RETROSPECTIVE:

Software Pipelining: An Effective Scheduling Technique for VLIW Machines

Monica S. Lam
Computer Systems Laboratory
Stanford University

ABSTRACT

The basic idea behind software pipelining was first developed by Patel and Davidson for scheduling hardware pipe-lines. As instruction-level parallelism made its way into general-purpose computing, it became necessary to automate scheduling. How and whether instructions can be scheduled statically have major ramifications on the design of computer architectures. Rau and Glaeser were the first to use software pipelining in a compiler for a machine with specialized hardware designed to support software pipelining. In the meantime, trace scheduling was touted to be the scheduling technique of choice for VLIW (Very Long Instruction Word) machines. The most important contribution from this paper is to show that software pipelining is effective on VLIW machines without complicated hardware support. Our understanding of software pipelining subsequently deepened with the work of many others. And today, software pipelining is used in all advanced compilers for machines with instruction-level parallelism, none of which, except the Intel Itanium, relies on any specialized support for software pipelining.

1. BACKGROUND

The story of software pipelining cannot be told without also telling the story of instruction-level parallelism (ILP) in computer architectures. The idea of software pipelining originated from Patel and Davidson as they studied the problem of creating hardware pipelines out of a collection of resources. In their model, each task, once initiated, flows from resource to resource synchronously according to the pipeline schedule. Resources may be reused in a pipeline; a new task can be initiated as long as its resource usage does not collide with those of previous tasks still in progress. The objective was to minimize the initiation interval between tasks and thus maximize the pipeline's throughput.

Patel and Davidson showed that inserting proper buffers or delays between resources in the pipeline can eliminate resource collisions and improve throughput. They proposed a branch-and-bound algorithm that inserts the minimum number of delays into a schedule that allows tasks be executed at a given initiation interval. It is easy to see the correspondence between this problem and software pipelining. Each task in the pipeline is similar to an iteration in a loop. The main difference is that tasks in their pipelines are independent of each other, whereas iterations in a loop may have dependences between them.

The evolution of statically scheduled ILP machines started with having micro-instructions controlling parallel and pipe-lined functional units on a processor. Kogge showed that Patel and Davidson's methodology can be applied to hand microprogramming. Rau and Glaeser developed the first compiler with software pipelining for the polycyclic architecture, which had a novel crossbar whose cross-points provided a programmable form of delay to support software pipelining[7].

Integrated floating-point adder and multiplier chips were first introduced by Weitek Inc. in 1984. This led to a flurry of activities in developing "high-performance" numerical engines, among which were the commercial Multiflow Trace computer and the CMU Warp systolic array. The latter research project provided the context in which this paper was written. The availability of integrated floating-point chips was, of course, just a milestone along the road to single-chip ILP processors. Today, all modern general-purpose machines employ ILP and instruction scheduling is needed in all optimizing compilers.

2. CONTRIBUTIONS OF THE PAPER

At the time the paper was written, trace scheduling[5] was considered to be the technique of choice for scheduling VLIW (Very Long Instruction Word) machines. This paper establishes software pipelining as a useful static scheduling technique for VLIW processors without requiring specialized architectural support.

This paper has three major results. First, instead of relying on specialized hardware support like a polycyclic interconnect, this paper shows that the same effect can be achieved by using modulo variable expansion and unrolling the generated code a small number of times. Second, unlike hardware pipelining, dependences may cross iteration boundaries in a loop, thus creating cycles in the dependence graph. This paper shows how scheduling one instruction in a strongly connected component can severely constrain the schedule of all other instructions in the same component. A heuristic technique must take this fact into consideration to be efficient. Third, this paper handles conditional statements in a loop using hierarchical scheduling.

This paper would have little impact had it not included a thorough evaluation of the technique. I owed the results of this work to my Ph. D. advisor, H. T. Kung, and the rest of the Warp research team. Warp was a great research project. We did not develop just one VLIW processor architecture, but a cooperating systolic array of ten VLIW machines. We built all the system software as well as a large image-processing library so that the machine could be used as a signal processor to drive a modified GM van autonomously. The project would not have been possible without Thomas Gross, who led the compiler effort, Onat Menzilcioglu, who designed and built the Warp processor, and John Webb, who led the image-processing effort. Many others contributed to the project including Marco Annaratone, Emmanuel Arnould, C. H. Chang, Robert Cohn, Peter Lieu, Abu Noaman, Ken Sarocky, John Senko and David Yam.

The Warp processor, with its many registers and very wide instruction word, was relatively easy to schedule. Even though I knew that my algorithm was unlikely to handle well machines with less resources, our goal to make Warp a fully working prototype left me with no time to explore algorithms for alternate processor architectures. After I graduated from Carnegie Mellon University, I joined the faculty at Stanford and worked on the problem of parallelization, which, interestingly enough, was the original topic for my Ph. D. dissertation.

3. SUBSEQUENT DEVELOPMENT

The most significant omission from my algorithm is the minimization of register usage, which is a problem analogous to minimizing delays in Patel and Davidson's formulation. Huff was one of the first to address the register usage issue and proposed a scheduling technique that minimizes register lifetimes[4].

How well software pipelining works depends greatly on the composition and the scheduling constraints of the loop to which it is applied. Loop transformations should first be performed before software pipelining so as to increase the loops' parallelism and data locality, improve their balance of resource usage, and reduce their register pressure[1, 11].

On the theoretical front, Feautrier[2] and Gao's research group[3, 6] showed that software pipelining with resource constraints and register usage minimization can be solved optimally with integer linear programming. It is interesting to note that their formulations also iterate through the target initiation intervals, just like the original heuristic technique.

"Is this theoretically elegant approach real?" I challenged Gao in 1995 and suggested settling this question by comparing optimal scheduling with SGI's state-of-the-art software pipeliner. The result of this challenge was one of my favorite papers, "Software Pipelining Showdown: Optimal vs. Heuristic Methods in a Production Compiler"[9]. The SGI scheduler was carefully tuned for the SGI R8000 and used multiple heuristic strategies to find the best schedule. The comparison showed that the heuristic schedules produced by the SGI compiler were near-optimal. What was more surprising was to discover how well the optimal scheduler handled the real complexities of a machine and how competitive the academic implementation was relative to a state-of-the-art industrial effort.

4. ARCHITECTURAL IMPLICATIONS

Software pipelining has a significant influence on computer architecture design. Because of its effectiveness on numerical applications, numerical engines such as digital signal processors do not need expensive dynamic scheduling hardware. Software pipelining is also useful for general-purpose machines with dynamic scheduling as it gives priority to operations on the critical path and utilizes hardware resources more effectively.

Software pipelining was originally applied to machines with specialized support. The polycyclic architecture had an expensive crossbar with programmable delays[7], whereas the Cydra 5 machine used rotating registers and predicated execution[8]. This paper suggests that specialized support is unnecessary. If we unroll the generated loop body a small number of times, all the hardware that is needed is an instruction cache, which is a common and generally useful machine feature. Besides, code expansion due to software pipelining is limited because only small inner loops can benefit from this technique. Software pipelining has been used in processors from HP, IBM, Intel, SGI, Sun and Texas Instrument, none of which has any specialized software pipelining support.

More recently, hardware support for software pipelining re-emerged in the Itanium architecture[10] in the form of rotating registers, similar to those used in the Cydra 5 machine. The Itanium, however, does not have a dynamic scheduler which is found in all other modern processor architectures. Software pipelining is applicable only to codes with predictable behavior like numerical applications; as such, it only expands the number of instructions in innermost loops slightly. On the other hand, the behavior of non-numeric applications is much less predictable; without a dynamic scheduler, an aggressive static scheduler needs to generate codes for many alternate paths, which can lead to code bloat. Specialized software pipelining support does not seem to be warranted on a general-purpose machine. By complicating the design unnecessarily, it makes other important functionality such as dynamic scheduling harder to provide.

REFERENCES

[1] S. Carr and K. Kennedy. Improving the ratio of memory operations to floating-point operations in loops. *ACM Transactions on Programming Languages and Systems*, 16(6):1768–1810, November 1994.

[2] Paul Feautrier. Fine-grain scheduling under resource constraints. In *The 7th Annual Workshop on Languages and Compilers for Parallel Computing*, pages 1–15, 1994.

[3] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 85–94, November 1994.

[4] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 258–267, June 1993.

[5] P. G. Lowney, S. M. Freudenberger, T. J. Karzes, W. D. Lichtenstein, Nix R. P., J. S. O'Donnell, and J. C. Ruttenberg. The Multiflow trace scheduling compiler. *The Journal of Supercomputing*, 7(1-2):51–142, 1993.

[6] Qi Ning and Guang R. Gao. A novel framework of register allocation for software pipelining. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 29–42, Charleston, South Carolina, 1993.

[7] B. R. Rau and C. D. Glaeser. Some scheduling techniques and an easily schedulable horizontal architecture for high performance scientific computing. In *Proceedings of the 14th Annual Workshop on Microprogramming*, pages 183–198, October 1981.

[8] B. R. Rau, D. W. L. Yen, W. Yen, and R. A. Towle. The Cydra 5 departmental supercomputer. *IEEE Computer*, 22(1):12–35, January 1989.

[9] J. Ruttenberg, G. R. Gao, A. Stoutchinin, and W. Lichtenstein. Software pipelining showdown: optimal vs. heuristic methods in a production compiler. In *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*, pages 1–11, 1996.

[10] M. S. Schlansker and B. R. Rau. EPIC: explicitly parallel instruction computing. *IEEE Computer*, 33(2):37–45, February 2000.

[11] M. E. Wolf, D. E. Maydan, and D.-K. Chen. Combining loop transformations considering caches and scheduling. In *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pages 274–286, December 1996.