Register-Sensitive Software Pipelining

Amod K. Dani
Veritas Software India Pvt. Ltd.
1179/3, Shivajinagar
Modern College Road
Pune 411 005, India

amod@veritas.com

V. Janaki Ramanan
Supercomputer Edn. & Res. Centre
Indian Institute of Science
Bangalore 560 012, India

ramanan@rishi.serc.iisc.ernet.in

R. Govindarajan
Supercomputer Edn. & Res. Centre
Computer Science & Automation
Indian Institute of Science
Bangalore 560 012, India

govind@{serc,csa}.iisc.ernet.in

Abstract

In this paper, we propose an integrated approach for register-sensitive software pipelining. In this approach, the heuristics proposed in the stage scheduling method of Eichenberger and Davidson [4] are integrated with the iterative scheduling method to obtain schedules with high initiation rate and low register requirements. The performance of our integrated software pipelining method was analyzed for a large number of loops taken from a variety of scientific benchmark programs. Our studies reveal that the stage scheduling heuristics facilitate better performance benefits when applied at the scheduling time, resulting in significant performance improvement over both the stage scheduling method and the slack scheduling method.

1 Introduction

Modulo scheduling or software pipelining is an aggressive instruction scheduling technique for loops. In modulo scheduling, instructions from successive iterations of the loop are initiated at a constant interval, known as the initiation interval (**II**) of the schedule. Several methods have been proposed for constructing software pipelined loops [3, 4, 6, 8, 9, 10, 11, 12].

Modulo scheduling inherently increases the register requirement as it overlaps the execution of a number of operations. The software pipelining methods reported in [3, 9, 11, 12] do not consider the register requirements while constructing the schedule. When the register pressure increases beyond the available registers, it causes register spill, which in turn, may increase the **II**.

In order to keep the register requirements low, the scheduling algorithm should try to reduce the *lifetimes* of variables [1]. This is achieved by a careful placement of the instructions in the schedule [4, 6, 8, 10]. Huff's bidirectional slack scheduling method [8] schedules an operation *early* or *late* depending on its number of stretchable input and output flow dependences. Llosa, *et al.*, describe a method that prioritizes the instructions using the hypernode reduction method to reduce the register requirements [10].

Eichenberger and Davidson use a two-step approach for register minimization [4]. Their method first obtains a schedule for the loop without any register requirement considerations. It then applies a set of heuristics, termed as stage scheduling heuristics, to shift the position of instructions in the schedule by multiples of **II**, subject to dependency constraints, to reduce the register requirements. The disadvantage of this two-step approach is that it can shift the operations and, thus, can reduce the lifetimes of the variables only by multiples of **II**. Reduction in lifetime by fractional values of **II** is not possible.

In this paper we present an integrated approach to software pipelining that achieves both high initiation rate and low register requirements. In our approach, we integrate the stage scheduling heuristics with the modulo scheduling method [12]. We refer to our scheduling method as Integrated Register-sensitive Iterative Software pipelining (IRIS) method. A salient feature in integrating the heuristics is that the resulting scheduling method attempts to schedule an instruction either *as early as possible* or *as late as possible* — an adaption of Huff's bidirectional scheduling method [8]. We observe that this integrated approach gives surprisingly significant performance improvements over the stage scheduling method as well as Huff's slack scheduling method [8]. However, our experiments reveal that there is still scope for improvement in terms of the register requirements of the schedules constructed by IRIS. Thus we apply the stage scheduling heuristic again on top of IRIS to obtain further improvements in register requirements.

The rest of the paper is organized as follows. We present a motivating example in Section 2. Section 3 describes the scheduling algorithm in detail. Experimental results are reported in Section 4. Section 5 describes the related work and Section 6 summarizes the conclusions.

2 Motivating Example

In this section we present a motivating example that illustrates the impact of the placement of the instructions on the register requirements of the schedule. We briefly discuss Eichenberger's stage scheduling method [4] in Section 2.2. Lastly, we motivate the work presented in this paper using the same example.

2.1 The Basic Iterative Schedule

Consider the loop L:

```
for(i=0;i<n;i++)
{
   z[i] = 2*x[i] + y[i] + z[i-1];
   w  = w + z[i] ;
}
```

The data dependence graph (DDG) for the loop is depicted in Fig. 1. The vertices of the DDG represent the instructions in the loop and edges (or arcs) represent the dependency between them. Each edge (i, j), from node i to j, is associated with a *latency* equal to the execution time of the producer instruction i. The *dependence distance* of edge (i, j) indicate how many iterations later the value produced by instruction i is consumed by its successor j.

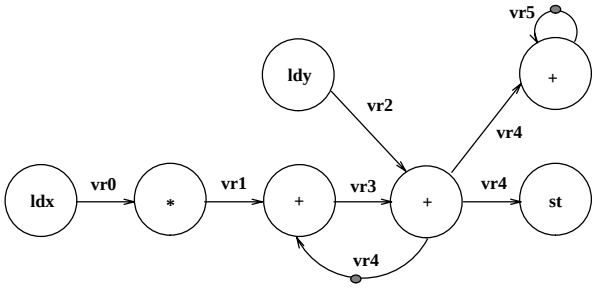

Figure 1: Motivating Example – Data Dependence Graph (DDG)

Consider a processor architecture with 2 add, 1 multiply, and 2 load/store units. Let the latencies of the functional units be 1, 4 and 1 cycles respectively. The initiation interval **II** for the example loop is 2. That is, we initiate a new iteration of the loop every 2 cycles. The schedule of various instructions in the loop are shown in Fig. 2(a). The repetitive *kernel* of the software pipelined schedule is shown in Fig. 2(b).

The *lifetime* of a value defined by an instruction stretches from the start of the execution of that instruction to a time until all its successors have used the value. The lifetimes of various values for the given schedule are shown in Fig. 2(c). The usage of registers in every time step of the

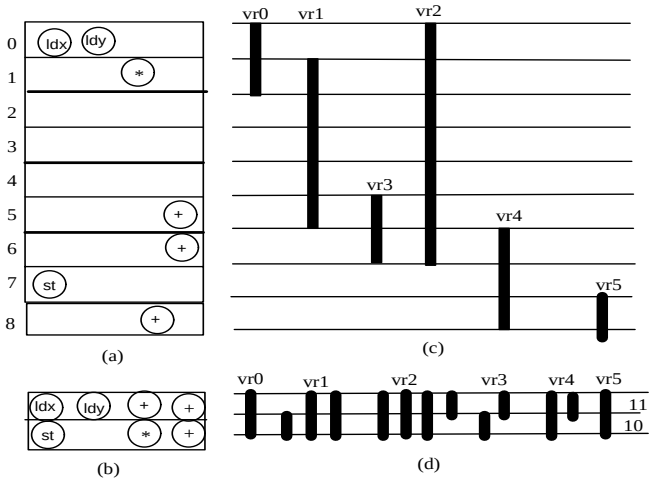

Figure 2: Motivating Example: (a) Schedule of the loop. (b) Kernel code. (c) Lifetimes of one iteration. (d) Lifetimes in the Repetitive Kernel

repetitive kernel is obtained by collapsing Fig. 2(c) to a table of **II** rows, using wraparound, as shown in Fig. 2(d). For example, the value in register `vr1` gets defined in cycle 1 and it remains live till its last use in cycle 5. This corresponds to (roughly) 3 live values in the repetitive kernel — the three vertical bars under `vr1` in Fig. 2(d).

The register requirement of a modulo schedule is typically characterized by its *Maxlive* — the maximum number of live values at any single cycle of the loop schedule [8, 5, 4]. The *Maxlive* for the schedule is 11 as shown in Fig. 2(d). Our aim is to reduce *Maxlive*, thereby minimizing the register requirements of the schedule.

2.2 Stage scheduling

Eichenberger and Davidson use a two-stage approach for reducing the register requirements of a software pipelined schedule [4]. In the first phase, they obtain a resource-constrained schedule using Rau's iterative software pipelining method [12]. Then they apply a set of stage scheduling heuristics that shift the position of instructions in the schedule by multiples of **II**, subject to dependency constraints, and based on a number of heuristics [4].

First we present a set of definitions from [4].

Definition 2.1 *A node in the DDG is said to be a* **Source Node** *(or* **Sink Node***) if it has only outgoing (or incoming) edges.*

Definition 2.2 *An edge in the DDG is a* **cut-edge***, if removing that edge from the DDG makes the underlying graph disconnected.*

Definition 2.3 (Skip Factor) *For each edge (i, j) from instruction i to j in the DDG, the* **skip factor** *of (i, j) for a given schedule S is defined as*

$$Skip_{i,j} = \left\lceil \frac{time_j - time_i - rdist_{i,j}}{\mathbf{II}} \right\rceil$$

where $time_i$ and $time_j$, respectively, refer to the time steps in which the first instance (corresponding to iteration 0) of instructions i and j are scheduled, and $rdist_{i,j}$ refer to the distance from instructions i to j in the repetitive kernel of the modulo schedule.

Skip factor of (i, j) roughly corresponds to the number of (integral) vertical bars for the lifetime of the associated register in the repetitive kernel, provided j is the last instruction to use the value of i.

We describe three of the four heuristics that are considered in the stage scheduling method. The Up/Down heuristics propagates the additional skip factors on edges upward/downward along the edges of the DDG. Up (Down) propagation results in scheduling the producer (consumer) node *late* (*early*) by $skip\_factor * \mathbf{II}$ time steps. The application of up propagation take place in the reverse topological order. For strongly connected components, this heuristic is applied several times. The Sink/Source heuristics identifies all the source/sink nodes in the DDG, ignoring cut-edges and self-loop edges. For a source node, it calculates the minimum skip factor among all the outgoing edges and propagates the minimum skip factor upward. Similarly, the minimum skip factor among the incoming edges of a sink node is propagated downward. Lastly, the Cut-Edge heuristics detects all cut-edges, once again ignoring self-loop edges, and removes the additional skip factor on the cut-edge, by up or down propagation. This has the effect of scheduling the nodes of the two biconnected components closer to each other, thereby reducing the register requirement for the cut-edge.

It was established in [4] that the above up and down propagations result in schedules that do not violate data dependencies. For example, the skip factor of the edge corresponding to register vr2 in the above schedule is 3. Applying up propagation, a skip factor of 2 can be eliminated. This results in scheduling the instruction *ldy* at time step $0 + 2 * II = 4$. The reader can verify that this reduces the register requirement for vr2 by 2, resulting in a *Maxlive* of 9 for the kernel.

2.3 Motivation for Integrated Register-Sensitive Software Pipelining

While stage scheduling succeeds in reducing the lifetimes of variables which have a minimum skip factor greater than 0, it only achieves reducing the integral part of the register requirement in Figure 2(d). Notice that scheduling *ldy* in time 5, while satisfies all dependency constraints of the loop, reduces the lifetime of vr2 to 2 cycles, *i.e.*, from cycle 5 to 6. The modified schedule and the register lifetimes are shown in Fig. 3. As can be seen from Fig. 3(d), the *Maxlive* for the schedule reduces to 8. Note that in the new schedule (shown in Fig. 3), the lifetime of vr2 is not only reduced by an (integral) multiples of **II**, but also by a fractional part.

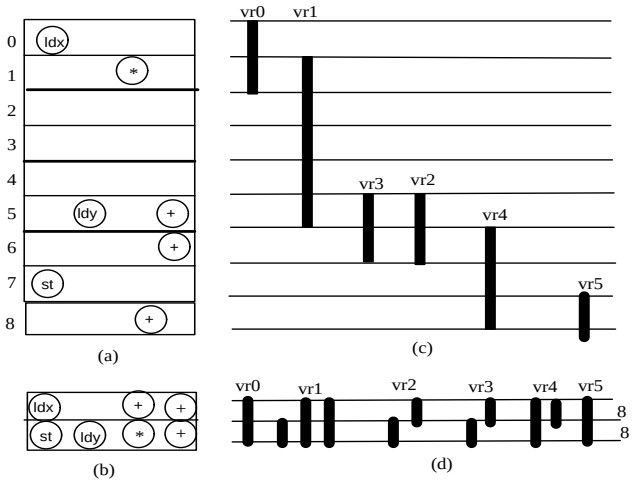

Figure 3: Motivating Example – IRIS Schedule: (a) Schedule of the loop. (b) Kernel code. (c) Lifetimes of one iteration. (d) Lifetimes in the Repetitive Kernel

In the following section we present our integrated approach to register-sensitive scheduling which uses the stage scheduling heuristics at scheduling time to carefully position the instructions in the schedule in order to minimize the *Maxlives*.

3 Integrated Register-Sensitive Iterative Scheduling (IRIS)

The IRIS algorithm is very similar to the iterative modulo scheduling algorithm [12] with the following modifications. IRIS uses a tighter bound for *Lstart*, the latest time step at which an instruction can be scheduled such that data dependency is not violated for its successors. The earliest time step *Estart* and *Lstart* are calculated using the method described in [8]. In scheduling an instruction, the scheduler can place an instruction closer to either *Estart* or *Lstart*. The scheduler achieves this by looking for a suitable time step in the direction from *Estart* to *Lstart* or vice-versa. The direction in which the search proceeds is determined by one of the following heuristics, adapted from [4].

Source Heuristic: If the instruction to be scheduled is a source node in the DDG, and any of its successor(s)

has already been scheduled by the scheduling method, then the instruction is scheduled as late as possible; *i.e.,* the search direction is from *Lstart* to *Estart*.

Sink Heuristic: If the instruction to be scheduled is a sink node in the DDG, and any of its predecessor(s) has already been scheduled by the scheduling method, then the instruction is scheduled as early as possible; *i.e.,* the search direction is from *Estart* to *Lstart*.

Cut-Edge Heuristic: Let (i, j) be a cut-edge in the DDG. If instruction j has already been scheduled, then instruction i is scheduled as late as possible; *i.e.,* the search direction is from *Lstart* to *Estart*. On the other hand, if instruction i has already been scheduled, then j is is scheduled as early as possible.

Default: If the instruction to be scheduled does not fall in any of the above categories, then it is scheduled as early as possible; *i.e.,* the search direction is from *Estart* to *Lstart*.

In all other aspects, IRIS is similar to the iterative modulo scheduling method. For example, IRIS uses height priority function for selecting the order in which instructions are scheduled and budget to decide when to discard the current partial schedule and try for the next **II** value [12]. The instruction ejection policy and reschedule are also similar to Rau's iterative scheduling method. The detailed algorithm is omitted here due to space constraints and the reader is referred to [2].

4 Experimental Results

We implemented the IRIS method and analyzed its performance on a set of 1066 benchmark loops extracted from a variety of scientific benchmark programs. Our machine model involves pipelines with structural hazards. The reservation tables for the various function units are presented in [2]. Certain performance statistics of IRIS schedules are reported in Table 1.

	Min.	Avg.	Max.
Sched. Time (mSec.)	0.1	27.7	600.1
II/**MII**	1.0	1.04	1.9
Maxlive	2	13.8	78

Table 1: Performance of IRIS

Next we compared the performance of IRIS with our implementation of the stage scheduling (SS) method [4] and Huff's slack scheduling method (HS) [8]. In both cases, our implementation faithfully followed the details given in the respective references. Each row in Table 2 reports the performance improvements, in terms of **II** and *Maxlive*[1], achieved by a method over others.

As can be seen from Table 2, IRIS performs better than SS in terms of both **II** (in 55 loops with an average improvement of 42%) and *Maxlive* (19% improvement in 385 benchmarks). Compared to HS, IRIS schedules are better in **II** in 200 loops with an average improvement of 23%. However there are some loops for which the SS or HS schedules are better than our IRIS schedules in terms of *Maxlive* (in the case of HS, this number (217) is large). This indicates that the *Maxlive* of IRIS schedules can be further improved.

Hence, we apply the stage scheduling heuristics again, after the IRIS scheduling. We refer to this method as IRIS+SS. This results in further improvement in *Maxlive*. Specifically, the application of stage scheduling heuristics on IRIS reduces the *Maxlive* in nearly 200 loops. This increases the number of benchmarks to nearly 500 where the *Maxlive* of IRIS+SS schedules are better than SS. Lastly, with the application of stage scheduling in IRIS+SS, the *Maxlives* of IRIS+SS schedules are better than HS in 291 loops, and worse than HS in 149 cases. Other comparison numbers are reported in Table 2.

In summary, we understand that the stage scheduling heuristics reduce *Maxlive* when applied at the scheduling time, as in IRIS, and also have the potential to further reduce the *Maxlive* when applied after the scheduling time.

5 Related Work

Software pipelining has been extensively studied [3, 4, 6, 8, 9, 10, 12]. Rau and Fisher provide a comprehensive survey of these works in [11]. The software pipelining methods reported in [4, 6, 8, 10] attempt to construct schedules with high initiation rate and low register requirements.

The stage scheduling method [4] is a two-step approach. In the first step an modulo schedule is constructed using the iterative method [12]. In the second step, the stage scheduling method shifts the position of certain instructions in the schedule by multiples of **II**, to reduce the register pressure of the iterative schedule. Whereas our IRIS method applies the stage scheduling heuristics at the schedule time to move the operations anywhere in the schedule. This allows reduction of both the integral and fractional parts of *Maxlive*.

An important difference between IRIS and the iterative scheduling method is that IRIS uses the stage scheduling heuristics to decide the search direction. In this sense, IRIS is also bidirectional like Huff's slack scheduling

[1] We compared the *Maxlives* of schedules produced by two methods only when both have the same **II**.

		over IRIS		over SS		over HS		over IRIS+SS	
		No. of Bench-marks	% Impro-vement	No. of Bench-marks	% Impro-vement	No. of Bench-marks	% Impro-vement	No. of Bench-marks	Impro-vement vement
II	IRIS Better	–	–	55	42.4	200	23.2	0	–
	SS Better	10	9.3	–	–	–	–	10	9.3
	HS Better	9	6.8	–	–	–	–	9	6.8
	IRIS+SS Better	0	–	55	42.4	200	23.2	–	–
Maxlive	IRIS Better	–	–	385	19.2	148	18.1	0	–
	SS Better	70	9.4	–	–	–	–	7	9.2
	HS Better	217	35.1	–	–	–	–	149	40.8
	IRIS+SS Better	219	11.4	493	18.7	213	16.2	–	–

Table 2: Comparison of Different Scheduling Methods

method [8]. IRIS differs from the slack scheduling algorithm in that the stage scheduling heuristics, rather than *stretchability* measures are used to decide the search direction.

Llosa, *et al.*, proposed the hypernode reduction modulo scheduling method [10] in which the instructions of the loops are ordered by reducing the nodes in the DDG to a *hypernode*. Lastly, methods based on linear and integer linear programming approaches have been proposed to obtain, respectively, minimum register iterative schedules [5] and rate-optimal schedule with minimum register requirements [6, 7].

6 Conclusion

In this paper we have proposed an integrated register-sensitive software pipelining (IRIS) method based on the heuristics proposed in the two-step stage scheduling method [4]. Experimental results reveal the potential benefits that stage scheduling heuristics can give when applied at or after the scheduling time. In particular applying the heuristics both at and after the scheduling gives significant performance improvement in **II** and *Maxlive* compared to the stage scheduling and the slack scheduling methods.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Co., Reading, MA, corrected edition, 1988.

[2] Amod K. Dani Software Pipelining for VLIW Architectures. M.E. Project Report, Dept. of Computer Science and Automation, Indian Institute of Science, Bangalore, 560 012, India, Jan. 1997.

[3] J. C. Dehnert and R. A. Towle. Compiling for Cydra 5. *Jl. of Supercomputing*, 7:181–227, May 1993.

[4] A. E. Eichenberger and E. S. Davidson. Stage scheduling: A technique to reduce the register requirements of a modulo schedule. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 338–349, Ann Arbor, MI, Dec. 1995.

[5] A. E. Eichenberger, E. S. Davidson, and S. G. Abraham. Minimum register requirements for a modulo schedule. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 75–84, San Jose, CA, Dec. 1994.

[6] R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 85–94, San Jose, CA, Dec. 1994.

[7] R. Govindarajan, E. R. Altman, and G. R. Gao. A framework for resource-constrained rate-optimal software pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1133-1149, Nov. 1996.,

[8] R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the ACM SIGPLAN '93 Conf. on Programming Language Design and Implementation*, pages 258–267, Albuquerque, NM, June 23–25, 1993.

[9] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of the SIGPLAN '88 Conf. on Programming Language Design and Implementation*, pages 318–328, Atlanta, GA, June 1988.

[10] J. Llosa, M. Valero, E. Ayguadé, and A. González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Ann. Intl. Symp. on Microarchitecture*, pages 350–360, Ann Arbor, MI, Dec. 1995.

[11] B. R. Rau and J. A. Fisher. Instruction-level parallel processing: History, overview and perspective. *Jl. of Supercomputing*, 7:9–50, May 1993.

[12] B. R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of the 27th Ann. Intl. Symp. on Microarchitecture*, pages 63–74, San Jose, CA, Dec. 1994.