

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

Alberto Ferreira de Souza  
Departamento de Informática  
Universidade Federal do Espírito Santo  
Av. Fernando Ferrari, S/N  
29060-970 – Vitória – ES – Brazil  
alberto@inf.ufes.br

Peter Rounce  
Department of Computer Science  
University College London  
Gower Street  
London WC1E 6BT - 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 an 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 (DIF) architecture. We also present comparisons between the DTSVLIW, pure VLIW, and the PowerPC620 processor. Our results show that the DTSVLIW scheduling algorithm has almost the same performance as the Greedy and FCFS. The results also show that the DTSVLIW performs better than the DIF for important machine configurations, better than pure VLIW implementations in most cases, and better than the PowerPC620 using equivalent hardware resources.*

## 1. Introduction

Recently, research on superscalars has incorporated into these architectures new features such as trace cache [1], value prediction, and instruction reuse [2], allowing the exploitation of more *instruction-level parallelism* (ILP). However, these features increase the implementation complexity, which can limit the processor clock rate. Moreover, simple superscalar machines with higher clock rates have proved to be more powerful than their more complex counterparts [3].

*Very Long Instruction Word* (VLIW) architectures [4] are potentially the most simple and direct way of exploiting ILP and have shown to perform better than superscalar using similar hardware [5]. In a classic VLIW architecture, ILP is achieved through a static code analysis that builds long instructions, each holding operations for all the machine's functional units. As the machine does not dynamically make any decisions about multiple operation issue, its hardware is simple and fast. 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 *instruction set architecture* (ISA). This problem is known as the *VLIW object code compatibility problem* and has limited the commercial interest in VLIW

machines [6].

A solution to the VLIW code compatibility problem is the Dynamic Instruction Formatting (DIF) concept proposed by Nair and Hopkins [7]. In a DIF machine, the original code is fetched and then dynamically formatted into blocks of VLIW instructions that are stored in a VLIW cache for subsequent execution in a VLIW engine. As with standard superscalar designs, code dependencies have to be handled, but this is only done when the code is formatted, 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. The architecture that forms the basis of this paper, the *dynamically trace scheduled VLIW architecture* (DTSVLIW) [8], follows the DIF concept. Our earlier work [9] demonstrates 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.

Figure 1 shows a block diagram of the DTSVLIW architecture. In a DTSVLIW machine, the Scheduler Engine fetches instructions from the Instruction Cache and executes them the first time using a simple pipelined processor – the Primary Processor. In addition, its Scheduler Unit dynamically schedules the trace produced during execution into VLIW instructions, placing them as blocks of VLIW instructions in the VLIW Cache. If the same code is executed again, it is fetched by the VLIW Engine from this cache and executed in a VLIW fashion. In a DTSVLIW machine, the Scheduler Engine provides object-code compatibility, and the VLIW Engine provides VLIW performance and simplicity.

Our main motivation for the development of the DTSVLIW came from the observation that even small instruction caches (16Kbyte or 4098 instructions) can achieve average hit rates higher than 99% with the SPEC92 and SPEC95 benchmark suites [10, 11]. This shows that there is strong temporal execution locality in programs. The DTSVLIW exploits temporal execution locality by scheduling the code into blocks of VLIW instructions on the first execution encounter and by executing it in the VLIW Engine on subsequent encounters.

To achieve performance, the DTSVLIW scheduling 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 [9] 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 time. However, the effectiveness of

the DTSVLIW scheduling 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 [12]; and the Greedy algorithm, used in the DIF. To perform the comparison, 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. We have then extended our simulator to fully implement a DIF machine and performed experiments to assess why the DTSVLIW performed better than the DIF in our earlier experiments [9]. We also have made comparisons between the DTSVLIW and the PowerPC620 and between the DTSVLIW and pure VLIW implementations. Our results show that the DTSVLIW scheduling algorithm, although simpler, has performance very similar to the FCFS and Greedy algorithms when operating with identical conditions. In addition, the results show that the DTSVLIW performs better than the DIF for important machine configurations because its register renaming mechanism is more cost-effective. Our experiments further show that the DTSVLIW performs better than pure VLIW implementations in most cases and that, using equivalent hardware resources, it performs better than the PowerPC620.

The paper is organised as follows. After this introduction, Section 2 presents the DTSVLIW architecture and its scheduling algorithm. Section 3 presents the DIF architecture and its scheduling algorithm. In Section 4, the FCFS algorithm is described and compared with the DTSVLIW and Greedy algorithms. Section 5 presents the experimental methodology, describes the experiments, and discusses the experimental results. In Section 6, the related work is discussed. Finally, in Section 7, our conclusions are presented together with future work proposals.

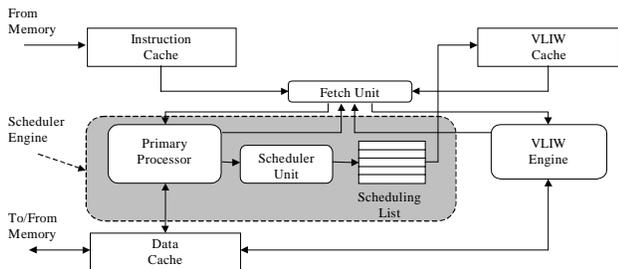


Figure 1: The Dynamically Trace Scheduled VLIW Architecture.

## 2. The DTSVLIW Architecture and its Scheduling Algorithm

In our DTSVLIW implementation, the Primary Processor executes Sparc-7 ISA [13] 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). A decode stage is not necessary as decoded instructions are saved in the VLIW Cache. The VLIW Cache is a simple set-associative cache, where a block of *long instructions* (the term used in the rest of this paper to refer to VLIW instructions) occupies a single cache line. Individual long instructions, however, are the unit of communication between the VLIW Cache and the rest of the DTSVLIW. For details of

how the DTSVLIW deals with exceptions, memory aliasing (disambiguation), and the execution of particular instructions (input/output, Sparc save and restore, etc) the interested reader may refer to [9].

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. Multicycle 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 [14].

The DTSVLIW’s 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 can produce optimum scheduling and consistently produces near-optimum scheduling [12]. 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 [9].

### 2.1 The DTSVLIW Scheduling Algorithm

The DTSVLIW scheduling 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; there is space for it in the next element; there is not a dependency with instructions in next element. Figure 2 shows an example of the algorithm scheduling a simple 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 (Figure 1) 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 in the list’s tail element; otherwise, it is inserted in a new tail element added to the list. In Figure 2, 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*).

An instruction inserted into the scheduling list in a clock cycle 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, 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 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  $i$ . In Figure 2, instruction 3 is installed in the fourth cycle, while instruction 8 is moved up in the ninth cycle.

```

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)

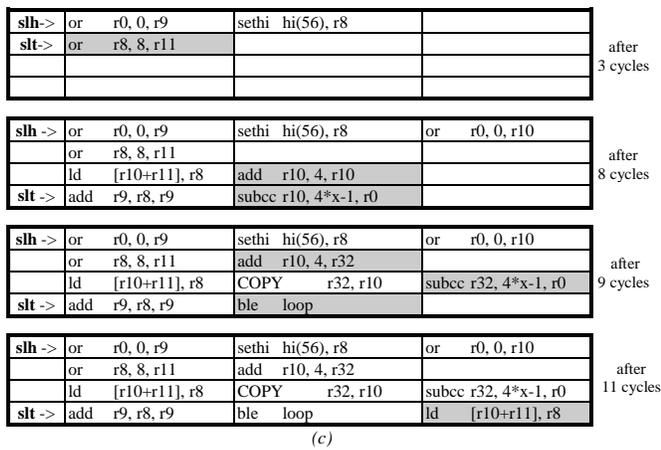


Figure 2: Scheduling algorithm running example. (a) C code fragment. (b) Assembly language version of the C code (c) Four snapshots of a three instructions wide and four long instructions deep scheduling list, filled with instructions coming from the Primary Processor after 3, 8, 9, and 11 cycles of the completion of the first instruction. The shaded instructions in each snapshot are also candidate instructions for promotion.

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 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 anti/output 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 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 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 [9].

## 2.2 Addressing

Instruction addressing has to change once instructions are scheduled into long instructions. There is one address for the whole block, and this is the address of the first instruction scheduled in the block. A block of long instructions is stored as a VLIW Cache line. 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 pipelining, and only a single bubble occurs when a branch is made out of a block and another block is hit in the VLIW Cache.

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 ports, and data cache ports with the Primary Processor. The cost in cycles of swapping between them is equal to the sum of the number of pipeline stages of both processors only. 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.

## 3. 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. This table has as many entries as

resources in the machine and records the earliest long instruction in which each resource is available [7]. 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. 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 only for integer registers.

The DIF architecture accesses its registers differently to the DTSVLIW. It has to translate each register specifier to access the register file during VLIW or sequential execution because of its renaming mechanism, while the DTSVLIW accesses its registers directly. The unit of communication between the DIF cache and its VLIW Engine is an entire block of long instructions, while the DTSVLIW accesses one long instruction per VLIW Cache access. A more detailed discussion of the differences between DTSVLIW and DIF is presented in [9].

#### 4. The FCFS Algorithm

The FCFS algorithm is a superset of the DTSVLIW and DIF algorithms and has historically been used for microcode compaction [12]. 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 [15] has shown that microcode compaction is a NP-complete problem, so a single-pass algorithm such as the FCFS is a cost-effective solution. Nevertheless, the FCFS algorithm can achieve optimum scheduling for execution time, as shown by Davidson et al. [12].

The FCFS algorithm was first proposed by Dasgupta and Tartar [16]; however, the description 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 at most one entry and one exit point (a SLM is a basic block of MOs). The algorithm takes MOs from the SLM in order and adds them to the end of an initially empty list of MIs. After a MO has been added, the next step is to move this MO up as far as it can go in the list. This groups them to form MIs with multiple MOs.

Here we use the FCFS algorithm to schedule instructions coming from a dynamic trace into long instructions, instead of MOs coming from a static SLM into MIs. 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 DIF algorithm differs from the FCFS algorithm in that it does not implement step 3 of FCFS; i.e., it 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, it moves instructions up the list one element at a time and there must be a slot available in the next element. This can cause premature installing of instructions that could be moved to a long instruction two or more entries up in the list, which can limit the code density and the achievable parallelism.

Table 1: Fixed Parameters

Primary Processor	<ul style="list-style-type: none"> <li>• four-stage (fetch, decode, execute, and write back) pipeline</li> <li>• no branch prediction hardware</li> <li>• taken branches cause a 2-cycle bubble in the pipeline</li> </ul>
Decoded Instruction Size	6 bytes

Table2: Benchmark programs

SPEC95 Benchmarks	Inputs	SPEC92 Benchmarks	Inputs
compress	400000 e 2231	compress	in
gcc	-O3 jump.i	eqntott	int_pri_3.eqn
go	40 19 null.in	espresso	cps.in
jpeg	vigo.ppm -GO	gcc	-O jump.i
m88ksim	dhry.big	xlisp	queens 7
perl	primes.pl		
vortex	vortex.in		
Xlisp	queens 7		

#### 5. 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 *instruction per cycle* performance measurement index used throughout this section 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 consumed by DTSVLIW execution.

The simulator receives as input executables generated by ordinary compilers that generate Sparc-7 ISA code and faithfully models the DTSVLIW. We have used the gcc 2.7.2 compiler with 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.

Model parameters that are invariant for simulations are presented in Table 1, and the SPEC benchmark programs used are shown in Table 2. The Primary Processor fully implements the Sparc-7 ISA. Except when stated otherwise, each program was run for 50 million or more instructions each experiment, as counted by the test machine.

### 5.1 Performance of the Three Scheduling Algorithms – Untyped Functional Units

In order to compare the three scheduling algorithms, we have chosen to use blocks with three different geometries (instructions per long instruction (width) versus long instructions per block (height)): 4x4, 8x8, and 16x16. The number of functional units in the VLIW Engine is equal to the number of instructions in a long instruction and functional units can execute any instruction (they are untyped). 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 with an unlimited number of renaming registers.

As shown in Figure 3, 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, which reduces the code density in the block and limits 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 3 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-*jpeg* run, but also seen in the 16x16-*m88ksim* run. Some particular combinations of instructions are particularly well suited to the simplified FCFS algorithm. The 16x16 performance on the *jpeg* benchmark is superior to all others for all three algorithms, and exceptionally so for the DTSVLIW algorithm.

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, flushing the block to the cache with the first line 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 block size and width 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.

The simulations described 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 a more likely scenario in an implementation. We discuss the performance of the three algorithms for machines with typed functional units next.

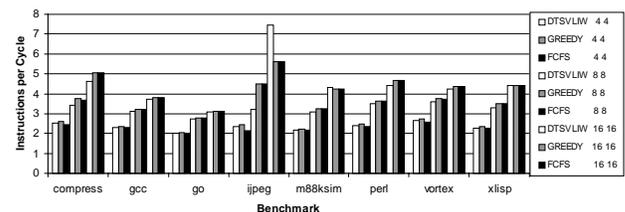


Figure 3: Performance of the DTSVLIW, Greedy, and FCFS algorithms – untyped F.U.

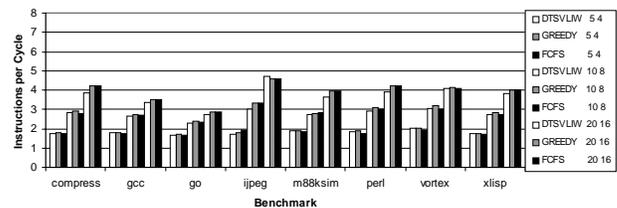


Figure 4: Performance of the DTSVLIW, Greedy, and FCFS algorithms – typed F.U.

### 5.2 Performance of the Three Scheduling Algorithms – Typed Functional Units

In order to evaluate the impact of typed functional units on 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 4. Table 3, Table 4, and Table 5 summarise the data shown in Figure 3 and Figure 4.

**Table 3: Summary of the results – 4x4 & 5x4 machine configurations**

	Average Performance (ipc)		Relative Performance		
	4x4	5x4	5x4 / 4x4	Algorithm / FCFS	
				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 4: Summary of the results – 8x8 & 10x8 machine configurations**

	Average Performance (ipc)		Relative Performance		
	8x8	10x8	10x8 / 8x8	Algorithm / FCFS	
				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 5: Summary of the results – 16x16 & 20x16 machine configurations**

	Average Performance (ipc)		Relative Performance		
	16x16	20x16	20x16 / 16x16	Algorithm / FCFS	
				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%

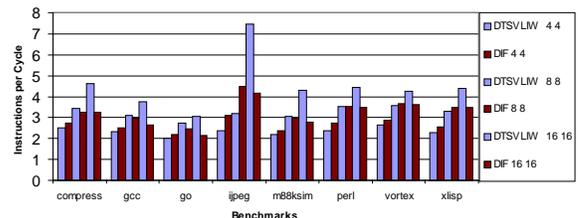
As a visual comparison between the graphs in Figure 3 and Figure 4 shows, the use of typed functional units causes significant performance loss, even though the typed configurations have 25% more instruction slots in each long instruction than the similar untyped ones. As shown in Table 3, Table 4, and Table 5, 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. The 20x16 configuration achieves from 83% to 89% of the 16x16 performance. Larger typed configurations show smaller performance losses, which indicates that their number of functional units is reasonably balanced for the available ILP.

The relative performance of the three algorithms did not change much from machines with untyped to machines with typed functional units. In Table 3, Table 4, and Table 5, 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.

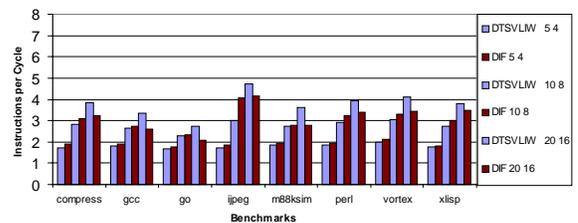
### 5.3 DTSVLIW versus DIF

The key 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 many 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’s scheduling algorithm in our simulator and have run experiments with DIF machine configurations and equivalent DTSVLIW configurations. We have used four instances for 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. The results are shown in Figure 5 and Figure 6.



**Figure 5: DTSVLIW versus DIF. Untyped functional units.**



**Figure 6: DTSVLIW versus DIF. Typed functional units.**

As the graph in Figure 5 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 heterogeneous functional units, the DTSVLIW advantage is smaller, as shown in Figure 6. 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 have more than a few instances per ISA register. The number register read and write ports grow with the number of functional units, so a

large multiported register file would render the machine clock rate too slow.

Figure 7 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 [7]. The parameters were: 2 branch units plus four homogeneous functional units; 2-way set-associative Instruction Cache with 128-byte lines and 16 lines per set (4-Kbyte), 2 cycle miss penalty; a direct-mapped Data Cache with 128 lines each of length 32 bytes (4-Kbyte), and a 2-cycle miss penalty; a two 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 [7]. 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 the graph in Figure 7 shows, the average performance of the two machines is similar: 2.4 instructions per cycle for the DTSVLIW and 2.2 for DIF, a difference of approximately 9% in favour of DTSVLIW. DIF performs better in compress and xisp, while DTSVLIW performs better in the remaining benchmarks.

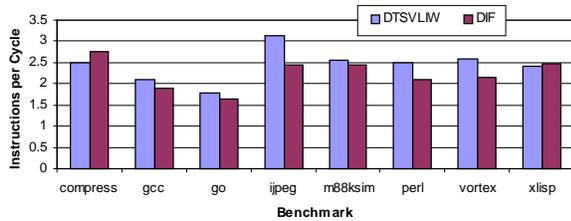


Figure 7: DTSVLIW versus DIF: more realistic models.

## 5.4 DTSVLIW versus VLIW

The VLIW research group at IBM is a leading group on VLIW compiler and architecture technologies. In [17] they have presented experimental performance results of various VLIW configurations using a powerful VLIW compiler. In Figure 8, we show the performance figures for programs from the SPECint92 and SPECint95 when running in the DTSVLIW and in the IBM's VLIW described in [17] and [18]. 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 8 shows, the VLIW outperforms the DTSVLIW in compress and eqntott for a large margin. However, for gcc, go, m88ksim, and xisp the DTSVLIW has consistent better performance for both machine configurations shown. These results show 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 scheduler algorithm has access to dynamic information not available to the VLIW compiler. We believe that, using the techniques employed in VLIW compilers, such as loop unrolling, software pipeline, and predication (if added to the DTSVLW ISA) the DTSVLIW would perform even better. Other compiler techniques could also be developed specifically to the DTSVLIW.

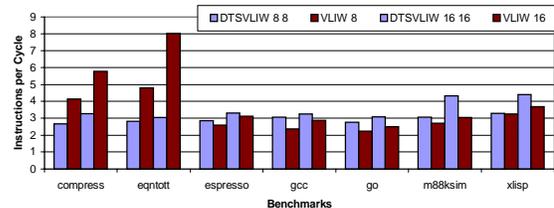


Figure 8: DTSVLIW versus VLIW.

## 5.5 DTSVLIW versus PowerPC620

The graph in Figure 9 shows a comparison between the performance of the PowerPC620, as described in [19], and the DTSVLIW. The PowerPC620 performance figures were taken from [19], and the parameters used there were: 4-instruction wide fetch, dispatch, complete, and writeback pipeline stages; single cycle integer, 2 cycles load, and 3 cycles 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 each functional unit of each kind, respectively; 32Kbyte, 8-way set associative instruction and data L1 caches, with an 8-cycle miss penalty (a perfect unified L2 cache was assumed); and a branch predictor with a 256-entry two-way branch target buffer (BTB) and a 2048-entry (two-bit counters) direct mapped branch history table (BHT).

The DTSVLIW was configured with a 4x8 block and functional units of the same type, with same latency, and in the same number of the PowerPC620. Although six

functional units are available in the VLIW Engine with this configuration, we have used 4-instruction wide long instructions in the DTSVLIW to give 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 dispatch them to the appropriated functional units. A branch predictor with the same characteristics of the PowerPC620's was used to try to reduce the extra cost added to next long instruction fetch misses by this dispatch stage. An 8-Kbyte, 8-way set associative instruction cache, and a 24-Kbyte, 8-way set associative VLIW Cache were used. These sizes were chosen to make this pair equivalent to the instruction cache of the PowerPC620. The DTSVLIW data cache was configured with the same characteristics of the PowerPC620's. In this simulation, the DTSVLIW was allowed to execute the same number of instructions executed in the PowerPC620 simulation [19].

As Figure 9 shows, the performance of the two machines is comparable, although the DTSVLIW performance is better overall. This is so because the DTSVLIW instruction window is larger than the PowerPC620, which allows more opportunities to find ILP. Note that a DTSVLIW implementation is likely to have significantly higher clock rate.

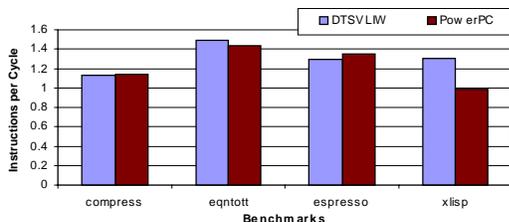


Figure 9: DTSVLIW versus PowerPC620.

We have used our DTSVLIW simulator to produce the experimental results for most of the experiments described in this paper. The only exceptions are experimental results for the Nair and Hopkins's DIF [7], in Figure 7, for the IBM's VLIW [17, 18], in Figure 8, and for the Diep's PowerPC620 [19], in Figure 9, 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 comparisons made must be seen with caution, because the results we have collected from the literature have been produced with different simulators based in the PowerPC ISA, running the benchmarks with possibly different inputs, and compiled with different compilers with possibly different compiler flags. To isolate the effect of each one of these factors (and possibly others) 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 5.3.) However, the results we would obtain are unlikely to be better than those we have collected from the literature.

## 6. Related Work

### 6.1 Previous Research on VLIW Architectures

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. 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 mechanism is the hardware equivalent of *superblock scheduling* [20], a compiler technique derived from trace scheduling [21]. However, in superblock scheduling 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 sets.

### 6.2 Tackling the VLIW Object Code Compatibility Problem

There are software and hardware approaches to tackle the VLIW object code compatibility problem. The simplest software technique is off-line recompilation of the programs source code. 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 installation of a complete set of new binaries. Binary translation [22] is a variant of this technique that can be performed without the source code, but machine upgrades still require either translation or reinstallation of 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 [23].

*Dynamic Rescheduling*, proposed by Conte and Sathaye [24], is another software technique that can be used to overcome the VLIW object code compatibility problem. When a non-compatible program is invoked in a VLIW system that implements dynamic rescheduling, the operating system translates the first page to a new page that is binary compatible with the system hardware. This process is repeated each time a new page fault occurs, and provides correct execution over different VLIW machine generations. Ebcioğlu and Altman [25], with their DAISY machine, extended dynamic rescheduling to *dynamic compilation*, in order to use a generic ISA. These techniques rely on the ability of the operating 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, because the translation is done on a page-fault basis, the operating system may not know much about the dynamic behaviour of branches in pages being translated, relying on heuristics to determine their outcome. Essentially, static scheduling is performed, as done by VLIW compilers, but in a much shorter time.

Rau [6] presented a new type of VLIW machine, named *dynamically scheduled VLIW* (DSVLIW), which tackles the software compatibility problem at the hardware level. However, despite its ability to implement a family of VLIW machines with different functional units' latency and the same ISA, the DSVLIW concept 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 [26] proposed the use of a fill unit [27] 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 this long instruction into a *shadow cache*. At the same time, the fill unit sends the long instruction to the functional units for execution. Fetch accesses hitting the shadow cache provide long instructions directly to the functional units. This design cannot exploit ILP extensively, as the proposed fill unit does not rename registers and works within a window of only one long instruction.

Banerjia and his colleagues presented a processor architecture similar to the Franklin and Smotherman proposal, called *Miss Path Scheduling* (MPS) architecture [28]. 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 [28]. Once in the cache, long instructions are fetched and issued to a VLIW core. MPS has three 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' DIF [7] is another improvement of the Franklin and Smotherman proposal that schedules blocks of long instructions as opposed to a single long instruction and is able to perform register renaming. The DTSVLIW is similar to the DIF, but it has a significantly different implementation (see Section 3). In addition, the DTSVLIW performs better than DIF for important machine configurations, as shown in Subsection 5.3.

### 6.3 Other Approaches for Exploiting ILP

A machine that follows the *trace cache* architecture [1] fetches instructions from the instruction cache and attempts to schedule and execute them across multiple functional units. During this process, the instructions are saved into the *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, which increases instruction fetch bandwidth and throughput.

Similar to the superscalar architecture, the trace cache architecture has instruction-scheduling overheads that lengthen the clock cycle time. Logic fan-out and wire delays are perhaps the most important of these scheduling overheads [5]. The fan-out overhead is caused by the logic that forwards the functional units results to the reservation stations, whereas the wire delay overhead is caused by the long wires necessary

to connect the functional units to the various reservation stations. In the near future, wire delays are likely to dominate the clock cycle time of superscalar-like machines [29]. VLIW machines, and likewise DTSVLIW machines, do not use hardware mechanisms equivalent to reservation stations and thus can have a faster clock rate than superscalars [5].

Value prediction has been proposed by Lipasti and Chen for predicting the future value of registers [30]. A machine employing value prediction uses a special hardware table that records history about register contents and uses this history for predicting their future values. In contrast, the instruction reuse technique uses a hardware table to store histories of instructions and their input values [31]. Experiments have shown that many instructions and groups of instructions having the same inputs are executed repeatedly [31]. Therefore, their results can be provided by this hardware table avoiding subsequent executions and improving the overall processor performance. The key difference between the value prediction and the instruction reuse techniques is the way they verify the validity of the values read from the tables. In the former, these values are used speculatively and validated later, while in the latter the values are validated before being used [2]. With either of these, a superscalar machine can exploit more ILP and, in this sense, these techniques are improvements on the superscalar architecture.

Superscalar machines using trace cache, value prediction, or instruction reuse are effective ways of exploiting ILP. However, superscalar machines employing these techniques are also complex devices and the impact of such complexity on the design cost and clock cycle time can be severe. We believe the DTSVLIW architecture is a simpler, more cost-effective alternative for general-purpose machines.

## 7. Conclusions and Future Work

The *dynamically trace scheduled VLIW* (DTSVLIW) architecture takes advantage of the repetitive and localised pattern of instruction fetch addresses in current programs. In a DTSVLIW machine, a code fragment is scheduled into long instructions and saved in a VLIW Cache upon its first execution. In subsequent executions, a VLIW Engine executes it in a VLIW fashion.

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 clock of VLIW designs: inherently faster than superscalar designs. The Primary Processor and the VLIW Engine of the DTSVLIW do not restrict the achievable clock rate. It is the Scheduler Engine that is the key to an efficient and high clock rate implementation. 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. 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: even more so if the highest rate clocking of the VLIW Engine is to be achieved.

If compared with a pure VLIW machine, the DTSVLIW should have approximately the same clock frequency and a slightly more complex design. However, the DTSVLIW offers backward code compatibility, the possibility of

implementing existing ISAs, and, according to our results, better performance in most cases. If compared with superscalars, the DTSVLIW offers simpler implementation, therefore faster clock rate and overall better performance.

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

## 8. References

- [1] E. Rotenberg, et al., "Trace Processors", Proc. of 30th Int. Symp. on Microarchitecture, pp. 138-148, 1997.
- [2] A. Sodani and G. Sohi, "Understanding the Difference Between Value Prediction and Instruction Reuse", Proc. of 31st Int. Symp. on Microarchitecture, pp. 205-215, 1998.
- [3] J. E. Smith and S. Weiss, "PowerPC 601 and Alpha 21064: A Tale of Two RISCs," IEEE Computer, pp. 46-58, June 1994.
- [4] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", IEEE Computer, pp. 45-53, July 1984.
- [5] T. Hara et al., "Performance Comparison of ILP Machines with Cycle Time Evaluation", Proc. of 23rd Int. Symp. on Computer Architecture, pp. 18-26, 1996.
- [6] B. R. Rau, "Dynamically Scheduled VLIW Processors", Proc. of 26th Int. Symp. on Microarchitecture, pp. 80-92, 1993.
- [7] R. Nair, M. E. Hopkins, "Exploiting Instructions Level Parallelism in Processors by Caching Scheduled Groups", Proc. of 24th Int. Symp. on Computer Architecture, pp. 13-25, 1997.
- [8] A. F. de Souza and P. Rounce, "Dynamically Trace Scheduled VLIW Architectures", Proc. of HPCN'98, in Lecture Notes on Computer Science, Vol. 1401, pp. 993-995, 1998.
- [9] A. F. de Souza and P. Rounce, "Dynamically Scheduling the Trace Produced During Program Execution into VLIW Instructions", Proc. of 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing - IPPS/SPDP'99, pp. 248-257, 1999.
- [10] 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.
- [11] M. J. Charney and T. R. Puzak, "Prefetching and Memory System Behaviour of the SPEC95 Benchmark Suite", IBM J. of Res. and Dev., Vol. 41, No. 3, May 1997.
- [12] S. Davidson, et al., "Some Experiments in Local Microcode Compaction for Horizontal Machines", IEEE Trans. on Computers, Vol. C-30, No. 7, pp. 460-477, 1981.
- [13] Sun Microsystems, "The Sparc Architecture Manual - Version 7", Sun Microsystems Inc., 1987.
- [14] A. F. de Souza and P. Rounce, "Effect of Multicycle Instructions on the Integer Performance of the Dynamically Trace Scheduled VLIW Architecture", Proc. of HPCN'99, in Lecture Notes on Computer Science, Vol. 1593, pp. 1203-1206, 1999.
- [15] D. J. DeWitt, "A Machine Independent Approach to the Production of Optimal Horizontal Microcode", Tech. Rep. 76 DT4, University of Michigan, Ann Arbor, August 1976.
- [16] S. Dasgupta and J. Tartar, "The Identification of Maximal Parallelism in Straight Line Microprograms", IEEE Trans. on Computers, Vol. C-25, pp. 986-991, 1976.
- [17] J. H. Moreno et al., "Simulation/Evaluation Environment for a VLIW Processor Architecture", IBM J. of Res. and Development, Vol. 41, No. 3, May 1997.
- [18] M. Moudgill et al., "Compiler/Architecture Interaction in a Tree-Based VLIW Processor", IBM Research Report RC20694, November 1996.
- [19] T. A. Diep, C. Nelson, and J. P. Shen, "Performance Evaluation of the PowerPC620 Microarchitecture", Proc. of the 22nd Int. Symp. on Computer Architecture, pp. 163-174, 1995.
- [20] W. W. Hwu et al., "The Superblock: An Effective Technique for VLIW and Superscalar Compilation", The Journal of Supercomputing, 7, pp. 229-248, 1993.
- [21] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", IEEE Trans. on Computers, Vol. C-30, No. 7, pp. 478-490, July 1981.
- [22] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, "Binary Translation", Communications of the ACM, Vol. 36, pp. 69-81, February 1993.
- [23] J. Turley, "Alpha Runs x86 Code with fx!32", Microprocessor Report, Vol. 10, March 1996.
- [24] T. M. Conte and S. W. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures", Proc. of the 28th Int. Symp. on Microarchitecture, pp. 208-218, 1995.
- [25] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", Proc. of the 24th Int. Symp. on Computer Architecture, pp. 26-37, 1997.
- [26] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue", Proc. of the 27th Int. Symp. on Microarchitecture, pp. 162-171, 1994.
- [27] S. Melvin, M. Shebanow, and Y. Patt, "Hardware Support for Large Atomic Units in Dynamic Scheduled Machines", Proc. of the 21st Int. Symp. on Microarchitecture, pp. 60-66, 1988.
- [28] S. Banerjia, S. W. Sathaye, K. N. Menezes, and T. Conte, "MPS: Miss-Path Scheduling for Multiple-Issue Processors", IEEE Trans. on Computers, Vol. 47, No. 12, pp. 1382-1397, December 1998.
- [29] D. Matzke, "Will Physical Scalability Sabotage Performance Gains?", IEEE Computer, September 1997.
- [30] M. H. Lipasti and J. P. Shen, "Exceeding the Data-Flow Limit Via Value Prediction", Proc. of the 29th Int. Symp. on Microarchitecture, pp. 226-237, 1996.
- [31] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse", Proc. of the 24th Int. Symp. on Computer Architecture, pp. 194-205, 1997.