# Dynamically Trace Scheduled VLIW Architectures

Alberto Ferreira de Souza 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.**

This paper presents a new architecture organisation, the *dynamically trace scheduled VLIW* (DTSVLIW), that can be used to implement machines that execute the code of current RISC or CISC instruction set architectures in a VLIW fashion, with backward code compatibility.

**Keywords:** VLIW, Superscalar, trace cache, scheduling.
**Area:** Architecture.

## Introduction

Superscalar architectures are currently of great interest as they have the potential to deliver *instruction level parallelism* (ILP), thus improving computer performance, without affecting the standard uniprocessor-programming model and with backward code compatibility. Current Superscalar machines fetch up to four instructions from cache each clock cycle, and attempt to schedule the execution of these in parallel across several functional units. Dependencies between instructions prevent full use of these units despite several algorithms to remove dependencies. Also, due to the physical layout of instruction caches, taken branches in the middle of a line cause the remainder of the line to be discarded, resulting in a *partial fetch*, reducing the instruction fetch bandwidth. This can occur on almost every fetch in current machines.

The *trace cache* architecture fetches instructions from the instruction cache and attempts to schedule them across multiple functional units [1]. The instructions that are executed are 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: this increases instruction bandwidth and throughput. This architecture is an improved Superscalar architecture and still has similar scheduling overheads.

VLIW processors [2] provide an alternative approach to ILP, using compiler techniques to construct lines of code to be executed in parallel. However, they cannot run existing instruction set architectures (ISAs) object code. It can be argued that the dynamic analysis made by of Superscalars is more effective than the static analysis of VLIWs. Much current research is examining ways to improve Superscalars and VLIW designs.

Object code compatibility is a problem for VLIW architectures: mapping a VLIW 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) has been presented by Rau [3]. This splits each instruction of a VLIW instruction into two components: *phase1* and *phase2*. The first is the original compiled instruction with an anonymous register assigned to the destination, while the second finally copies the result into the correct destination. This allows the compiled instructions to be executed on the available hardware regardless of dependencies on other instructions in the long instruction. When these have completed, the phase2 copy instructions can execute. 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 [4] with their DAISY machine can translate dynamically from the object code of an existing ISA architecture to the object code of a VLIW. When a fetch first accesses an address, a “VLIW translation missing” exception occurs into a Virtual Machine Monitor (VMM), implemented in software. The exception is handled by a fast algorithm that translates code starting in this address from base architecture instructions to VLIW primitives, which are scheduled into VLIW instructions stored in a protected section of the virtual address space. 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 reusability is probably appreciable, a hardware translation is possibly advantageous.

The Dynamic Instruction Formatting (DIF) concept (Nair and Hopkins [5]) performs hardware re-formatting of the fetched code. The original code is executed on a primary engine (a simple, less aggressively ILP processor). At the same time, the executed code is re-formatted into the DIF VLIW cache for execution by a VLIW engine. As with standard Superscalar designs, code dependencies have to be handled, but this is only done when the code is reformatted, not each time it is fetched from the DIF cache. This allows the extra speed of the VLIW engine to be fully utilised.

**The DTSVLIW Machine**

This paper presents the *dynamically trace scheduled VLIW* (DTSVLIW), that implements the DIF concept. This architecture organisation (Fig. 1) has 2 processing engines and 2 caches, an instruction cache for the original compiled code, executed by a scheduler engine, and a VLIW cache for VLIW instructions built from the code trace produced by the scheduler engine. The scheduler engine consists of a sequential processor in the current design (that implements the Sparc-7 ISA) and the key element, the scheduler unit. This uses a pipelined implementation of the FCFS (Fist Come First Served) scheduling algorithm, traditionally used in microcode compaction [6]. Nair and Hopkins [5] suggested, but did not present, a pipelined implementation. The organisation of our VLIW engine and VLIW cache are also different from theirs. The implemented version of the FCFS algorithm runs over a circular list – the *scheduling list* – which has a fixed number of long instructions, one per element of the list. Each element of the scheduling 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.

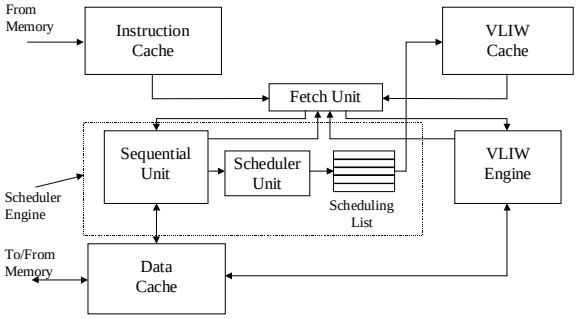


**Fig. 1.** DTSVLIW Machine.

In each clock cycle, the candidate instructions in each list element are checked for dependencies against the long instructions in the same and the next list element. If there is no dependency a candidate instruction is promoted to the next element, unless it is already in the head element; if there is a dependency the instruction is installed in its *companion* long instruction. Installing can be just that or it can involve splitting the instruction, as in the DSVLIW, by renaming the output register, promoting the instruction, and installing a copy instruction into the companion long instruction. An instruction coming from the Sequential Unit becomes a *candidate instruction*, depending on dependencies, in either the tail element of the list or in a new entry appended to the list. In the latter case, if there are no spare list elements, the whole list is installed in the VLIW cache as a block. This facilitates the addressing system.

When an instruction is first *placed* in the long instruction at the head of the scheduling list, the original memory address of this instruction becomes the long instruction's address, which is saved with it in the VLIW cache. The following long instructions in the list receive the same address plus a *specific-line* identification field, which is just the enumeration of its position in the list. This unification of long instructions into a *block* necessitates only a few bits to specify the fall-through long instruction address during VLIW execution. The last long instruction in a block has a special address field pointing to the fall-through block of long instructions.

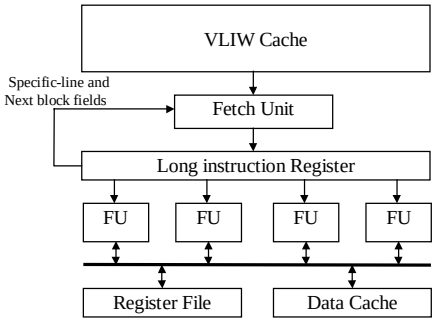


**Fig. 2.** The VLIW Engine

The VLIW Engine (Fig. 2) has a simple fetch-execute, two-stage pipeline. A decode stage is not necessary, since the long instructions are saved in the VLIW cache

already decoded. The fetch stage uses the specific-line field of current long instruction, a conditional branch target address, or the address of a fall-through block. All conditional branches are resolved in the execute stage. If one branch follows a different direction than that observed during scheduling during execution, the penalty is just one cycle if the target is in the VLIW cache.

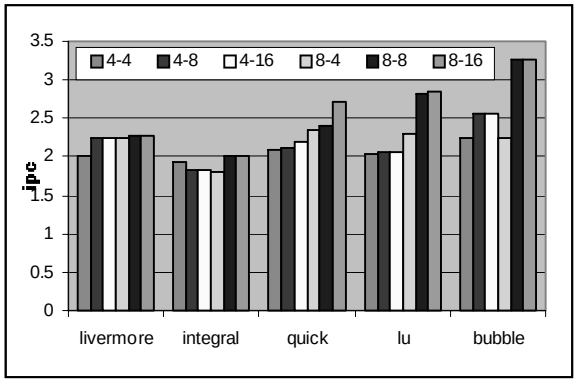


**Fig. 3.**

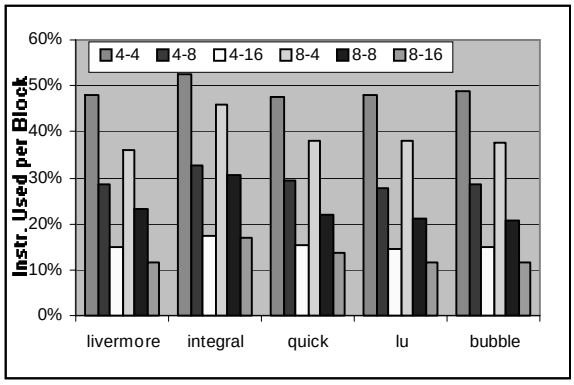


**Fig. 4.**

**Experiments and Discussion**

A simulator of the DTSVLIW was implemented and execution driven simulations was performed. One execution driven simulator permits a very precise evaluation of a new architecture, however it demands CPU power. For this reason, a set of small benchmarks was selected. The suite of test programs used includes *Livermore Loop 24* (9999 instructions executed); *Integral* evaluation of a function by the trapezoidal rule (30623 instructions); *Quick-Sort (62317); LU* decomposition (77158); and *Bubble-Sort (38245816)*. These programs were translated to Sparc code by the gcc compiler with -O2 optimisation. The number of instructions executed per cycle (ipc) achieved is presented in Fig. 3 (the legend specifies *block size - long instruction size*).

The current version of the FCFS algorithm is not able to take into account the latencies of multicycle instructions during scheduling: instructions are presumed to execute in one cycle. It is not acceptable to design a VLIW machine with all instruction with the same latency, since it will invalidate its best quality, the fast clock rate. Single cycle operation has only been chosen to see if the DTSVLIW merits further study in which case a multilatency version would be investigated. The current results encourage further investigations and a multilatency scheduler is been developed.

The algorithm used by the Scheduler Unit is not able to fill all long instructions completely in all blocks, even for small long instruction widths (4 instructions), resulting in poor utilisation of the silicon area used by the VLIW cache (Fig. 4.). The copy instructions generated by the algorithm augment the VLIW code size, demanding even more VLIW cache space. In order to make viable a cost-effective implementation of DTSVLIW machines, a different VLIW cache organisation may be necessary.

**References**

[1] J. E. Smith, S. Vajapeyam, "Trace Processors: Moving to Forth-Generation Microarchitectures", *IEEE Computer*, September 1997.

[2] J. A. Fisher, "The VLIW Machine: A Multiprocessor for Compiling Scientific Code", *IEEE Computer*, pp. 45-53, July 1984.

[3] B. R. Rau, "Dynamically Scheduled VLIW Processors", *Proceedings of the 26th Annual Symposium on Microarchitecture*, pp. 80-92, 1993.

[4] K. Ebcioglu, E. R. Altman, "DAISY: Dynamic Compilation for 100% Architectural Compatibility", IBM Research Report RC20538, 82 pages. 1996.

[5] R. Nair, M. E. Hopkins, "Exploiting Instructions Level Parallelism in Processors by Caching Scheduled Groups", IBM Research Report RC20628, 17 pages, 1996.

[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.