the last long instruction in a block, in which case the next fetch is made using the address of the instruction that follows the block, also stored in the cache line. This mechanism requires only two instruction addresses to be stored in a cache line. 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. 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.

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 functional units, register file's ports, and data cache's ports with the Primary Processor. The cost in cycles of swapping between them is equal to the sum of a number of pipeline stages of both processors only (the stages discarded in one processor plus the stages refilled in the other).

On a VLIW Cache miss, the Primary Processor takes over execution, fetching from the last PC value produced by the VLIW Engine. The Fetch Unit does not issue fetches to the VLIW Cache again until an instruction arrives at the execution 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.

III. The DIF Architecture and its Scheduling Algorithm

In contrast to the DTSVLIW, which uses the scheduling list for scheduling, 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 [NAI97]. 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.

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 [SUN87]. Although most Sparc processors have only 128 integer registers, a single exit map for such processor, with four instances of each register, would require 256 bits just for the integer registers.

The DIF architecture accesses its register file differently to the DTSVLIW. The DIF has to translate each register specifier to access its register file due to its renaming mechanism, while the DTSVLIW accesses its registers directly. Again different from the DTSVLIW, which fetches one long instruction per VLIW Cache access, the unit of communication between the DIF cache and its VLIW Engine is an entire block of long instructions. A more detailed discussion of the differences between DTSVLIW and DIF is presented in [DES99a].

IV. The FCFS Algorithm

The FCFS algorithm is a superset of the DTSVLIW and DIF algorithms and has historically been used for microcode compaction [DAV81]. Microcode compaction is

the process of combining *microoperations* (MOs) into *microinstructions* (MIs) in a way that reduces the space required by the microprogram and, hopefully, the time needed for the microprogram execution. DeWitt [DEW76] has shown that microcode compaction is a 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. [DAV81].

The description of the FCFS algorithm presented here is based on that of Davidson et al. The original algorithm operates over a list of MOs coming from a *straight-line microcode* segment (SLM), which is a sequence of MOs containing no branch MO except perhaps one at the end, and no entry point except at the beginning (a SLM is a basic block of MOs). The algorithm takes MOs from the SLM and groups them to form MIs with multiple MOs. Here, however, we use the FCFS algorithm to schedule instructions coming from a trace (produced dynamically during program execution) into long instructions. 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 are as follows.

1. Take one instruction from the trace and, if there is no dependency, add it to the last long instruction of the 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 BLOCK_SIZE, save the previous list's contents in the VLIW Cache 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 lack of a suitable slot in any long instructions above the one in the tail and the list is smaller than BLOCK_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.

The difference between the DIF and the FCFS algorithms is that DIF does not implement the step 3 of FCFS; i.e. the DIF 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. 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 achievable parallelism.

TABLE I
Fixed Parameters

Primary Processor	• four-stage (fetch, decode, execute, and write back) pipeline • no branch prediction hardware • taken branches cause a 2-cycle bubble in the pipeline
Decoded Instruction Size	6 bytes
Instruction Latency	1 cycle
N. of Renaming Registers	integer = f.p. = memory = flags = 256

TABLE II
Benchmark programs

Benchmark	*Input*
compress	400000 e 2231
gcc	-O3 jump.i
go	40 19 null.in
ijpeg	vigo.ppm -GO
m88ksim	dhry.big
perl	primes.pl
vortex	vortex.in
xlisp	queens 7

V. Methodology and Experimental Results

A simulator of the DTSVLIW has been implemented in C (23K lines of code), and execution-driven simulation performed to produce the results reported here. All results were produced with the simulator running in *test mode* in order to guarantee correct simulation. 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 program counter becomes equal to the DTSVLIW's. 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 for experimental evaluation because in this mode it is possible to measure the precise number of instructions necessary for the execution of a program, since the test machine can provide it. A DTSVLIW simulator alone cannot provide this number due to copy instructions and instructions executed speculatively. The *instructions per cycle* performance index used throughout this section has been produced dividing the number of instructions necessary to execute the program, as counted by the test machine, by the number of cycles consumed by DTSVLIW execution.

The simulator receives as input executables generated by the gcc compiler and faithfully models the DTSVLIW. Model parameters that are invariant for simulations are presented in Table I, and the benchmark programs used –

the SPECint95 benchmark suite – are shown in Table II. The Primary Processor fully implements the Sparc-7 ISA.

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 execute 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.)

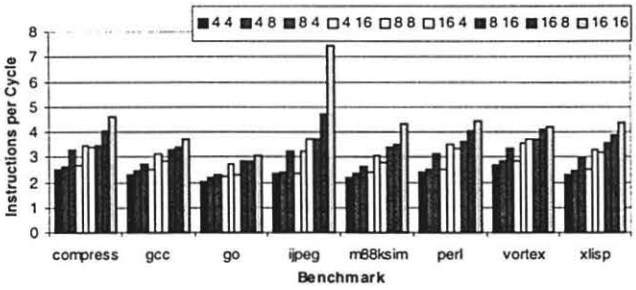

Fig. 3: Variation of parallelism with the block size and geometry

A. *Effect of the Block Size and Geometry on the Performance of the DTSVLIW*

Figure 3 shows the effect of the block size, in terms of the number of instructions, and of the block geometry – instructions per long instruction (width) versus long instructions per block (height) – on the DTSVLIW performance. The numbers in the legend are instructions per long instruction and long instructions per block, respectively. Since we are only interested in the performance of the algorithms, the experiments leading to the results in this figure and throughout this section were performed with perfect instruction and data caches (no miss penalty), large VLIW Cache (3072-Kbyte), and no next long instruction miss penalty. Adding these and other factors is likely to hide the difference in the effectiveness of the algorithms.

As the graph shows, the performance of DTSVLIW machines with the same block sizes and different geometry is significantly different. For example, the performance of a machine with 4x8 configuration is lower than the machine with 8x4 configuration 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, while a large number of long instructions in a block implies many renaming registers and more resource consuming hardware for recovering from exceptions [DES99a]. A large number of long instructions in a block can also have an impact on the required size of the VLIW Cache for the same performance [DES99a]. 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 and 16x4 geometry in the majority of the SPECint95 benchmarks. The DTSVLIW benefits from large block sizes but not linearly. As the graph in Figure 3 shows, a 16-fold increase in the number of instructions of a block (from 4x4 to 16x16) does not quite double its performance on average.

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 loop iteration can be scheduled into a single block, allowing instructions from these iterations to overlap, extracting much greater parallelism (in Figure 2, instruction 5 of the second loop iteration overlaps with instructions of the first).

The results presented in Figure 3 show better overall DTSVLIW performance than preliminary results published elsewhere [DES98b]. The improvements are due to more accurate implementation of the scheduling algorithm: previously we were very conservative with instructions dealing with different sizes of data (byte, word, and long word), which resulted in false data dependencies.

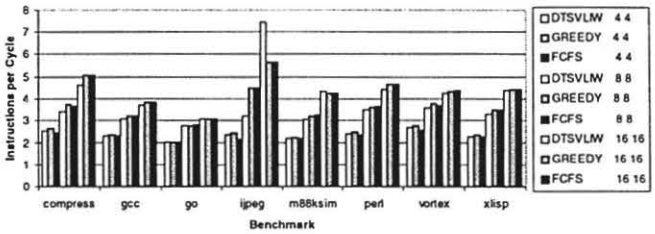

Fig. 4: Performance of the DTSVLIW, Greedy, and FCFS algorithms – untyped F.U.

B. *Performance of the Three Scheduling Algorithms – Untyped Functional Units*

In order to permit visualising the differences between the three scheduling algorithms, we have chosen to use blocks with three different geometries: 4x4, 8x8, and 16x16. As shown in Figure 4, all three scheduling algorithms perform very similarly. 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, reducing the code density in the block and limiting the achievable parallelism. Blocking in this fashion does not occur for the other algorithms. However, the DTSVLIW algorithm is expected to provide a much more feasible and faster implementation, and the results in Figure

4 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 also seen in the 16x16-m88ksim run. It would seem that some particular combination of instructions is particularly well suited to the simplified FCFS algorithm. The 16x16 performance on the ijpeg benchmark is superior to all others for all three algorithms, and exceptionally so for the DTSVLIW algorithm.

The full FCFS is as good 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 order to take full advantage of long instructions added to the top of the block, the FCFS algorithm needs to be unbounded in the block size. Larger geometries improve its relative performance, reflecting the movement towards an unbounded block. The difference between the Greedy and the full FCFS algorithm is just the extra step where, when resource constraints solely restrain the movement of an instruction to a higher position, a new long instruction is added at the top of the block for the instruction. In some cases, this extra long instruction 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, flushing the block to the cache with the first long instruction only partially filled, reducing the parallelism. The Greedy algorithm does not add the new long instruction at the front 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 block size and width reduces the resource blocking of instructions and also allows for more instructions to be added after resource blocking occurs. This gives more opportunity for the added front long instruction to be filled.

The simulations discussed until here were performed with untyped functional units, i.e. all functional units could execute all instructions; however, machines using typed functional units are a more likely scenario in an implementation. We discuss the performance of the three algorithms for machines with typed functional units next.

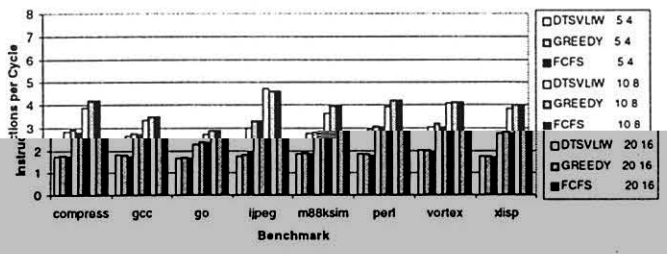

Fig. 5: Performance of the Three Algorithms – typed F.U.

TABLE III
Summary of the results – 4x4 & 5x4 machine configurations

	Average Performance (ipc)		Relative Performance 5x4 / 4x4	Algorithm / FCFS	
	4x4	5x4		4x4	5x4
DTSVLIW	2.35	1.80	77%	103%	100%
GREEDY	2.41	1.83	76%	106%	102%
FCFS	2.28	1.80	79%	100%	100%

TABLE IV
Summary of the results – 8x8 & 10x8 machine configurations

	Average Performance (ipc)		Relative Performance 10x8 / 8x8	Algorithm / FCFS	
	8x8	10x8		8x8	10x8
DTSVLIW	3.24	2.77	85%	92%	97%
GREEDY	3.55	2.91	82%	100%	102%
FCFS	3.53	2.85	81%	100%	100%

TABLE V
Summary of the results – 16x16 & 20x16 machine configurations

	Average Performance (ipc)		Relative Performance 20x16 / 16x16	Algorithm / FCFS	
	16x16	20x16		16x16	20x16
DTSVLIW	4.53	3.77	83%	103%	96%
GREEDY	4.40	3.94	89%	100%	100%
FCFS	4.41	3.93	89%	100%	100%

C. *Performance of the Three Scheduling Algorithms – Typed Functional Units*

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: 5x4 – with 2 integer, 1 load/store, 1 floating point, and 1 branch units, and a 4 long instructions height block; 10x8 – with twice the number of functional units and twice the number of long instructions of the previous configuration; and 20x16 – with four times the number of functional units and four times the number of long instructions of the 5x4 configuration. The results are presented in graph form in Figure 5. Table III, Table IV, and Table V summarise the data shown in Figure 4 and Figure 5.

As a visual comparison between the graphs in Figure 4 and Figure 5 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. As shown in Table III, the 5x4 configuration achieves from 77% to 79% of the 4x4 average performance for the three algorithms, while the 10x8 configuration achieves from 81% to 85% of the 8x8 performance, as shown in Table IV. The 20x16 configuration achieves from 83% to 89% of the 16x16 performance, as can be seen Table V. Larger typed configurations show smaller performance losses, which indicates that their number of functional units is reasonably balanced for the available instruction-level parallelism.

The relative performance of the three algorithms did not change much from machines with untyped to machines with typed functional units. In Table III, Table IV, and Table V, 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 92% to 106% percent for configurations with untyped functional units, and from 96% to 102% for typed configurations; i.e., 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.

VI. Conclusions and Future Work

The *dynamically trace scheduled VLIW* (DTSVLIW) architecture takes advantage of the instruction execution locality in current programs. In a DTSVLIW machine, code fragments are scheduled into long instructions and saved in a VLIW Cache upon their first execution. In subsequent executions, a VLIW Engine executes them in a VLIW fashion.

The design of the DTSVLIW architecture has been driven by the requirement to develop an architecture which can be effectively implemented to realise the fast clocking of VLIW machines: inherently faster than superscalar machines. The Primary Processor and the VLIW Engine of the DTSVLIW do not restrict the achievable clock rate. The key to an efficient and high clock rate implementation is the Scheduler Engine. The simplified version of the FCFS scheduling algorithm used by the DTSVLIW has a complexity that is readily implementable, and requires far fewer resources than the Greedy algorithm used by the DIF architecture as have been shown in our previous work [DES99a]. The results in this paper further demonstrate the effectiveness of the DTSVLIW algorithm in that there is no significant reduction in performance over the other candidate scheduling algorithms, even though these algorithms are expected to be much more difficult to implement.

The DTSVLIW architecture opens several new avenues of research. Next long instruction prediction, suitable compiling techniques, and new VLIW Cache organisations are examples of issues that we will investigate in future work.

References

[DAV81] S. Davidson, et al., "Some Experiments in Local Microcode Compaction for Horizontal Machines", IEEE Transactions on Computers, Vol. C30, No. 7, pp. 460-477, July 1981.

[DES98a] A. F. de Souza and P. Rounce, "Dynamically Trace Scheduled VLIW Architectures", Proceedings of The HPCN'98, in Lecture Notes on Computer Science, Vol. 1401, pp. 993-995, 1998.

[DES98b] 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.

[DES99a] A. F. de Souza and P. Rounce, "Dynamically Scheduling the Trace Produced During Program Execution into VLIW Instructions", Proceedings of The 2nd IPPS/SPDP Merged Symposium: 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing, pp. 248-257, 1999.

[DES99b] 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 HPCN'99, in Lecture Notes on Computer Science, Vol. 1593, pp. 1203-1206, 1999.

[DEW76] D. J. DeWitt, "A Machine Independent Approach to the Production of Optimal Horizontal Microcode", Technical Report 76 DT4, University of Michigan, Ann Arbor, August 1976.

[EBC97] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", Proceedings of The 24th International Symposium on Computer Architecture, pp. 26-37, 1997.

[FIS84] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", IEEE Computer, pp. 45-53, July 1984.

[HAR96] T. Hara, et al., "Performance Comparison of ILP Machines with Cycle Time Evaluation", Proceedings of The 23rd International Symposium on Computer Architecture, pp. 18-26, 1996.

[HWU87] W. W. Hwu, and Y. N. Patt, "Checkpoint Repair for Out-of-order Execution Machines", Proceedings of The 14th International Symposium on Computer Architecture, pp. 18-26, 1987.

[NAI97] R. Nair, M. E. Hopkins, "Exploiting Instructions Level Parallelism in Processors by Caching Scheduled Groups", Proceedings of The 24th International Symposium on Computer Architecture, pp. 13-25, 1997.

[RAU93] B. R. Rau, "Dynamically Scheduled VLIW Processors", Proceedings of The 26th International Symposium on Microarchitecture, pp. 80-92, 1993.

[ROT97] E. Rotenberg, et al., "Trace Processors", Proceedings of The 30th International Symposium on Microarchitecture, pp. 138-148, 1997.

[SMI94] J. E. Smith and S. Weiss, "PowerPC 601 and Alpha 21064: A Tale of Two RISCs," IEEE Computer, pp. 46-58, June 1994.

[SOD98] A. Sodani and G. Sohi, "Understanding the Difference Between Value Prediction and Instruction Reuse", Proceedings of The 31st International Symposium on Microarchitecture, pp. 205-215, 1998.

[SUN87] Sun Microsystems, "The Sparc Architecture Manual – Version 7", Sun Microsystems Inc., 1987.