RESIS: A New Methodology for Register Optimization in Software Pipelining *

Fermín Sánchez and Jordi Cortadella

Departament d'Arquitectura de Computadors
Universitat Politècnica de Catalunya
e-mail: {fermin,jordic}@ac.upc.es
http://www.ac.upc.es/homes/{fermin,jordic}

Abstract. This paper presents a new technique to reduce the register pressure in pipelined schedules. A two-step approach is proposed: minimizing the *SPAN* of the loop and rearranging operations within a basic block. Experimental results show that further improvements on the schedules found by the best existing techniques can be obtained at the expense of a negligible computational cost.

1 Introduction

In parallel architectures, a loop is usually executed by means of software pipelining techniques. These techniques attempt to find a schedule that contains instructions belonging to different iterations. The number of registers required to execute the schedule may be reduced by adding *spill code* [1] (storing some variables in memory). When there are not enough registers to execute the schedule, some techniques increase the expected number of cycles (initiation interval or *II*) and schedule the loop again [2]. Recent experiments have demonstrated that this approach may never converge [3].

Scheduling followed by register allocation may require much spill code [4]. On the contrary, register allocation followed by scheduling may reduce the potential parallelism [5]. In this paper, we will show that scheduling followed by register allocation may obtain optimal results in most cases.

The rest of the paper is organized as follows: Section 2 presents the formalism to represent a loop and a schedule. Section 3 shows three different lower bounds on the number of registers required to execute a loop. Section 4 presents an overview of *RESIS*, the approach presented in this paper, and an example. Sections 5 and 6 present the two main steps of *RESIS*: *SPAN reduction* and *incremental scheduling*. Some results are reported in Section 7.

2 Obtaining a DG from a schedule

A loop can be represented as a labelled directed dependence graph, $DG(V, E)$. Each vertex $u \in V$ represents an instruction of the loop body. Each edge $e \in E$

* This work was supported by the Ministry of Education and Science of Spain, under contract CICYT TIC-95-0419

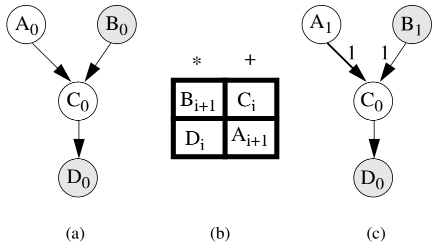

Fig. 1. Obtaining a DG from a schedule (a) Initial DG (b) Pipelined schedule by assuming 1 adder and 1 multiplier (c) Equivalent DG associated to the schedule

corresponds to a data dependence between two instructions. Labels of the DG are defined by two mappings as follows:

- $\lambda(u)$, defined on vertices, denotes the iteration index of u ($\lambda(u) \geq 0$). $\lambda(u) = i$ will be denoted by u_i in the DG. Initially, $\forall u \in V,\ \lambda(u) = 0$.
- $\delta(u, v)$, defined on edges, denotes the number of iterations traversed by the dependence. $\delta(u, v) = 0$ corresponds to an intra-loop dependence (ILD). $\delta(u, v) > 0$ corresponds to a loop-carried dependence (LCD).

In general, a schedule of a loop can be represented by a matrix containing the instructions of the loop. Each row of the matrix represents a cycle of the schedule. The schedule contains instructions that may belong to different loop iterations. Thus, placing instruction u_i at row j indicates that instruction u from iteration $i + k$ is executed at cycle j in the kth iteration. Figures 1(a) and 1(b) show an example of loop and schedule representation.

A schedule obtained by means of software pipelining is in general associated to a DG different from the initial one, but equivalent to it (representing the same loop). Such a equivalence is described by means of the rules of *retiming*. *Retiming* [6] transforms a DG in a way such that the index of the nodes (λ's) and the distance of the dependences (δ's) may be different in both DGs. A $DG = (V, E)$ is equivalent to another one $DG' = (V, E)$ if the following condition holds:

$$\forall (u, v) \in E \ : \ \lambda(v) - \lambda(u) + \delta(u, v) = \lambda'(v) - \lambda'(u) + \delta'(u, v) \tag{1}$$

To build the DG associated to the schedule, the value of λ' for each instruction is taken from the schedule, and the value of δ' for each dependence is computed by using Equation (1). Since initially $\lambda(u) = 0$ for each $u \in V$, we conclude that:

$$\delta'(u, v) = \lambda'(u) - \lambda'(v) + \delta(u, v) \tag{2}$$

As an example, DGs from Figures 1(a) and 1(c) are equivalent, since equation (1) holds for each dependence. The schedule from Figure 1(b) can be easily obtained from the DG in Figure 1(a) by using any software pipelining approach. However, note that it can also be obtained from the DG in Figure 1(c) by using any algorithm for scheduling basic blocks. This idea has been used in [7, 8] to perform software pipelining with resource and timing constraints.

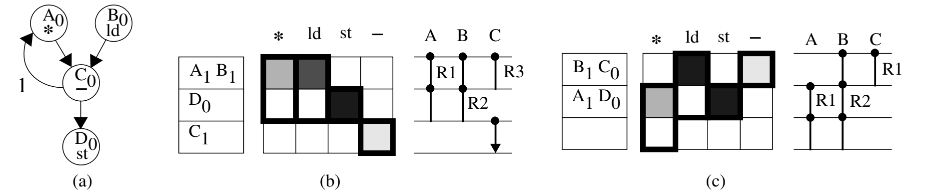

Fig. 2. Example of register optimization

3 Lower bounds on registers

A tight lower bound on the number of registers required by a schedule is the maximum number of variables whose lifetimes overlap at any cycle ($MaxLive$). [9]. Such a lower bound may not be reached, since a register assignment with such requirements may not exist (see [10] for an example).

The number of registers required when all variables have the minimum lifetime and their overlapping is minimized is also a lower bound, that can be computed as [11]:

$$\left\lceil \sum_{e \in E} minimum\_variable\_lifetime(e)/II \right\rceil \tag{3}$$

An *absolute lower bound* (ALB) on the number of registers can be computed by using Equation (3) in a DG in which $\lambda(u) = 0$ for each $u \in V$. ALB might be considered as a lower bound for any schedule of the loop [10].

A *relative lower bound* (RLB) on the number of registers can be computed by using Equation (3) with the DG associated to a schedule.

In general, $ALB \leq RLB \leq MaxLive$. Note that ALB is calculated by using the initial loop, $MaxLive$ is calculated by using the final schedule and RLB is calculated by using the DG associated to the schedule.

4 RESIS: strategy overview

This paper proposes *RESIS* (REduce Span and Incremental Scheduling), an approach aiming at reducing the register requirements of a schedule. *RESIS* works in two separate steps:

1. *SPAN reduction*: First, the DG associated to the schedule is built. Variable lifetimes are shortened by reducing the iteration index of some instructions and scheduling the new DG again.
2. *Incremental scheduling*: Variable lifetimes are reduced by moving some instructions within the schedule, attempting to reduce $MaxLive$.

As a single example, Figure 2(a) shows the SPEC-SPICE loop 10. We assume a result latency of one cycle for subtract and store, two cycles for multiply and load and an architecture with one FU (functional unit) fully pipelined of each type. Figure 2(b) presents the schedule found by HRMS [12], requiring 3 registers. Figure 2(c) shows the improvement achieved by *RESIS*. The index of instruction C has been reduced, and instructions A and C have been moved to a different cycle. As a result, the number of registers required is decreased by one.

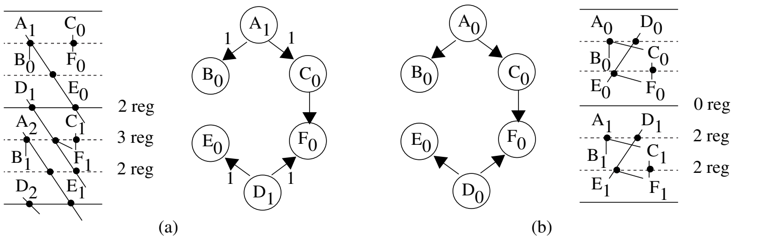

Fig. 3. Example of *SPAN* reduction (a) Schedule example ($MaxLive = 3$) and DG associated (b) DG after *SPAN* reduction and scheduling with $MaxLive = 2$

5 SPAN reduction

5.1 Strategy overview

The *SPAN* of a DG is defined as $\lambda_{max} - \lambda_{min} + 1$, where λ_{max} and λ_{min} are the maximum and minimum values for λ. In general, a reduction of the *SPAN* in a DG leads to a reduction in the variable lifetimes in any associated schedule.

The *SPAN* reduction phase works as follows. First of all, the DG associated to the given schedule, DG', is built by using Equation (2) for each dependence (we assume the initial DG is known). Following this, the maximum value for λ (λ_{max}) is computed by exploring all nodes in DG'. Then, this value is iteratively decreased while the following three conditions hold:

- a DG with minimum *SPAN* is not found (minimum *SPAN*=1)
- the critical path (CP) of the current DG is not longer than the expected *II*
- the number of registers estimate for the current schedule is greater than the *absolute lower bound* ($MaxLive >$ALB)

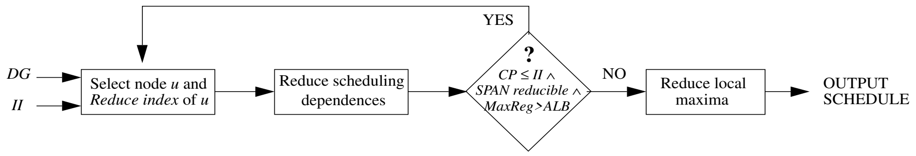

Fig. 4. Flow diagram of *SPAN* reduction

Figure 3 shows an example of the effectiveness of reducing the *SPAN* in an architecture with 2 FUs. Figure 4 shows a flow diagram of the algorithm used to reduce the *SPAN*. The execution time of the algorithm [2] is $O(V^3E + VE^2)$ [13]. The following sections explain in detail each one of the steps in the diagram.

5.2 Selecting a node to reduce the SPAN

In order to reduce the *SPAN*, two different approaches may be used: reducing λ_{max} or increasing λ_{min}. A transformation called *reduce_index* is used[3] to reduce

[2] V and E are the number of nodes and edges in a DG respectively.
[3] A similar transformation is used to increase λ_{min}.

λ_{max}. *Reduce_index(u)* is based on *retiming*, and it is only applied to nodes so that the transformed DG has non-negative dependences. Among all the nodes in the DG, the node which will produce the DG with the shortest CP is selected. *Reduce_index(u)* decreases $\lambda(u)$ by also transforming $\delta(e)$ for the incoming and outgoing edges of u as follows:

- $\lambda'(u) = \lambda(u) - 1$
- $\forall (u,v) \in E$, $\delta'(u,v) = \delta(u,v) - 1$
- $\forall (v,u) \in E$, $\delta'(v,u) = \delta(v,u) + 1$

5.3 Reducing the number of scheduling dependences

Let L_u be the execution time for an instruction u. As shown in Figure 5, for an expected II of the schedule, data dependences can be classified as follows [13]:

- Positive Scheduling Dependences (PSDs): $\delta(u,v) < \frac{L_u}{II}$
- Negative Scheduling Dependences (NSDs): $\frac{L_u+II-1}{II} > \delta(u,v) \geq \frac{L_u}{II}$
- Free Scheduling Dependences (FSDs): $\delta(u,v) \geq \frac{L_u+II-1}{II}$

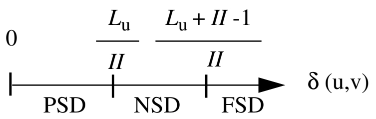

Fig. 5. Types of scheduling dependences according to the value of $\delta(u,v)$

PSDs constrain the scheduling process more than NSDs, and FSDs do not constrain the schedule. In fact, some NSDs do neither constrain the scheduling process [13]. The *SPAN* reduction algorithm attempts to reduce the number of PSDs and NSDs without increasing the *SPAN* by transforming them into FSDs. Function *reduce_sched_depend* performs such a task (see Figure 6).

DGs containing fewer PSDs are considered *better* for scheduling. For the same number of PSDs, the DG with fewer NSDs is considered as the *best*. An edge is selected to be retimed only once if a better DG is not found. The heuristics used to select an edge for retiming are as follows: (i) head of a CP[4], (ii) tail of a CP, (iii) head or tail of a path not critical. The scheduling algorithm used is a *list scheduling* (see [13] for details).

5.4 Reducing local maxima

On one hand, the algorithm to reduce *SPAN* is based on reducing the index of nodes whose index is λ_{max}. Therefore, no reduction is done with nodes having smaller indices. However, the index of such nodes may also be reduced, also reducing the variable lifetimes and thus register requirements. On the other hand, the function *reduce_sched_depend* increases the distance of some dependences. As

[4] A CP is a path formed only by PSDs.

```
function reduce_sched_depend(DG, II);
DG1 := DG;
Repeat
    sched := scheduling(DG, II);
    if no schedule has been found then
        e :=select_edge(DG, λmax);
        if edge_selected then
            DG :=retiming(DG, e);
            if (DG is better than DG1)
                then DG1 := DG;
until (no edge selected or schedule found);
return true if a schedule has been found;
```

```
function reduce_local_max(DG, SCH, II);
DG1 := DG;
loop
    u :=select a node from DG1;
    exit if no node selected;
    DG1 :=reduce_index(DG1, u);
    new_sched :=scheduling(DG1, II);
    if schedule found then
        DG := DG1;
        SCH := new_sched;
    else undo reduce_index(DG1, u);
return SCH;
```

Fig. 6. Functions *reduce_sched_depend* and *reduce_local_maxima*

a side effect, the indices of some nodes may be unnecessarily increased. Given the previous argumentation, some indices smaller than λ_{max} may also be reduced after reducing the *SPAN*. These nodes are called *local maxima*. Figure 6 shows the algorithm used to reduce *local maxima*.

6 Incremental scheduling

6.1 Strategy overview

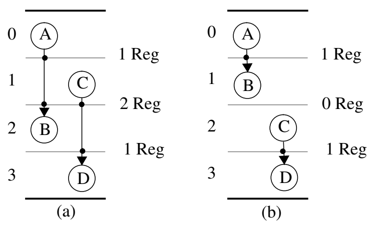

Fig. 7. Reducing registers by *incremental scheduling*.

After reducing the *SPAN*, *RESIS* tries to reduce register requirements by re-arranging some instructions without changing their iteration indices. This step is denoted *incremental scheduling* (Figure 7 shows an example). Code rearranging strategies have previously been proposed by other authors [14, 15]. Two different moves are considered:

- *Re-schedule*: moves an instruction from the current cycle to another cycle if sufficient resources are available.
- *Swap*: swaps the scheduling of two instructions that have a similar execution pattern (both instructions use the same resources at the same cycle).

The execution time of *incremental_scheduling* is $O(V^3)$ [13]. The *incremental scheduling* algorithm is as follows:

1. Compute the RLB. This is done to stop the search when a schedule requiring such registers is found.
2. Compute $MaxLive$. The algorithm ends if $MaxLive = RLB$.
3. Select a cycle c requiring $MaxLive$ registers.

4. Select an instruction to move across cycle c. The lifetime of the variable read or written by the selected instruction must be alive at cycle c. An instruction is selected only once if no movements are carried out within the schedule.
5. Move (*re-schedule*) the selected instruction until crossing cycle c, decreasing the number of registers required at cycle c. *Swapping* the instruction for another one is only done when *re-scheduling* the instruction is not successful.
6. If no successful move can be done with the selected instruction, select another one to move across cycle c and go to 5. If the instruction is moved successfully, go to 2. The process is repeated until no instruction can be moved.

6.2 Moving an instruction

A move consists of moving a node (*forward* or *backwards*) across a given cycle c, attempting to prevent the variable lifetime from being alive at cycle c. In order not to change the iteration index of any instruction, no movement can be done across the boundaries of the schedule.

- *Re-scheduling*: Assume that u was scheduled at cycle $S(u)$. If u must be moved forward, try to reschedule u from cycle $c+1$ to cycle $ALAP(u)$ If u must be moved backwards, try to reschedule u from cycle c to cycle $S(u)$.
- *Swapping*: In order to *swap* u with another node, a node v is selected among those that have a similar execution pattern as u. The *swapping* is recursively done by following the same algorithm as that used to move u (first by *re-scheduling* v, and by *swapping* v for another node x only when v cannot be successfully re-scheduled).

7 Experimental Results

We have borrowed from [16] a set of benchmark loops selected from assorted scientific programs such as Livermore Loops, SPEC, Linpack and Whetstone. As in [16], we assume a unit result latency for add, subtract, store, and move instructions, a result latency of 2 cycles for multiply and load, and a result latency of 17 cycles for divide. We also assume that all the FUs are fully pipelined in a superscalar architecture with 1 FP adder, 1 FP multiplier, 1 FP divisor and 1 load/store unit. Lifetime for a dependence $e=(u,v)$ has been considered from the starting of u to the starting of v.

In order to show the efficacy of *RESIS*, we have executed the algorithm over the schedules generated by HRMS [12]. Table 1 shows the reduction obtained in the number of registers. For each benchmark, the first column shows the initiation interval of the found schedule. The next two columns show the *absolute* (ALB) and the *relative* (RLB) lower bounds. RLB has been computed by using the final schedule. The next column (OPT) shows the actual minimal register requirements. This number has been calculated by using an integer linear programming approach [17]. The next columns show the register requirements ($MaxLive$) of the schedule found by HRMS and by *RESIS* after each step (*SPAN* reduction and incremental scheduling), as well as the CPU-time used in a Sparc-10/40 workstation. Last column ($diff$) shows the register reduction achieved.

When comparing ALB and RLB to the optimal register requirements, we find that the proposed lower bounds are very close to $MaxLive$. This suggests that ALB and RLB are a good estimation for $MaxLive$.

Note that, despite HRMS is a very good algorithm from the point of view of register pressure, *RESIS* achieves improvements in more than 20% of the cases. The optimization is achieved by both the *SPAN* reduction phase and the incremental scheduling phase. The short time used to calculate the final schedule suggest that *RESIS* can be suitable to be integrated in a parallel compiler.

The schedule found after *incremental scheduling* is optimal in a more than 90% of the cases. However, we think that integrating both *SPAN reduction* and *incremental scheduling* in a unique task may still obtain better results.

Table 1. Register optimization in HRMS

Application Program		II	Lower bounds ALB	RLB	OPT	HRMS MaxLive	RESIS MaxLive after SR	MaxLive after IS	time (secs)	diff
SPEC SPICE	Loop1	1	3	3	3	3	3	3	0.06	
	Loop2	6	3	4	5	5	5	5	0.08	
	Loop3	6	1	2	2	2	2	2	0.07	
	Loop4	11	8	8	8	8	8	8	0.07	
	Loop5	2	1	1	1	1	1	1	0.07	
	Loop6	2	14	14	15	15	15	15	0.12	
	Loop7	3	8	14	15	15	15	15	0.06	
	Loop8	3	2	3	5	5	5	5	0.06	
	Loop9	6	4	4	7	7	7	7	0.11	
	Loop10	3	2	2	2	3	**2**	2	0.02	-1
SPEC DODUC	Loop1-f	20	4	4	5	7	7	**6**	0.09	-1
	Loop2	21	2	2	3	4	4	**3**	0.12	-1
	Loop3	20	2	2	3	4	4	**3**	0.16	-1
	Loop7	2	18	18	18	18	18	18	0.02	
SPEC-FP.	Loop1	20	2	2	2	2	2	2	0.02	
TOMPCAT	Loop1	22	2	3	6	7	7	**6**	0.19	-1
Livermore	Loop1	3	5	5	6	7	7	7	0.16	
	Loop5	3	3	3	3	3	3	3	0.02	
	Loop23	9	5	8	10	11	**10**	10	0.20	-1
Linpack	Loop1	2	4	5	5	5	5	5	0.02	
Whetstone	Loop1	17	2	4	5	5	5	5	0.15	
	Loop2	6	4	5	6	6	6	6	0.09	
	Loop3	5	3	4	4	4	4	4	0.02	
	Cycle1	4	1	1	1	1	1	1	0.02	
	Cycle2	4	2	2	2	2	2	2	0.02	
	Cycle4	4	4	4	4	4	4	4	0.02	
	Cycle8	4	8	8	8	8	8	8	0.02	

8 Conclusions

In this paper we have presented *RESIS*, a new algorithm for register optimization based on reducing the maximum number of variables whose lifetime overlaps at any cycle. *RESIS* is divided into two steps, namely *SPAN reduction* and *incremental scheduling*.

Results show that *RESIS* may reduce the register requirements of schedules obtained by using any software pipelining approach. This is because it performs a global optimization of the variable lifetimes.

References

1. G. J. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN82 Symp. on Compiler Construction*, pages 201–207, June 1982.
2. J. Wang, A. Krall, M. A. Ertl, and C. Eisenbeis. Software pipelining with register allocation and spilling. In *Proc. of the 27th Annual Int. Symp. on Microarchitecture (MICRO27)*, pages 95–99, November 1994.
3. J. LLosa. *Reducing the Impact of Register Pressure on Software Pipelined Loops*. PhD thesis, Universitat Politècnica de Catalunya (Spain), 1996.
4. D. G. Bradlee, S. J. Eggers, and R. R. Henry. Integrating register allocation and instruction scheduling for RISCs. In *Proc. of the 4th Int. Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 122–131, April 1991.
5. S. Pinter. Register allocation with instruction scheduling. *ACM SIGPLAN Notices*, 28(26):248–257, 1993.
6. C.E. Leiserson, F. Rose, and J. Saxe. Optimizing synchronous circuitry by retiming. In *Proc. of the 3rd Caltech Conf. on VLSI*, pages 87–116, March 1983.
7. F. Sánchez and J. Cortadella. Maximum-throughput software pipelining. In *Proc. of the Int. Conf. on Massively Parallel Computing Systems (MPCS)*, pages 483–490, May 1996.
8. F. Sánchez and J. Cortadella. Time-constrained loop pipelining. In *Proc. of the Int. Conf. Computer-Aided Design (ICCAD)*, pages 592–596, November 1995.
9. B.R. Rau, M. Lee, P. P. Tirumalai, and M. S. Schlansker. Register allocation for software pipelining loops. In *Proc. of the ACM SIGPLAN92 Conf. on Programming Languages Design and Implementation*, pages 283–299, June 1992.
10. F. Sánchez and J. Cortadella. RESIS: A new methodology for register optimization in software pipelining. Technical Report RR-1996/15, UPC-DAC, April 1996.
11. R. A. Huff. Lifetime-sensitive modulo scheduling. In *Proc. of the 6th Conf. Programming Languages Design and Implementation*, pages 258–267, 1993.
12. J. LLosa, M. Valero, E. Ayguadé, and A. González. Hypernode reduction modulo scheduling. In *Proc. of the 28th Annual Int. Symp. on Microarchitecture (MICRO28)*, pages 350–360, November 1995.
13. F. Sánchez. *Loop Pipelining with Resource and Timing Constraints*. PhD thesis, Universitat Politècnica de Catalunya (Spain), 1995.
14. A. Sharma and R. Jain. InSyn: Integrated scheduling for DSP applications. In *Proc. of the 30th Design Automation Conf. (DAC)*, pages 349–354, June 1993.
15. J. Vanhoof, K. Van Rompaey, I. Bolsens, G. Goossens, and H. De Man. *High-Level Synthesis for Real Time Digital Signal Processing*. Kluwer Academic Pub., 1993.
16. R. Govindarajan, E. R. Altman, and G. R. Gao. Minimizing register requirements under resource-constrained rate-optimal software pipelining. In *Proc. of the 27th Annual Int. Symp. on Microarchitecture (MICRO27)*, pages 85–94, November 1994.
17. J. Cortadella, R. M. Badia, and F. Sánchez. A mathematical formulation of the loop pipelining problem. Technical Report UPC-DAC-1995-36, Department of Computer Architecture (UPC), October 1995.