DTSVLIW: VLIW Performance with Sequential Code

Alberto Ferreira de Souza[1] and Peter Rounce[2]

[1] Departamento de Informática
Universidade Federal do Espírito Santo
Av. Fernando Ferrari, S/N – 29060-970 – Vitória – ES – Brazil
{alberto@inf.ufes.br}
[2] Department of Computer Science
University College London
Gower Street, London WC1E 6BT - UK
{p.rounce@cs.ucl.ac.uk}

Abstract—

Due to the temporal execution locality present in programs, even small instruction caches (16-Kbyte) can provide processors with fast access to instructions most of the time. The *Dynamically Trace Scheduled VLIW* (DTSVLIW) architecture exploits programs' temporal execution locality by executing code in two distinct modes. In the first execution encounter, fragments of the code are executed in sequential mode (in a simple pipelined processor), scheduled into blocks of VLIW instructions and cached in a VLIW cache by the DTSVLIW's Scheduler Engine. In subsequent encounters, the DTSVLIW's VLIW Engine executes these blocks in VLIW mode. In this paper, we present experiments which show that DTSVLIW machines can perform better than Superscalar machines with equivalent hardware and better than VLIWs with the same degree of parallelism, while keeping the fast clock of the latter. We also discuss how the DTSVLIW compares with the Trace Cache and EPIC architectures.

Keywords— **DTSVLIW, EPIC, Superscalar, VLIW**

I. INTRODUCTION

An *Instruction Set Architecture* (ISA) is a contract between a class of programs and a set of processor implementations [RAU 93a]. Usually, this contract is concerned with the instructions format and the interpretation of the bits that constitute each instruction. However, in the case of systems that exploit *Instruction-Level Parallelism* (ILP), this contract extends to information embedded in the programs regarding the available parallelism between the instructions of the programs. Special functions are performed by ILP exploiting systems in order to find and take advantage of ILP, and different forms of ISA contract divide these functions between the compiler and the hardware differently. These functions can be summarised as follows:

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

Figure 1 shows some forms of ISA contract for ILP exploiting systems.

Superscalar [JOH 91] machines execute sequential ISA code and exploit ILP; therefore, their hardware has to determine dependencies and independencies between several instructions, and bind several instructions to resources at the same time, dynamically. The hardware for doing this is in the main data path of Superscalar machines. Because of this, Superscalar machines with elaborate instruction scheduling hardware have slower clocks than simpler Superscalar machines, and the latter may have a better performance than the former due to their fast clocks [SMI 94].

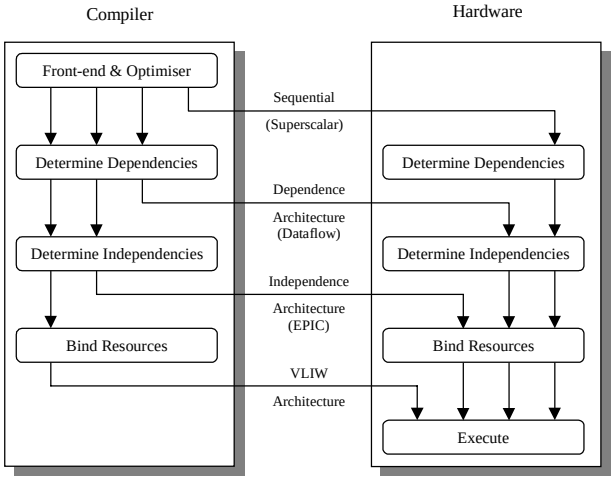

Fig.1 Different forms of ISA contract

Superscalar and *Very Long Instruction Word* (VLIW) [FIS 84] machines are at opposite extremes regarding ILP exploitation. In standard VLIW systems, the compiler is responsible for all functions that have to be performed in order

to exploit ILP. This allows the VLIW hardware to be simpler and faster than the Superscalar's, but makes the VLIW ISA contract very restrictive. Developments in compiler or hardware technology following a VLIW ISA specification may allow for greater parallelism than that which can be expressed within this VLIW ISA specification. To take advantage of these developments, the VLIW ISA may have to be changed; this creates the VLIW object-code compatibility problem [RAU 93b].

Dynamically Trace Scheduled VLIW (DTSVLIW) machines [DES 98] execute sequential code and exploit ILP. Their hardware, similar to that of Superscalars, has to determine dependencies and independencies between instructions and to bind instructions to resources dynamically. However, different from Superscalars, DTSVLIW machines perform the functions related to ILP exploitation with one instruction at a time, producing VLIW code that is cached and thereafter executed many times.

Figure 2 shows a block diagram of the DTSVLIW architecture. In a DTSVLIW machine, the Scheduler Engine fetches instructions from the Instruction Cache and executes them the first time using a simple pipelined processor – the Primary Processor. In addition, its Scheduler Unit dynamically schedules the trace produced during 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 fetched by the VLIW Engine from this cache and executed in a VLIW fashion. In a DTSVLIW machine, the Scheduler Engine provides object-code compatibility, and the VLIW Engine provides VLIW performance and simplicity.

The DTSVLIW architecture is a variant of the DIF architecture [NAI 97], proposed by Nair and Hopkins. However, 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 in that there is no significant reduction in performance over the DIF algorithm, even though the latter is expected to be much more difficult to implement.

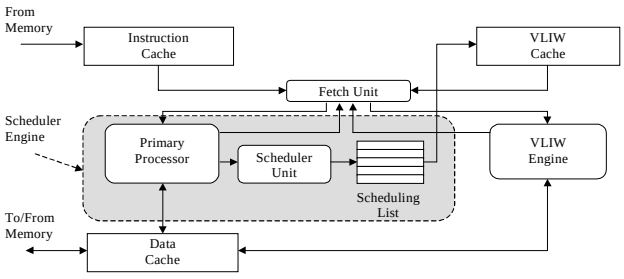

Fig.2 The Dynamically Trace Scheduled VLIW Architecture

Our main motivation for the development 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 strong temporal execution locality in programs. The DTSVLIW exploits temporal execution locality by scheduling the code into blocks of VLIW instructions on the first execution encounter and by executing it in the VLIW Engine on subsequent encounters. In this paper, we present experiments which show that, because of their characteristics, DTSVLIW machines can perform better than Superscalar machines with equivalent hardware and better than VLIWs with the same degree of parallelism, while keeping the fast clock of the latter. We also discuss how the DTSVLIW compares with the Trace Cache [ROT 96] and *Explicit Parallel Instruction Computing* (EPIC) [GWE 97] architectures.

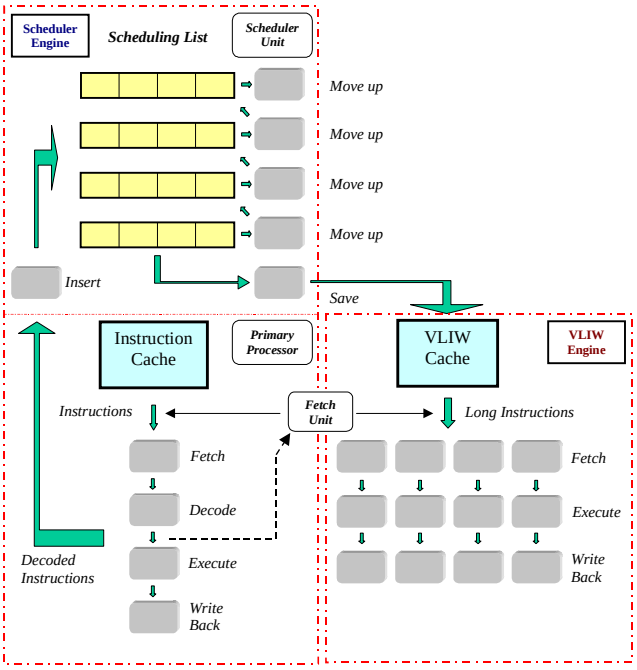

Fig.3 A DTSVLIW Machine

II. THE DTSVLIW ARCHITECTURE

The symbolic diagram of a DTSVLIW machine is shown in Figure 3. 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 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 3) 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 3). 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 DTSVLIW implementation, the Primary Processor executes Sparc-7 ISA [SUN 87] code, while the VLIW Engine executes a sub-set. The VLIW Engine has a simple fetch – execute – write-back pipeline for each functional unit (multicycle instructions execute in pipelined functional units). A decode stage is not necessary as decoded instructions are saved in the VLIW Cache. The VLIW Cache is a simple set-associative cache, where a block of long instructions occupies a single cache line. Individual long instructions are the unit of communication between the VLIW Cache and the rest of the DTSVLIW. Details of how the DTSVLIW deals with exceptions, memory aliasing (disambiguation), and the execution of particular instructions, e.g. Sparc save and restore, are in [DES 99a].

A. The Scheduler Engine

The Scheduler Engine is composed of the Primary Processor plus the Scheduler Unit (Figure 3). The Primary Processor is a simple pipelined processor capable of executing all instructions of the Sparc-7 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, 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 from 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 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 [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 3). 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.

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.

Conditional and indirect branches do not move up. They are installed when inserted 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.

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 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 line and is identified by the address of the first instruction installed in it. Each cache line holds this address and the address of the following block.

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.

For more details about the DTSVLIW architecture please refer to [DES 00], where preliminary DTSVLIW's performance results are also presented.

TABLE 1
FIXED PARAMETERS

Primary Processor	• four-stage (fetch, decode, execute, and write back) pipeline • no branch prediction hardware • taken branches cause a 2-cycle bubble in the pipeline
Decoded Instruction Size	6 bytes

TABLE 2
BENCHMARK PROGRAMS

SPEC92 Benchmarks	***Inputs***	***SPEC95 Benchmarks***	***Inputs***
compress	in	go	40 19 null.in
eqntott	int_pri_3.eqn	m88ksim	dhry.big
espresso	cps.in		
gcc	-O jump.i		
xlisp	queens 7		

III. EXPERIMENTS

A simulator of the DTSVLIW has been implemented in C, and execution-driven simulation performed to produce the results reported here. All results were produced with the simulator running in *test mode* in order to guarantee correct simulation. 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. Execution of these alternates and the Sparc ISA state of both machines is compared regularly. If it is not equal, an error is signalled and the simulation interrupted. The test mode has been very useful because in this mode it is possible to measure the precise number of instructions necessary for the execution of a program. A DTSVLIW simulator alone cannot provide this number due to copy instructions and instructions executed speculatively. The *instruction per cycle* performance measurement index used here is produced by dividing the number of instructions executed by the test machine, by the number of cycles consumed by the DTSVLIW execution.

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

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

A. *DTSVLIW versus PowerPC620*

The graph in Figure 4 shows a comparison between the performance of the PowerPC620 Superscalar processor, as described in [DIE 95], and a DTSVLIW processor with equivalent characteristics, when running four programs of the SPECint92. The PowerPC620 performance figures were taken from [DIE 95], and the parameters used there were: 4-instruction wide fetch, dispatch, complete, and writeback pipeline stages; single cycle integer, 2 cycles load, and 3 cycles floating point instruction latency; 3 integer, 1 load/store, 1 floating point, and 1 branch functional units, with 2, 3, 2, and 4 reservation stations for each functional unit of each kind, respectively; 32-Kbyte, 8-way set associative instruction and data L1 caches, with an 8-cycle miss penalty (a perfect unified L2 cache was assumed); and a branch predictor with a 256-entry two-way branch target buffer (BTB) and a 2048-entry (two-bit counters) direct mapped branch history table (BHT).

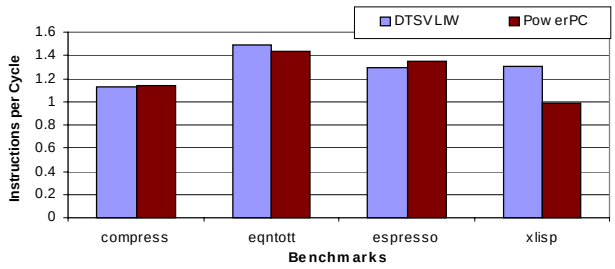

Fig.4 DTSVLIW versus PowerPC620

The DTSVLIW was configured with a 4x8-block (four instructions wide long instructions and eight long instructions deep) and functional units of the same type, same latency, and in the same number as the PowerPC620. Although six functional units are available in the VLIW Engine with this configuration, we have used 4-instruction wide long instructions in the DTSVLIW to give the same dispatch width for both machines. An extra dispatch pipeline stage was added to the VLIW Engine pipeline to account for the logic necessary to unpack the long instructions coming from the VLIW Cache and dispatch them to the appropriate functional units. A branch predictor with the same characteristics of the PowerPC620's was used to try to reduce the extra cost added to next long instruction fetch misses by this dispatch stage. An 8-Kbyte, 8-way set associative instruction cache, and a 24-Kbyte, 8-way set associative VLIW Cache were used. These sizes were chosen to make this pair equivalent to the instruction cache of

the PowerPC620. The DTSVLIW data cache was configured with the same characteristics of the PowePC620's. In this simulation, the DTSVLIW was allowed to execute the same number of instructions executed in the PowerPC620 simulation [DIE 95].

As Figure 4 shows, the performance of the two machines is comparable, although the DTSVLIW performance is better overall. This is so because the scheduling list of the DTSVLIW is larger than the instruction window (all reservation stations) of the PowerPC620, which allows more opportunities for finding ILP. Although larger than the instruction window of the PowerPC620 (15 instructions), the scheduling list of the DTSVLIW (32 instructions) is simpler. The complexity of the instruction window of the PowerPC620 is proportional to the number of reservation stations times the number of functional units (15 * 6 = 80, see [JOH 91]), while the complexity of the scheduling list of the DTSVLIW is proportional to the number of candidate instructions times the number of instructions per long instruction (8 * 4 = 32, see [DES 99d]). Note that a DTSVLIW implementation is likely to have a significantly higher clock rate than a PowerPC620, because, different from the PowerPC620's, the DTSVLIW scheduling hardware is not in its main data path. In addition, due to the instruction fetch bandwidth problem of Superscalar machines (see Section IV), the PowerPC620 performance in terms of ILP can be seen as a high-end performance for standard Superscalar machines. The DTSVLIW described in this section, on the other hand, is a low-end DTSVLIW and larger DTSVLIW configurations can be implemented with increasing performance returns [DES 00].

B. *DTSVLIW versus VLIW*

The VLIW research group at IBM is a leading group on VLIW compiler and architecture technologies. In [MOR 97] they have presented experimental performance results of various VLIW configurations using a powerful VLIW compiler. In Figure 5, we show the performance figures for programs from the SPECint92 and SPECint95 when running in two DTSVLIW configurations and in two IBM VLIWs described in [MOR 97] and [MOU 96]. The benchmarks m88ksim and go are from SPECint95 while the others are from the SPECint92. One VLIW configuration used in the IBM experiments has been set with 8 *untyped functional units* (able to execute all instructions) and the other with 16. The instruction latencies have been set at 1-cycle for integer (including load/store), 3-cycle for integer multiply, 10-cycle for integer divide, and 3-cycle for floating-point. The number of added registers for renaming has been 64-integer, 64-floating-point, and 16-condition in the 8-wide configuration, and 128-integer, 128-floating-point, and 32-condition in the 16-wide configuration. The experiments have been performed with perfect instruction and data caches (no miss penalty).

We have configured two DTSVLIW machines with parameters identical or equivalent to those used in the two IBM VLIW machines. One DTSVLIW configuration has an 8x8-block and the other a 16x16-block, both with untyped functional units. The instruction latencies have been all set at 1-cycle, which is a value lower than that used in the IBM's experiments for integer multiply, integer divide, and floating-point instructions. However, the Sparc-7 ISA does not have integer multiply or divide instructions but only multiply-step, which can execute in one cycle. Since the benchmarks are all integers, the number of floating-point instructions executed is zero or negligible; therefore, the different latencies used for floating-point instructions do not constitute a problem. The number of renaming registers used during the DTSVLIW simulations has never exceeded the number used in the IBM's simulations in any combination of benchmark program and machine configuration. Our experiments have also been performed with perfect instruction and the data caches.

As the graph in Figure 5 shows, the VLIW outperforms the DTSVLIW in compress and eqntott for a large margin. However, for gcc, go, m88ksim, and xlisp both DTSVLIW machine configurations have consistently better performance. These results show that, in most cases, the DTSVLIW algorithm is able to find more parallelism than a state-of-the-art VLIW compiler under similar conditions. This is possible because the DTSVLIW scheduler algorithm has access to run time information not available to the VLIW compiler. We believe that a conjunction of the DTSVLIW architecture and compiler technology, such as loop unrolling, software pipeline, and predication, would perform even better than shown, in particular with compress and eqntott. Published results corroborate this view, showing that the use of such optimisations does significantly improve performance, in particular the use of predication in the eqntott and compress benchmarks [AUG 98]. Other compiler techniques could also be developed specifically to the DTSVLIW.

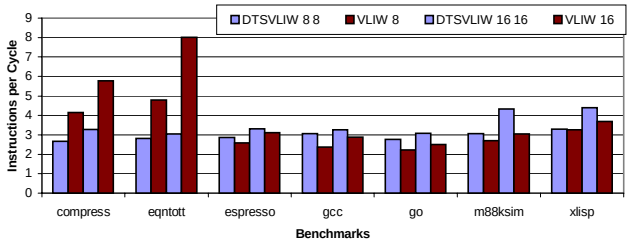

Fig.5 DTSVLIW versus VLIW

IV. DISCUSSION

In the integer programs of the SPEC92 benchmark suite, on average 19% of the executed instructions are branches [PAT 96 (page 105)] and on average 62% of them change the control flow [PAT 96 (page 166)]. This means that about 12% of the instructions executed in these programs change the control flow – almost one in eight. Current Superscalar machines fetch up to four instructions each clock cycle. For the next generation, it is going to be possible to fetch eight or more instructions per cycle. This means that almost every fetch will contain a branch that will change the control flow. Since these branches are distributed evenly throughout the address space, many of these fetch cycles (almost half for an 8-wide fetch) will be only partially effective. In addition, instead of

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

Several high bandwidth fetch mechanisms based on the conventional instruction cache have been proposed [CON 95, SEZ 96, YEH 93]. In such mechanisms, on every cycle instructions from non-contiguous locations in the instruction cache are fetched and assembled into dynamic sequences using information collected by dynamic branch predictors. To do this, branch target tables are inspected and pointers are generated to all non-contiguous instruction blocks. A moderately to highly interleaved instruction cache is accessed and provides multiple lines simultaneously. These lines are aligned by an alignment network, which then sends the instructions to the decode stage of the Superscalar processor.

The disadvantage of these high bandwidth fetch mechanisms is their complexity. Sophisticated dynamic branch predictors, interleaved multiport instruction caches, and complex alignment networks are required to make them work.

A. DTSVLIW versus Enhanced Superscalar Architectures

The *Trace Cache* architecture avoids the complexity of high bandwidth fetch mechanisms by caching dynamic instruction sequences, rather than only the information for constructing them [ROT 96]. A machine that follows this architecture fetches instructions from the instruction cache and attempts to schedule them across multiple functional units exactly as a Superscalar. These instructions are then grouped by a Fill Unit [MEL 88] and placed in a trace cache, which stores them in execution order, as opposed to the static order determined by the compiler. On an instruction fetch, the trace cache will provide a line of instructions if available. This line can encompass more than one line from the instruction cache through merging of lines affected by partial fetches caused by taken branches: this increases instruction fetch throughput.

Trace Cache architectures are basically Superscalar architectures enhanced with a Fill Unit and a trace cache. Their Fill Unit is equivalent to the Scheduler Unit of DTSVLIWs. However, the Scheduler Unit performs all functions necessary to ILP extraction on a DTSVLIW machine. In contrast, Superscalars implemented according to the standard Trace Cache architecture use their Fill Unit only to ameliorate the fetch bottleneck. Nevertheless, the DTSVLIW and Trace Cache can be seen as members of the same family of architectures. They can be put in a scale of how many ILP extraction functions are performed by their Scheduler Unit/Fill Unit and how many of these functions are performed by their parallel execution-core. Figure 6 shows such a scale and the positions of the DTSVLIW and Trace Cache architectures.

Like standard VLIW and Superscalar, the DTSVLIW and Trace Cache architectures are at opposite extremes. As shown in Figure 6, the Trace Cache architecture's execution-core performs all ILP extraction functions, while the DTSVLIW's performs none. On the other hand, the DTSVLIW's Scheduler Unit performs all ILP extraction functions, while the Trace Cache architecture's Fill Unit performs none.

One might argue that the DTSVLIW sequential execution during the scheduling phase would strongly affect its performance. However, in the DTSVLIW or in any architecture that always operate in parallel, such as the Trace Cache, the first time a fragment of code is executed it is likely to cause data and instruction cache misses that will determine the performance during its execution. In addition, no machine has comprehensive information about the dynamic behaviour of branches of fresh code fragments, and the more parallel the machine the larger the effect of branch mispredictions. Furthermore, our results show that most of the time the DTSVLIW is executing code in VLIW-mode if the VLIW Cache is large enough to hold pre-scheduled code [DES 99a]. Results with the Trace Cache architecture also show that the code is found in the trace cache most of the time [ROT 97].

Real implementations usually distance themselves from standard architecture definitions due to real world constraints. For example, in the DTSVLIW configuration compared with the PowerPC620 in Section III-A, we have left to the execution-core the task of dispatching the instructions to specific functional units. Researchers working with the Trace Cache architecture have also suggested moving some ILP related functions from the execution-core to the Fill Unit in order to allow implementations with fast clock [ROT 97, VAJ 97, FRI 98]. Other Superscalar enhancements such as value prediction [FU 98, NAK 99] and instruction reuse [SOD 97] may also be incorporated into both architectures.

Although future research may produce variants of the DTSVLIW and Trace Cache architectures that are more closely together in the scales shown in Figure 6, their pure definitions still represent useful models to understand the nature of the ILP that can be exploited dynamically. Ultimately, the available VLSI technology and design expertise will determine which of the two extremes represented by the DTSVLIW and Trace Cache architectures will reach the marketplace. We believe, however, that a point close to the DTSVLIW's in the scales of the Figure 6 is a better option. This view is corroborated by the evolution of Alpha microprocessors. They implement the simplest RISC ISA (which means simple execution-core) in the mass microprocessor market and are "*performance leaders since their introduction in 1992*" [KES 99] mainly, we believe, due to their simple high-clock-speed-tailored implementation.

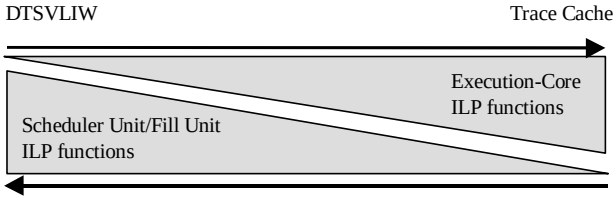

Fig.6 Distribution of ILP related functions

B. DTSVLIW versus EPIC

The term Explicitly Parallel Instruction Computing (EPIC) was coined recently by Intel and Hewlett Packard in their joint

announcement of the IA-64 ISA [GWE 97]. EPIC ISAs require the compiler to determine the dependencies and the independencies between instructions (Figure 1). However, the hardware interpreter of an EPIC ISA is responsible for binding the independent instructions, specified by the compiler, to the machine resources. This is in contrast with VLIW systems where the compiler is responsible for determining dependencies and independencies, and also for binding resources.

In the Intel/Hewlett Packard EPIC ISA, or IA-64 for short, each three instructions are grouped together into 128-bit sized and aligned containers called *bundles*. Each bundle contains three 41-bit *instruction slots* and a 5-bit *template field* [INT 99]. The template field specifies two properties: the mapping of instruction slots to execution unit types, and stops within the current bundle. The stops define *instruction groups*. An instruction group is a sequence of instructions starting at a given bundle address and slot number and including all instructions at sequentially increasing slot numbers and bundle addresses up to the first stop or taken branch. Therefore, instruction groups can encompass several bundles. The IA-64 specifies dependency restrictions that allow the processor to execute all (or any subset) instructions within a legal instruction group in parallel or serially with the end result being identical. That is, two or more instructions within a group should not write to the same register (output dependency), or read from and write to the same register (true and anti dependencies). Memory reads and writes to the same address are allowed within groups, however. The IA-64 compiler has to generate code according to these dependency restrictions. To facilitate this task, the IA-64 exposes to the compiler a large number of registers: 128 integer, 128 floating-point, 64 predicate, and a large number of other registers.

Conditional branch instructions use *predicate registers* to decide their outcome. In addition, these registers are used to implement predicated execution (see [DUL 98] for details). Instruction hoisting ([DUL 98]) is also supported by IA-64.

Because the compiler specifies the dependencies and independencies between instructions, the IA-64 processor only has to fetch one or more bundles, to identify the stops, and to send the independent instructions to the respective functional units. Memory dependencies can be dealt with via compiler assisted memory disambiguation, compiler and ISA assisted speculative memory access [INT 99], and the use of an *Address Resolution Buffer* (ARB) [FRA 92]. The new IA-64 processor still executes legacy 8086 ISA code and all is upgrades, but it does so using special modes that may not take full advantage of the new EPIC processor core [INT 99].

The EPIC architecture relies on compiler technologies such as predication, instruction hoisting, loop unrolling, software pipelining, etc, to exploit the ILP available in programs. The DTSVLIW architecture, on the other hand, does not rely on the compiler to exploit ILP and can achieve performance executing legacy sequential code. In addition, the DTSVLIW architecture can be employed to emulate legacy ISA code in IA-64 processors, taking advantage of the EPIC core. It can also be used to execute EPIC code directly, collecting dynamic branch behaviour information and organising the code during the scheduling phase of program execution to increase the processor performance.

V. CONCLUSIONS

The DTSVLIW architecture can be used to implement machines that execute sequential code in a VLIW fashion, delivering ILP with backward code compatibility. The DTSVLIW takes advantage of code execution locality by executing programs in two distinct modes: one sequential, in its Scheduler Engine, the other (much more frequent) parallel, in its VLIW Engine. Our experimental results confirm that the DTSVLIW architecture takes advantage of code execution locality by showing that its performance is overall better than the Superscalar's and VLIW's. The DTSVLIW performs better than the Superscalar because its scheduling list is larger than the instruction window of the Superscalar. It performs better than the VLIW because it has access to run time information not available to the VLIW compiler.

We argue that DTSVLIW and Trace Cache are members of the same family of architectures. Their main difference is in how they divide the ILP extraction functions internally. The Trace Cache architecture's execution-core performs all ILP extraction functions, while the DTSVLIW's execution-core performs none. On the other hand, the DTSVLIW's Scheduler Unit performs all ILP extraction functions and the Trace Cache architecture's Fill Unit performs none. We believe that simplicity in the main execution-core results in overall better performance. This view is corroborated by the evolution of the Alpha family of microprocessors, "*performance leaders since their introduction in 1992*" [KES 99].

EPIC architectures rely on the compiler to expose ILP to their hardware. The DTSVLIW architecture does not; in addition, it can be employed to emulate legacy code in new EPIC processors, taking advantage of the EPIC core. It can also be used to execute EPIC code directly, collecting run time information and organising the code during the scheduling to increase performance.

VI. REFERENCES

[AUG 98] D. I. August et al., "Integrated Predicated and Speculative Execution in the IMPACT EPIC Architecture", Proc. of the 25th Int. Symp. on Computer Architecture, pp. 227-237, 1998.

[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, May 1997.

[CON 95] T. M. Conte, K. N. Menezes, P. M. Mills, and B. A. Patel, "Optimization of Instruction Fetch Mechanisms for High Issue Rates", Proc. of the 22nd Int. Symp. on Computer Architecture, pp. 333-344, 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. (www.ippsxx.org/library.htm)

[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 00] A. F. de Souza and P. Rounce, "Dynamically Scheduling VLIW Instructions", to appear in the J. of Parallel and Distributed Computing, 2000.

[DIE 95] T. A. Diep, C. Nelson, and J. P. Shen, "Performance Evaluation of the PowerPC620 Microarchitecture", Proc. of the 22nd Int. Symp. on Computer Architecture, pp. 163-174, 1995.

[DUL 98] C. Dulong, "The IA-64 Architecture at Work", IEEE Computer, pp. 24-31, July 1998.

[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 92] M. Franklin and G. S. Sohi, "The Expandable Split Window Paradigm for Exploiting Fine-Grain Parallelism", Proc. of the 19th Int. Symp. on Computer Architecture, pp. 58-67, 1992.

[FRI 98] D. H. Friendly, S. J. Patel, and Y. N. Patt, "Putting the Fill Unit to Work: Dynamic Optimizations for Trace Cache Microprocessors", Proc. of the 31st Int. Symp. on Microarchitecture, pp. 173-181, 1998.

[FU 98] C.-H. Fu, M. D. Jennings, S. Y. Larin, and T. M. Conte, "Value Speculation Scheduling for High Performance Processors", Proc. of the 8th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 262-271, 1998.

[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.

[GWE 97] L. Gwennap, "Intel, HP make EPIC Disclosure", Microprocessor Report, Vol. 11, No. 14, pp. 1-9, October 27, 1997.

[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 Corp., Order No. 245188-001, May 1999.

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

[KES 99] R. E. Kessler, "The Alpha 21264 Microprocessor", IEEE Micro, pp. 24-36, March-April 1999.

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

[MOR 97] J. H. Moreno et al., "Simulation/Evaluation Environment for a VLIW Processor Architecture", IBM J. of Res. and Dev., Vol. 41, No. 3, May 1997.

[MOU 96] M. Moudgill et al., "Compiler/Architecture Interaction in a Tree-Based VLIW Processor", IBM Research Report RC20694, November 1996.

[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.

[NAK 99] T. Nakra, R. Gupta, and M. L. Soffa, "Value Prediction in VLIW Machines", Proc. of the 26th Int. Symp. on Computer Architecture, pp. 258-269, 1999.

[PAT 96] D. A. Patterson and J. L Hennessy, "Computer Architecture: A Quantitative Approach, Second Edition", Morgan Kaufmann Publishers, Inc., 1996.

[RAU 93a] B. R. Rau and J. A. Fisher, "Instruction-Level Parallelism: History, Overview, and Perspective", The J. of Supercomputing, Vol. 7, pp. 9-50, 1993.

[RAU 93b] B. R. Rau, "Dynamically Scheduled VLIW Processors", Proc. of the 26th Int. Symp. on Microarchitecture, pp. 80-92, 1993.

[ROT 96] E. Rotenberg, S. Bennett, and J. E. Smith, "Trace Cache: A Low Latency Approach to High Bandwidth Instruction Fetching", Proc. of the 29th Int. Symp. on Microarchitecture, pp. 24-34, 1996.

[ROT 97] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. E. Smith, "Trace Processors", Proc. of the 30th Int. Symp. on Microarchitecture, pp. 138-148, 1997.

[SEZ 96] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, "Multiple-Block Ahead Branch Predictors", Proc. of the 7th Int. Conf. on Architectural Support for Programming Languages and Operating Systems, pp. 116-127, 1996.

[SMI 94] J. E. Smith and S. Weiss, "PowerPC 601 and Alpha 21064: A Tale of Two RISCs", IEEE Computer, pp. 46-58, June 1994.

[SOD 97] A. Sodani and G. S. Sohi, "Dynamic Instruction Reuse", Proc. of the 24th Int. Symp. on Computer Architecture, pp. 194-205, 1997.

[SUN 87] Sun Microsystems, "The Sparc Architecture Manual – Version 7", Sun Microsystems Inc., 1987.

[VAJ 97] S. Vajapeyam and T. Mitra, "Improving Superscalar Instruction Dispatch and Issue by Exploiting Dynamic Code Sequences", Proc. of the 24th Int. Symp. on Computer Architecture, pp. 1-12, 1997.

[YEH 93] T.-Y. Yeh, D. T. Marr, and Y. N. Patt, "Increasing the Instruction Fetch Rate via Multiple Branch Prediction and a Branch Address Cache", Proc. of the 7th Int. Conf. on Supercomputing, pp. 67-76, 1993.