

SPECint95 Performance of an Implementation of the Dynamically Trace Scheduled VLIW Architecture

Alberto Ferreira de Souza¹ and Peter Rounce
Department of Computer Science
University College London
Gower Street, London WC1E 6BT - UK
a.souza@cs.ucl.ac.uk, p.rounce@cs.ucl.ac.uk

Abstract. *Dynamically trace scheduled VLIW* (DTSVLIW) architectures can be used to implement machines that execute code of current RISC or CISC instruction set architectures in a VLIW fashion, delivering instruction level parallelism with backward code compatibility. This paper presents preliminary SPECint95 performance measurements of the DTSVLIW architecture, obtained with a simulator which has been implemented in C.

1. Introduction

Object code compatibility is a problem for *Very Long Instruction Word* (VLIW) architectures: mapping a VLIW *instruction set architecture* (ISA) to implementations with different hardware latencies and varying levels of parallelism is not generally possible. To get over this, a *dynamically scheduled VLIW* (DSVLIW) was presented by Rau [1]. However, despite the ability to implement a family of VLIW machines with different functional units' latency and the same ISA, this concept cannot be used to implement an existent sequential ISA. Ebcioglu and Altman [2] with their DAISY machine can translate dynamically from the object code of an existing ISA architecture to the object code of a VLIW using a Virtual Machine Monitor (VMM) implemented in software. The DAISY machine concept relies on the ability of the VMM to translate code fast, and in the reusability of this code. Since the VMM is implemented in software, the cost of the translation is necessarily high. Although code reusability is probably appreciable, a hardware translation is possibly advantageous.

The Dynamic Instruction Formatting (DIF) concept (Nair and Hopkins [3]) performs hardware re-formatting of the fetched code. The original code is executed on a primary engine (a simple processor, less aggressive in exploiting parallelism) and, at the same time, re-formatted into the DIF VLIW cache for execution by a VLIW engine. This paper describes an architecture organisation that implements the DIF concept, named *dynamically trace scheduled VLIW* (DTSVLIW) [4]. In order to evaluate the DTSVLIW architecture, a parametric simulator was implemented and execution driven simulation was performed using the SPECint95 benchmark suite. Experimental results presented here show that the DTSVLIW can achieve *instruction level parallelism* (ILP) higher than 4 instructions per cycle with some machine configurations.

This paper is organised as follows. In the next section, the DTSVLIW architecture is described. In Section 3, the experimental methodology and the results of the experiments carried out to evaluate the DTSVLIW architecture are presented. Section 4 contains the conclusions and future work.

2. The DTSVLIW Architecture

The DTSVLIW has two execution engines: the Scheduler Engine and the VLIW Engine; and two caches for instructions: the Instruction Cache and the VLIW Cache. The Scheduler Engine fetches instructions from the Instruction Cache and executes the original code for the first time using simple pipelined hardware. In addition, the Scheduler Engine *dynamically schedules the trace* produced during the execution into VLIW instructions and saves them as blocks of VLIW instructions into the VLIW Cache. If the same code is executed again, it is fetched by the VLIW

¹ Sponsored by CAPES (Brazilian Government Agency).

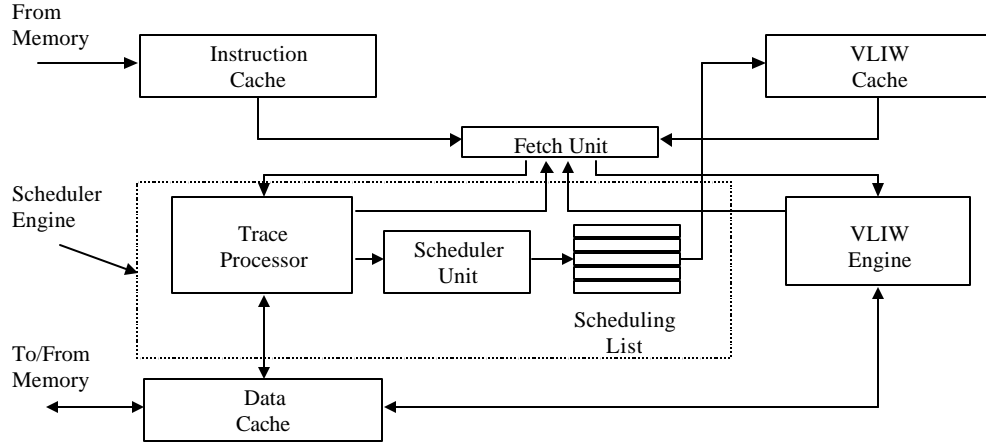


Figure 1: The DTSVLIW architecture.

Engine and executed in a VLIW fashion. A block diagram of the DTSVLIW is presented in Figure 1.

The Scheduler Engine of the DTSVLIW consists of the Trace Processor and the Scheduler Unit. The Trace Processor of the DTSVLIW presented in this paper is a simple pipelined scalar Sparc processor. As this implementation executes Sparc-7 ISA [5] code, the Trace Processor is capable of executing all instructions of this ISA. The Scheduler Unit uses a pipelined implementation of the FCFS (First Come First Served) scheduling algorithm, traditionally used in microcode compaction [6].

The FCFS algorithm has been chosen as the scheduling algorithm of the DTSVLIW for three reasons. First, as its name states, 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 [6]. Finally, the FCFS algorithm is easy to implement in hardware in a pipelined fashion. The implemented version of the FCFS algorithm runs over a circular list that was named *scheduling list*.

The scheduling list has a fixed number of long instructions, one per element of the list. Each element of the list has also a *candidate instruction*. A valid candidate instruction is scheduled into a list element in the preceding clock cycle, and is an aspirant member of the long instruction in the element. A valid candidate instruction still may be moved to a higher position in the list. An instruction with execution finished in the Trace Processor in one cycle can be *inserted* in the scheduling list in the subsequent cycle. If the scheduling list is not empty and depending on dependencies, the incoming instruction becomes a candidate instruction either in the tail element of the list or in a new entry added to the list. In the latter case, if there are no spare list elements, the list is made empty, the instruction is inserted in the empty list, and the whole previous list content is sent to the VLIW Cache as a *block*.

The VLIW Cache is set associative with set size equal to one block of long instructions of scheduling list size. In the VLIW Cache, each long instruction can be accessed directly. The VLIW Engine fetches VLIW instructions from the VLIW Cache and has a simple two-stage fetch-execute pipeline. A decode stage is not necessary since the long instructions are saved in the VLIW Cache already decoded. All conditional and indirect branches are resolved in the execute stage of the VLIW Engine. The direction taken during scheduling, and recorded into the VLIW Cache, is used during execution to determine a possible misprediction. If a conditional or indirect branch target is different than that observed while scheduling during VLIW execution, the current fetch is annulled and the program counter receives the new target. Consequently, a branch misprediction causes a one cycle deep bubble in the pipeline.

3. DTSVLIW Experimental Evaluation

A simulator of the DTSVLIW has been implemented in C (19K lines of code), and execution driven simulation performed to produce the results reported here. The simulator receives as input binary executable programs generated by the gcc compiler and faithfully models the execution performed by the DTSVLIW. Model parameters that are invariant for simulations are shown in Table 1.

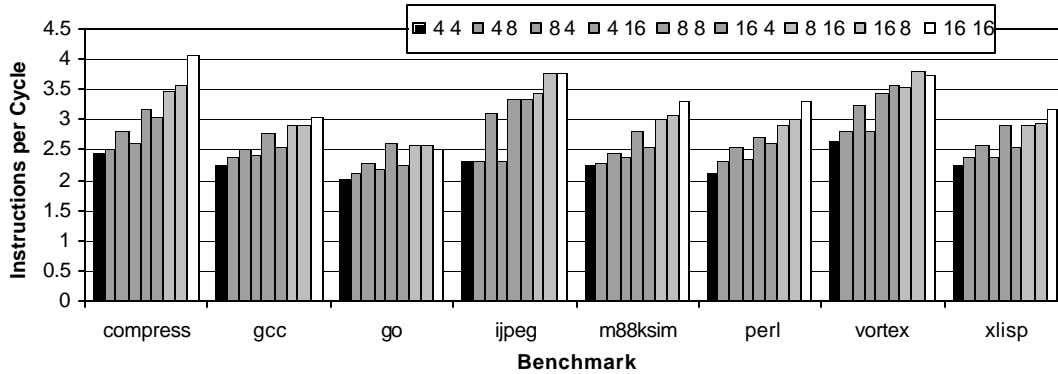
The benchmark programs used in the experiments were the SPECint95 set. All benchmarks were compiled with optimisation flags `-O -mflat`. Each benchmark program was allowed to run 50 million or more instructions on each experiment.

Trace Processor	<ul style="list-style-type: none"> four-stage (fetch, decode, execute, and write back) pipeline no branch prediction: not-taken branches cause a 3 cycle bubble in the pipeline instructions following a load requiring the data loaded cause a one-cycle bubble
Instruction & Data Caches	perfect
VLIW Cache Associativity	4 way
Decoded Instructions Size	6 bytes
VLIW Engine	homogeneous: functional units can execute any instruction
Instructions Latency	1 cycle
VLIW Engine Lists Size	load = store = checkpoint recovery store = (long instruction size * block size) entries
Number of Renaming Registers	integer = f.p. = memory = flags = 256 registers
Scheduler Unit Pipe	inserting/splitting and moving up/saving = 1/block size/1 stages

Table 1: Fixed Parameters

Figure 2 shows the effect of the block geometry on the performance of the DTSVLIW. To ensure the absence of extraneous effects, the experiments leading to the results in this figure were performed with perfect instruction and data caches (no miss penalty), large VLIW Cache (3072-Kbyte), and no next long instruction miss penalty. The numbers in the legend are instructions in a long instruction and long instructions in a block, respectively.

As the graph in the figure shows, the performance grows with both long instruction size and block size. However, the performance of blocks with the same number of instructions but different geometry is significantly different. For example, the performance of a machine configuration with 4 instructions per long instruction and 8 long instructions per block is lower than a configuration with 8 instructions per long instructions and 4 long instructions per block. The DTSVLIW architecture benefits from large long instruction and block sizes, but not linearly. A sixteen-fold increase in the number of instructions in a block (from 4x4 to 16x16) does not double the performance of this



implementation on any benchmark.

Figure 2: The DTSVLIW performance versus block geometry

The results presented in Figure 2, represents the highest achievable SPECint95 performance of this DTSVLIW implementation. When the VLIW Cache is smaller the performance of the DTSVLIW is expected to be lower because premature flushing of useful scheduled blocks due to replacement by new blocks. Figure 3 shows the impact of different VLIW Cache sizes on the performance of a DTSVLIW machine with 8 instructions per long instruction and 8 long instructions per block. As the graph shows, some benchmark programs do not demand a large VLIW Cache size in order to exploit the performance of the DTSVLIW. Compress and ijpeg appear to be very insensitive to the VLIW Cache size, achieving the same performance for a wide range of sizes. From the graph of Figure 3 it is possible to infer that a VLIW Cache of 384-Kbyte is suitable for a DTSVLIW machine with the specified parameters.

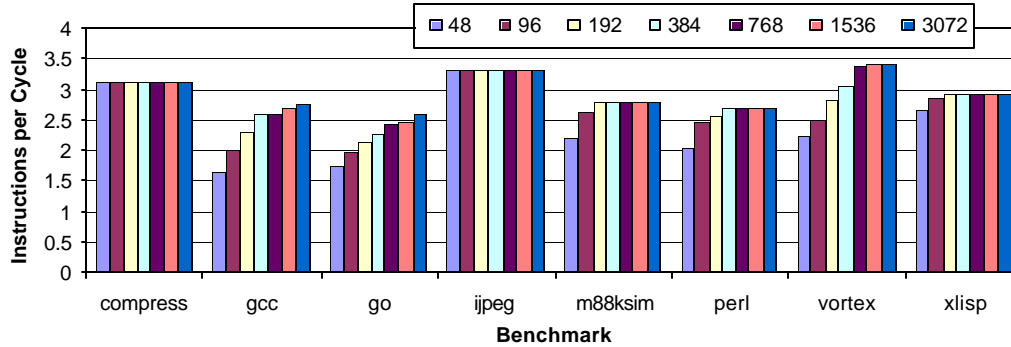


Figure 3: The DTSVLIW performance versus the VLIW Cache size in Kbytes

4. Conclusion and Future Work

This paper presents preliminary ILP performance measurements of the DTSVLIW architecture. The DTSVLIW architecture can be used to implement machines that execute code of current RISC or CISC ISA in a VLIW fashion, delivering ILP with backward code compatibility. This architecture takes advantage of the repetitive and localised pattern of instruction fetch addresses in current programs. The first time that program segments are executed, they are scheduled into long instructions and saved in a VLIW Cache; in the following executions, a VLIW Engine executes them in a VLIW fashion.

A DTSVLIW simulator has been implemented, parameterised, and instrumented. The effect of some parameters of the architecture on its performance has been evaluated using this execution driven simulator running the SPECint95 benchmark suit. The results show that ILP of more than 4 instructions per cycle can be achieved with the DTSVLIW.

The DTSVLIW architecture opens several new avenues of research. Next long instruction prediction, new VLIW Cache organisations, and new exception handling mechanisms are just a few examples.

Acknowledgements

The authors would like to acknowledge Eliseu Chaves Filho, from COPPE/UFRJ, for providing the source code of his scalar SPARC simulator, which is the base of the DTSVLIW simulator. The authors also thank Antonio Liotta, Jorge Ortega-Arjona, Tom Quick, and the anonymous referees for providing helpful comments on this paper.

5. References

- [1] B. R. Rau, "Dynamically Scheduled VLIW Processors", *Proc. of the 26th International Symposium on Microarchitecture*, pp. 80-92, 1993.
- [2] K. Ebcioglu, E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", *Proc. of the 24th International Symposium on Computer Architecture*, pp. 26-37, 1997.

- [3] R. Nair, M. E. Hopkins, "Exploiting Instructions Level Parallelism in Processors by Caching Scheduled Groups", *Proc. of the 24th International Symposium on Computer Architecture*, pp. 13-25, 1997.
- [4] A. F. de Souza and P. Rounce, "Dynamically Trace Scheduled VLIW Architectures", *Lecture Notes on Computer Science*, Vol. 1401, pp. 993-995, April 1998.
- [5] Sun Microsystems, "The Sparc Architecture Manual – Version 7", *Sun Microsystems Inc.*, 1987.
- [6] S. Davidson, D. Landskov, B. D. Shriver, P. W. Mallett, "Some Experiments in Local Microcode Compaction for Horizontal Machines", *IEEE Transactions on Computers*, Vol. C30, No. 7, pp. 460-477, July 1981.