A Comparative Analysis Between EPIC Static Instruction Scheduling and DTSVLIW Dynamic Instruction Scheduling

Sandro C. Santana[1], Alberto F. 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
{camata, 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—

To achieve performance, *Explicitly Parallel Instruction Computing* (EPIC) systems take the responsibility of extracting instruction-level parallelism (ILP) from the hardware and give it to the compiler. They expose a large part of the hardware control at the conventional machine level. *Dynamically Trace Scheduled VLIW* (DTSVLIW) systems, on the other hand, leave the responsibility of extracting ILP to the hardware and use conventional compilers. Their hardware uses a simple hardware implemented scheduling algorithm – executed dynamically – to exploit ILP and achieve performance. This work examines three compiler/EPIC architecture combinations (SGI PRO64 C++ Compiler/IA64, Intel C++ Compiler 5.0.1/IA64 and Trimaran 2.0/HPL-PD) and compares these with a compiler/DTSVLIW architecture combination (gcc/Alpha-DTSVLIW). Our experiments show that, on average, the DTSVLIW architecture achieves better performance than EPIC because its dynamic scheduler, although much simpler, harnesses more ILP due to its exploitation of execution-time information invisible to the EPIC compiler's scheduler.

I. Introduction

A large and steady increase in microprocessor systems performance has been achieved in recent years, largely without rewriting programs in parallel form, changing algorithms or languages, and mostly without recompiling programs. The Superscalar architecture of recent microprocessors [1] has played a part in this by its exploitation of the instruction-level parallelism (ILP) available in existing programs. However, this has been at the expense of a considerable increase in hardware complexity, which may restrict further performance gains. To resolve this problem many architectures have been proposed, including the *Explicitly Parallel Instruction Computing* (EPIC) [2] and the *Dynamically Trace Scheduled VLIW* (DTSVLIW) [3] architectures.

The EPIC architecture is a more elaborated form of *Very Long Instruction Word* (VLIW) architecture [4] that moves the responsibility of detecting and extracting ILP out of the hardware into the compiler while improving the hardware to exploit the available ILP better. With EPIC, the compiler is responsible for scheduling the sequential code specified by the programmer into parallel EPIC instructions.

DTSVLIW architectures execute programs in two phases: one sequential, the other parallel. The first occurs when a fragment of code is first seen during execution, the second when the same fragment of code needs to be re-executed. In the sequential phase, the instructions are fetched from the instruction cache and executed by a simple pipelined processor. Concurrently, these instructions are scheduled into VLIW instructions and saved in blocks in a VLIW cache. In the parallel phase, the scheduled VLIW instructions are fetched from the VLIW cache and executed by a VLIW processor.

In this work, we compare the performance of the complex **static** schedulers used by the EPIC compilers with the simple **dynamic and hardware-implemented** scheduler of the DTSVLIW architecture. We report studies of the performance of programs compiled by three EPIC compilers: IA64 SGI PRO64 C++ Compiler version 0.13, IA64 Intel C++ Compiler 5.0.1 for Linux beta version build 20010418, and HPL-PD Trimaran 2.0 Compiler. SPECInt95 programs have been compiled by these for two different EPIC instruction set architectures (ISAs), HPL-PD [5] and IA64 [6], and have been executed in instrumented execution-driven EPIC simulators. The best performance measurements have been compared with similar ones made on an execution-driven DTSVLIW simulator configured with hardware equivalent to that of the EPIC machines and running the same programs, but compiled with the gcc compiler.

Our results show that the Intel (compiler)/IA64 (ISA) combination has a performance 19.1% better than the SGI/IA64, while the combination Trimaran/HPL-PD with equivalent hardware has a much poorer performance than the others. However, the Trimaran/HPL-PD combination with unlimited hardware resources achieved performance 37.7% better than the Intel/IA64 on average. Comparing the best EPIC results with the DTSVLIW results we observed that the

DTSVLIW achieves performance at least 37.6% better than any EPIC combination with equivalent hardware resources and breaks even with EPIC combinations with unlimited resources. The evidence shows that the DTSVLIW achieves better performance because its scheduler, although much simpler than EPIC's, has access to dynamic information, not available to static compilers, allowing exploitation of more ILP.

II. VLIW AND EPIC ARCHITECTURES

A VLIW machine has multiple functional units executing in parallel and controlled by a single long instruction, holding a sub-instruction for each functional unit. Often operations cannot be found for all units and "nop" sub-instructions must be inserted, increasing code size. When a long instruction is fetched, its sub-instructions are directly sent for execution without dependency checks: the VLIW compiler has sole responsibility for detecting ILP and scheduling instructions. This reduces the hardware making VLIW machines simpler and faster. However, a program compiled for a specific VLIW machine must be recompiled to run on another with different functional units either in type or number. This is known as the VLIW code compatibility problem.

The EPIC architecture is an evolution from VLIW. It can execute several instructions concurrently, with the compiler identifying and exploiting the ILP available in programs. However, unlike VLIWs, the EPIC hardware is responsible for binding instructions to functional units. This allows the same program to be executed by EPIC machines with different levels of parallelism, solving the code compatibility problem while keeping most of the positive characteristics of VLIWs.

A. Static Scheduling

Compiler performance is vital for VLIW and EPIC systems. This scheduling of program code into groups of instructions for concurrent execution can be done over the whole program and operates after traditional compiler optimization techniques (*loop invariant motion*, *common subexpression elimination*, *induction variable simplification*, *inline*, *loop unrolling*, etc. [7]).

To facilitate scheduling, instructions are usually grouped in *basic blocks*[1], and a graph constructed with basic blocks as vertices and the control path between them as edges. Basic blocks are often small – 5 to 20 instructions on average [8] – which strongly limits the ILP inside them.

To exploit ILP beyond basic blocks limits, *Trace Scheduling* [9] uses heuristics or profiling to select the most probable path through the program's graph. The basic blocks belonging to this "trace" form a new unique block and their instructions are scheduled as if they belonged to a single basic block, so instructions can be moved outside the limits of their basic blocks. This scheduling generally ignores conditional branches, and compensation code must be inserted in all edges leaving or leading to the trace to ensure correct program execution (book-keeping [9]). The compiler repeats this scheduling process on all other traces in the program's graph in the order of their probability of execution.

To alleviate book-keeping operations pressure, which increases object code size, the *Superblock Scheduling* technique [10] builds superblocks. These are traces with no *side entrances*, where control can only lead to the first instruction. A superblock is built from an ordinary trace via *tail duplication*, which copies instructions of the trace from the side entrance to the end of the trace, and redirects the side entrance to this new trace. *Hyperblock Scheduling* produces hyperblocks that are essentially superblocks but which may contain predicated instructions [11] and, therefore, may contain instructions belonging to more then one control path.

All the above are global scheduling techniques that can be applied to acyclic segments of code. Although they can be applied to cyclic code (loops), they cannot exploit the ILP available in loops efficiently. Software Pipelining [12] and loop unrolling [13] are designed to do this. A software pipelining compiler intercalates instructions belonging to different loop iterations so that the ILP available in the loop can be exposed to the hardware.

These static scheduling techniques, and others, e.g. Enhanced Pipeline Percolation Scheduling [14], have been used in compilers for both VLIW and EPIC machines.

B. The EPIC HPL-PD and IA64 ISAs

An EPIC ISA must provide the compiler with facilities for expressing the parallel execution plan of the programs. However, it must leave to the hardware the task of binding instructions to resources. It must implement mechanisms to check, at execution time, the legality of execution of instructions moved by the compiler across basic block limits. Here we present two EPIC ISAs that do this.

In 1994, Hewlett Packard (HP) published the specification of the *HP Laboratories PlayDoh* (HPL-PD) EPIC architecture [5]. This is a generic parameterized experimental EPIC of 32 bits. It provides speculative versions of all instructions except branches and stores. Exceptions during speculative execution are not signaled immediately, only later, if a non-speculative instruction uses a result that is invalid because of an exception. Load instructions can be speculatively moved above stores with the compiler scheduling a check instruction after the store to identify address aliasing, in which case the hardware re-executes the load. A branch instruction can be split into sub-instructions to allow the earlier execution of parts of the branch to improve ILP and to give static prediction information to allow the prefetch of instructions from the computed target address. *Rotating registers* [15] are provided for the support of Software Pipelining.

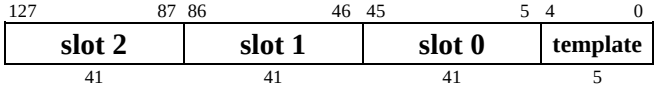

Fig. 1: *bundle* format

[1] a group of instructions with a single entry at the beginning and no conditional branches except at the end

The IA64 EPIC ISA is a result of an alliance between HP and Intel [2]. It incorporates all HPL-PD characteristics previously described. Its compiler must group instructions into 128-bit *bundles* of three instructions, each 41 bits, and a 5-bit *bundle template* (Figure 1).

This template informs the hardware of the functional units used by the instructions and of instruction dependencies. The possible dependencies are encoded by the identification of *stops* between instructions. A *stop* between instruction specifies that these instructions must be executed in separate cycles. Thus, there are bundles that require a minimum of three clocks to execute; however, two or more bundles may execute in a single cycle if there are no stops within and between them.

An IA64 machine can send several bundles for execution simultaneously and the hardware is responsible for dispatching instructions in the bundles to appropriate functional units. This ISA, while taking advantage of VLIW characteristics, allows for code compatibility; but does require part of the complex dispatch and issue hardware of Superscalars [1].

III. The DTSVLIW Architecture

The DTSVLIW architecture provides an alternative solution to the VLIW code compatibility and backward compatibility problems. The symbolic diagram of a DTSVLIW machine (Figure 2) 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 Primary Processor executes the original code first, and the code trace this produces 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 is re-executed.

While the Primary Processor is executing the code, the Fetch Unit (Figure 2) issues different addresses to the Instruction Cache and the VLIW Cache. The *program counter* (PC) content is issued to the former, while the address of the instruction in the execute stage of the Primary Processor is issued to the latter (dashed arrow in Figure 2). The machine uses this address because at this point it knows 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 tagged 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 resumes 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 starts scheduling 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.

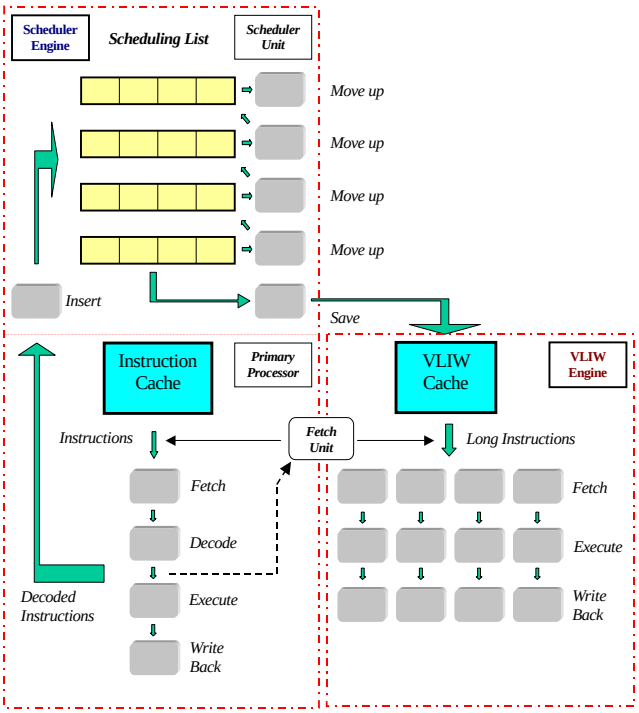

Fig. 2 A DTSVLIW Machine

The DTSVLIW architecture is a variant of the DIF architecture of Nair and Hopkins [16]. The DTSVLIW achieves similar or better performance to the DIF, but with a simpler architecture [17]. Our results 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 [18].

In our current DTSVLIW implementation, the Primary Processor executes Alpha 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. There is no decode stage as decoded instructions are saved in the VLIW Cache, which 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. We have previously presented details on dealing with exceptions, memory aliasing (disambiguation), and the execution of particular instructions [19].

A. DTSVLIW dynamic scheduling

The DTSVLIW Scheduler Engine performs *superblock scheduling* dynamically. In VLIW and EPIC compilation systems, superblocks are built in two steps. First, traces are

selected using heuristics or profiling. Second, *tail duplication* is applied to the traces (see Section II). In a DTSVLIW machine, the trace for scheduling is dynamically produced by the Primary Processor executing the original sequential program, and it is this trace that is scheduled, as it is produced, by the Scheduler unit into VLIW blocks. Each block of long instructions may encompass many basic blocks. The scheduling allows any branch inside any block to branch outside its block without side effects, due to register renaming and memory disambiguation. 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 to that followed during scheduling, these paths will cause the scheduling of new blocks. This is equivalent to tail duplication. Compilers, when performing superblock scheduling, select traces statically and these traces must be suitable for all a program's input data sets. In contrast, a DTSVLIW machine performs dynamic trace selection, which adjusts to the input data set and restricts the data set's impact on the machine's performance.

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 [20]. This algorithm was chosen for three reasons [17]. First, it operates with one instruction at a time and considers instructions in the strict order that they appear during program execution, perfectly fitting the DTSVLIW mode of operation. Second, the FCFS algorithm produces optimum or near-optimum scheduling [20]. Finally, the FCFS algorithm is easy to implement in hardware in a pipelined fashion [17].

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 is not at the head of the list; there is space for it in the next element; 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 are *move up* and *install* examples on 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 destination. Example:

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

Conditional and indirect branches do not move up. They are installed when placed in the scheduling list and establish a *tag* for their long instruction. Subsequent instructions installed in this long instruction receive the last established tag. When, during VLIW execution, the VLIW Engine evaluates a branch, it validates its tag if it executes the same as when scheduled. Only instructions with valid tags have their results written in the machine state. Thus, the copy instruction in the example is only executed in VLIW mode if the conditional branch (beq) follows the same direction observed during scheduling.

When there is no space for an incoming instruction, the list is flushed to the VLIW Cache 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, on a pipelined basis, one long instruction per cycle. Insertion of further incoming instructions overlaps the saving of the old block [19]. A VLIW block is tagged in the cache with the address of its first installed instruction and with that of the following block.

Load and store instructions can also be split, which can cause memory aliasing [4] and exceptions. We have previously presented details on how the DTSVLIW deals with these situations [17]. We have also proved that the core operations performed by the DTSVLIW's scheduling algorithm have the complexity of an integer adder and, as such, should not increase the clock cycle time [17]. Multicyle instructions impact upon the operation and performance of the architecture. Their scheduling has to respect dependencies in any of their cycles [21]. This can restrict the packing of instructions into long instructions limiting parallelism.

IV. A Comparison Between the EPIC Static Instruction Scheduling and the DTSVLIW Dynamic Instruction Scheduling

To compare the EPIC static scheduling with the DTSVLIW dynamic scheduling, we have used three execution-driven machine simulators: the SKI [22] and the Trimaran HPL-PD [23] EPIC simulators, and our DTSVLIW simulator.

The SKI simulator interprets IA64 ISA code. At the end of program execution, this simulator presents the number of executed instructions (including nop's) and the number of stops found. The simulator considers that the latency of all instructions is one, and that there is no limitation of resources (the number of slots in a long instruction and the number of functional units). Then, the number of stops reported by the

SKI simulator represents the smallest number of cycles that would be required for the execution of the interpreted program. In a silicon interpretation, besides limited resources, the latency of the instructions would not all be one cycle and there would be cache misses, branch mispredictions, etc., that would require a much larger number of cycles. However the cycle count of the SKI simulator does display the performance of the compiler scheduler, which is why we have used it.

The Trimaran HPL-PD simulator interprets the set of instructions defined in the HPL-PD [5] specification. This simulator is parameterized and one can choose the number of registers and functional units, latency of the instructions, etc.

The DTSVLIW simulator interprets Alpha ISA [24] code. It receives as input any programs compiled for the OSF-1 operating system and faithfully executes them according to the DTSVLIW architecture model described in Section III. The simulated DTSVLIW machine also incorporates the block compaction mechanisms described in [25]. The simulator is parameterized and one can choose the number of registers and functional units, latency of the instructions, etc.

All simulators interpret only the code that executes in user mode, including the code of libraries linked into the program. Operating system calls are detected, converted to calls to the host operating system and executed there, with results being copied back into the execution context.

TABLE 2
PROGRAMS AND PARAMETERS USED

Programs SPECInt95	Parameters
099.go	9 9
124.m88ksim	dcrand.lit
129.compress95	30000 q 2131
130.li	queens 7
132.ijpeg	vigo.ppm.fast –GO
134.perl	Primes.pl
147.vortex	vortex.in

In all our experiments, we have used the programs of SPECInt95 (except gcc, because its SPEC95 source is not 64-bit compatible, thus impossible to compile with the available EPIC compilers). The programs and the inputs used are listed in Table 2. EPIC executable code has been generated by the compilers: IA64 SGI PRO64 C++ Compiler version 0.13, the IA64 Intel C++ Compiler 5.0.1 for Linux beta version build 20010418, and the HPL-PD 2.0 Trimaran Compiler. The gcc 2.7.2 compiler has been used for compiling to the Alpha ISA code.

A. Simulation Parameters

A.1 Compilers

Although it is not mentioned in the documentation of the Intel EPIC compiler, we believe that it uses trace scheduling or one of its variations (hyperblock scheduling, for example). Its flag –O2 turns on all classic optimizations (global register allocation, register variable detection, common subexpression elimination, etc) and software pipelining. The flag –O3, provides the same optimizations as flag –O2, but also turns on: prefetching, scalar replacement and loop transformation. The flag -ip turns on optimizations between procedures (interprocedural optimizations) such as inline function expansion and interprocedural constant propagation. We have used two compiler configurations: one, referred to as Intel–O2.IP, uses flags –O2 and –ip; the other, referred to as Intel–O3.IP, uses flags –O3 and –ip. When compiling with both configurations, we used profiling information collected from previous runs of each program with the same inputs. This gave us the best performance.

TABLE 3
DTSVLIW PARAMETERS

Primary Processor	• 4-stage (fetch, decode, execute, writeback) pipeline • no branch prediction hardware • taken branches cause a 2-cycle pipeline bubble
Decoded Instruction Size	6 bytes
Instructions Latency	1 cycle
VLIW Cache	Four way set associative, blocks of 15x16 instructions, 3072-Kbyte
Instruction Cache	perfect (no miss penalty)
Data Cache	perfect (no miss penalty)
Number of renaming regs.	128

The SGI compiler was modified to use more functional units (15 of each type) than that available in the Intel Itanium processor and to have latency of 1 for all instructions. The –O2 flag turns on the majority of this compiler's optimizations. The optimizations in this level are conservative, in the sense that they are always beneficial, providing improvements proportional to the time spent on compilation. The flag –O3, on the other hand, turns on more aggressive optimizations than flag –O2. These are generally beneficial, but can compromise performance in some cases. Our configuration, SGI–O2, uses flag–O2, while SGI–O3 uses flag –O3.

We have set the HPL-PD 2.0 Trimaran Compiler to generate code for a Trimaran HPL-PD machine with infinite resources by activating the unlimited resources option and deactivating all the register restriction options. Two configurations were parameterized for different block formation – superblock and hyperblock. These two configurations are referred to as MAX SB and MAX HB. Another two HPL-PD 2.0 Trimaran Compiler configurations were set (again with Superblock or Hyperblock scheduling) to generate code for a Trimaran HPL-PD machine with 15 functional units of each existing type (integer, floating point, memory and branch) and 128 registers of each existing type (general purpose, floating-point, predicate and branch target). These configurations are referred to as the T15 SB and T15 HB. These configurations are very optimistic from the point of view of implementation.

Flags –O3 and –unrolloops had been used with the gcc to generate the code for the Apha ISA.

A.2 Machine Simulators

It was not necessary to set any parameter for the SKI simulator.

The Trimaran HPL-PD was set with: (i) unlimited registers and functional units, for the compiler configurations MAX SB and MAX HB, and (ii) 15 functional units of each existing type and 128 registers of each existing type, for the compiler configurations T15 SB and T15 HB. The latency was set to 1 for all instructions.

For the DTSVLIW simulator, we have used the parameters shown in Table 3. The count of instructions executed given for the DTSVLIW includes only the instructions that would be executed in a scalar machine, i.e., nops and copy instructions added to long instructions during scheduling are not counted (to count these instructions would erroneously inflate the ILP achieved by the DTSVLIW). Instructions executed and cycles spent in the Primary Processor during scheduling are, of course, counted.

The instructions and data caches of all machines studied were specified as ideal (without miss penalty and with single cycle access). All parameters used were chosen in order to isolate the subject of study: the quality of the scheduling of the sets of compiler/ISA/machine architecture.

B. Experiments

In this section we show performance measurements for each individual benchmark running on each configuration studied and average performances also. Jacob and Mudge [26], and Giladi and Ahituv [27] 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 and the arithmetic mean for indices directly related to the execution time like number of instructions executed and cycle count. Therefore, when appropriate, we use either the harmonic mean or the arithmetic mean.

B.1 Intel x SGI x Trimaran Compiler

Figure 3 shows the count of instructions executed for each SPECInt95 program on the different EPIC simulators with the compiler configurations: Intel–O2.IP, Intel-O3.IP, SGI–O2, SGI–O3, T15 SB, THB, MAX SB, MAX HB. The arithmetic mean (A.M.) of the instructions executed is also shown.

Figure 3 shows that the Intel and SGI compilers produce little variation in the instructions executed as a function of the level of optimization. However, in most cases, the instruction count is larger when the optimization is more aggressive. For the Trimaran configurations we cannot actually distinguish between more or less aggressive, but only different optimizations. With limited resources, T15 SB and T15 HB, hyperblock scheduling generates programs that execute less instructions in the majority of the cases compared to superblock, although this trend is not confirmed in the cases of go and xlisp, possibly due to the use of predication in code fragments that are not good candidates for this technique. For Trimaran with unlimited resources, MAX SB and MAX HB, we see a more uniform behavior, where the formation of hyperblocks always leads to the same or fewer instructions executed. Comparing compilers, we can see that the Intel compiler always generates code that results in less executed instructions than the SGI compiler and that the Trimaran compiler with limited resources produces, on average, results between Intel and SGI or, with unlimited resources, code that results in the execution of less instructions than these two.

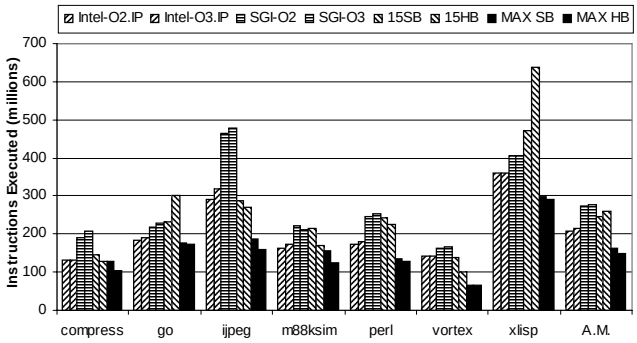

Fig. 3 EPIC: Instructions executed

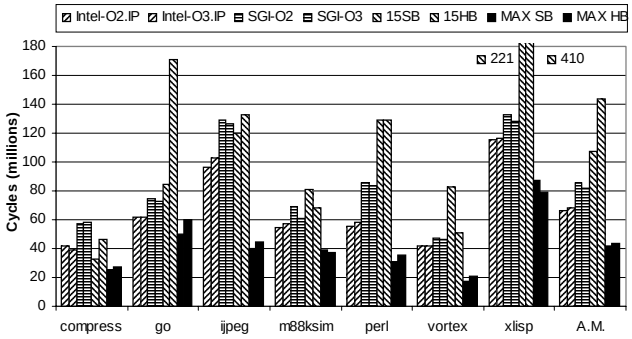

Fig. 4 EPIC: Required Cycles

Figure 4 shows the count of execution cycles for the SPECInt95 programs with the same configurations as Figure 3. The cycle count represents the best measure of performance, since it indicates the execution time of each program. From Figure 4, we can perceive that the Intel–O2.IP configuration gave, on average, a slightly better performance than Intel–O3.IP (about 2.3%). This shows that more aggressive optimizations do not always results in performance gains (please note that the Intel–O3.IP configuration may perform better than Intel–O2.IP if memory latencies are considered). The Intel configurations show, on average, a better performance than the SGI configurations (about 19 %). The SGI–O3 configuration gave, on average, a slightly better performance than the SGI–O2 (about 3.3% better). The Trimaran configurations with limited resources, T15 SB and T15 HB, show the worse performance, where the instruction count for the programs *go* and *xlisp* has a strong negative impact on the average performance. As expected, Trimaran MAX SB and MAX HB obtained the best performance in all programs on average, about 37.7% better than Intel–O2.IP, the runner up. However, perhaps surprisingly, the configuration

with superblock scheduling generated programs that require less cycles than that with hyperblock scheduling for all programs except m88ksim and xlisp, although not by much.

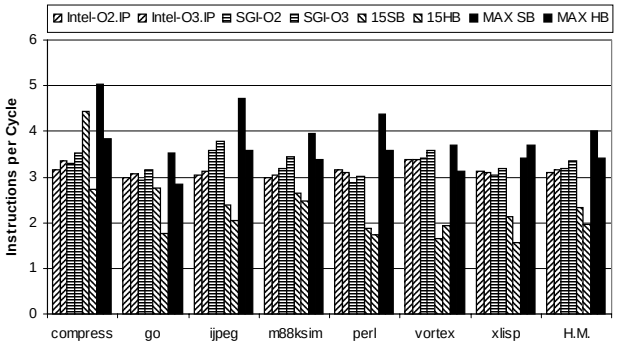

Fig. 5 EPIC: Instructions per Cycle

Figure 5 shows the average instructions per cycle executed for the SPECInt95 programs, along with their harmonic mean (H.M.) for the same configurations of Figure 3. These figures indicate the degree of ILP achieved. As Figure 5 show, the Intel and SGI compilers do not obtain ILP much larger than three – the size of the IA64 ISA bundle – instructions per cycle. The T15 SB and T15 HB configurations show worse ILP for all programs with the exception of compress with T15 SB. Again MAX SB and MAX HB achieve the best results on average, with the advantage to the MAX SB configuration.

B.2 DTSVLIW x EPIC

In order to compare the EPIC scheduling with that of the DTSVLIW, we have selected the best performing EPIC configurations for each compiler: Intel– O2.IP, SGI–O3 and MAX SB. Figure 6 presents the instruction execution count for each of these configurations (results from Figure 3) plus those for the DTSVLIW. The gcc compiler produces programs that always result in the execution of fewer instructions than that generated by the Intel and SGI compilers. This is expected. The facilities for conditional execution in the Alpha ISA are represented only by a few conditional move instructions little used by the gcc compiler, while the IA64 ISA has ample mechanisms for conditional execution, intensively used by the compilers, which leads to the execution of many extra instructions. In addition, compilers for the IA64 ISA generate many nop instructions due to the restrictions imposed by the template field for valid bundle formation. Programs generated by gcc have, on average, the same instruction count as those from the Trimaran with unlimited resources. The Trimaran does not generate nops; thus, as our results show, at least with unlimited resources it is possible to make good use of predication for executing fewer instructions.

Figure 7 shows the cycle count for the execution of each SPECInt95 program. The combination gcc/DTSVLIW surpasses the combinations Intel-compiler/IA64 and SGI-compiler/IA64 by a wide margin on average. The DTSVLIW only loses in one program, go, for the Intel/IA64 combination, but by a small margin. Compared with the Trimaran/HPL-PD combination, however, the gcc/DTSVLIW only wins in m88ksim and xlisp; however, on average the gcc/DTSVLIW performance is equivalent to Trimaran/HPL-PD.

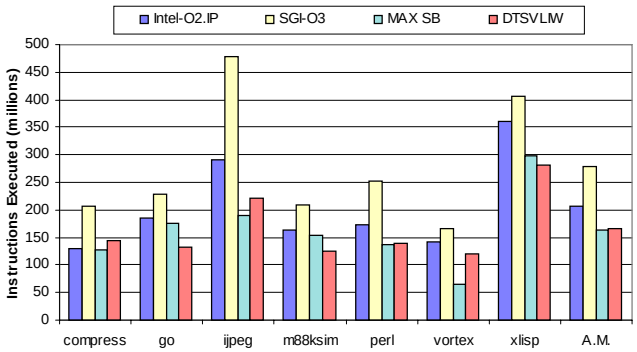

Fig. 6 DTSVLIWxEPIC: Instructions Executed

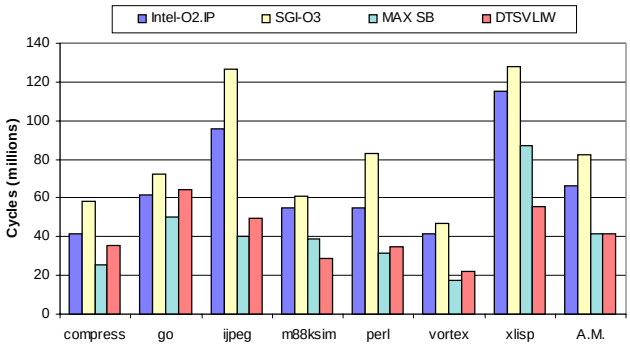

Fig. 7 DTSVLIWxEPIC: Required Cycles

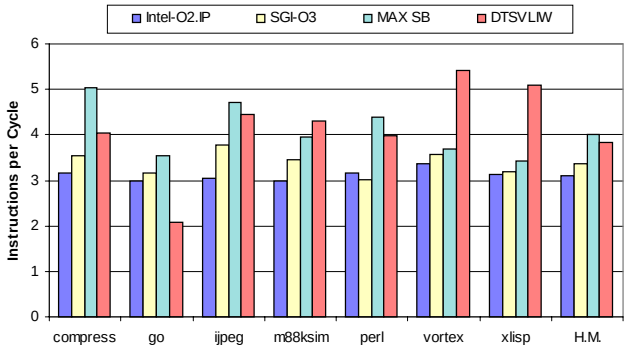

Fig. 8 DTSVLIWxEPIC: Instructions per Cycle

Figure 8 shows the instructions per cycle for each combination under study. The combination gcc/DTSVLIW has been able to explore parallelism levels only inferior to the ones achieved with an EPIC compiler compiling for an unlimited machine and being executed in a machine with unlimited resources. The single exception is again the go benchmark. This benchmark has many hard to predict branches. These branches force the DTSVLIW to reschedule many blocks, which results in less ILP.

The DTSVLIW machine configuration used in the experiments is optimistic, but much less optimistic than the EPIC machine emulated by the SKI simulator and even less so than that emulated by the Trimaran simulator. Our results show that the DTSVLIW needed, on average, practically the same number of cycles, had only 4.4% inferior ILP and executed 1.5% more instructions than the Trimaran MAX SB. But compared to the Intel–O2.IP configuration, which achieves better results than the SGI–O3, the DTSVLIW, on average, needed 37.6% fewer cycles, achieved a 23.4% larger ILP and executed 19.5% fewer instructions.

V. Conclusion

We have presented an experimental comparative analysis between static EPIC scheduling and dynamic DTSVLIW scheduling. In our experiments, the DTSVLIW scheduling obtained, on average, practically the same performance achieved with the Trimaran EPIC compiler configured with unlimited resources, which had the best results among the EPIC compilers used. However, the DTSVLIW machine obtained its performance using a configuration significantly less optimistic than the EPIC machines. Thus, it is likely that the DTSVLIW obtained the demonstrated performance because its scheduling hardware makes good use of dynamic information, not available to the compilers, regarding the execution of the test programs. The static schedulers of the EPIC compilers have access only to averaged information obtained via profiling.

It is important to note that the DTSVLIW architecture may had been disfavored because the code executed by the DTSVLIW simulator was generated by a compiler not specific for this architecture, while for the EPIC architecture specific compilers have been used. Moreover, the existing differences between the EPIC ISAs and the Alpha ISA have not been considered. These questions should be clarified with future analyses of the performance of the DTSVLIW architecture running EPIC code and/or running code generated by a compiler specific for the DTSVLIW.

References

[1] JOHNSON, M. Superscalar Microprocessor Design. Prentice-Hall, 1991.

[2] GWENNAP, L. Intel, HP make EPIC Disclosure. Microprocessor Report, Vol. 11, No. 14, pp. 1-9, October 27, 1997.

[3] DE SOUZA, A. F.; ROUNCE, P. A.. Dynamically Trace Scheduled VLIW Architectures. Proc. of the HPCN'98, in Lecture Notes on Computer Science, Vol. 1401, pp. 993-995, 1998.

[4] FISHER, Joseph. The VLIW Machine: A multiprocessor for Compiling Scientific Code. IEEE Computer, pp. 45-53, Jul 1984.

[5] KATHAIL, Vinod; SCHLANSKER, Michael; RAU, B. Ramakrishna. HPL-PD Architecture Specification: Version 1.0. HPL-93-80, Feb 1994.

[6] INTEL Corporation. IA64 Application Developer's Architecture Guide, 1999.

[7] AHO, A.; SETHI, R.; ULLMAN, J. D.. Compilers - Principles Techniques and Tools. Addison-Wesley Publishing Company, USA, 1986.

[8] PATTERSON, D. A.; HENNESSY, J. L.. Computer Architecture: A Quantitative Approach, Second Edition. Morgan Kaufmann Publishers, Inc., 1996.

[9] FISHER Joseph. Trace Scheduling: A Technique for Global Microcode Compaction. IEEE Transactions on Computers, v. C-30, n.7, pp. 478-490, Jul 1981.

[10] HWU, Wen-mei W.; et al. The Superblock: An Effective Technique for VLIW and Superscalar Compilation. The Journal of Supercomputing, v.7, pp. 229-248, 1993.

[11] PARK, J. R. H.; SCHLANSKER, M. S. On Predicated Execution. Technical Report HPL-91-58, HP Laboratories, Palo Alto, CA, May 1991.

[12] ALLAN, V. H.; JONES, R. B.; LEE, R. M.; ALLAN, S. J. Software Pipeline. ACM Computing Surveys, Vol. 27, No. 3, September 1995.

[13] RAU, B. R. FISHER, J. A. Instruction-Level Parallelism: History, Overview, and Perspective. The Journal of Supercomputing, Vol. 7, pp. 9-50, 1993.

[14] NAKATANI, T.; EBCIOGLU, K. Making Compaction-Based Parallelization Affordable. IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 9, pp. 1014-1029, 1993.

[15] SCHLANSKER, Michael; RAU, B. Ramakrishna; MAHLKE, Scott; KATHAIL, Vinod. Achieving High Levels of Instruction-Level Parallelism with Reduced Hardware Complexity. HPL-96-120, Nov 1994.

[16] NAIR, R.; HOPKINS, M. E.. Exploiting Instructions Level Parallelism in Processors by Caching Scheduled Groups. Proceedings of the 24th Annual International Symposium on Computer Architecture, pp.13-25, 1997.

[17] DE SOUZA, A. F.; ROUNCE, P. A. Dynamically Scheduling VLIW Instructions. Journal of Parallel and Distributed Computing, n.60, pp.1480-1511, 2000.

[18] DE SOUZA, A. F.; ROUNCE, P. A. On the Scheduling Algorithm of the Dynamically Trace Scheduled VLIW Architecture. Proc. of the International Parallel and Distributed Processing Symposium - IPDPS'2000, pp. 565-572, 2000.

[19] DE SOUZA, A. F.. Integer Performance Evaluation of the Dynamically Trace Scheduled VLIW Architecture, PhD Thesis, Department of Computer Science, University College London, 1999.

[20] Davidson, S.; et al. Some Experiments in Local Microcode Compaction for Horizontal Machines. IEEE Trans. on Computers, Vol. C-30, No. 7, pp. 460-477,

1981.

[21] DE SOUZA, A. F.; ROUNCE, P. 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.

[22] HP Laboratories. Ski IA64 Simulator Reference Manual. Ver 1.0L, Apr 2000.

[23] HP Laboratories; New York University, ReaCT-ILP Group; University of Illinois, IMPACT Group. Trimaran: An Infrastructure for Compiler Research in Instruction Level Parallelism, 1998.

[24] Digital Equipment Corporation. Alpha Architecture Handbook. Digital Equipment Corporation, 1992.

[25] DE SOUZA, A. F. Improving the DTSVLIW Performance via Block Compaction. Proc. of the 13th Symp. on Computer Architecture and High Performance Computing, pp. 98-105, 2001.

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

[27] GILADI, R.; AHITUV, N. "SPEC as a Performance Evaluation Measure", IEEE Computer, pp. 33-42, August 1995.