Software Pipelining and Register Pressure in VLIW Architectures: Preconditionning Data Dependence Graphs is Experimentally Better Than Lifetime-Sensitive Scheduling

Frédéric Brault [1,2,3], Benoît Dupont-de-Dinechin [3], Sid-Ahmed-Ali Touati [1,2], Albert Cohen [1]
[1] INRIA-Saclay [2] University of Versailles Saint-Quentin en Yvelines
[3] Kalray, Montbonnot, France

Abstract

Embedding register-pressure control in software pipelining heuristics is the dominant approach in modern back-end compilers. However, aggressive attempts at combining resource and register constraints in software pipelining have failed to scale to real-life loops, leaving weaker heuristics as the only practical solutions. We propose a decoupled approach where register pressure is controlled before scheduling, and evaluate its effectiveness in combination with three representative software pipelining algorithms. We present conclusive experiments in a production compiler on a wealth of media processing and general purpose benchmarks.

1 Introduction

Most SoftWare Pipelining (SWP) methods used in production combine registers and resources constraints in a single optimisation process. Several papers highlight the effectiveness of such approaches on small kernels [1, 2, 3, 4]. Unfortunately, combined methods do not scale well to real-world applications where loops may have hundreds of instructions[1], leaving weaker heuristics as the only practical alternatives. In this paper, we consider three SWP algorithms representative of common SWP strategies, and we demonstrate that a decoupled approach scales better on a real VLIW architecture and in a production compiler.

[1]Loop splitting may be used sometimes to reduce the size of a loop, but may be prohibited by the strongly connected components of the DDG; also, loop splitting increases the loop overhead, and may reduce data locality utilisation.

To the best of our knowledge, SIRA [5] (Schedule Independent Register Allocation) is the only optimization framework capturing periodic register constraints before SWP, enforcing them through a preconditioning of the data dependence graph (DDG). This framework has four advantages compared tp other register optimisation methods: (1) it does not interfere with the SWP method itself, allowing the scheduling algorithm to focus on its core throughput objective; (2) it is formally defined with proven theorems; (3) its theoretical model is compatible with most existing VLIW architectures (e.g., multiple registers types and delayed accesses to registers, modeling of buffers and rotating register files, etc.); (4) it is a released free software and the implementation is independent of an existing compiler [?].

2 Experimental setup

Our experimental setup is based on `st200cc`, a production compiler from STMicroelectronics based on `Open64`, whose code generator has been extensively rewritten in order to target the STMicroelectronics ST200 VLIW processor family. The `st200cc` compiler augments the `Open64` code generator with super-block instruction scheduling optimisations, and includes three variants of software pipeliners: (1) an optimal expensive one using integer linear programming that computes a minimal *II* under resource and data dependence constraints; (2) a heuristic software pipeliner under resource constraints based on the unwinding approach that allows the optimisation of larger loops (multiple hundreds of instructions and arcs); (3) a lifetime sensitive heuristic based on a generalised variant of decomposed software pipelining [2, 6]. The number of architectural registers considered in our experiments is 32 general purpose registers and 4 branch registers. These numbers are representative of the size of embedded VLIW characteristics currently used in the market.

We insert SIRA, the DDG preconditioner, just before the SWP pass. For each variant of the three possible SWP methods, we study the impact on the final code quality when we activate or deactivate SIRA; combinations of these options result in 6 different compilation flows.

We consider the MEDIABENCH and FFMPEG benchmarks, representative of multimedia VLIW computing, as well as all C and C++ SPEC CPU2000 applications (there is no Fortran compiler for ST231).

For each benchmark, we generate 6 code variants depending on the 6 possible compilation flows: the 3 classical SWP options plus the 3 SIRA-constrained ones. We then analyse the code quality when we apply a DDG preconditioning through SIRA just before SWP. The two following sections demonstrate that this is a better choice than relying on combined resource/register-constrained SWP.

2.1 Impact of dependence graph preconditionning on *II*

For each benchmark, and for each variant of SWP, we measured the *II* variation as $\frac{\sum II_0 - \sum II_1}{\sum II_0}$ where II_0 is the *II* produced by the compiler if we do not activate SIRA, and II_1 is the produced *II* by the compiler when we apply SIRA before SWP. Tab. 1 shows the results, where O-SWP stands for the optimal SWP (with integer linear programming), U-SWP stands for the unwinding heuristic SWP, and LS-SW stands for the lifetime-sensitive SWP. As we can see, *II* always decreases when applying SIRA before SWP (for any of the 3 used SWP methods). Note that the reduction obtained against optimal SWP is sometimes exaggerated due to the timeout on the integer linear programming solver, resulting in a suboptimal schedule on the larger loops.

Bench	O-SWP	U-SWP	LS-SWP
FFMPEG	-22.1%	-26.93%	-2.86%
MEDIABENCH	-15.86%	-24.69%	-3.4%
SPEC2000	-14.89%	-27.03%	-4.93%

Table 1: *II* variation

2.2 Impact of dependence graph preconditionning on spill code

For each benchmark, and for each variant of SWP, we measured the spill count reduction as $\frac{\sum \#\text{spill}_0 - \sum \#\text{spill}_1}{\sum \#\text{spill}_0}$ where $\#\text{spill}_0$ is the number of spill operations produced by the compiler if we do not activate SIRA, and $\#\text{spill}_1$ is the number of spill operations produced by the compiler when we apply SIRA before SWP. Tab. 1 shows the results. As we can see, the spill count always decreases in impressive proportions when we apply SIRA before SWP (for any of the 3 used SWP methods), even against a lifetime SWP.

Bench	O-SWP	U-SWP	LS-SWP
FFMPEG	-50.8%	-61.8%	-12.34%
MEDIABENCH	-46.9%	-65.15%	-24.45%
SPEC2000	-39.31%	-62.16%	-24.91%

Table 2: Spill count reduction

3 Conclusion

We integrated SIRA into a production compiler for embedded VLIW processors, and evaluated its potential to reduce the spill code and the initiation interval in the real world. Our experimental results are conclusive: using SIRA significantly decreases both *II* and spills, for all schedulers; of course, results are less impressive on a lifetime-sensitive scheduler, since the heuristic already reduce register pressure; nevertheless, the combination of SIRA with an aggressive scheduler outperforms a lifetime-sensitive heuristic.

Acknowledgments F. Brault was partly supported by a thesis scholarship from Digiteo foundation (contract number 2007-10D). This work was conducted while B. Dupont-de-Dinechin was working at STMicroelectronics.

References

[1] Santosh G. Nagarakatte and R. Govindarajan, "Register Allocation and Optimal Spill Code Scheduling in Softw are Pipelined Loops Using 0-1 Integer Linear Programming Formulation," in *CC*, Springer, Mar. 2007.

[2] Benoît Dupont-de-Dinechin, "Parametric Computation of Margins and of Minimum Cumulative Register Lifetime Dates," in *LCPC*, 1996.

[3] Richard A. Huff, "Lifetime-sensitive modulo scheduling," in *PLDI*, (New York, NY, USA), pp. 258–267, ACM, 1993.

[4] Alexandre E Eichenberger and Edward S. Davidson, "Efficient formulation for optimal modulo schedulers," *SIGPLAN Notice*, 1997.

[5] Sid-Ahmed-Ali Touati and Christine Eisenbeis, "Early Periodic Register Allocation on ILP Processors," *PPL*, 2004.

[6] Jian Wang and Christine Eisenbeis and Martin Jourdan and Bogong Su, "Decomposed software pipelining: A new perspective and a new approach," *IJPP*, vol. 22, pp. 351–373, June 1994.