Effect of Multicycle Instructions on the Integer Performance of the Dynamically Trace Scheduled VLIW Architecture

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. *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 (ILP) with backward code compatibility. This paper presents the effect of multicycle instructions on the performance of a DTSVLIW architecture running the SPECint95 benchmarks.

1 Introduction

The classic approaches to providing ILP are VLIW and superscalar architectures. With VLIW, the compiler is required to extract the parallelism from the program and to build *Very Long Instruction Words* for execution. This leads to fast and (relatively) simple hardware, but puts a heavy responsibility on the compiler, and object code compatibility[4] is a problem. In Superscalar, the extraction of parallelism is done by the hardware which dynamically schedules the sequential instruction stream on to the functional units. The hardware is much more complex, and therefore slower than a corresponding VLIW design. The peak instruction feed into the functional units is lower for Superscalar. A number of approaches[1][2][3] have been examined that marry the advantages of the contrasting designs: the Superscalar dynamic extraction of ILP, the simplicity of the VLIW architectures. The approach presented here follows that first presented by Nair and Hopkins[3]. Our architecture, the *dynamically trace scheduled VLIW architecture* (DTSVLIW) [4], demonstrates similar results to theirs in providing significant parallelism, but with a simpler architecture that should be much easier to implement. In our earlier work[5], we had zero latency load/store instructions. Here we present results with more realistic latencies.

2 The DTSVLIW Architecture

The DTSVLIW has two execution engines: the Primary Processor and the VLIW Engine, and two caches for instructions: the Instruction Cache and the VLIW Cache. The Primary Processor, a simple pipelined processor, fetches instructions and does the first execution of this code. The instruction trace it produces is *dynamically scheduled* by the Scheduler Unit into VLIW instructions, saved in blocks to the VLIW Cache for re-execution by the VLIW Engine. The Primary Engine, executing the Sparc-7 ISA, provides object-code compatibility; the VLIW Engine VLIW performance and simplicity. The Scheduler unit works in parallel with the Primary Engine. Scheduling does not impact on VLIW performance as it does in Superscalar.

2.1 The Scheduler Algorithm

The major design problem is the scheduling, which has to maximise the parallelism extracted from the trace, while not extending the machine cycle time. The Scheduler Unit uses a pipelined version of the First Come First Served (FCFS) scheduling algorithm[6]. FCFS has advantages for hardware implementation: it operates with one instruction at a time in execution order; it produces optimum or near-optimum scheduling[6]. We showed it to be suitable for pipelined implementation in [5]. The Scheduler Unit uses a circular *scheduling list*, to build a block of VLIW instructions, using out-of-order execution, register renaming, and speculative execution. An instruction arriving from the Primary Engine is placed at the end of the block, moving up the block on subsequent clock cycles, dependencies allowing. The block starts with one element, increasing to a fixed block maximum. Only one instruction in each block element has to be checked for moving up on a cycle: moving up produces out-of-order execution. Speculative execution moves an instruction up past conditional branches, but delays its write-back stage until the branch outcomes are determined.

For a multicycle instruction, two copies, A and B, are inserted, separated by the instruction latency, into the scheduling list to identify the list region where the instruction is active. B's role is for dependency checking against instructions moving up. A and B are treated partly as other instructions, partly as one instruction, e.g. their separation is kept constant. B is not saved in the VLIW Cache. Scheduling a multicycle instruction lengthens a block by the latency of the instruction, impacting efficiency since the longer block is more difficult to fill, reducing parallelism.

3 Methodology and Experiments

A simulator of the DTSVLIW has been implemented in C. All results were produced with the simulator running in *test mode*: a *test machine* is run in tandem with the DTSVLIW. Comparison of the state of the 2 machines after an instruction or a VLIW block completes validates the DTSVLIW machine. The test machine measures the instructions executed to determine the ILP achieved. Model parameters, benchmark programs (SPECint95), together with the input sets used can be found in [5]. Each program was run for 50 million or more instructions, as counted by the test machine.

3.1 Effect of the Block Size and Geometry

Fig 1 shows the effect of the block size in terms of the number of instructions and block geometry (instructions per VLIW instruction (width) versus VLIW instructions per block (length) on performance. The experiments were performed with perfect instruction and data caches, large VLIW Cache (3072-Kbyte), and no next VLIW instruction miss penalty. The performance of machines with the same block sizes and different geometry is significantly different. Thus, the performance with 4x8 geometry is lower than with 8x4 geometry for all benchmarks. The block width and length affect the machine cost in different ways. Block width impacts on the number of functional units, data cache ports, and register file ports; the block length on the number of renaming registers, the length of load/store and checkpoint recovery lists [5], and the required size of the VLIW Cache for the same performance. To increase just the width or just the length of the block does not appear to be the best approach. A DTSVLIW with 8x8-block geometry generally performs better than with 4x16 and 16x4 geometries.

Benefits from large block size do not grow linearly. The performance of the 16x16 geometry on the ijpeg benchmark is extraordinary. This benchmark spends most of its execution in one loop. With a large enough block size, more than one iteration of the loop can be scheduled into a single block, allowing instructions from these iterations to be overlapped, extracting much greater parallelism.

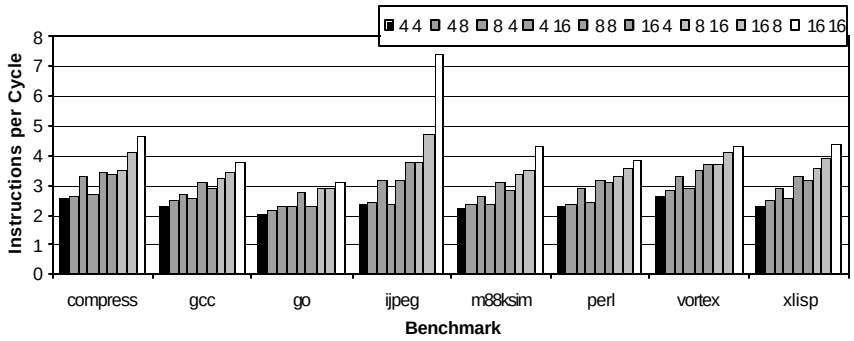


Fig 1. Variation of parallelism with the block size and geometry

3.2 Effect of the Load/Store Instructions Latency

Fig 2 shows the effect of the load/store instructions latency on an 8x8 geometry: LxSy stands for loads with latency *x* and stores with latency *y*. Latency is the number of cycles before an instruction's results can be used. Load latency has a severe impact – 25.4% average performance loss with 1-cycle and 50.2% with 2-cycle latency, because loads are frequent and their data is usually required imminently. Store latency does not have such a strong impact, as stored data is usually not imminently required.

With longer blocks the impact of Load/Store latency is smaller. For 8x16 (Fig 3) there is 20.5% average performance loss with 1-cycle latency and 42.6% with 2-cycle latency. With a longer block, there is more opportunity to accommodate instructions in the empty VLIW instructions created by the multicycle scheduling. This results in better scheduling and performance, but the latency impact is still high.

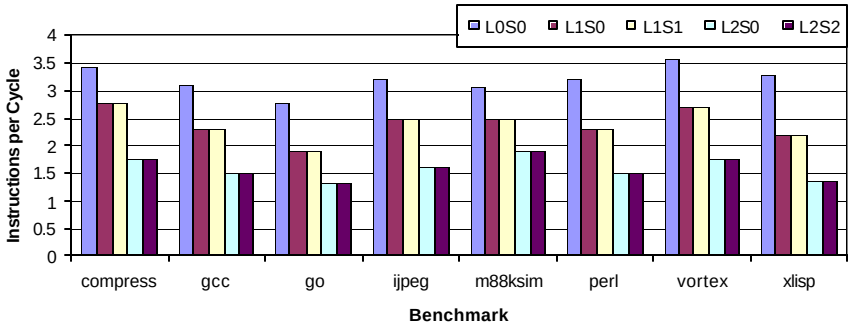
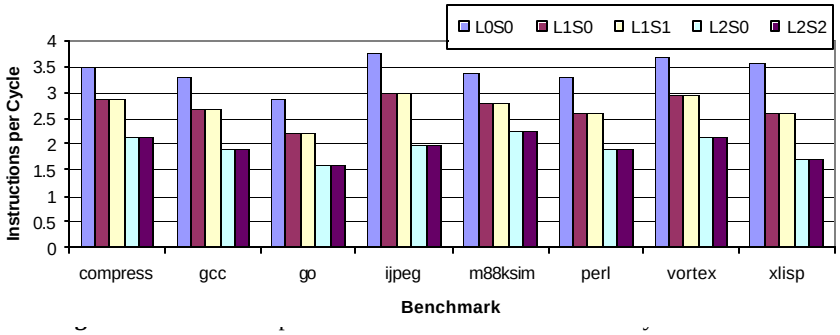


Fig 2. Variation of the parallelism with the load/store instructions latency – 8x8-block



4 Conclusion

The results show that the DTSVLIW can achieve ILP as high as 7 and average ILP superior to 4 with a large machine geometry. Multicycle load instructions impose a severe performance penalty on the DTSVLIW architecture and it is clear that it is important to get their latency as close to zero as possible: single cycle load operation. Single cycle stores are not so important. Low load/store latency (2 cycles) is achievable with a high frequency clock as demonstrated in the DEC-Alpha[7]. We calculated across all our benchmark results the average number of VLIW cycles per program with 8x16-block geometry of 98.57%, strongly suggesting that the DTSVLIW architecture is effective in taking advantage of its VLIW Engine. The Primary Processor and the VLIW Engine in the DTSVLIW can have high clock rates. The simplicity of the scheduling algorithm means that a similar high clock rate should be achieved for the Scheduler Unit, leading to an overall clocking rate similar to, if not higher than, high clock rate superscalar architectures, but achieving higher ILP.

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", *Proceedings of HPCN'98, in Lecture Notes on Computer Science*, Vol. 1401, pp. 993-995, April 1998.
5. A. F. de Souza and P. Rounce, "Dynamically Scheduling the Trace Produced during Program Execution into VLIW Instructions", *To be published in the Proceedings of 13th International Parallel Processing Symposium*, 1999.
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.
7. J. Keller, "The 21264: A Superscalar Alpha Processor with Out-of-Order Execution", *9th Microprocessor Forum*, 1996.