# Improving DTSVLIW Performance via Block Compaction

Alberto F. de Souza

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

*Abstract*—

***Dynamically Trace Scheduled VLIW* (DTSVLIW) machines have two execution engines and two instruction caches: a Scheduler Engine and a VLIW Engine, and an Instruction Cache and a VLIW Cache. The Scheduler Engine fetches instructions from the Instruction Cache and executes them singly, the first time, using a simple pipelined processor. In addition, it dynamically schedules the instruction trace produced during this execution into VLIW instructions, groups them as blocks of VLIW instructions, and saves these blocks into the VLIW Cache. If the same instruction trace must be executed again, it is fetched by the VLIW Engine from this cache and executed in VLIW fashion. Due to code temporal execution locality, machines that follow the DTSVLIW architecture spend most of the time operating in VLIW mode, which results in instruction-level parallelism (ILP) comparable or better than other current approaches for exploiting ILP, such as superscalar or pure VLIW. However, the scheduled blocks may contain a large number of nop instructions in VLIW instructions' slots that the Scheduler Engine has been unable to fill with useful instructions. In this paper we present two techniques for compacting blocks that allow removing part of these nop instructions dynamically. Our experiments show that DTSVLIW machines that employ these techniques can perform 13.2% better than DTSVLIW machines that do not.**

*Keywords*— **DTSVLIW, VLIW, instruction compaction**

## I. INTRODUCTION

*Dynamically Trace Scheduled VLIW* (DTSVLIW) machines [DES 98] execute sequential code and exploit instruction-level parallelism (ILP). Their hardware, similar to that of Superscalars [JOH 91], has to determine dependencies and independencies between instructions and to bind instructions to resources dynamically. However, different from Superscalars, DTSVLIW machines perform these operations with one instruction at a time. This enables DTSVLIW implementations with fast clocks, while allowing ILP comparable or even better than that of superscalars or pure VLIWs [DES 00a, DES 00b].

Figure 1 shows a block diagram of the DTSVLIW architecture. It exploits programs' temporal execution locality by executing code in two distinct modes. First, fragments of code are executed in sequential mode by the Primary Processor (Figure 1) and, at the same time, scheduled into blocks of VLIW instructions by the Scheduler Unit, which also caches these blocks into the VLIW Cache. If the same code has to be executed again, it is fetched by the VLIW Engine from this cache and executed in a VLIW fashion.

The DTSVLIW architecture is a variant of the DIF architecture [NAI 97], proposed by Nair and Hopkins. As our earlier work [DES 99a] demonstrates, the DTSVLIW achieves similar or better performance to the DIF, but with a simpler architecture. The results in [DES 99c] further demonstrate the effectiveness of the DTSVLIW scheduling algorithm, which shows no significant reduction in performance over the DIF algorithm, even though the latter is expected to be much more difficult to implement, as we have also shown.

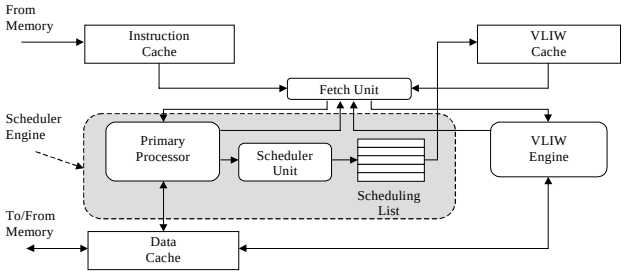


Fig.1 The Dynamically Trace Scheduled VLIW Architecture

Our main motivation for developing of the DTSVLIW 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 [GEE 93, CHA 97]. This shows that there is a strong temporal execution locality in programs. The DTSVLIW exploits temporal execution locality by scheduling code into blocks of VLIW instructions on the first execution encounter and by executing it in the VLIW Engine on subsequent encounters. However, frequently the Scheduler Engine is not able to fill the VLIW instructions completely, leaving than with many nop (no operation) instructions. In fact, our experiments show that, in the standard DTSVLIW architecture, only 19.9% of the VLIW instructions saved are filled with useful scalar instructions on average. In this paper we tackle this problem in two ways. The fist is a simple mechanism that was first proposed in [NAI 97]. It consists in

holding VLIW Cache fetch attempts until blocks being scheduled hold a certain number of instructions. The second, which can be used together with the first, consists of a mechanism for compacting the blocks of VLIW instructions before saving them into the VLIW Cache. The use of the first mechanism alone can improve DTSVLIW performance in 9.1% and block usage in 36.4%, while that the use of both can improve the DTSVLIW performance in 13.2% and block usage in 70.4%, as the experiments presented in this paper show.

This paper is divided in 6 sections. After this introduction, the DTSVLIW architecture is presented in more detail in Section 2. In Section 3, we describe the two mechanisms for improving DTSVLIW performance. In Section 4, we present our experimental set-up, results and discussion. In Section 5, we discuss related work and, in Section 6, our conclusions.

## II. THE DTSVLIW ARCHITECTURE

The symbolic diagram of a DTSVLIW machine is shown in Figure 2. It has two caches for instructions and two processing engines. The Instruction Cache stores fragments of original compiled code while the VLIW Cache stores blocks of *long instructions* (the term used in the rest of this paper to refer to VLIW instructions). The Primary Processor executes the original code first. 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.

While the Primary Processor is executing the code, the Fetch Unit (Figure 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 Figure 2). We use this instruction' address because at this point we know for sure that this instruction must be executed. If this instruction has been executed before, there may be a block with its address in the VLIW Cache. On a VLIW Cache hit, the VLIW Engine takes over execution. The block being constructed by the Scheduler Unit is flushed to the VLIW Cache – this block is made to point at the hit block. The contents of all but the write back pipeline stage of the Primary Processor are annulled and the PC receives the memory address that hit the VLIW Cache. In subsequent cycles, the VLIW Engine controls the PC.

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

In our current DTSVLIW implementation, the Primary Processor executes Alpha [DIG 92] code, while the VLIW Engine executes a sub-set. The VLIW Engine has a simple fetch – dispatch – execute – write-back pipeline. 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 occupies a single cache block. Individual long instructions are the unit of communication between the VLIW Cache and the rest of the DTSVLIW. Details on how the DTSVLIW deals with exceptions, memory aliasing (disambiguation), and the execution of particular instructions are in [DES 99a].

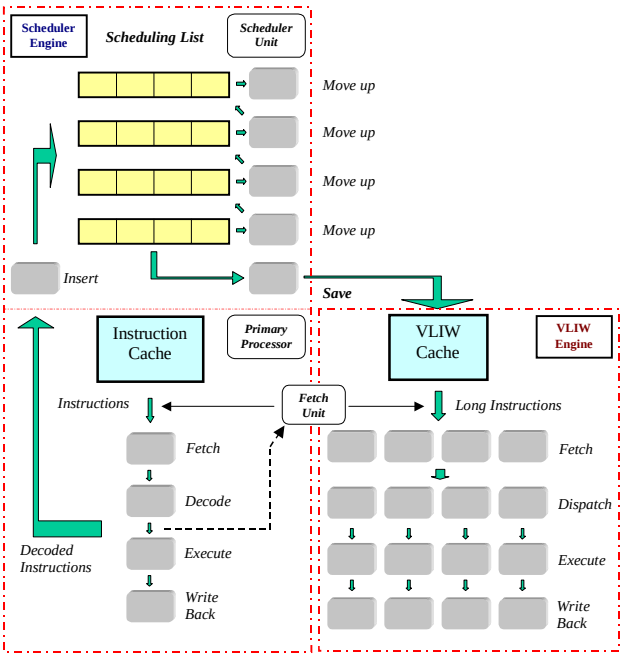


Fig.2 A DTSVLIW Machine

### A. *The Scheduler Engine*

The Scheduler Engine is composed of the Primary Processor plus the Scheduler Unit (Figure 2). The Primary Processor is a simple pipelined processor capable of executing all instructions of the Alpha instruction set architecture (ISA). 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 Engine performs *superblock scheduling* dynamically. Superblock scheduling [HWU 93] is a compiler technique derived from *trace scheduling* [FIS 81]. 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, through the creation of 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 branch outside its block without side effects, thanks to register renaming and memory disambiguation (see below). 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 from that followed during scheduling, these paths will cause the scheduling of new blocks. This is equivalent to tail duplication. However, when performing superblock scheduling, compilers select traces statically and these traces must be suitable for all programs' input data sets. In contrast, DTSVLIW machines perform dynamic trace selection and, as such, can achieve performance in all input data sets.

### *B. The Scheduling Algorithm*

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 [DAV 81]. 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 [DAV 81]. Finally, the FCFS algorithm is easy to implement in hardware in a pipelined fashion, as we have shown in [DES 99a].

A broad overview of the DTSVLIW scheduling algorithm is that a valid instruction in the decode pipeline stage of the Primary Processor is inserted at the end of the scheduling list on the next clock cycle (Figure 2). 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; and there is not a dependency with instructions in the next element.

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 – there is a pipeline of candidates for promotion. If an instruction cannot move up, it is installed into its current long instruction.

Below there is a move up example in a 2x2 scheduling list (the shaded instruction is a candidate instruction and the destination register is the rightmost):

| | | | | | |
|---|---|---|---|---|---|
| sub r1, r2, r3 | | move up ⇒ | sub r1, r2, r3 | add r4, r5, r6 |
| add r4, r5, r6 | | | | | |

Install example (the instruction is not moved up):

| | | | | |
|---|---|---|---|---|
| sub r1, r2, r3 | | install ⇒ | sub r1, r2, r3 | |
| add r3, r4, r5 | | | add r3, r4, r5 | |

If there is a control, output, or anti dependency on a candidate instruction, it can still move up, but has to be *split*. The split is done by renaming the candidate instruction's output, moving up the renamed instruction, and by inserting a *copy instruction* permanently in the long instruction slot previously occupied by the candidate instruction. This copy instruction copies the renaming register content to the instruction's original output. Example:

| | | | | |
|---|---|---|---|---|
| sub r1, r2, r3 | | split ⇒ | sub r1, r2, r3 | add r4, r5, r32 |
| beq r3, 1000 | add r4, r5, r6 | | beq r3, 1000 | COPY r32, r6 |

Conditional and indirect branches do not move up. They are installed when inserted into the scheduling list and establish a *tag* for their long instruction. All instructions subsequently installed 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. Therefore, the copy instruction shown in the example above is only executed in VLIW mode if the conditional branch (beq) follows the same direction observed during scheduling.

When there is no free element for an incoming instruction, the 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 pipelined 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 [DES 99a]. A block of long instructions is stored as a VLIW Cache block and is identified by the address of the first instruction installed in it. Each cache block holds this address and the address of the following block.

Load and store instructions can also be split, which can cause memory aliasing [FIS 84] and exceptions. Please refer to [DES 00b] for details on how the DTSVLIW deals with these situations. In [DES 00b] we also prove that the core operations performed by the DTSVLIW's scheduling algorithm have the complexity of an integer adder and, as such, should not impact negatively the DTSVLIW clock cycle time. Multicyle instructions impact upon the operation and performance of the architecture. Their scheduling, described in [DES 99b], has to respect dependencies in any of their cycles. This can restrict the packing of instructions into long instructions limiting parallelism.

## III. IMPROVING DTSVLIW PERFORMANCE WITH BLOCK COMPACTION

### *C. VLIW Fetch Starting Point*

When a DTSVLIW machine is scheduling code, every time a valid instruction reaches the Primary Processor's execute pipeline stage, a VLIW fetch can be attempted with its address. On a VLIW Cache hit, the VLIW Engine takes over execution and the block being scheduled is saved into the VLIW Cache.

If no special action is taken, the blocks produced this way can have any number of long instructions, from 1 to block size, but small blocks (1 or just a few long instructions) are likely not to have much parallelism. However, instead of always allowing VLIW fetches, we can easily allow VLIW fetches only when the size of the block being scheduled is near its maximum. That is, we can establish a starting point for VLIW fetches associated with the number of instructions or long instructions of the current block being scheduled, forcing the production of larger and, hopefully, more compact (parallel) blocks. In Section 4, we show that this simple modification alone can improve DTSVLIW performance in 9.1% on average.

### *D. VLIW Block Compaction*

During the scheduling, many long instructions' slots may not be filled due to control, resource or data dependencies. These slots receive nop instructions. These take the space, in the VLIW Cache, of possibly useful instructions, which might reduce DTSVLIW performance. We introduce here a mechanism for compacting blocks of long instructions prior to saving them into the DTSVLIW VLIW Cache. It operates at the *save* pipeline stage indicated in Figure 2.

A block of long instructions is saved in a pipelined fashion, one long instruction at a time [DES 99d]. A compacting buffer can be added to this pipeline to hold *compacted long instructions*, which will form compacted blocks into the VLIW Cache. The difference between a compacted and an uncompacted long instruction is that the former does not have nops, unless it is at the end of a block.

Several long instructions may be packaged together into the compacting buffer, but the last one may only partially fit – part of this last long instruction may have to be put into the following compacted long instruction. In order to reconstruct the long instructions during VLIW execution, a mechanism must be provided to identify where each long instruction starts and ends. This mechanism can be implemented with control fields added to each compacted long instruction, as shown in Figure 3.

Two blocks are shown in Figure 3: an uncompacted block and its correspondent compacted block. Each long instruction slot of the long instructions in these blocks can hold a useful instruction or a nop (empty slot). Valid instructions are labelled <LI*long instruction* number>-<I*nstruction number*>.

Each compacted long instruction carries its correspondent control fields, as shown in Figure 3b. In a compacted long instruction, there is one control field per instruction slot. These control fields specify where a long instruction, compacted into the compacted long instruction, ends. The first compacted long instruction of Figure 3b holds the first long instruction of Figure 3a plus one instruction of the second long instruction of Figure 3a. The first control field of this compacted long instruction contains the number 3, which specify that there is a long instruction inside this compacted long instruction that ends at the third instruction. The first control field also specify where the second long instruction starts, while the second control field specify where the second long instruction ends and the third starts, and so on.

| | | | |
|---|---|---|---|
| LI0-I0 | LI0-I1 | LI0-I2 | |
| LI1-I0 | LI1-I1 | | |
| LI2-I0 | LI2-I1 | LI2-I2 | LI2-I3 |
| LI3-I0 | LI3-I1 | | |
| LI4-I0 | | | |
| LI5-I0 | LI5-I1 | LI5-I2 | |

**(a)**

⇓

control fields

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | 0 | - | - | LI0-I0 | LI0-I1 | LI0-I2 | LI1-I0 |
| 1 | 0 | - | - | LI1-I1 | LI2-I0 | LI2-I1 | LI2-I2 |
| 1 | 3 | 4 | - | LI2-I3 | LI3-I0 | LI3-I1 | LI4-I0 |
| 3 | 4 | - | - | LI5-I0 | LI5-I1 | LI5-I2 | |

**(b)**

Fig.3 Block compaction. (a) Uncompacted block. (b) Compacted block.

The second control field of the first compacted long instruction of Figure 3b holds zero. The value zero indicates that the last long instruction inserted into this compacted long instruction is incomplete: the rest of it is in following compacted long instruction. The value zero marks, then, the end of a compacted long instruction, and any remaining control fields after a zero are not used (these contains a "-" in Figure 3b).

The last control field of the third compacted long instruction in Figure 3b contains the value 4. This value is equal to the compacted long instruction width and indicates that a long instruction ends at the last instruction of the compacted long instruction. As with the case of a control field with value zero, a control field with value equal to the size of the compacted long instruction marks the end of the compacted long instruction and any remaining control fields after this are not used.

During VLIW execution, the control fields of the compacted long instructions in the pipeline buffers of the fetch and dispatch VLIW Engine pipeline stages (Figure 2) are analysed by the logic of the dispatch stage. Guided by these control fields, the dispatch stage collects instructions from the buffers and reconstructs the original long instructions produced by the Scheduler Unit during scheduling. These long instructions are then sent to the functional units for execution.

The most important positive impact of employing this compacting block technique is that it allows scheduling lists larger than the limit imposed by the VLIW Cache block geometry. As shown in Figure 3, blocks with more than four long instructions can fit in a block with four compacted long instructions only, for example. Larger scheduling lists allow better performance, as our experiments show next.

## IV. EXPERIMENTS

Execution-driven simulations were performed for producing the results reported here using our new DTSVLIW simulator, which uses the simplescalar-3.0 tool set [AUS 97]

as its core. The simulator faithfully models the DTSVLIW and receives as input executables produced by ordinary compilers that generate Alpha ISA code. We have used the gcc 2.7.2 compiler with optimisation flags –O3 –unrolloops.

TABLE 1
FIXED PARAMETERS

| | |
|---|---|
| Primary Processor | • four-stage (fetch, decode, execute, and write back) pipeline<br>• no branch prediction hardware<br>• taken branches cause a 2-cycle bubble in the pipeline |
| Decoded Instruction Size | 6 bytes |
| Instructions Latency | 1 cycle |
| VLIW Cache | Four way set associative, 3072-Kbyte |
| Instruction Cache | perfect (no miss penalty) |
| Data Cache | perfect (no miss penalty) |
| Number of renaming regs. | unlimited |

TABLE 2
BENCHMARK PROGRAMS

| ***Benchmark*** | ***Inputs*** | ***Instructions Executed*** |
|---|---|---|
| compress | 30000 q 2131 | 144153036 |
| gcc | -O3 jump.i | 176479034 |
| go | 9 9 | 132169125 |
| ijpeg | vigo.ppm -GO | 220880247 |
| m88ksim | -c < ctl.in | 125045424 |
| perl | primes.pl | 139264287 |
| vortex | vortex.in | 120451770 |
| xlisp | queens 7 | 280939082 |

Model parameters that are invariant for simulations are presented in Table 1. They form an almost perfect DTSVLIW configuration, which we have choose to use to ensure the absence of extraneous effects and to allow the appreciation of the variables under study. The SPECint95 benchmark programs used are shown in Table 2, together with its inputs and the number of instructions executed. All programs were executed until termination.

The following subsections present the effects of the VLIW fetch starting point and compacted blocks on DTVSLIW performance.

### *A. Effect of the VLIW Fetch Starting Point*

The graph in Figure 4 shows the impact of the VLIW fetch starting point on DTSVLIW performance. We have used a DTSVLIW machine with 16 instructions per long instruction and 16 long instructions per block (16x16-block geometry) for this experiment. The legend of Figure 4 shows the different VLIW fetch starting points used: at any block size (baseline), at half of the maximum block size (1/2), at three quarters of the maximum block size (3/4), and at full block (VLIW fetches are only allowed when scheduling 16th long instruction). In addition to the individual performance of each benchmark, we have also added to the graph the harmonic mean (H.M.) of all benchmarks to appreciate the average performance of each configuration.

As the graph in Figure 4 shows, the strategy of discarding some possible VLIW fetch opportunities to favour the production of larger blocks is worthwhile. The machine configuration with VLIW fetch starting point at 1/2 of the maximum block size performs better than the baseline configuration in all benchmarks. The other two configurations shown in the graph do not have this consistent behaviour, though. The configuration with starting point at 3/4 of the maximum block size has better performance than the 1/2 and baseline configurations in all benchmarks but gcc and go. The configuration that starts issuing VLIW fetches only at full block has a performance inferior to the previous three configurations discussed in the gcc and go benchmarks; it has also a worse or almost the same performance than the 3/4 configuration in vortex and compress, respectively. This happens because, with this configuration, the DTSVLIW does not have many opportunities to perform VLIW fetches and spends too much time executing code in the Primary Processor, since VLIW fetches are only allowed when the block is full.

The benchmarks compress, gcc, go, and vortex show clearly that there is a performance maximum when the VLIW fetch starting point is near the 1/2 – 3/4 range of block sizes. The harmonic means confirm this and show a DTSVLIW performance 8.3% and 9.1% better than the baseline for the 1/2 and 3/4 configurations, respectively. For a silicon implementation, a larger set of experiments must be made in order to determine the adequate VLIW fetch stating point.

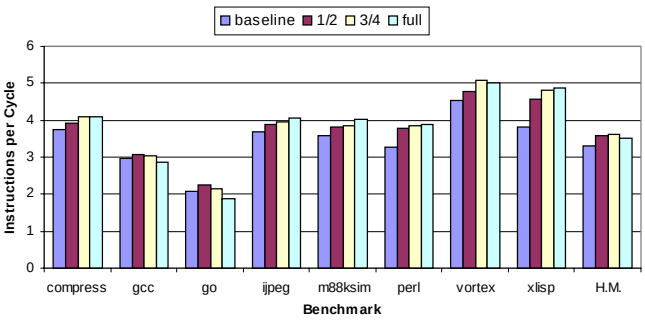


Fig.4 Impact of VLIW fetch starting point.

### *B. Effect of Compacted Blocks*

The graph in Figure 5 shows the impact of the compacting block mechanism on DTSVLIW performance. In order to allow visual comparison with the DTSVLIW performance results shown in the previous subsection, we have added to Figure 5 the previous baseline configuration results, named baseline in the legend, and best performing configuration results, 3/4 in the legend. The cbaseline results shown in Figure 5 are of the DTSVLIW basic configuration described in Table 2 enhanced with the compacting block mechanism, while the c3/4 results are of the DTSVLIW configuration that put together a VLIW fetch starting point at three quarter of block size and the compacting block mechanism. Note that the block size used now is not the number of long instructions in the block but the actual number of instructions in the block – the block size is limited to 16 *compacted long instructions*, but blocks may have more than 16 long instructions.

The experimental results present in Figure 5 confirm that the compacting block mechanism improves the DTSVLIW performance: the c3/4 configuration performs better than the others in all but the gcc and go benchmarks, and looses in these two for a small margin. On average, the c3/4 configuration performs 13.2% better than the baseline – an improvement of 4.1% against the previous best performing DTSVLIW without compacting blocks (13.2% - 9.1%).

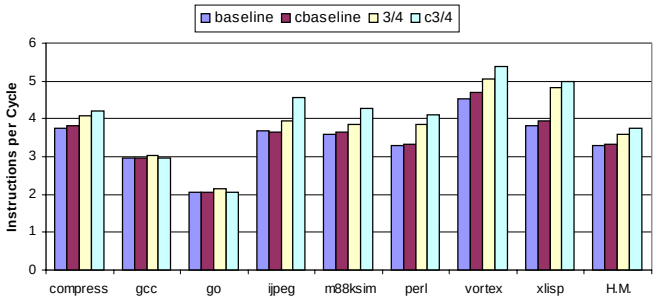


Fig.5 Impact of the compacting block mechanism.

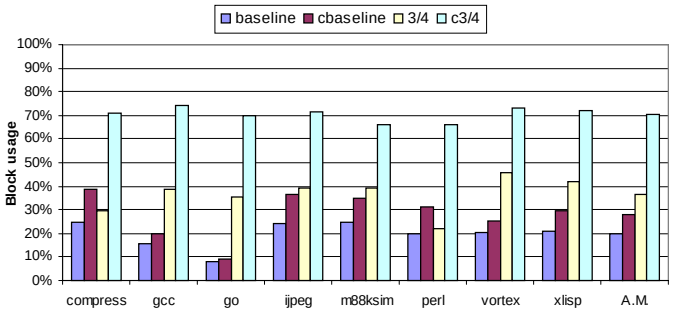


Fig.6 Percentage of block usage.

Each VLIW Cache block of the configurations under study can hold 16x16 instructions maximum (256 instructions). Figure 6 shows, for the same DTSVLIW configurations shown in Figure 5, the percentage of this maximum capacity that is actually used. We use arithmetic mean (A.M.) in Figure 6, since the data shown are just percentages and not ratio of rates [JAC 95], as is the case of the previous figures. As the graph in Figure 6 shows, both compacting block mechanisms improve block capacity usage, especially when used together, as in the c3/4 configuration. In the baseline configuration, only 19.9% of block capacity is used on average, while in the cbaseline, 28.1%. In the 3/4 configuration, 36.4% of block capacity is used on average and in the c3/4, 70.4%. Although these improvements in block capacity usage do correlate to performance in m88ksim, vortex and xlisp, this is not the case for the other programs under study, since configurations with better block usage does not show better performance. This can be explained with the help of Figure 7.

In order to execute a program, a DTSVLIW machine spends cycles executing code either in the Primary Processor or in the VLIW Engine. Figure 7 shows the percentage of cycles each DTSVLIW configuration spent executing code in the Primary Processor for each benchmark program. As the graph in Figure 7 shows, except for gcc, go and vortex, the DTSVLIW spent more than 95% of the time executing code in the VLIW Engine. The time spent scheduling impacts on performance, since the DTSVLIW executes one instruction per cycle maximum during scheduling. The compacting block mechanisms are effective for the gcc and go benchmarks (Figure 6); however, they significantly increase the scheduling time (Figure 7), which take back any performance improvement provided by block compaction. The compress and perl benchmarks show better compaction with the cbaseline configuration than with the 3/4 configuration and this does not translates to performance. As Figure 7 shows, the DTSVLIW spent too little time scheduling these benchmarks for the cbaseline configuration (less than 0.3% of the cycles) and just a few blocks were necessary for their whole execution – these blocks were better compacted with this configuration than with the 3/4 configuration just by chance.

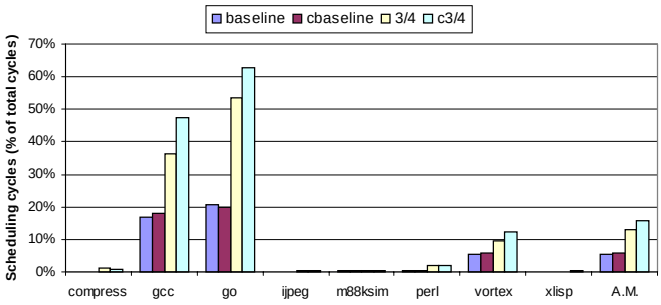


Fig.7 Percentage of scheduling cycles.

## V. RELATED WORK

In order to take advantage of code execution locality to exploit ILP, several techniques have been proposed. Franklin and Smotherman [FRA 94] investigated the use of a fill unit [MEL 88] to compact a dynamic stream of scalar instructions into long instructions. The fill unit accepts decoded instructions from the machine decoder, compacts them into a long instruction, and saves the 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 that hit in the shadow cache provide long instructions directly to the functional units. However, the fill unit does not rename registers, resulting in a reduction in the capacity to deal with data dependencies, and works within a window of one long instruction only. For these reasons, it cannot exploit ILP extensively. Nair and Hopkins [NAI 97] suggested a VLIW based architecture named DIF (Dynamic Instruction Formatting), which is an improvement of the Franklin and Smotherman proposal. It allows register renaming and schedules blocks of long instructions. The DTSVLIW architecture is similar to the DIF and was developed shortly after it. The DTSVLIW performance is equivalent to DIF's, but this is achieved with fewer hardware resources [DES 99a]. The core logic of the DTSVLIW's Scheduler Unit is straightforward to implement, being comparable to an adder,

and, as such, is much more feasible than that of the DIF [DES 99a].

In *dynamic rescheduling*, proposed by Conte and Sathaye [CON 95], when a program is invoked the operating system schedules its first page into a more parallel piece of code and saves it into a new page. This process is repeated each time a new page fault occurs. Only scheduled pages are executed. Ebcioglu and Altman [EBC 97] extended the concept of dynamic rescheduling to dynamic compilation, in order to translate and schedule any ISA to the machine's hardware. The Crusoe processors [KLA 00], for example, use dynamic compilation to translate x86 (Intel) code to a VLIW ISA code dynamically. Dynamic rescheduling and dynamic compilation rely on the ability of the operating system to translate and schedule code rapidly, and on the reusability of this code. However, since they are implemented in software, the cost of translating and scheduling are high. The DTSVLIW uses hardware for scheduling and, as such, should provide better performance.

Discarding some VLIW Cache fetch opportunities to favour the production of larger blocks, as a mechanism of improving performance, was first studied in [NAI 97] for the DIF architecture. Here we have extended this study to the DTSVLIW and shown that it improves performance and block usage as well. The mechanism we have proposed for coding the compacted long instructions is similar to the 5-bit *template field* of the Intel's IA64 ISA instruction *bundles* [INT 99]. However, these template fields specify the mapping of instruction slots to execution unit types and stops within the bundles, while the control fields of our compacted instructions only specify stops.

## VI. CONCLUSIONS

DTSVLIW machines exploit ILP using a hardware implemented dynamic code scheduler and a VLIW execution core. The scheduler produces blocks of VLIW instructions by executing and scheduling one instruction at a time. These blocks are saved in a VLIW cache and thereafter executed by the VLIW core, which realizes the parallelism found during scheduling. Code execution locality compensates for the time spent scheduling. However, the scheduler is not able to fill the blocks completely – only 19.9% of the blocks capacity is used in the standard DTSVLIW architecture. In this paper, we have studied two mechanisms for improving block usage: to hold VLIW fetch attempts until the blocks being scheduled reach a certain number of VLIW instructions, and to remove nop instructions from the blocks before saving them in the VLIW cache. Our experiments show that these mechanisms, used together, increase block usage to 70.4% of their maximum capacity and improve DTSVLIW performance in 13.2% on average.

Better used blocks provide better performance because they contain more scheduling effort. However, this effort is made sequentially and, in programs with poor execution locality, like gcc and go, may not pay off. Nevertheless, according to our experiments, this extra scheduling effort does not hurt the DTSVLIW performance even in these cases.

## REFERENCES


[AUS 97] T. Austin and D. Burger, "The SimpleScalar Tool Set", Technical Report TR-1342, Computer Science Department, University of Wisconsin – Madison, June 1997.

[CHA 97] 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, pp. 265-285, May 1997.

[CON 95] T. M. Conte and S. W. Sathaye, "Dynamic Rescheduling: A Technique for Object Code Compatibility in VLIW Architectures," *Proc. of the 28th Ann. Int. Symp. on Microarchitecture*, pp. 208-218, 1995.

[DAV 81] 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.

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

[DES 99a] 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.

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

[DES 99c] A. F. de Souza and P. Rounce, "On the Effectiveness of the Scheduling Algorithm of the Dynamically Trace Scheduled VLIW Architecture", *Proc. of the 11th Brazilian Symp. on Computer Architecture and High Performance Computing*, pp. 167-174, 1999.

[DES 99d] A. F. de Souza, "Integer Performance Evaluation of the Dynamically Trace Scheduled VLIW Architecture", PhD Thesis, Department of Computer Science, University College London, 1999.

[DES 00a] A. F. de Souza and P. Rounce, "DTSVLIW: VLIW Performance with Sequential Code", *Proc. of the 12th Brazilian Symp. on Computer Architecture and High Performance Computing*, pp. 271-278, 2000.

[DES 00b] A. F. de Souza and P. Rounce, "Dynamically Scheduling VLIW Instructions", J. of Parallel and Distributed Computing **60**, pp. 1480-1511, December 2000.

[DIG 92] Digital Equipment Corporation, "Alpha Architecture Handbook", Digital Equipment Corporation, 1992.

[EBC 97] K. Ebcioglu and E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility," *Proc. of the 24th Ann. Int. Symp. on Computer Architecture*, pp. 26-37, 1997.

[FIS 81] J. A. Fisher, "Trace Scheduling: A Technique for Global Microcode Compaction", IEEE Trans. on Computers, Vol. C-30, No. 7, pp. 478-490, 1981.

[FIS 84] J. A. Fisher, "The VLIW Machine: A Multiprocessor

for Compiling Scientific Code", IEEE Computer, pp. 45-53, July 1984.

[FRA 94] M. Franklin and M. Smotherman, "A Fill-Unit Approach to Multiple Instruction Issue," *Proc. of the 27th Ann. Int. Symp. on Microarchitecture*, pp. 162-171, December 1994.

[GEE 93] 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.

[HWU 93] W. W. Hwu, et al., "An effective Technique for VLIW and Superscalar Compilation", J. of Supercomputing, Vol. 7, pp. 229-248, 1993.

[INT 99] Intel, "IA-64 Application Developer's Architecture Guide", Intel, Order Number: 245188-001, May 1999.

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

[JOH 91] M. Johnson, "Superscalar Microprocessor Design", Prentice-Hall, 1991.

[KLA 00] A. Klaiber, "The Technology Behind Crusoe™ Processors", Transmeta Corporation, January 2000.

[MEL 88] S. Melvin, M. Shebanow, and Y. Patt, "Hardware Support for Large Atomic Units in Dynamic Scheduled Machines," *Proc. of the 21st Ann. Int. Symp. on Microarchitecture*, pp. 60-66, 1988.

[NAI 97] 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.