Journal of Parallel and Distributed Computing **60**, 1480–1511 (2000)
doi:10.1006/jpdc.2000.1661, available online at http://www.idealibrary.com on IDEAL®

Dynamically Scheduling VLIW Instructions

A. F. de Souza

Departamento de Informática, Universidade Federal do Espírito Santo, Av. Fernando Ferrari, S/N, 29060-970, Vitória, ES, Brazil
E-mail: alberto@inf.ufes.br

and

P. Rounce

Department of Computer Science, University College London, Gower Street, London WC1E 6BT, United Kingdom
E-mail: p.rounce@cs.ucl.ac.uk

Received September 15, 1999; revised May 17, 2000; accepted May 17, 2000

Very long instruction word (VLIW) machines potentially provide the most direct way to exploit instruction-level parallelism; however, they cannot be used to emulate current general-purpose instruction set architectures. In addition, programs scheduled for a particular implementation of a VLIW model cannot be guaranteed to be binary compatible with other implementations of the same machine model with a different number of functional units or functional units with different latencies. This paper describes an architecture, named *dynamically trace scheduled VLIW* (DTSVLIW), that can be used to implement machines that execute code of current RISC or CISC instruction set architectures in a VLIW fashion, with backward code compatibility. Preliminary measurements of the DTSVLIW performance, obtained with an execution-driven simulator running the SPECint95 benchmark suite, are also presented. © 2000 Academic Press

Key Words: VLIW; DTSVLIW; dynamic instruction scheduling; ILP.

1. INTRODUCTION

Very long instruction word (VLIW) machines can execute several scalar operations in a single clock cycle [12]. They have long instructions (hundreds to thousands of bits), with fields to control each of their many functional units. These long instructions are fetched from memory, one per processor clock cycle, and issued to functional units that operate in parallel. In VLIW systems, the compiler has complete responsibility for creating a package of operations that can be simultaneously issued. VLIW processors do not dynamically make any decisions about multiple operation issue, and thus they are simple and fast. However, the assumptions built into the code by the compiler about the VLIW hardware prevent code compatibility between different implementations of the same VLIW *instruction set architecture* (ISA). VLIW processors with different levels of hardware parallelism require recompilation of the source code. For instance, the code generated for a VLIW processor with four operations per VLIW instruction could not run in another VLIW processor with three operations per VLIW instruction without recompilation. This problem, known as *the VLIW object–code compatibility problem*, has limited the commercial interest in VLIW architectures [29].

Recently, a new architectural concept named *dynamic instruction formatting* (DIF) has been proposed [26]. A machine implementing this concept can overcome the VLIW object–code compatibility problem by executing the program in two distinct phases: one sequential and the other parallel. During the sequential execution phase, the code is formatted as VLIW instructions. In the parallel phase, a VLIW engine executes the formatted code. In this paper, we present an architecture that follows the DIF concept. This architecture, named dynamically trace scheduled very long instruction word (DTSVLIW) architecture [5], has been conceived independent of DIF, which has permitted an implementation significantly different from that suggested by the proponents of DIF. We have shown that the DTSVLIW is easier to implement than DIF and delivers equivalent performance with fewer hardware resources [6, 8, 9].

Figure 1 shows a block diagram of the DTSVLIW architecture. In the DTSVLIW architecture, the scheduler engine fetches instructions from the instruction cache and executes them first using a simple pipelined processor—the primary processor. In addition, its scheduler unit dynamically schedules the trace produced during this execution into VLIW instructions, placing them as blocks of VLIW instructions in the VLIW cache. If the same code is executed again, it is then fetched by the VLIW engine from this cache and executed in a VLIW fashion. In the DTSVLIW architecture, the scheduler engine provides object–code compatibility, and the VLIW engine provides VLIW performance and simplicity.

Executing code in two distinct modes, one sequential and one parallel, results in four positive characteristics. First, code compatibility between different machine

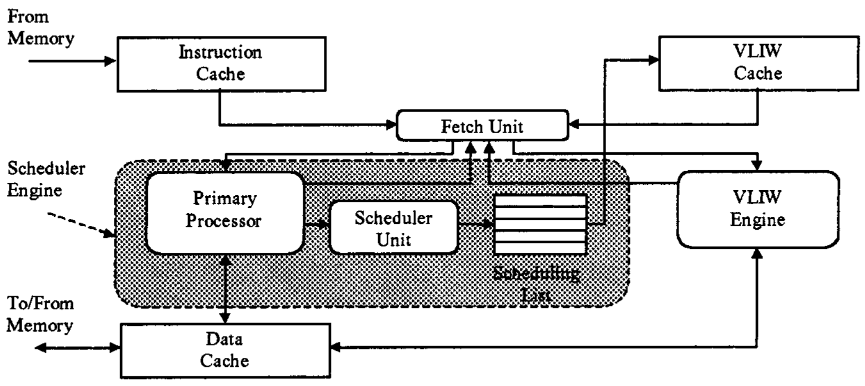

FIG. 1. The dynamically trace scheduled VLIW (DTSVLIW) architecture.

generations is facilitated. Second, complex instructions can be dealt with in sequential mode, with complex instructions being decomposed into several simpler operations which can later be executed in parallel mode. Third, the task of finding parallelism is simplified, as the scheduler unit receives no more than one instruction (or operation) per cycle and, therefore, can have a simple and fast hardware implementation. Fourth, instruction exceptions can be dealt with in sequential mode: in the case of an instruction exception during parallel execution, a DTSVLIW machine can switch to sequential mode to deal with the exception. However, in order to take advantage of these characteristics, a DTSVLIW machine has to reuse the blocks of VLIW instructions saved in the VLIW cache many times.

1.1. Research Motivation

The main motivation for the research described here came from the observation that even small instruction caches (16 Kbyte or 4098 instructions) can achieve average hit rates higher than 99% with the SPEC92 and SPEC95 benchmark suites [2, 14, 34]. This shows that there is strong temporal execution locality in programs. The DTSVLIW exploits temporal execution locality by converting the code into blocks of VLIW instructions in the first execution encounter and by executing it in the VLIW engine in subsequent encounters. To take advantage of the DTSVLIW architecture characteristics, the converting algorithm has to be effective in producing VLIW code. In addition, it has to be simple enough not to render the clock cycle time longer than that determined by the VLIW engine design. The results achieved with this research so far support the view that this can be achieved [6, 9].

1.2. Paper Overview

In this paper, after reviewing related work, we describe the DTSVLIW in detail and show that it can be implemented without taking the clock cycle time far from that determined by a pure VLIW design. To better understand the DTSVLIW architecture, we investigate the effect of various architectural parameters on its performance via experiments. We then compare the DTSVLIW performance with that of the DIF architecture.

2. RELATED WORK

2.1. Tackling the VLIW Object–Code Compatibility Problem

Existing techniques to get over the VLIW object–code compatibility problem can be divided into software and hardware techniques. The simplest software technique to overcome the VLIW object-code compatibility problem 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 [31] 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 [18].

Dynamic rescheduling, proposed by Conte and Sathaye [3], is another software technique that can be used to overcome the VLIW object–code compatibility problem. When a noncompatible 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. Ebcioglu and Altman [10], 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 behavior 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. This imposes limitations on the scheduling quality, impacting on system performance.

Rau [29] presented a new type of VLIW machine, named *dynamically scheduled VLIW* (DSVLIW), which tackles the VLIW object–code compatibility problem at the hardware level. 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 [13] proposed the use of a fill unit [25] to compact a dynamic stream of scalar instructions. Their fill unit accepts decoded instructions from the machine decoder, compacts them into a *long instruction* (the term used in the rest of this paper to refer to VLIW instructions), and saves this 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 *instruction-level parallelism* (ILP) extensively, as the proposed fill unit does not rename registers and works within a window of only one long instruction. Similar to this design is that proposed by Banerjia and his colleges, the *miss path scheduling* (MPS) architecture [1]. The main differences between the two proposals is that MPS schedules blocks of long instructions as opposed to a single long instruction and that the scheduling hardware is placed between the instruction cache and the next level of memory. In MPS, instruction scheduling is performed at instruction cache misses and the blocks of long instructions formed are saved in a special instruction cache. 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 likely to have not been executed before. Therefore, dynamic branch behavior information is not likely to be available at scheduling time.

Nair and Hopkins [26] suggested a VLIW-based machine organization named DIF (dynamic instruction formatting), which also follows the Franklin and Smotherman proposal [13]. The DIF machine incorporates two engines: the VLIW engine and the primary engine. The latter is a simple processor, less aggressive in exploiting parallelism, which executes instructions of a generic ISA when first fetched. Simultaneously with the execution of a code sequence, this engine reformats (schedules) the code, generating groups of long instructions as opposed to a single long instruction. These groups, which can encompass many basic blocks, are saved in a special cache—the DIF cache. Following accesses to the same sequence will hit the DIF cache, and the long instructions fetched will be executed by the VLIW engine.

In a DIF machine, the instructions are executed during scheduling; therefore, useful execution occurs during scheduling time. Register renaming is performed and the dynamic branch behavior is recorded into the scheduled blocks. This allows for more parallelism than the MPS and all previously mentioned proposals. The DTSVLIW is similar to the DIF, but it has a significantly different implementation. The DTSVLIW differs from the DIF in its register renaming mechanism, VLIW engine register access, communication between the VLIW cache and the VLIW engine, and scheduling algorithm. The DTSVLIW architecture was first presented in [5]. A detailed description of the DTSVLIW is presented in Section 3 and the differences between the DTSVLIW and the DIF are presented in Section 3.8.

2.2. Other Approaches for Exploiting ILP

A machine that follows the *trace cache* architecture [30] fetches instructions from the instruction cache and attempts to schedule and execute them across multiple functional units using, for example, Tomasulo's algorithm [36]. 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, however, 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 [17]. 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 [24]. VLIW machines, and likewise DTSVLIW machines, do not need hardware mechanisms equivalent to reservation stations in their main data path and do not suffer from their characteristic forwarding logic and forwarding wire delay overheads. Forwarding logic and forwarding wires are of course necessary in VLIW and DTSVLIW machines. However, they only connect functional units' outputs to functional units' inputs and not to several reservation stations at the input of each functional unit. Therefore, they can have a faster clock than superscalar-like machines even considering wire delays [17].

Superscalar machines using trace cache, value prediction [23], or instruction reuse [33] 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.

There are, off course, other approaches for exploiting ILP [16, 22, 37], but they either require rewriting the source code or do not allow current sequential ISAs to be employed without modification.

3. THE DTSVLIW ARCHITECTURE

The symbolic diagram of a DTSVLIW machine is shown in Fig. 2. It has two caches for instructions and two processing engines. The instruction cache stores fragments of the original compiled code while the VLIW cache stores *blocks* of long instructions. The original code is executed first by the primary processor. The code trace produced during this execution is scheduled by the scheduler unit into blocks of long instructions that are saved in the VLIW cache. The VLIW engine executes these long instructions if an already scheduled code fragment has to be executed again.

In a DTSVLIW machine, the VLIW engine and the primary processor never operate at the same time and no machine state has to be transferred between them, as they share the DTSVLIW machine state. This simplifies the design of both, even allowing the VLIW engine to share its functional units with the primary processor. The cost in cycles of swapping the execution between the VLIW engine and the primary processor is equal to the sum of a number of pipeline stages of each one only (the pipeline stages discarded in one plus the pipeline stages refilled in the other).

While the primary processor is executing the code, the fetch unit (Fig. 2) issues different addresses to the instruction cache and the VLIW cache. To the instruction cache is issued the *program counter* (PC) content. To the VLIW cache is issued the address of the instruction in the execute stage of the primary processor (dashed arrow in Fig. 2). If this instruction has been executed before, there may be a block with its address in the VLIW cache. On a VLIW cache hit, the VLIW engine takes over execution. The block being constructed by the scheduler unit is flushed to the VLIW cache—this block is made to point at the hit block. The contents of all but the write back pipeline stage of the primary processor are annulled and the PC receives the memory address that hit the VLIW cache. In subsequent cycles, the VLIW engine controls the PC.

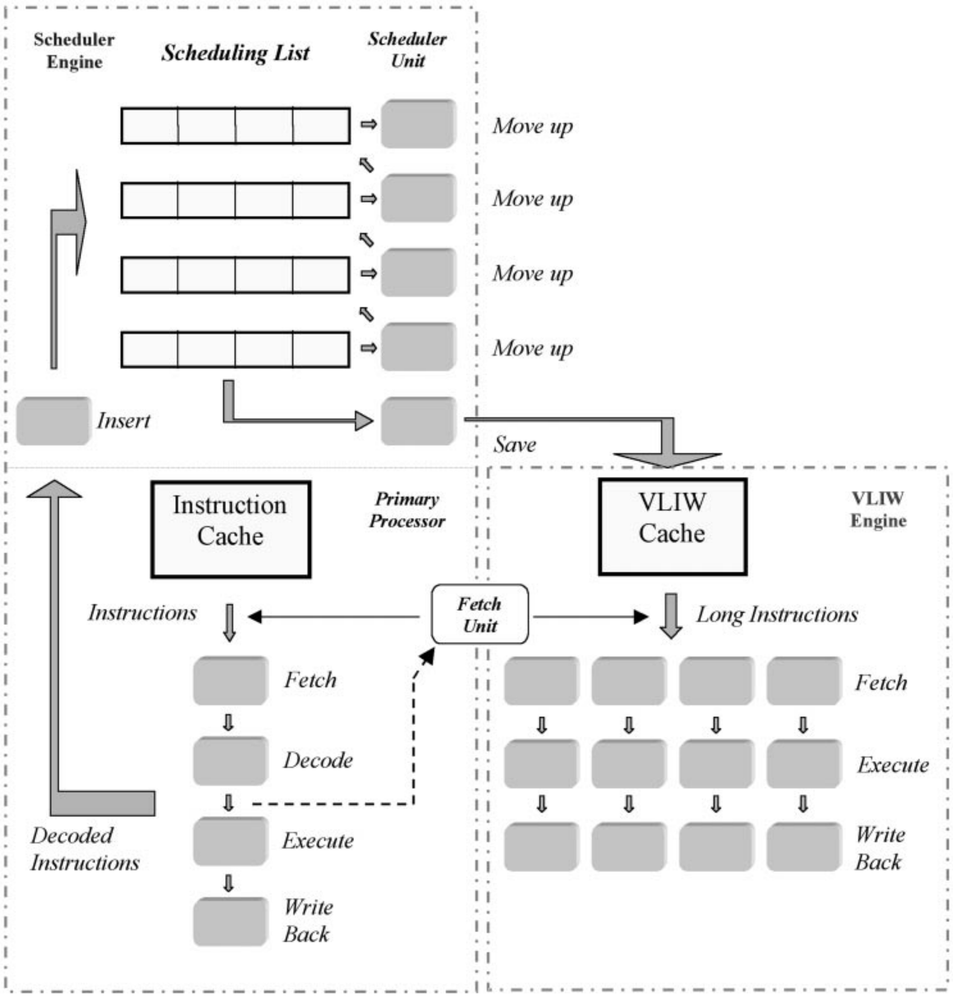

FIG. 2. A DTSVLIW machine.

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

The key issues to be resolved in the DTSVLIW architecture are the scheduling of the instruction trace into long instructions and the addressing within these long instructions. The primary processor and the VLIW engine themselves are not a challenge. In this section, the scheduler engine operation and the VLIW engine long instruction addressing are examined together with other relevant aspects of the DTSVLIW architecture. Full details of the architecture can be found in [8].

3.1. The Scheduler Engine

The scheduler engine is composed of the primary processor plus the Scheduler Unit (Fig. 2). The primary processor is a simple pipelined processor capable of executing all instructions of the Sparc Version 7 ISA [35]—the ISA we have chosen for the implementation of the DTSVLIW machine described in this paper. When a valid instruction moves from the decode pipeline stage to the execute pipeline stage, the primary processor sends it to the scheduler unit. The scheduler unit implements in hardware a simplified version of the first come first served (FCFS) algorithm, which historically has been used to statically schedule microcode [4]. 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 [4]. Finally, the FCFS algorithm, as presented here, is easy to implement in hardware in a pipelined fashion (see Section 3.5).

3.1.1. *The scheduling algorithm.* The DTSVLIW's scheduler engine performs *superblock scheduling* dynamically. Superblock scheduling [20] is a compiler technique derived from trace scheduling [11]. A superblock is a block of instructions encompassing many basic blocks in which control may only enter at the top, but may exit from one or more locations. In a compilation system, superblocks are built in two steps. First, traces are selected using heuristics or profiling. Second, *tail duplication* is applied to the trace to eliminate any side entrances, via creating a unique piece of code for each side entrance.

In a DTSVLIW machine, the execution trace produced by the primary processor feeds the scheduler unit, which schedules the instructions into blocks of long instructions and saves these blocks into the VLIW cache. Each block of long instructions may encompass many basic blocks. Scheduling is performed in a way that allows any branch inside any block to exit without side effects. The unique entry point of each block is its first instruction. Therefore, if a path in the program leads to an instruction inside an existent block, or a branch inside a block follows a path different than that followed during scheduling, these paths will cause the scheduling of new blocks. This is equivalent to tail duplication. 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 data sets.

A core scheduling operation performed by the DTSVLIW is the *move up* operation, which moves instructions through a list of long instructions inside the machine to produce compact long instructions (see Section 3.1.2). This operation is similar to the *move-op with renaming* operation of the *enhanced pipeline percolation scheduling* technique [27]. However, the application of the move-op operation requires the evaluation of all execution paths that transverse the instruction being moved; on the other hand, the application of the move up operation requires the evaluation of the trace path only. The move-op operation is applied in a sequential fashion by the compiler, while the move up operation is applied in a pipelined parallel fashion by the hardware. The application of the move-op operation can cause the generation of new long instructions whereas the move up operation does not.

The DTSVLIW schedules the code trace observed during execution into long instructions, all with the same format. For each branch, the long instruction holds the target address for branching out of the block; however, each long instruction has a default next long instruction target. This simplifies the implementation of fast fetch hardware.

3.1.2. *The scheduling algorithm implementation.* The implemented version of the FCFS algorithm acts on a list, the *scheduling list*. This list has a number of elements equal to *BLOCK_SIZE* (a hardware constant) and each element contains one long instruction and a *candidate instruction*, which holds an instruction for scheduling into the long instruction. A broad overview of the algorithm is that an instruction coming from the primary processor in one clock cycle is placed at the end of the scheduling list on the next clock cycle. On each subsequent cycle, this instruction can *move up* to the next higher element in the list if it has not reached the head of the list, there is space for it in the next element, and there is not a dependency with instructions in the next element. Fig. 3 shows an example of the algorithm scheduling a fragment of code that adds all elements of a vector. In Fig. 3, slh and slt stand for scheduling list head and tail, respectively, and the destination register of the instructions is the rightmost. The scheduling algorithm ignores the nop instruction. The details of the algorithm's operation follow.

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

An instruction inserted with the first method is a candidate for moving up the list on subsequent clock cycles. There can only ever be a single candidate instruction in a long instruction, but each long instruction in the list may have a candidate for promotion. Thus, candidate instructions of different long instructions can be moved up in parallel in a clock cycle. After an instruction has been inserted into the end of the list, the next step is to move this instruction up as far as it can go in the list of long instructions. An instruction can move up from long instruction i to long instruction $i-1$ if it has no true data dependency on any instruction in the long instruction $i-1$ and there is a suitable slot available. If the instruction cannot move up, it is *installed* in long instruction i. In Fig. 3, 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 anti-dependency on any instruction in i, or
- there is a control dependency on any instruction in i (there is a conditional branch or indirect branch in i)

However, in such cases, the candidate instruction has to be *split*. The split is done by renaming either the candidate instruction's output that has caused the

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

(a)

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

(b)

slh->	or	r0, 0, r9	sethi	hi(56), r8			
slt->	or	r8, 8, r11					after 3 cycles

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

slh ->	or	r0, 0, r9	sethi	hi(56), r8	or	r0, 0, r10	
	or	r8, 8, r11	add	r10, 4, r32			after
	ld	[r10+r11], r8	COPY	r32, r10	subcc	r32, 4*x-1, r0	9 cycles
slt ->	add	r9, r8, r9	ble	loop			

slh ->	or	r0, 0, r9	sethi	hi(56), r8	or	r0, 0, r10	
	or	r8, 8, r11	add	r10, 4, r32			after
	ld	[r10+r11], r8	COPY	r32, r10	subcc	r32, 4*x-1, r0	11 cycles
slt ->	add	r9, r8, r9	ble	loop	ld	[r10+r11], r8	

(c)

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

output/anti-dependency or all outputs if there is a control dependency and by inserting a *copy instruction* permanently in the current slot in long instruction *i*. This copy instruction performs the copy of the renaming register (or the renaming registers) content to the instruction's original output (or instruction's original outputs). In Fig. 3, instruction 7 is split in the ninth cycle. Copy instructions do not cause data dependencies and they can be overwritten by other instructions that write to the same registers that they write to during scheduling.

Allowing instructions to cross basic block limits imposed by conditional and indirect branches provides speculative execution. This is implemented by splitting instructions and moving up their first part past conditional or indirect branches, leaving the copy part behind. 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. For example, in Fig. 3, the second instance of instruction 5 receives the tag established by instruction 9 in the eleventh cycle. During VLIW execution, the VLIW engine evaluates 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. If a conditional or indirect branch does not follow the same direction during execution, the copy part of the split instruction is not executed, not committing the corresponding instruction.

When there is no free element for an incoming instruction, the scheduling list content is sent to the VLIW cache as a *block* and the incoming instruction is inserted into the list as the first instruction of a new block. 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 (see [8] for details).

3.1.3. *Multicycle instructions handling.* Multicyle instructions, such as integer divide, floating point multiply, or (sometimes) loads and stores, impact upon both the operation and performance of the architecture. Their scheduling requires special care to respect dependencies in any of their cycles. To schedule multicycle instructions within the DTSVLIW, extra features were added to the scheduling list [7]. These are an extra candidate instruction for each element of the list, the *candidate instruction B*, and an extra slot in each VLIW instruction for each multicycle functional unit, the *slot B*. Two instances, A and B, of a multicycle instruction are inserted into the scheduling list. These are just copies of the original instruction and have cross-references to each other's position in the scheduling list. The purpose of the A and B instructions is to delimit the scheduling list elements in which the instruction is active to prevent instructions with dependencies being scheduled in these elements. The primary role of the B instruction is for dependency checking against instructions moving up.

Instance A is inserted into the tail of the scheduling list. Instance B, on the other hand, is inserted into the *scheduling list tail* + (*instruction latency* − 1) position of the scheduling list in the candidate instruction B and slot B; the scheduling list tail register is made to point at this element. If this exceeds the maximum block size, a new block is started at instance A's position in the list. After insertion, the scheduler unit handles these two instances as other instructions, except that:

- Instance B does not suffer or cause resource dependencies.
- Instance B does not suffer or cause data dependencies related to its inputs.
- Instance A does not suffer or cause data dependencies related to its outputs but only control dependencies, in which case A's output is renamed and instance B is split. The copy instruction generated is not placed in a B slot, but in a normal one.
- Instances A and B move up together; if A cannot move up both are permanently placed in their current long instructions (B can always move up if A can move up).
- If instance B suffers an output or anti data dependency it is split and A is also renamed.

- Instance B is not saved in the VLIW cache and is only used for scheduling purposes.

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

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

3.1.4. *Nonschedulable instructions handling.* A few instructions are too complex for the VLIW engine and are always executed by the primary processor. When such an instruction is sent to the Scheduling unit, it flushes the scheduling list to the VLIW cache.

3.2. *DTSVLIW Long Instruction Format*

The format of the long instructions of the DTSVLIW can be appreciated with the help of Fig. 4. In this figure, two long instructions are shown, and each of them has five instructions. The individual instruction tags are represented by the shaded fields. The first long instruction does not have conditional or indirect branches; therefore, all instructions have tags with the value zero. Instructions with tags equal to zero are executed unconditionally. The second long instruction has two conditional branches. In this long instruction, only the branch *ble (0) loop* is executed unconditionally. The *(0)* in this branch indicates which of the many DTSVLIW's conditional code registers (CC registers) the branch is testing. These CC registers can be renamed as shown in Fig. 4. The instruction *subcc r1*, *r3*, *r0*: *(1)* writes into the conditional code register 1, as indicated by the *(1)*. The copy instruction *COPY cc1*, *cc0* copies the content of the cc1 to cc0: the original *subcc* was split and moved up by the scheduler unit. The branch *ble (0) loop* changes this long instruction's tag value to 1, and three instructions receive this new tag value, including the second branch, *bz (1) exit*. When installed, this second branch changes the tag again, and the instruction *add r8*, *r2*, *r3* receives the new tag value, 2. When executing this long instruction, the VLIW engine validates the tags 1 and 2 by checking if the branches

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

FIG. 4. DTSVLIW long instruction format examples.

follow the same direction observed during scheduling. Only instructions with valid tags, including branches, have their results written to the machine state.

3.3. *VLIW Engine Instruction Addressing*

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

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

3.4. *The VLIW Engine*

The VLIW engine of the DTSVLIW has a simple fetch-execute-write back pipeline for each functional unit (multicycle instructions execute in pipelined functional units with more than one execute stage). A decode stage is not necessary as decoded instructions are saved in the VLIW cache. 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. All conditional and indirect branches are resolved in the execute stage of the VLIW engine.

3.5. *Scheduler Unit Implementation*

The scheduler unit can be implemented in a pipelined fashion as depicted in Fig. 5. One or more pipeline stages can be used for inserting instructions into the scheduling list, each scheduling list entry can be made a pipeline stage, and none,

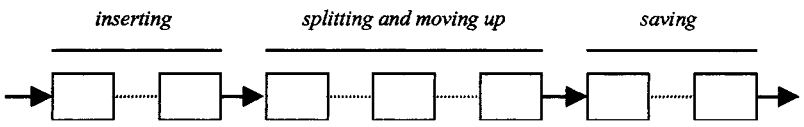

FIG. 5. Scheduler unit pipeline.

one, or more pipeline stages can be used for saving the scheduled long instructions into the VLIW cache. The checking operations required on the scheduling list on each clock cycle are just comparison operations between each candidate instruction and the instructions in the current and next element of the list. Each check operation is independent. However, the decision to install, split, or move up a candidate instruction may depend on a chain of decisions as long as the scheduling list. Nevertheless, the information necessary to each one can be gathered in a way similar to carry propagation in carry-lookahead adders [28, Appendix A], and the logic required can be made as fast as an *and-or* gate delay. It can be proved with the help of Fig. 6.

In Fig. 6, the value of CRd(i), CTd(i), COd(i), Rd(i), Td(i), Od(i), Ad(i), and Cd(i) for each element *i* of the list ($0 < i < \text{block size} - 1$) is available at the beginning of each clock cycle after the comparators delay (*xor* gate delay). Invalid candidate instructions never produce CRd(i), CTd(i), or COd(i) signals. Valid candidate instructions could influence the Rd(i), Td(i), Od(i), and Ad(i) signal values; for this reason, their companion position is used for disabling the comparators associated with the slot where the companion instruction is. CRd(i) is also disabled if there is more than one slot available in $i-1$ for candidate instruction *i*.

Let us analyze the installing case first. A valid candidate instruction must be installed on true data dependencies or resource dependencies. So, if Td(i) is true there is an instruction already installed in long instruction $i-1$ causing a true data dependency on the candidate instruction *i*. In this case, the candidate instruction in

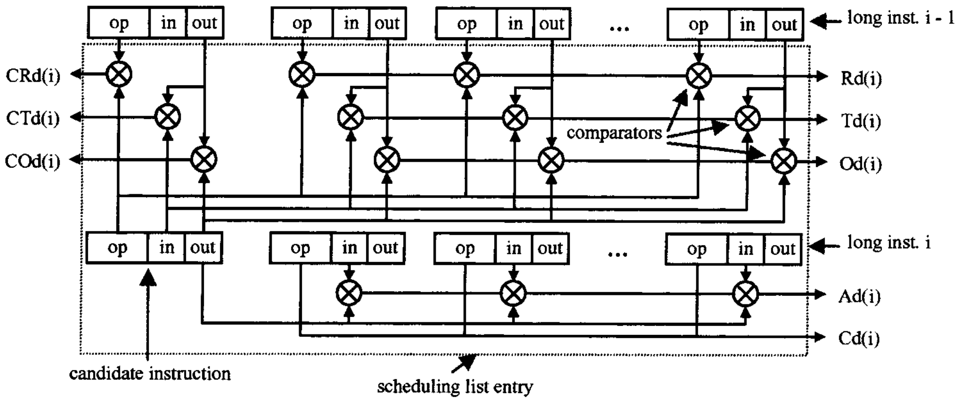

FIG. 6. Scheduling list. Rd(i), Td(i), Od(i), Ad(i), and Cd(i) stand for resource dependency, true data dependency, output data dependency, anti-data-dependency, and control dependency in candidate instruction i, respectively. CRd(i), CTd(i), and COd(i) stand for resource dependency, true data dependency, and output data dependency on candidate instruction i caused only by the candidate instruction in long instruction $i-1$, respectively.

i must be installed. If only CTd(i) is true one cannot tell whether or not the candidate instruction should be installed, because the candidate instruction in $i-1$ might move up in this cycle. The same can be said about Rd(i) and CRd(i) signals. Nevertheless, using the position of the candidate instruction in the list, which is recorded in each valid scheduling list entry, an *install signal* can be computed for each candidate instruction in the scheduling list as follows:

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

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

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

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

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

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

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

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

3.6. Load/Store Instructions and Memory Aliasing Detection

Load and store instructions can be split and moved up by the scheduling algorithm without restrictions. For the dependency test, their data addresses are compared with the data address of other load/store instructions, while the registers they use (including those used to compute data addresses) are compared with registers of other instructions (including load/store). *Memory renaming registers* provide for the renaming of memory positions. Memory aliasing [12] can occur, as the memory address observed during scheduling is not necessarily the same during VLIW execution. To detect memory aliasing and generate memory aliasing exceptions during VLIW execution, load and store instructions receive two extra fields when they are scheduled: the *order* and the *cross bit* fields. The order field receives the load/store insertion order, which is copied from the *load/store order counter*. This counter is zeroed every time the scheduling list is found empty and is incremented every time a load/store is inserted into the scheduling list. The cross bit field is set in the load/store when it is placed in a long instruction containing a store or a memory copy instruction generated from a store split.

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

3.7. *Exception Handling*

Exceptions (interrupts) may be generated by the execution of some instructions, such as load/store (page faults, access violations) or divide (divide by zero). However, split instructions should not signal exceptions until their copy part is executed. To avoid uncommitted instructions generating exceptions but to still allow true exceptions to be handled, an *exception bit* is added to each renaming register of the DTSVLIW. When a split instruction generates an exception, the exception bit of its renamed destination register is set and execution proceeds normally. If this register is read by any other instruction, the exception bit is propagated to the destination registers of this instruction. If an input register with an exception bit set is committed to the machine state, an exception is signaled.

The DTSVLIW uses the *checkpointing* exception handling mechanism, proposed by Hwu and Patt [19]. Checkpointing occurs at the beginning of the execution of each block of long instructions, when all registers that make up the Sparc ISA state are saved in shadow registers. Store instructions executed in the block cause the data they overwrite in the data cache to be saved in the *checkpoint recovery store list*. This list contains the address, data overwritten, and data type.

If the VLIW engine detects an exception during the execution of a block, the scheduler engine enters a *recovery mode* of execution. In this mode, registers are restored from the shadow registers, each entry of thecheckpoint recovery store list is written back into the data cache, and the load and store lists are emptied. If the exception detected is an aliasing exception, the VLIW cache entry containing the block that caused the exception is invalidated. Execution is then resumed.

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

There are alternative schemes to this, but these need further research [8].

3.8. *Differences between DTSVLIW and DIF*

The DTSVLIW architecture differs from the DIF architecture in the organization of the cache used by the VLIW engine, in its scheduling algorithm, in its register renaming, and in the VLIW engine register access mechanism. The unit of communication between the DIF cache and its VLIW engine is an entire block of long instructions, whereas the DTSVLIW machine accesses one long instruction per VLIW cache access. This should make the DTSVLIW VLIW cache implementation simpler than the DIF VLIW cache. A DIF machine schedules instructions using a hardware table, which has as many entries as resources in the machine and records the earliest long instruction in which each resource is available. Its proposed scheduler implements the greedy algorithm, by checking all resources necessary for each new instruction against this table and scheduling the instruction in the earliest long instruction possible. The DTSVLIW uses a simplified pipelined version of the first come first served algorithm, which operates over a list of long instructions. An instruction has only to be checked for dependencies against other instructions in its current and next position in the list, as opposed to all resources available in the machine. Instead of using copy instructions to implement register renaming, a DIF machine has a number of instances of each ISA register and extra bits are added to each register specifier to specify the register being used during VLIW execution. A register-mapping table is used to access the current ISA register set. Renaming is performed by specifying the extra bits during scheduling and by copying the new register mapping—the *exit map*—to the register-mapping table every time the execution leaves a block. Each exit point of a block (all branches and the final long instruction) has to carry its own exit map. This mechanism may not be practical for machines with a large number of physical registers, however. The Sparc ISA, for example, allows processor implementations with as many as 520 integer registers due to its register windows. Although most Sparc processors have only 128 integer registers, a single exit map for such a processor with four instances of each register would require 256 bits only for integer registers. The DTSVLIW splits instructions with the purpose of renaming registers to overcome data and control dependencies and the copy instructions generated are simpler to handle than mapping tables.

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

4. EXPERIMENTAL METHODOLOGY

We have implemented a parameterized and instrumented simulator of the DTSVLIW architecture to perform the experiments described in this paper. This simulator executes ordinary programs for the Sparc Version 7 ISA compiled with standard compilers. Integer programs from the SPEC95 benchmark suite have been compiled and used as input for the simulator.

4.1. The DTSVLIW Simulator

The DTSVLIW simulator has been implemented in C (23 K lines of code) and performs *execution-driven* simulation. To guarantee correct simulation results, all results have been produced with the simulator running in a special mode called *test mode*. The test mode puts two machines to run together: the DTSVLIW and a *test machine* with the same characteristics of the primary processor of the DTSVLIW. The DTSVLIW starts first, and every time an instruction or a block of long instructions is completed, the simulator switches to the test machine, which runs until its PC becomes equal to the DTSVLIW PC. The Sparc ISA state of both machines is compared and, if not equal, an error is signalled and the simulation interrupted. The test mode has been very useful not only to validate the execution but also because in this mode it is possible to measure the precise number of instructions necessary for the execution of a program, which the test machine can provide. A DTSVLIW simulator alone cannot provide this number due to copy instructions and instructions executed speculatively.

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

The DTSVLIW parameters that are invariant for all simulations are presented in Table I, while the parameters that we have varied to appreciate their influence on the DTSVLIW performance are shown in Table II together with their default values. Except when stated otherwise, the default values were used in the simulations. Each program was run for 50 million or more instructions each experiment, as counted by the test machine. We have chosen to run this number of instructions because this is optimistically the number of instructions that a DTSVLIW machine is capable of executing between operating system context switches. (Supposing that the DTSVLIW can execute five instructions per cycle, a clock rate of 1 GHz, and one context switch every 10 ms.)

4.2. Benchmark Programs

The SPECint95 benchmark programs and their respective input data used in the experiments reported in this paper are shown in Table III. All programs have been compiled with the gcc 2.7.2 compiler, using optimization flag −O. In this level of optimization, the gcc compiler performs several optimisations such as automatic register allocation, common subexpression elimination, invariant code motion from loops, induction variable optimizations, constant propagation and copy propagation and filling of delay slots. We could have used the higher levels of optimization,

TABLE I

Fixed Parameters

Trace processor	Four-stage (fetch, decode, execute, and write back) pipeline
	Not taken branches cause a 2-cycle bubble in the pipeline (the Sparc ISA's delayed branches allow for zero-bubble taken branches)
	Instructions following a load, requiring the data loaded cause a one-cycle bubble in the pipeline
Decoded instruction size	6 bytes
VLIW engine list sizes	Load = store = checkpoint recovery store = unlimited
Scheduler unit pipeline	Inserting = 1 stage
	Splitting and moving up = block-size stages
	Saving = 1 stage

TABLE II

Variable Parameters

Parameter	Default value
Number of VLIW engine functional units	Equal to the long instruction size
VLIW engine functional units type	Untyped (can execute any instruction)
Number of renaming registers	Unlimited
Next long instruction miss penalty	No penalty (0-cycle)
Instructions latency	1-cycle
VLIW cache size	3072-Kbyte
VLIW cache associativity	4-way
Instruction cache	Perfect (no miss penalty)
Data cache	Perfect (no miss penalty)

−O2 or −O3; however, these levels include optimizations 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.

4.3. *Metrics*

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

We refrain from averaging benchmark performances most of the time and show performance measurements for each individual benchmark. However, sometimes averages are useful. Jacob and Mudge [21] and Giladi and Ahituv [15] have discussed which average should be used when dealing with computer performance indices and have suggested the use of the harmonic mean for indices like IPC. Therefore, when appropriate we use the harmonic mean. Nevertheless, we also show, for extra clarity, the arithmetic mean between parentheses at the side of the harmonic mean in the form (4.1 u.a.m.), where u.a.m. means *using arithmetic mean.*

TABLE III

Benchmark Programs and Input Data

Benchmarks	Inputs
compress	20000 q 2131
gcc	-03 jump.i
go	40 19 null.in
ijpeg	vigo.ppm –GO
m88ksim	dhry.big
perl	primes.pl
vortex	vortex.in
xlisp	queens 7

5. EXPERIMENTS

In this section, we present and discuss the experiments carried out to evaluate the integer performance of the DTSVLIW architecture. We start by examining the effect of some architecture parameters on the DTSVLIW performance. We then present comparisons between the DTSVLIW and the DIF.

5.1. Effect of Some Architectural Parameters on the DTSVLIW Performance

5.1.1. *Block Size and Geometry.* Fig. 7 shows the effect of the block size (in number of instructions) and block geometry (instructions per long instruction (width) versus long instructions per block (height)) on the DTSVLIW performance. To ensure the absence of extraneous effects, we have used perfect instruction and data caches (no miss penalty), large VLIW cache (3072 Kbyte), and no next long instruction miss penalty to produce the results shown in Fig. 7. The numbers in the figure's legend are instructions per long instruction and long instructions per block, respectively.

As the graph in the figure shows, the performance of machines with the same block sizes and different geometry is significantly different. For example, the performance of the machine with 4×8-block geometry is lower than the machine with 8×4-block geometry for all benchmark programs. The block width and height affect the cost of implementing a DTSVLIW machine in different ways. Large long instructions imply many functional units, data cache ports, and register file ports. A Large number of long instructions in a block imply many renaming registers and long load/store and checkpoint recovery store lists. Increasing 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 8×8-block geometry performs better than machines with 4×16-block geometry and 16×4-block geometry in the majority of the SPECint95 benchmarks. The DTSVLIW benefits from large block sizes but not linearly. A 16-fold increase in the number of instructions of a block (from 4×4 to 16×16) does not quite double its performance.

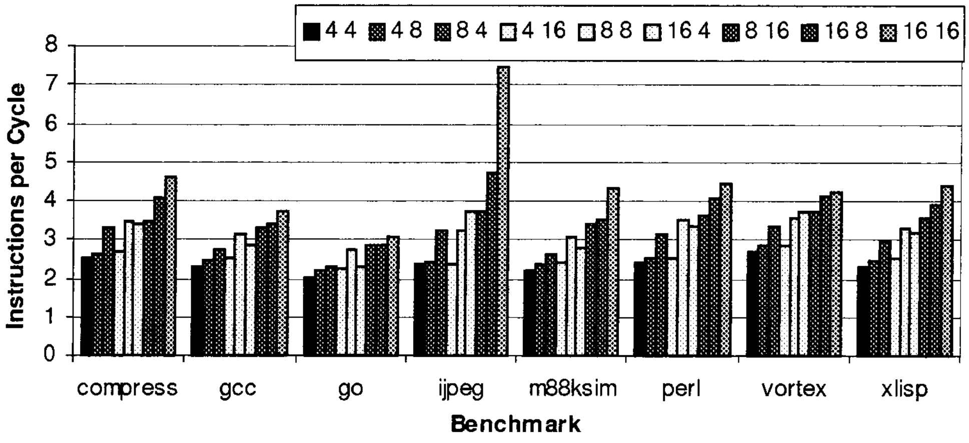

FIG. 7. Variation of parallelism with the block size and geometry.

The performance of the 16x16 configuration on the ijpeg benchmark is extraordinary and has been investigated. This benchmark spends most of its execution in one loop. With a large enough block size, more than one iteration of the loop can be scheduled into a single block, allowing instructions from these iterations to be overlapped, extracting much greater parallelism. (In Fig. 3, instruction 5 of the second loop iteration overlaps with instructions of the first.)

5.1.2. *VLIW Cache Size.* The results of Fig. 7 represent the highest achievable SPECint95 performance of the DTSVLIW for the block sizes shown. However, when the VLIW cache is smaller the performance is expected to be lower due to premature flushing of useful scheduled blocks by replacement blocks, leading to the need to rebuild the blocks flushed. This requires the primary processor to run, reducing parallelism.

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

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

5.1.3. *VLIW Cache Associativity.* Fig. 9 shows the effect of the VLIW cache associativity on the performance of the DTSVLIW. Two cache sizes are presented; 96- and 384-Kbyte, and the associativity is varied from 1 to 8. The figure shows that ijpeg is insensitive to the VLIW cache associativity in this range; however, m88ksim, perl, xlisp, and compress (for the 96-Kbyte cache) benefit from extra

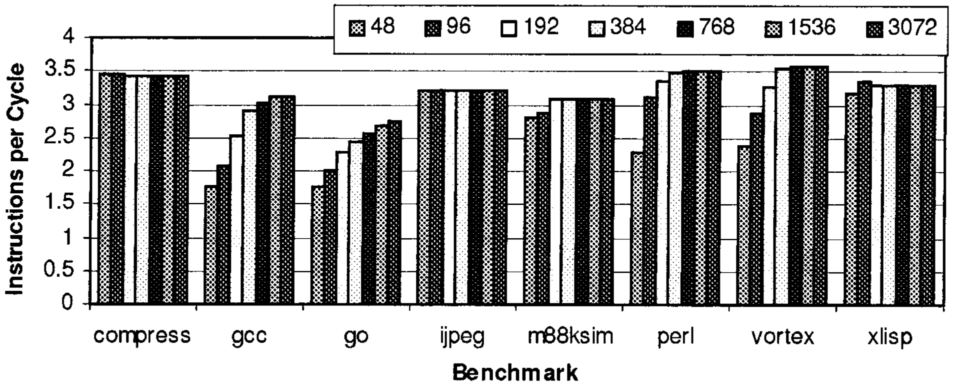

FIG. 8. Variation of the parallelism with the VLIW cache size.

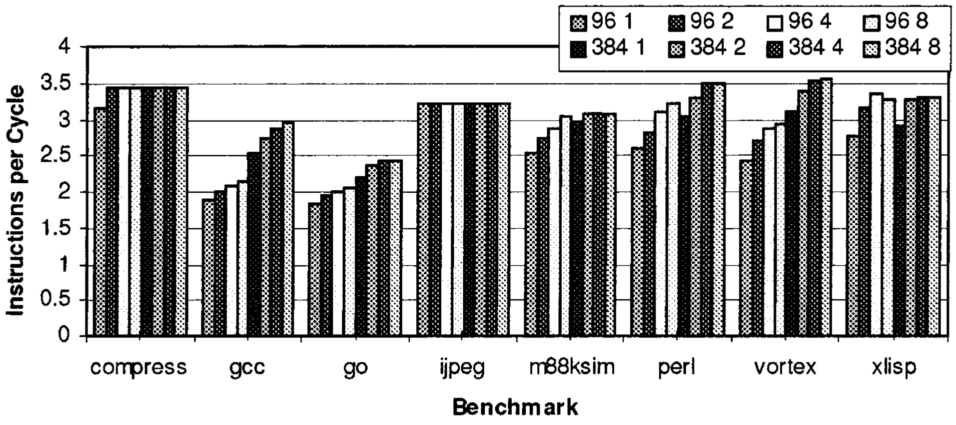

FIG. 9. Variation of parallelism with VLIW cache associativity.

associativity. From Figs. 8 and 9 it is possible to infer that a two- or four-way set-associative 384-Kbyte VLIW cache offers a cost-effective solution for a DTSVLIW machine with 8×8-block geometry.

5.1.4. *Multicycle instructions latency.* The Sparc 7 ISA does not have integer divide or multiply instructions, but only a multiply-step instruction that executes in a single cycle [35]. Therefore, from the set of integer instructions, only loads and stores require more than one cycle to execute in the Sparc 7 ISA. The graph in Fig. 10 shows the effect of the load–store instructions latency on the performance of the DTSVLIW with 8x8-block geometry. In the figure's legend, LxSy stands for load instructions with latency of x and store instructions with latency of y. In these results, we express the latency as the number of cycles necessary for the load/store execution and the functional units are fully pipelined.

As the graph in the figure shows, the latency of load instructions has a severe impact in the DTSVLIW performance—25.9% (25.3% u.a.m.) performance loss with 2-cycle and 50.7% (50.2% u.a.m.) with 3-cycle load latency. This occurs because load instructions are frequent in integer code and loaded data is usually required shortly after the load instructions. On the other hand, the latency of store

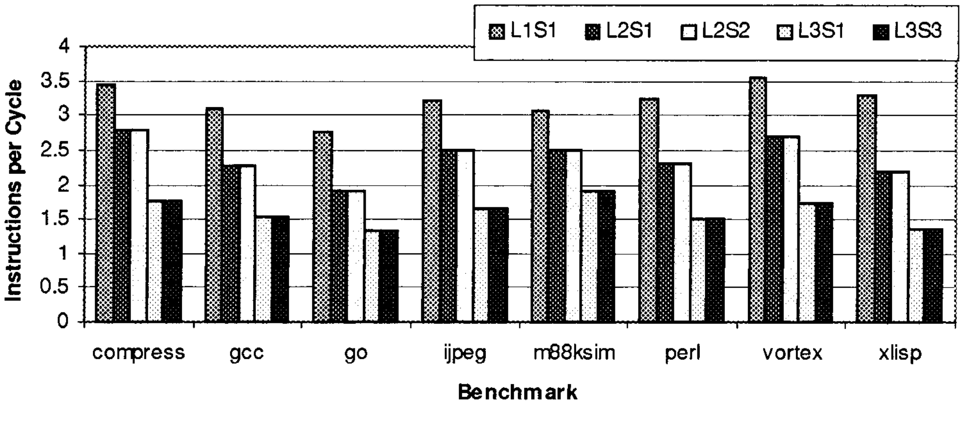

FIG. 10. Variation of the parallelism with the load/store instructions latency—8×8-block.

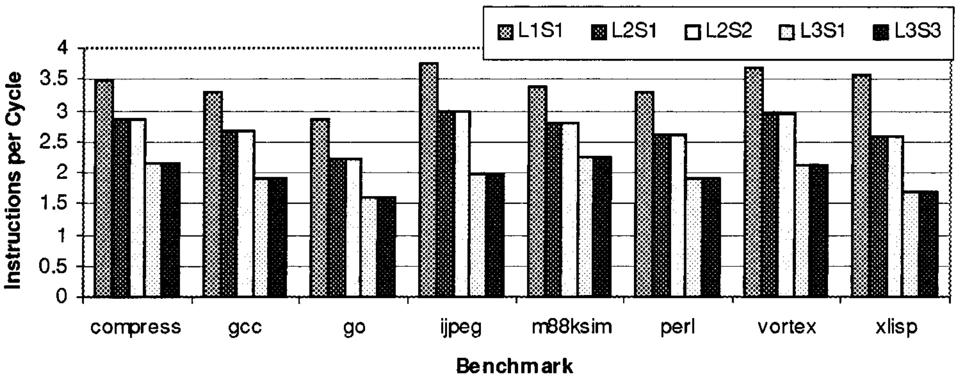

FIG. 11. Variation of the parallelism with the load/store instructions latency—8 × 16-block.

instructions does not have a strong impact on the DTSVLIW performance. Store instructions are also frequent in integer code; however, the stored data is often not required again for a relatively large number of instructions.

The graph in Fig. 11 shows the impact of load–store latency on the performance of a DTSVLIW machine with 8x16-block geometry. The impact is smaller with this geometry — 20.6 % (20.4 % u.a.m.) performance loss with 2-cycle load latency and 43.1 % (42.7 % u.a.m.) with 3-cycle load latency on average. With a longer block, the scheduler unit has more opportunities to accommodate instructions in the empty VLIW instructions created by the scheduling of multicycle loads. This results in better scheduling and better performance, but the latency impact is still high and there are costs for using long blocks (see Section 5.1.1).

As mentioned in Section 3.1.3, the primary processor does not pipeline multicycle instructions but retains them on its execute pipeline stage until they complete execution. Table IV presents the percentage of cycles the DTSVLIW spends waiting for these multicycle instructions to complete in the primary processor. As the table shows, the cost of waiting in the primary processor is very small and it is not an issue in the impact of the load/store latency on the DTSVLIW performance. Table IV also presents the percentage of VLIW execution cycles for the DTSVLIW with 8x16-block geometry. This machine configuration executes 98.56 % (98.57 % u.a.m.) of the cycles in VLIW mode on average. This strongly suggests that the DTSVLIW architecture is effective in taking advantage of its VLIW engine.

There are two simple approaches for reducing the impact of the load instruction latency on the DTSVLIW performance. The first is always to implement loads with

TABLE IV

Percentage of Cycles Waiting Load/Store Latency in the Primary Processor and Percentage of VLIW Execution Cycles

	compress	gcc	go	ijpeg	m88ksim	perl	vortex	xlisp	H.Mean	A.Mean
Waiting cycles (8 × 8)	0.14 %	0.11 %	0.08 %	0.00 %	0.01 %	0.01 %	0.03 %	0.01 %	0.00 %	0.05 %
Waiting cycles (8 × 16)	0.11 %	0.18 %	0.21 %	0.00 %	0.01 %	0.02 %	0.05 %	0.02 %	0.01 %	0.08 %
VLIW cycles (8 × 16)	97.97 %	98.20 %	97.49 %	99.98 %	99.87 %	96.37 %	98.89 %	99.80 %	98.56 %	98.57 %

1-cycle latency. This would, in some cases, almost double the DTSVLIW clock cycle length and, therefore, it may not be a cost-effective alternative. The second is to implement loads with 1-cycle latency and to use a small and fast data cache (8-Kbyte direct mapped, for example). This would increase the Data cache miss rate, but it may be a more cost-effective alternative. Another approach to further reduce the impact of loads on the DTSVLIW performance is to use hardware- or software-implemented data prefetching [38]. This approach can also be used together with one of the former approaches.

5.1.5. *A feasible DTSVLIW implementation.* So far, the results presented have been produced under ideal assumptions to allow appreciation of individual architecture parameters. However, the DTSVLIW architecture permits straightforward implementation using current *Very large scale integrated* (VLSI) circuit technologies if reasonable design parameters are used. The graph in Fig. 12 presents the performance of a DTSVLIW machine with a set of parameters that permits implementation using available technology. These parameters are:

- Blocks with 16 long instructions and 8 instructions per long instruction.

- 12-wide VLIW engine, with typed (specialized) functional units and 2-cycle next long instruction miss penalty. The functional units used are: 5 integer, 3 load–store, 2 floating-point, and 2 branch funtional units. Although this VLIW engine has 12 functional units, the VLIW fetch is 8-instruction wide because the block is 8-instruction wide. This is the main reason why we are using 2-cycle next long instruction miss penalty. One extra cycle has been added to the VLIW engine pipeline to allow the unpacking of 8-instruction wide long instructions, which are fetched from the VLIW cache, into the 12-instruction wide long instructions required by VLIW engine. The scheduler unit is conscious of the number of functional units available and schedules the 8-instruction wide long instructions respecting their availability per cycle.

- 2-cycle latency load instructions and 1-cycle latency store, integer (the Sparc 7 ISA does not have integer divide or multiply but only multiply-step, which can execute in one cycle), branch, and floating-point instructions. Latency of one cycle

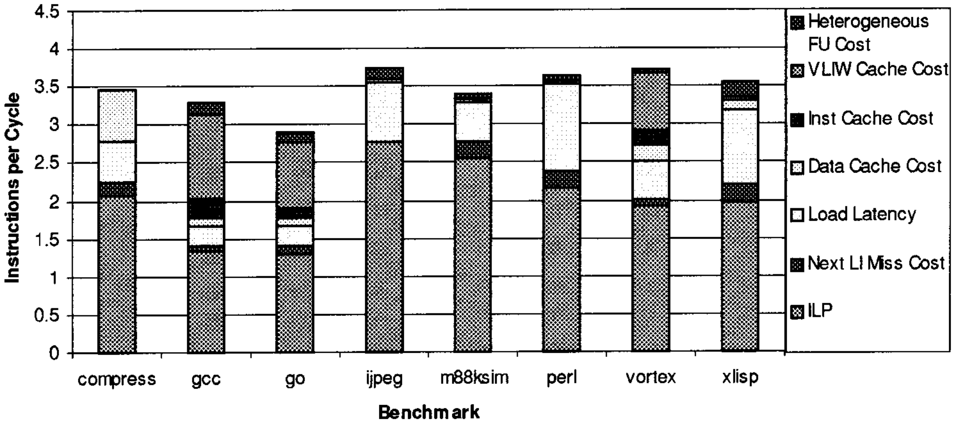

FIG. 12. Performance of a feasible DTSVLIW machine.

is a low latency for floating-point instructions; nevertheless, this latency has been used because the benchmarks are integers and, therefore, the number of floating-point instructions executed is zero or negligible.

- 192-Kbyte 4-way set-associative VLIW cache with 2-cycle fully pipelined access.
- 8-Kbyte 2-way set-associative instruction cache with 1-cycle access and 8-cycle miss penalty.
- 32-Kbyte 2-way set-associative data cache with 2-cycle fully pipelined access and 8-cycle miss penalty.
- Perfect (pre-initialized with all instructions and data) unified second level cache.
- The number of entries of the VLIW engine lists (load, store, and checkpoint recovery store) and the number of renaming registers has been left unlimited. However, the maximum number of entries required for these lists and the maximum number of renaming registers used during the simulation have been measured and are shown in Table V.

Figure 12 puts together, in the form of stacked bars, the result of various simulations to allow appreciation of the impact of various architectural parameters in the performance of this DTSVLIW machine. The first bar is the performance of the DTSVLIW with the parameters presented. The following bars represent the extra performance that would be added if the corresponding cost (shown in the legend) was removed. Table VI shows these costs as percentages of the maximum performance, together with other relevant information. Some items in this table do not have a harmonic mean because it cannot be computed when the list of values contains zeros.

As Fig. 12 and Table VI show, the load instruction latency is the principal contributor to the reduction of this DTSVLIW machine performance, and its cost is significant for all benchmark programs used. On the other hand, the second most important parameter that affects the machine performance—the VLIW cache size—has a significant impact only on the gcc, go, and vortex

TABLE V

Resource Consumption of a Feasible DTSVLIW Machine

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

TABLE VI

Performance of a Feasible DTSVLIW Machine

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

benchmarks. It is important to note, however, that, although large, this VLIW cache can hold only 256 blocks and only 35% (36% u.a.m.) of the instructions saved in these blocks during the simulations have been valid (Table VI, last row). Therefore, if nop instructions had not been saved in the VLIW cache, its capacity would have been better used and the performance of gcc, go, and vortex would have been significantly better. In addition, if the VLIW cache capacity had been better used, the instruction cache could have been even smaller than described. Instruction cache misses cause significant performance losses only for gcc, go, and vortex and, if the VLIW cache capacity is large enough, the size of the Instruction cache can be smaller than the size used [8].

Next long instruction misses have a small, although significant, impact on the machine performance. If nop instructions were not saved in the VLIW cache, a more elaborated VLIW fetch would be required, which would result in the need for the next long instruction miss penalty to be even higher than 2-cycle. However, the next long instruction address can be predicted and the number of misses reduced using techniques similar to those used in dynamic branch prediction.

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

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

A performance of 1.89 (2.01 u.a.m.) instructions per cycle in a machine with 12 functional units seems to be low. However, experiments with the PowerPC 620, an aggressive Superscalar machine with six functional units, have shown an average (arithmetic mean) of 1.2 instructions per cycle only [28, p. 341]. Taking into consideration that DTSVLIW machines can be implemented with clock speed higher than equivalent Superscalar machines such as the PowerPC, it appears to be worth implementing DTSVLIW machines with current technology. Simple machines with fast clocks have proved to be more powerful than their more complex counterparts [32]. In addition, DTSVLIW machines using equivalent hardware can perform better than Superscalars [8, 9].

5.2. DTSVLIW versus DIF

Figure 13 shows a comparison between a DTSVLIW and a DIF machine. The performance data of the DIF machine and the parameters used for both machines have been collected from [26]. The parameters were: 2 branch units plus 4 untyped functional units; 2-way set-associative instruction cache with 128-byte lines, 16 lines per set (4 Kbyte), and 2 cycle miss penalty; direct-mapped data cache with 128 lines each of length 32 bytes (4 Kbyte), and a 2-cycle miss penalty; 2-way set associative VLIW cache with 512x2 blocks; and a block size of 6 long instructions of 6 instructions each.

From this data and assuming an instruction size of 6 bytes for both machines, the DTSVLIW VLIW cache size is 216 Kbyte and the DIF VLIW cache size 463 Kbyte. The DIF VLIW cache is larger due to the DIF register renaming system. For each block exit point, the DIF machine requires 19 bytes for the exit map [26]. The number of renaming registers is different for the same reason. Four instances of each integer and floating point register were required in the DIF simulation, i.e., 96 integer and 96 floating point extra registers for renaming, while the maximum number of integer and floating point renaming registers required in the DTSVLIW simulation was 18 and 6, respectively.

As can be seen in Fig. 13, the average performance of the two machines is similar: 2.4 (2.4 u.a.m.) instructions per cycle for the DTSVLIW and 2.2 (2.2 u.a.m.) for DIF, a difference of approximately 10% (10% u.a.m.) in favor of DTSVLIW. DIF performs better in compress and xlisp, while DTSVLIW performs better in the remaining benchmarks. These results must be viewed with caution though, because the experiments carried out with the DIF implementation have used a trace

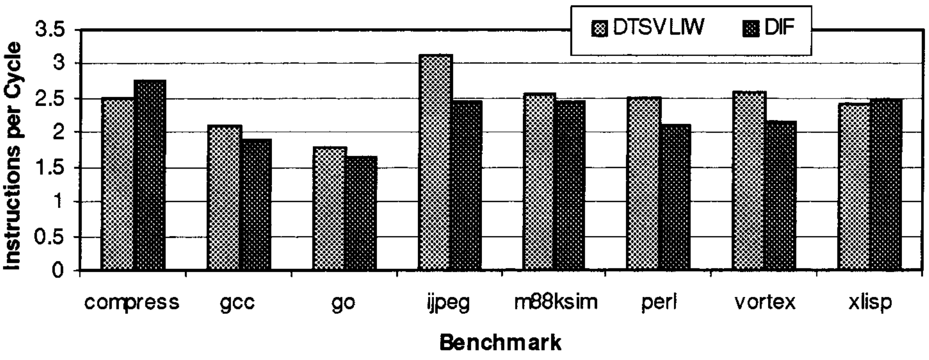

FIG. 13. DTSVLIW versus DIF.

simulator based in the PowerPC ISA, running the benchmarks with possibly different inputs and compiled with a different compiler with possibly different compiler flags. Nevertheless, a similar performance between the two machines was expected, since both implement the same concept, although in different ways.

6. CONCLUSION

This paper presents an architecture named the *dynamically trace scheduled VLIW*. This can be used to implement machines that execute code of current RISC or CISC ISAs in a VLIW fashion, delivering instruction-level parallelism with backward code compatibility. The architecture takes advantage of the code execution locality of current programs: Using the proposed architecture, the first time that a code fragment is executed, it is scheduled into long instructions and saved in a VLIW cache. In subsequent executions, a VLIW engine executes it in a VLIW fashion.

The experimental evaluations presented in this paper have shown that the DTSVLIW performance is basically similar to that of the DIF [26], but it is achieved with fewer hardware resources: 18 integer and 6 FP renaming registers in the DTSVLIW simulation, 96 integer and 96 FP in the DIF; 216-Kbyte DTSVLIW VLIW cache, 463-Kbyte DIF VLIW cache. As detailed in Section 3.5, the core logic of the scheduler engine is straightforward to implement, being comparable to an adder, and as such seems to be much more feasible than that of the DIF.

The design of the DTSVLIW architecture has been driven by the requirement to develop an architecture which can be effectively implemented to realize the fast clocking that can be achieved with 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 primary processor and the VLIW engine in the DTSVLIW can have high clock rates. The simplicity of the scheduling algorithm in the DTSVLIW means that a similar high clock rate should be achieved in an implementation of the scheduler unit, leading to an overall clocking rate similar to, if not higher than, high clock rate Superscalar architectures, but achieving higher ILP [8, 9].

The DTSVLIW architecture opens several new avenues of research. Next long instruction prediction, new VLIW cache organizations, and DTSVLIW-tailored compilers are just a few examples that will be investigated in future work.

REFERENCES

1. S. Banerjia, S. W. Sathaye, K. N. Menezes, and T. Conte, MPS: Miss-path scheduling for multiple-issue processors, *IEEE Trans. Comput.* **47**, 12 (December 1998), 1382–1397.
2. M. J. Charney and T. R. Puzak, Prefetching and memory system behaviour of the SPEC95 Benchmark Suite, *IBM J. Res. Devel.* **41**, 3 (May 1997), 265–285.

3. T. M. Conte and S. W. Sathaye, Dynamic rescheduling: A technique for object code compatibility in VLIW architectures, *in* "Proceedings of the 28th Annual International Symposium on Microarchitecture," pp. 208–218, Assoc. Comput. Mach., New York, 1995.
4. S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, Some experiments in local microcode compaction for horizontal machines, *IEEE Trans. Comput.* **30**, 7 (July 1981), 460–477.
5. A. F. de Souza and P. Rounce, Dynamically trace scheduled VLIW architectures, *in* "Proceedings of the High-Performance Computing and Networking 1998 – HPCN'98," Lecture Notes in Computer Science, Vol. 1401, pp. 993–995, Springer-Verlag, Berlin/New York, April 1998.
6. A. F. de Souza and P. Rounce, Dynamically scheduling the trace produced during program execution into VLIW instructions, *in* "Proceedings of the Second Merged Symposium IPPS/SPDP'1999: 13th International Parallel Processing Symposium & 10th Symposium on Parallel and Distributed Processing," pp. 248–257, IEEE Computer Society, Los Alamitos, CA, April 1999.
7. A. F. de Souza and P. Rounce, Effect of multicycle instructions on the integer performance of the dynamically trace scheduled VLIW architecture, *in* "Proceedings of the High-Performance Computing and Networking 1999 – HPCN'99," Lecture Notes in Computer Science, Vol. 1593, pp. 1203–1206, Springer-Verlag, Berlin/New York, April 1999.
8. A. F. de Souza, "Integer Performance Evaluation of the Dynamically Trace Scheduled VLIW Architecture," Ph.D. thesis, Department of Computer Science, University College London, University of London, September 1999.
9. A. F. de Souza and P. Rounce, On the scheduling algorithm of the dynamically trace scheduled VLIW architecture, *in* "Proceedings of the 14th International Parallel and Distributed Processing Symposium – IPDPS'2000," pp. 565–572, IEEE Computer Society, Los Alamitos, CA, May 2000.
10. K. Ebcioglu and E. R. Altman, DAISY: Dynamic Compilation for 100% Architectural compatibility, *in* "Proceedings of the 24th Annual International Symposium on Computer Architecture," pp. 26–37, Assoc. Comput. Mach., New York, 1997.
11. J. A. Fisher, Trace scheduling: A technique for global microcode compaction, *IEEE Trans. Comput.* **30**, 7 (July 1981), 478–490.
12. J. A. Fisher, The VLIW machine: A multiprocessor for compiling scientific code, *IEEE Comput.* (July 1984), 45–53.
13. M. Franklin and M. Smotherman, A fill-Unit approach to multiple instruction issue, *in* "Proceedings of the 27th Annual International Symposium on Microarchitecture," pp. 162–171, Assoc. Comput. Mach., New York, 1994.
14. J. D. Gee, M. D. Hill, D. N. Pnevmatikatos, and A. J. Smith, Cache performance of the SPEC92 Benchmark Suite, *IEEE Micro* (August 1993), 17–27.
15. R. Giladi and N. Ahituv, SPEC as a performance evaluation measure, *IEEE Comput.* (August 1995), 33–42.
16. L. Hammond, B. A. Nayfeh, and K. Olukotun, A single-chip multiprocessor, *IEEE Comput.* (September 1997), 79–85.
17. T. Hara, H. Ando, C. Nakanishi, and M. Nakaya, Performance comparison of ILP machines with cycle time evaluation, *in* "Proceedings of the 23rd Annual International Symposium on Computer Architecture," pp. 213–224, Assoc. Comput. Mach., New York, 1996.
18. R. J. Hookway and M. A. Herdeg, DIGITAL FX!32: Combining emulation and binary translation, *Digital Tech. J.* **9**, 1 (January 1997), 3–11.
19. W. W. Hwu, and Y. N. Patt, Checkpoint repair for out-of-order execution machines, *in* "Proceedings of the 14th Annual International Symposium on Computer Architecture," pp. 18–26, Assoc. Comput. Mach., New York, 1987.
20. W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, The Superblock: An effective technique for VLIW and superscalar compilation, *J. Supercomput.* **7** (1993), 229–248.

21. B. Jacob and T. Mudge, "Notes on Calculating Computer Performance," Technical Report CSE-TR-231-95, Advanced Computer Architecture Lab, EECS Department, University of Michigan, March 1995.

22. C. E. Kozyrakis, S. Perissakis, D. A. Patterson, T. Anderson, K. Asanovic, N. Cardwell, R. Fromm, J. Golbus, B. Gribstad, K. Keeton, R. Thomas, N. Treuhaft, and K. Yelick, Scalable processors in the billion-transistor era: IRAM, *IEEE Comput*. (September 1997), 75–78.

23. M. H. Lipasti and J. P. Shen, Exceeding the data-flow limit via value prediction, *in* "Proceedings of the 29th Annual International Symposium on Microarchitecture," pp. 226–237, Assoc. Comput. Mach., New York, 1996.

24. D. Matzke, Will physical scalability sabotage performance gains?, *IEEE Comput*. (September 1997), 37–39.

25. S. Melvin, M. Shebanow, and Y. Patt, Hardware support for large atomic units in dynamic scheduled machines, *in* "Proceedings of the 21st Annual International Symposium on Microarchitecture," pp. 60–66, Assoc. Comput. Mach., New York, 1988.

26. R. Nair and M. E. Hopkins, Exploiting instruction-level parallelism in processors by caching scheduled groups, *in* "Proceedings of the 24th Annual International Symposium on Computer Architecture," pp. 13–25, Assoc. Comput. Mach., New York, 1997.

27. T. Nakatani and K. Ebcioglu, Making Compaction-based parallelization affordable, *IEEE Trans. Parallel Distrib. Systems* **4**, 9 (September 1993), 1014–1029.

28. D. A. Patterson and J. L. Hennessy, "Computer Architecture: A Quantitative Approach, Second Edition," Morgan Kaufmann, San Mateo, CA, 1996.

29. B. R. Rau, Dynamically scheduled VLIW processors, *in* "Proceedings of the 26th Annual International Symposium on Microarchitecture," pp. 80–92, Assoc. Comput. Mach., New York, 1993.

30. E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, Trace processors, *in* "Proceedings of the 30th Annual International Symposium on Microarchitecture," pp. 138–148, Assoc. Comput. Mach., New York, 1997.

31. R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson, Binary translation, *Comm. Assoc. Comput. Mach.* **36** (February 1993), 69–81.

32. J. E. Smith and S. Weiss, PowerPC 601 and Alpha 21064: A tale of two RISCs, *IEEE Comput*. (June 1994), 46–58.

33. A. Sodani and G. S. Sohi, Dynamic instruction reuse, *in* "Proceedings of the 24th Annual International Symposium on Computer Architecture," pp. 194–205, Assoc. Comput. Mach., New York, 1997.

34. SPEC: Standard Performance Evaluation Corporation, available at http://www.specbench.org.

35. Sun Microsystems, "The Sparc Architecture Manual – Version 7," Sun Microsystems, Inc., 1987.

36. R. M. Tomasulo, An efficient algorithm for exploiting multiple arithmetic units, *IBM J. Res. Devel.* **11**, 1 (January 1967), 25–33.

37. D. M. Tullsen, S. J. Eggers, and H. M. Levy, Simultaneous multithreading: Maximazing on-chip parallelism, *in* "Proceedings of the 22nd Annual International Symposium on Computer Architecture," pp. 392–403, Assoc. Comput. Mach., New York, 1995.

38. S. P. VanderWiel and D. J. Lilja, When caches aren't enough: Data prefetching techniques, *IEEE Comput*. (July 1997), 23–30.

ALBERTO FERREIRA DE SOUZA received his B.Eng. (Cum Laude) in electronic engineering and his M.Sc. in computing and systems' engineering from the Universidade Federal do Rio de Janeiro (COPPE/UFRJ), Brazil, in 1988 and 1993, respectively, and his Ph.D. in computer science from the University College London, United Kingdom, in 1999. He is currently a senior lecturer of computer science at the Departamento de Informática of the Universidade Federal do Espírito Santo, Brazil. His research interests include computer microarchitecture, compilers, neural networks, and neuroscience. He is a member of the IEEE and ACM.

PETER ROUNCE received his Ph.D. from the University of London in 1986 and B.Sc. in physics from the University of Sussex in 1970. He is currently a senior lecturer in the Department of Computer Science at University College London, having been a lecturer in the department since 1981. His research interests encompass computer architecture, both design and implementation. His early career included a period as a software engineer for ICL.