Code Scheduling for VLIW/Superscalar Processors with Limited Register Files

Tokuzo Kiyohara

Media Research Laboratory
Matsushita Electric Industrial Co., Ltd.
Kadoma-shi, Osaka, 571 Japan

John C. Gyllenhaal

Coordinated Science Laboratory
University of Illinois, Urbana-Champaign
Urbana, IL 61801

Abstract

Moderate size register files can limit the performance of loop unrolling on multiple issue processors. With current scheduling heuristics, a breadth-first scheduling of iterations occurs, increasing register pressure and generating excessive spill code.

A heuristic is proposed that causes a more depth-first scheduling of unrolled iterations. This heuristic reduces the overlapping of the unrolled iterations and as a result, reduces register pressure. The experimental evaluation shows increased performance on processors with 32 or 64 registers. In addition, the performance of dependency removing optimizations is stabilized, so that applying additional optimizations is more likely to increase performance.

1 Introduction

In multiple instruction issue processors, such as VLIW and superscalar processors, scheduling code for efficient usage of their function units requires many independent instruction sequences. Loop unrolling, combined with techniques to minimize or remove dependencies, has been shown to improve performance [1]. But with the current prescheduling heuristics and limited register file sizes, increasing the amount a loop is unrolled or using more advanced optimizations can decrease performance due to increased spill code.

This paper presents a new heuristic for use in the prescheduling of unrolled loops for processors with limited register file sizes. The heuristic reduces spill code which stabilizes the performance of loop unrolling and dependence removing optimizations.

In previous work, Hwu and Chang [2] showed that a prescheduling, register allocation, postscheduling sequence extracts more performance than postscheduling alone. Goodman and Hsu [3] showed that a prepass scheduler can avoid introducing excessive spill code by switching between two scheduling algorithms when the number of available registers passes a threshold. A promising alternative to the above, used by Multiflow [4], is to combine register allocation and code scheduling by treating registers as a resource. For this paper, the proposed heuristic is applied to the scheduler described by Hwu and Chang, but is also applicable to any other list scheduling [5] based system (such as Goodman and Hsu's scheduler).

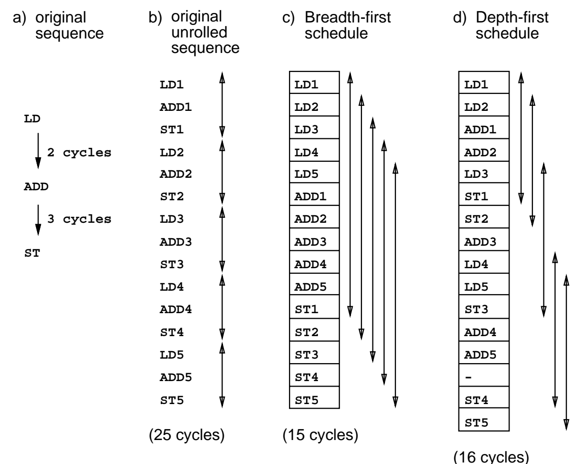


Figure 1: Breadth-first and depth-first scheduling of an unrolled loop

2 Scheduling to Reduce Spilling

Current scheduling algorithms, for a k-issue processor, choose the k highest priority instructions from those available each scheduled cycle (referred to as

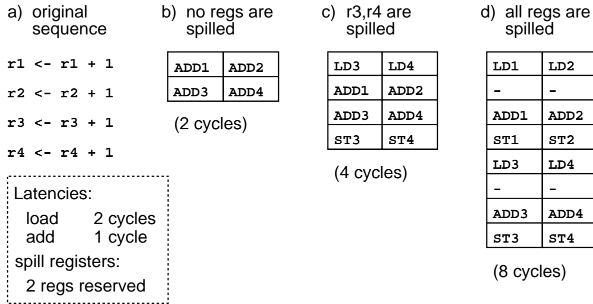


Figure 2: Effect of spilling registers

the **breadth-first** algorithm). The dominant heuristic measure for the priority is the instruction's height, which is defined as the length, in cycles, of the longest code sequence dependent on that instruction [6, 7]. Register pressure is proportional to how many times the loop is unrolled (since all iterations tend to be overlapped to some degree) and how much the iterations can be overlapped (which depends on the dependences between iterations). As a result, the performance of this algorithm with a limited register file is extremely sensitive to the amount the loop is unrolled and the dependence removing optimizations applied to the loop.

This paper proposes the use of the following **depth-first** algorithm that tends to reduce spill code by biasing the iteration's priorities so less overlap occurs. In each unrolled iteration, all the operation priorities (generated by the breadth-first algorithm) are biased (increased) by

$$(num_of_iterations - iteration_num) * bias_value$$

The bias value is chosen so large that there is no overlap in priority values between the iterations.

The effect of this heuristic is illustrated in Figure 1 on the unrolled body (Figure 1b) of a DOALL loop (Figure 1a). The breadth-first algorithm (Figure 1c) overlaps more iterations than the depth-first algorithm (Figure 1d). This overlap can result in a more compact schedule since it increases the number of instructions ready to schedule each cycle. It also increases the register pressure, and if this increases spilling during register allocation, the spill overhead can negate the benefits of the more compact schedule.

3 Hiding Spill Code Overhead

Three factors make hiding spill code overhead difficult: the spill code's occupation of instruction slots, the need to hide the latency of the spill loads, and the scheduling restrictions caused by dependencies between spill registers.

Instruction	Latency	Instruction	Latency
INT ALU	1	FP ALU	3
INT multiply	3	FP conversion	3
INT divide	10	FP multiply	3
branch	1/1-slot	FP divide	10
memory load	2	memory store	1

Table 1: Instruction latencies

The effect of spill code is illustrated in Figure 2. The fragment is prescheduled and register allocated for a two issue, in-order execution processor. In Figure 2b, all the virtual registers are allocated into physical registers. In Figure 2c, r3 and r4 are spilled out. The latency of the loads are hidden by moving the loads above the adds and the 2-cycle overhead is caused solely by the need for extra instruction slots. In Figure 2d, all the virtual registers are spilled out. LD3 and LD4 cannot be moved above ST1 and ST2 because only two spill registers are available. This results in the introduction of idle slots. It's possible that the idle slots could be filled, but the artificial dependencies caused by register allocation limits code movement.

In practice, when the ratio of spill code to total code size is small, most of the spill code can be moved into idle slots, and latencies hidden by code movement. But after some threshold, it becomes extremely difficult to hide the spill code overhead and that is when performance degrades severely. As issue rate increases, this threshold lowers because it becomes more difficult to hide the latency of the spill code.

4 Evaluation Methodology

The performance analysis is done using 29 unrolled loop nests (only inner loop unrolled) drawn from the PERFECT club benchmark suite [1]. They were compiled and scheduled using the IMPACT-I compiler [7]. Scheduling consists of a prescheduling (using either the breadth-first or depth-first algorithm), register allocation, postscheduling (always using the depth-first algorithm) sequence. Note that only prescheduling is varied because only it can effect register usage.

The underlying microarchitecture is assumed to have homogeneous function units, deterministic instruction latencies (Table 1), CRAY-1 style interlocking, and in-order execution. The instruction set is a RISC assembly language similar to the MIPS R2000

Level	Optimizations
Level 1	loop unrolling and conventional scalar processor optimizations
Level 2	register renaming and operation migration (+ Level 1)
Level 3	operation combining, strength reduction, and height reduction (+ Level 2)
Level 4	induction and accumulator variable expansion (+ Level 3)

Table 2: Definition of optimization levels

instruction set. The execution time of each loop nest, assuming a 100% cache hit rate, is derived using execution-driven simulation.

This study varies the processor's issue rate (2, 4 and 8 issue), floating point register file size (32[/16], 64[/32], 128[/64] or an unlimited number of single[/double] precision floating point registers), and the optimization level (Table 2) [1]. The register allocator uses a graph coloring algorithm that utilizes profile information in its priority calculations. For all register file sizes, 8 single [4 double] precision floating point registers are reserved as spill registers. They are allocated in a round robin fashion and a peephole optimization removes redundant spill loads and stores. The integer register file size is fixed at 64 registers.

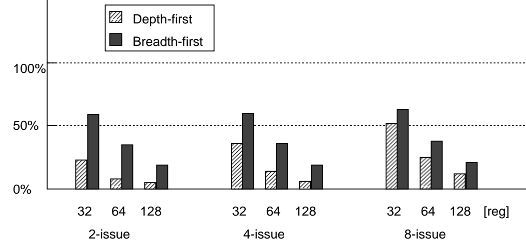
5 Results

Since the performance of the depth-first algorithm comes from reducing spill code, the loops are broken up into the following two groups: Group-1, made up of 19 loops, where moderate amounts of spill code is generated by the breadth-first algorithm and Group-2, containing 10 loops, where large amounts of spill code is generated.

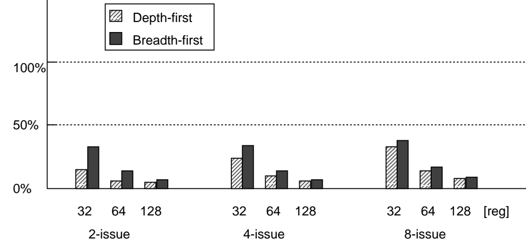
Figure 3 shows the percentage increase in code size due to spill code for the loops optimized at level 4 and scheduled with both algorithms. This figure shows that the depth-first algorithm successfully reduces register pressure (therefore spill code) in unrolled loops. The spill code generated by the breadth-first algorithm is independent of issue-rate because it maximizes iteration overlap even when less slots need to be filled.

Register presure can also be reduced by applying fewer dependence removing optimizations. For example, if the optimization level is reduced to 1, less than 20% spill code is generated for the group-2, 32 registers case, and much less for larger register file sizes.

a) All Loops



b) Group-1



c) Group-2

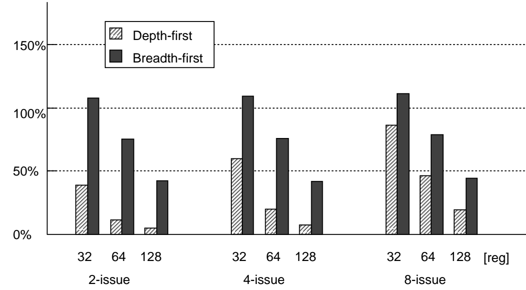
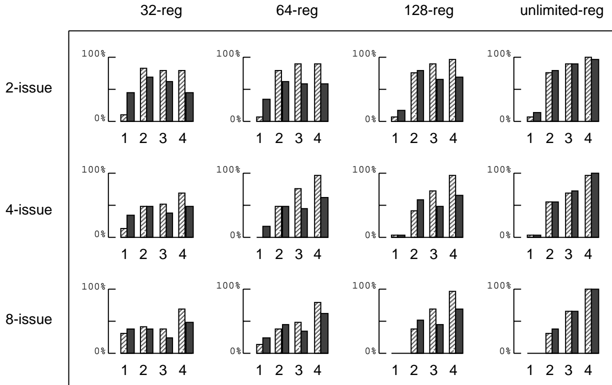


Figure 3: Percentage increase in code size at optimization level 4 due to spill code

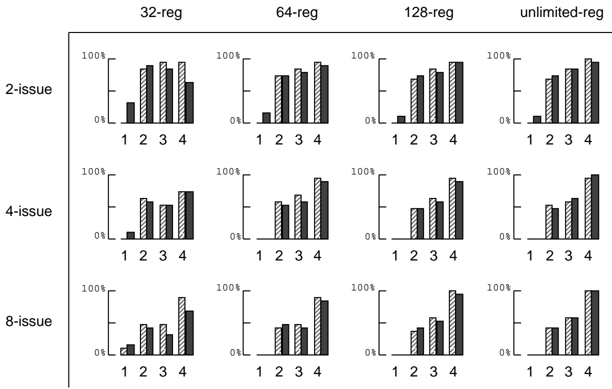
This yields a speedup of approximately 1.7 for issue 2, and 1.9 for issue 4 and 8 (for all register file sizes).

The effect of varying the optimization level is presented in Figure 4. The vertical bars indicate the percentage of loop nests that achieve at least 95% of the performance achievable by the best optimization level (for each loop nest and processor configuration) if the indicated optimization level is used. This figure shows the lack of stability in the optimizations performance when the number of registers are limited. For the breadth-first algorithm, there is no clear choice about what optimization level to use in the 128 or less register cases. For the depth-first algorithm, although the choice is still not clear, average performance is maximized by choosing optimization level 4. A exception is the group-2, 32 register case, where the extreme register pressure (code size almost doubles due to spill

a) All Loops



b) Group-1



c) Group-2

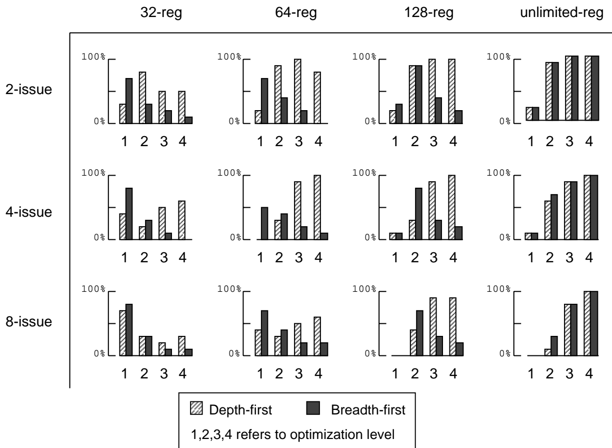


Figure 4: Percentage of loop nests that achieve at least 95% of the performance achievable by the best optimization level (for each loop nest and processor configuration) if the indicated optimization level is used

code) causes severe performance degradation at optimization levels above 1.

In Figure 5, the shaded bars show the speedup of the loops using optimization level 4 for various register file sizes and issue rates. In addition, the white bars indicate the speedup if, for each loop, the optimization level that yielded the best performance was chosen (as Figure 4 indicated, this is not always the highest optimization level). The base configuration for the speedup calculations is a single issue processor with an unlimited number of registers using conventional scalar processor optimizations. As expected, group-1 does not benefit from depth-first scheduling (-5% to 4% improvement for 64 registers) because spill code does not significantly degrade performance. However, group-2 benefits greatly (69% to 111% improvement for 64 registers) due to the spill code reduction. This averages out to an overall performance improvement ranging from 14% to 18% for 64 registers. Similar results occur with 32 registers with the overall performance improvement ranging from 8% to 38% (the 8% coming from the 8-issue model with 32 registers which is not realistic). The white bars show that when breadth-first scheduling is given too much scheduling freedom (from dependence removing optimizations), performance is often lost due to excessive spill code. The depth-first algorithm is more stable, so that applying additional dependence removing optimizations usually yields improved performance.

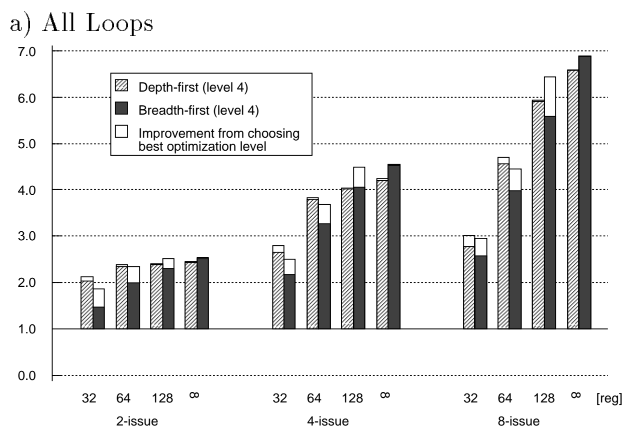
6 Conclusion

Moderate size register files can limit the performance of loop unrolling. This is partially due to the current scheduling heuristics interweaving of unrolled iterations. This breadth-first scheduling of iterations increases register pressure and generates excessive spill code when the number of registers is limited.

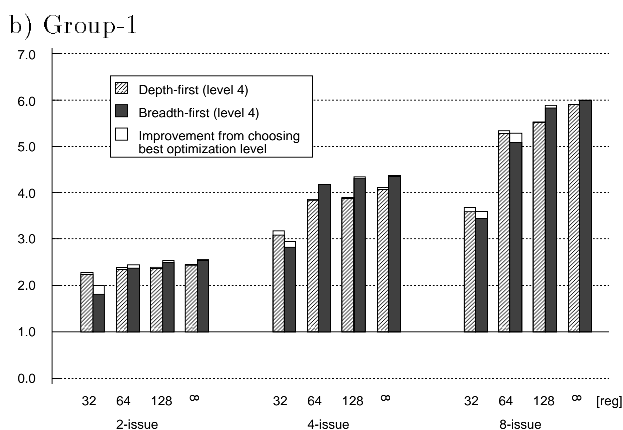
In order to reduce the performance degradation, a heuristic was proposed that causes a more depth-first scheduling of unrolled iterations. This reduces the overlapping of the unrolled iterations, which decreases the register pressure. But it also tends to produce a less efficient schedule for unlimited or large register files.

The experimental evaluation shows that this heuristic increases performance on processors with 32 or 64 registers. In addition, scheduling with this heuristic is more stable, so that applying additional dependence removing optimizations is less likely to decrease performance. Both of these features make this heuristic

a) All Loops



b) Group-1



c) Group-2

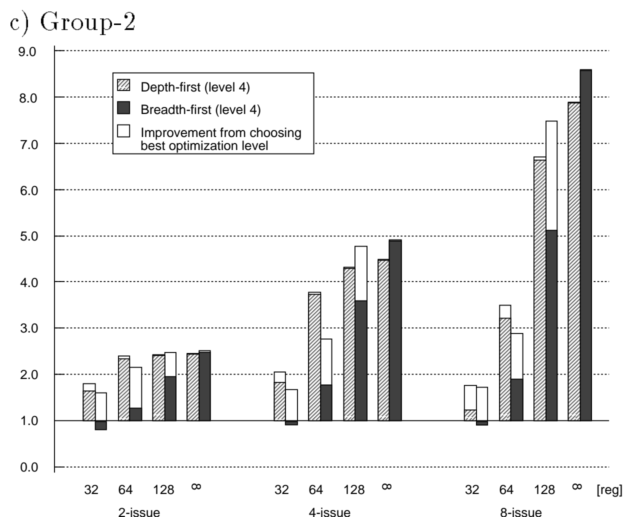


Figure 5: Speedup (shaded bars) from using the highest optimization level (level 4) and the improvement (white bars) from using the best optimization level for each loop

valuable for multiple issue processors with limited register files.

Acknowledgements

The authors would like to thank Wen-mei Hwu and all members of the IMPACT research group for their comments and suggestions. Special thanks to the anonymous referees whose comments and suggestions helped to improve the quality of this paper significantly.

John Gyllenhaal was supported by a NSF fellowship.

References

[1] S. A. Mahlke, W. Y. Chen, J. C. Gyllenhaal, W. W. Hwu, P. P. Chang, and T. Kiyohara, "Compiler code transformations for superscalar-based high-performance systems," in *Proceedings of Supercomputing 92*, November 1992.

[2] W. W. Hwu and P. P. Chang, "Exploiting Parallel Microprocessor Microarchitectures with a Compiler Code Generator," in *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 45–53, June 1988.

[3] J. R. Goodman and W. C. Hsu, "Code Scheduling and Register Allocation in Large Basic Blocks," in *Proceedings of the 1988 International Conference on Supercomputing*, pp. 442–452, July 1988.

[4] Freudenberger, S. and Ruttenberg, J., "Phase Ordering of Register Allocation and Instruction Scheduling," in *Code Generation - Concepts, Tools, Techniques*, Dagstuhl, Germany, May 1991.

[5] Edward G. Coffman, Jr., ed., *Computer and Job/Shop Scheduling Theory.* John Wiley & Sons, 1976.

[6] J. R. Ellis, *Bulldog: a compiler for VLIW architectures.* Combridge, MA: The MIT Press, 1986.

[7] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.