Software Prefetching for Software Pipelined Loops

F. Jesús Sánchez and Antonio González

Department of Computer Architecture
Universitat Politècnica de Catalunya
Campus Nord - c./ Jordi Girona, 1-3 - Mòdul D6
08034 - Barcelona (SPAIN)
E-mail: {fran,antonio}@ac.upc.es

Abstract

This paper investigates the interaction between software pipelining and different software prefetching techniques for VLIW machines. It is shown that processor stalls due to memory dependences have a great impact into execution time. A novel heuristic is proposed and it is show to outperform previous proposals.

1. Introduction

Software pipelining represents a family of loop scheduling techniques that tries to exploit ILP by executing in parallel consecutive iterations of a loop. The most popular scheme is called modulo scheduling, and it consists of finding a fixed pattern of operations (of length *II* or *initiation interval*) from distinct iterations([3]).

Several schemes have been proposed in the literature with the goal of minimize the *II* and/or register pressure, but none of them has evaluated the effect of memory. When software pipelining is applied in VLIW architectures, where instruction latencies and scheduling are fixed at compile-time, execution time can be highly degraded due to the stall time provoked by dependences with memory instructions. Even if a nonblocking cache is used, true dependences with previous memory operations at a near distance[1] can make the processor to stall afterwards. The choice of scheduling all loads using the cache-miss latency requires considerable ILP and increases register pressure([1]).

Different techniques to improve memory behavior exist and are well-known, and software prefetching is one of them. The main idea of this method is to bring to cache the data that will be used in a near future([2]).

In this paper we investigate the interactions between software pipelining and software prefetching in a VLIW architecture. Some alternatives to perform software prefetching are described, and a novel heuristic is presented. An evaluation in execution time terms is reported as well as some conclusions.

1.Almost all modulo scheduling schemes use a fixed cache-hit latency for all memory operations

2. Software prefetching schemes

Software prefetching is an effective technique to tolerate memory latency. When it is used with a nonblocking cache, this technique allows the processor to hide part or all the memory latency by overlapping the fetch of data and the computation.

Software prefetching can be performed through two alternative schemes: binding and nonbinding prefetching. The first alternative, also known as early scheduling of memory operations, moves memory instructions away from those instructions that depend on them. The second alternative introduces in the code special instructions, which are called prefetch instructions. These are nonfaulting instructions that perform a cache lookup but do not modify any register.

In the study presented in this paper we have evaluated two techniques of binding prefetching:

- *Early scheduling always* (ESA): all memory operations of the loop are scheduled using cache-miss latency.
- *Early scheduling according to locality* (ESL): schedule instructions that have some type of locality using the cache-hit latency and schedule the remaining ones using the cache-miss latency.

We have also evaluated three distinct schemes for inserting prefetch instructions (nonbinding prefetch):

- *Insert prefetch always* (IPA): insert a prefetch instruction for every memory operation.
- *Insert prefetch according to temporal locality* (IPT): insert prefetch for those references without temporal locality even if they exhibit spatial locality.
- *Insert prefetch according to locality* (IPL): insert prefetch for those instructions without any type of locality.

3. A novel software prefetching technique

The proposed software prefetching scheme is called *cache sensitive modulo scheduling* (CSMS), and it tries to minimize both the compute time and the stall time. These terms are not independent and reducing one of them may

1060-3425/98 $10.00 (c) 1998 IEEE

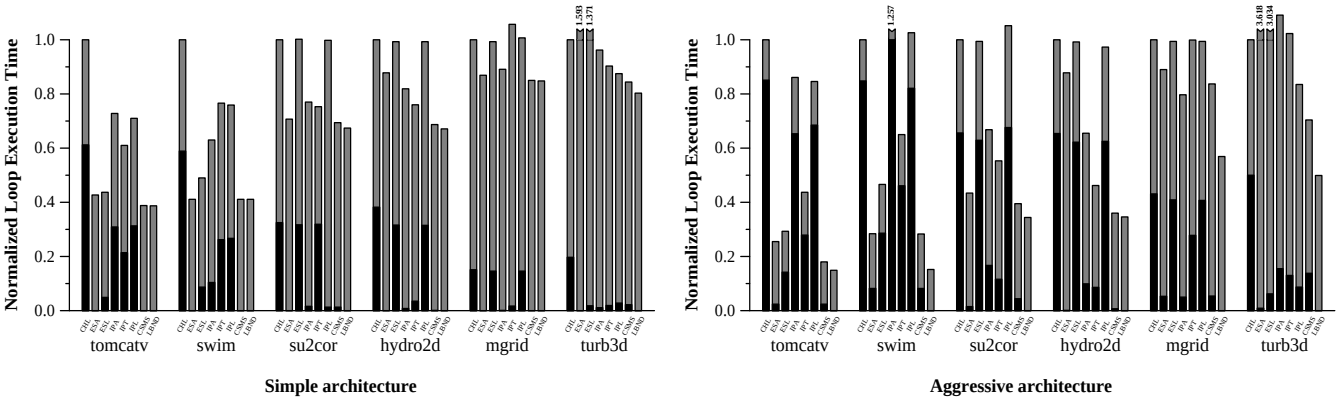

Figure 1. Software prefetching schemes performance

result in an increase in the other. The proposed algorithm tries to find the best trade-off between the two terms.

The CSMS algorithm is based on early scheduling of some selectively chosen memory operations. Scheduling a memory operation using the cache-miss latency can hide almost all memory latency without increasing much the number of instructions (as opposed to the use of prefetch instructions). However, it can increase the execution time in three ways:

- It may increase the register pressure, and therefore, it may increase the *II* due to spill code.
- It may increase II_{rec} because the latency of memory operations is augmented.
- It may increase the *SC* (stage counter) because the length of individual loop iterations may be increased.

Two of the main issues of the CSMS algorithm is the reduction of the impact of recurrences on the *II* and the minimization of the stall time. The problem of the cost of the prolog and epilog is handled by computing two alternative schedules. Both focus on minimizing the stall time and the *II*. However, one of them reduces the impact of the prolog and the epilog at the expense of an increase in the stall time whereas the other does not care about the prolog and epilog cost. Then, depending on the number of iterations of the loop, the most effective one is chosen.

The algorithm consists of creating two dependence graphs, one using the cache-miss latency for scheduling each memory operation, and another one using cache-miss or hit latency according to a static locality analysis. The effect of recurrences that limit de initiation interval is reduced by changing, from cache-miss to cache-hit, the latency of some memory operations (following a locality order) until this recurrence minimizes the *II*. An upper bound in the number of iterations of the loop help us to choose between the scheduling of both graphs.

More details about the CSMS algorithm are reported in [4].

4. Some performance results

The performance of the software prefetching schemes has been studied for some SPECfp95 benhmarks, and for two VLIW architectures: simple (4-issue and the cache-miss latency is 10 cycles) and aggressive (8-issue and the cache-miss latency is 20 cycles).

In addition to the above-mentioned schemes,we have measured the scheduling using cache-hit latency always (CHL) and a lower bound of the execution time (LBND).

In Figure 1 results are presented. Black bars represents stall time, and grey bar represents compute time, all of them normalized to CHL. It is show that the CSMS scheme achieves the best trade-off between stall and compute time, and its performance is close to the lower bound.

5. Conclusions

In this paper we have compared the effect that some software prefetching techniques have in software pipelined loops for VLIW architectures. We have seen that the proposed CSMS scheme significally outperforms previous proposals.

References

[1] S.G. Abraham, R.A. Sugumar, B.R. Rau and R. Gupta, “Predictability of Load/Store Instruction Latencies”, in *Procs. of 26th Int. Symp. on Microarchitecture*, pp. 129-152, Dec. 1993

[2] D. Callahan, K. Kennedy and A. Porterfield, “Software Prefetching”, in *Procs of the IV Symp. on Arch. Support for Prog. Lang. and Oper. Syst. (ASPLOS)*, pp. 40-52, April 1991

[3] M.S. Lam, “Software Pipelining: an Effective Scheduling Technique for VLIW Machines”, in *Procs. of Conf. on Prog. Lang. Desing and Impl. (PLDI)*, pp. 43-53, May 1991

[4] F.J. Sánchez and A. González, “Cache Sensitive Modulo Scheduling”, in *Procs. of 30th Int. Symp. on Microarchitecture*, Dec. 1997