UNIVERSITAT POLITÈCNICA DE CATALUNYA

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Autor: Fermín Sánchez

October, 1995

7

RESIS : REGISTER OPTIMIZATION

7.1 INTRODUCTION

The number of available registers is limited in both parallel architectures and HLS systems. In parallel architectures it is given by the processor architecture. In HLS systems, the maximum area of the chip limits the number of components to be integrated, including registers. For this reason, we propose in this chapter an algorithm to reduce the number of registers required by a schedule. If the target architecture has sufficient registers, the schedule may be directly executed. Otherwise, techniques such as *spill code* [Cha82] may be used in parallel architectures to store some variables in memory, reducing even more the number of required registers.

This chapter begins with a revision of the previous work done in register optimization (see Section 7.2). The previous work includes a description of the solutions proposed when the number of registers required by the schedule is greater than that available in the architecture. Some techniques proposed in the literature to reduce the register requirements in a schedule are briefly described. Finally, previous work in register allocation and register renaming techniques is presented. Both register allocation and register renaming have been previously addressed by several authors, and for this reason they are beyond the scope of this work. We approximate the number of registers required by a schedule by the maximum number of variables whose lifetime overlaps at any cycle.

In the same way that the analytical calculation of the minimum initiation interval allows us to compare the throughput of a schedule to the throughput of an optimal one, we propose in Section

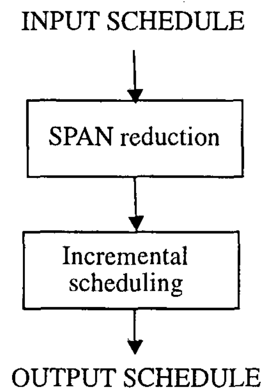

Figure 7.1 Flow diagram of *RESIS*

7.3 lower bounds on the number of registers. Although these lower bounds are often not reachable, we obtain an estimated number of registers close to them.

7.1.1 Strategy overview

In this chapter we propose a two-step approach aimed at reducing the number of registers of a schedule:

- *SPAN reduction*: first of all, the variable lifetimes are reduced by decreasing the iteration time. This is done by reducing the index of some instructions and by rescheduling the obtained π-graph again. This step may change the cycle at which each instruction has been scheduled, since the π-graph is modified and scheduled again. Section 7.4 describes *SPAN reduction*.
- *Incremental scheduling*: after reducing the iteration time, variable lifetimes are reduced by moving instructions within the schedule. This is done without modifying the index of any instruction. To do so, the cycle at which further variable lifetimes overlap is found. The instructions to be moved are selected among those that produce a result whose lifetime traverses such a cycle. This step is detailed in Section 7.5.

The approach is called *RESIS* (REduce Span and Incremental Scheduling). Figure 7.1 shows the flow diagram of *RESIS*.

7.2 PREVIOUS WORK

Consider a data dependence $e = (u, v)$ in a loop with $\delta(e) = d$. The data produced by instruction u at iteration i will be consumed by instruction v at iteration $i+d$. When v is scheduled immediately after the finishing of the execution of u, some architectures allow the FUs used by u to give the

produced data to the FUs used by v, without being previously stored. This technique is known as *forwarding* or *bypassing* [HP90]. When v is not immediately scheduled after u, the result produced by u must be temporarily stored in a register until it is used for v.

Since registers have not been considered by *UNRET* , the found schedule may need more registers than those that are available in the architecture. In this case, the schedule is not correct and a new schedule must be produced. The number of registers may be reduced by storing some variables in memory (instead of in registers). This is done by adding *spill code* [Cha82] to the loop body. The addition of *spill code* modifies the dependence graph of the loop, since new instructions *load* and *store* must be included. Therefore, the *MII* may change and, in general, the throughput of the loop may decrease. However, *spilling* may decrease the register requirement without degradation of the software pipelining performance if the spilling decision (which variable must be spilled to memory) is efficiently controlled [WKEE94]. [BEH91] observed that scheduling followed by register allocation requires much spill code. However, register allocation followed by scheduling reduces the potential parallelism too much. The same conclusion is supported in [Pin93], where scheduling and register allocation are solved simultaneously. [BEH91] proposes a list scheduling with a priority function which tries to reduce the number of alive variables when the register limit is exceeded. Register allocation is done after list scheduling. *Cut reduction*, a technique for closely coupling scheduling and register allocation is presented in [DGD94]. *Cut reduction* considers the entire scheduling solution space. The approach makes use of calculating the maximum number of alive variables in a graph. This calculation is based on retiming and it is done in linear time. The number of alive variables is reduced to within the constraints during a scheduling preprocessing step, adding extra constraints to the graph based on estimations of their impact on the schedule length. The actual scheduling does not have to take register constraints into account, and can even introduce any arbitrary amount of software pipelining to reduce the execution time of the application. A similar approach is proposed in [Rim89]. In this microcode compilation approach, the decisions on register allocation are local in nature, and there is no back-tracking on them.

When there are not enough available registers, some techniques increase the expected initiation interval and schedule the loop again [WKEE94]. Intuitively, for the same loop, a schedule larger than the other one executes fewer instructions in parallel, and therefore it requires less registers at the same time. However, recent experiments have demonstrated that this approach may never converge [LLo95]. Spilling may also be considered after increasing the expected *II*.

Several approaches have been proposed to perform *register allocation*. Techniques which use a *register allocation graph* are presented in [CAC+81, Cha82, CH84, TS86, BCKT89]. Each node in the *register allocation graph* represents a variable. An edge between two nodes indicates that the variables do not overlap their lifetimes, and therefore they can be stored in the same register (a variable is said to be alive between the time it is generated (written) and the last use (read) of it). The aim of this techniques is to determine the minimum number of cliques that cover the graph. [GRVD87, Sto92] present an alternative approach based on the use of a *conflict graph*. In the *conflict graph*, each node represents a variable, and an edge between two nodes exists when both variables cannot share the same register because their lifetime overlaps. Other approaches are based on *interval graphs* [KP87] and *cyclic interval graphs* [HGAM92, MLVV93]. An *interval graph* contains information that is not available neither in *register allocation graphs* nor in *conflict graphs*. Given a set of lifetimes, the *interval graph* contains only the overlapping information of any two lifetimes in the set. *Cyclic interval graphs* are an appropriate representation for variable lifetimes in loops. Coloring a *cyclic interval graph* with the minimum number of colors is known

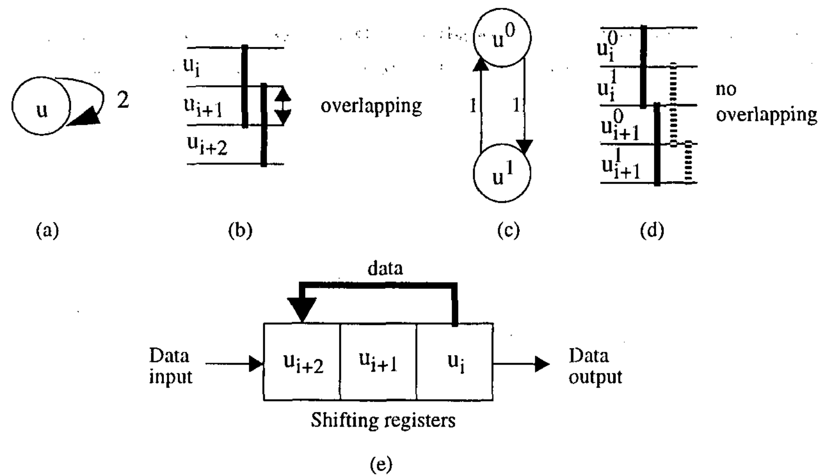

Figure 7.2 Register assignment in a superscalar architecture
(a) Example of π-graph π
(b) A possible schedule of π. Code generation is not possible
(c) π unrolled twice
(d) A possible schedule of π^2. Code generation is now possible
(e) Example of rotating register files

to be an NP-Hard problem [GJMP80]. Consequently, no polynomial-time algorithm is known to solve this problem.

In some cases, two different data dependences may exist between the same two instructions. For example, the instruction $A[i] := A[i-1] + A[i-2]$ produces two data dependences, $e_1 = (A, A)$ and $e_2 = (A, A)$, with $\delta(e_1) = 1$ and $\delta(e_2) = 2$. Both dependences produce a recurrence. The recurrence produced by e_1 constrains the initiation interval more than the one produced by e_2. However, the recurrence produced by e_2 produces a variable lifetime greater than the one produced by e_1. Therefore, e_1 must be used to compute *MII*, but e_2 must be used to compute the number of registers required for the schedule.

Compilers for superscalar architectures and VLIW processors generate code by using the schedule found by a software pipelining approach. *Code generation* requires register assignment, and sometimes it is not possible to create a new loop body because no feasible register assignment exists. Figure 7.2 shows an example. For the sake of simplicity, we will assume that u is a sum, and the architecture has an adder which adds in a single cycle. Figure 7.2(b) shows a feasible schedule of the π-graph from Figure 7.2(a). Although the schedule is correct, the generation of code is not possible because the variable lifetime required by two consecutive instances of u is overlapped. Therefore, the result of u_i cannot be written in the same register as the result of u_{i+1}, given that u_{i+1} rewrites the register, and thus the value written by u_i is not available to be read by u_{i+2}. Two different techniques may be used to overcome this problem: *static renaming via modulo variable expansion* and *dynamic remapping via the use of rotating register files*.

Static renaming via modulo variable expansion

Figure 7.2(c) shows the π-graph unrolled twice. The schedule throughput is similar to that obtained without previous unrolling. However, code generation is now possible, since variable lifetimes are not overlapped. The results produced by u^0 and u^1 are written in different registers, and therefore no overlapping exists between variable lifetimes. The result produced by u_i^0 is available to be read by u_{i+1}^0, and the same happens with u^1. Figure 7.2(d) shows this argumentation. This technique is called *modulo variable expansion* [Lam88], which is a variation of the variable expansion technique used in vectorizing compilers [KKP+81]. *Modulo variable expansion* always requires loop unrolling. The variable expansion transformation identifies those variables that are redefined at the beginning of every iteration of a loop, and expands each variable into a higher dimension variable, so that each iteration can refer to a different location. When *modulo variable expansion* is applied, code sequences for consecutive iterations differ in the registers used, thus lengthening the schedule. In order to generated code, the number of times the loop body must be unrolled is [Lam88]:

$$\left\lceil \frac{\max_{e \in E}(\text{variable_lifetime}(e))}{II} \right\rceil$$

Since the loop may previously have been unrolled to find a maximum throughput schedule, a new unrolling may produce a code much too long. In order to avoid *code explosion*, variable lifetimes must be minimized.

Dynamic remapping via the use of rotating register files

Some architectures, such as the *Polycyclic architecture* [RG81], the URPR-1 [SWT+90], the SRFA [UP93] or the Cydra 5 Supercomputer [SM88, Rau88, RYYT89] provide hardware support for code generation. This is done in the form of *rotating register files* and *predicated execution* [RST92, DHB89].

Consider saving the series of values generated by an instruction in its own infinite pushdown stack. Old values can be read from anywhere within the stack, and new values can be pushed on top, but a value cannot be modified once it has been put onto the stack; that is, the stack enforces a *dynamic single assignment* discipline [Rau91]. Since each value pushed onto the stack has the same lifetime, only a constant portion of the stack is alive at any given moment. In particular, once the schedule has been found, each stack acts like a finite shifter that shifts its values once every *II* cycles. A *rotating register file* can be thought of as a concatenation of these shifters (end to end) into a finite circular queue, as illustrated in Figure 7.2(e). By using *rotating registers files*, the variables are spaced sufficiently far apart in the register file in order to ensure that no part of a variable lifetime overlaps any part of another. This style of allocation is called *blades allocation* [RLTS92].

Predicated execution allows an instruction to be conditionally executed based on the value of the predicate (boolean value) associated with it. *Predicated execution* facilitates effective software pipelining of loops containing conditional branches. If-conversion [AKPW83] may use predicates to eliminate all branches from the loop body. Moreover, the generation of a prologue and an epilogue of the loop can be avoided by using *predicated execution*.

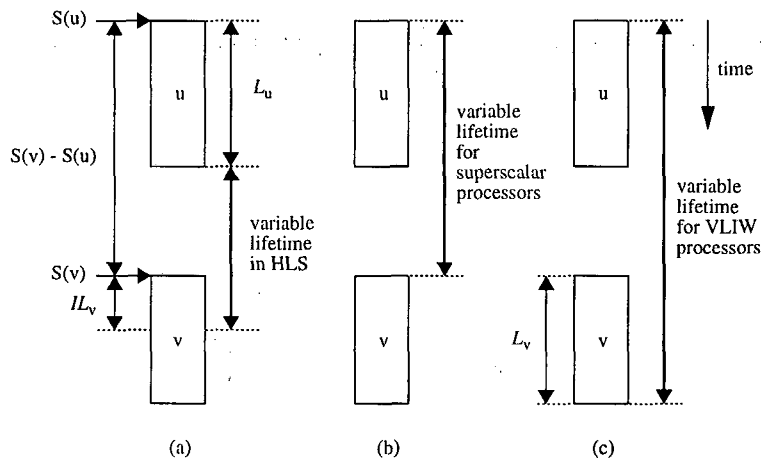

Figure 7.3 Variable lifetime for different architectures
(a) High-level synthesis
(b) Superscalar processors
(c) VLIW processors

7.3 LOWER BOUNDS ON REGISTER PRESSURE AND *RESIS* STRATEGY

7.3.1 Variable lifetime

For a dependence $u \rightarrow v$, the variable lifetime spreads from the completion of u to the cycle in which the FUs executing v no longer require the input data. For a given node u with n outgoing edges in the π-graph, only one register (and not n) is necessary to store the result computed by u. Therefore, only the edge (u, v) with the longest lifetime variable is taken into account to compute the number of registers required to store the result of u in the loop execution. The variable lifetime depends on the architecture in which the loop is executed. We will consider three different types of architectures: high-level synthesis systems, superscalar processors and VLIW processors.

- In HLS systems, the result produced by u is stored in the register in the last cycle of the execution of u. This result must be available until it is used by v. Since v may use pipelined FUs, the result produced by u must be available during the IL_v first cycles of the execution of v, as shown in Figure 7.3(a). Therefore, the variable lifetime related to a dependence $e = (u, v)$ is:
$$S(v) - S(u) - L_u + IL_v$$
- Superscalar processors present the *stall* problem. A *stall* occurs when an instruction must wait (one or more clock cycles) because someone/several of the resources it uses is/are not available. The resource requirements are dynamically computed before issuing an instruction. Therefore, although the register is not written until the last cycle, it must be available from the start of the execution to avoid a *stall*. Thus, for a dependence $e = (u, v)$, the variable lifetime spreads from the starting of u to the starting of v (v may start when all resources it

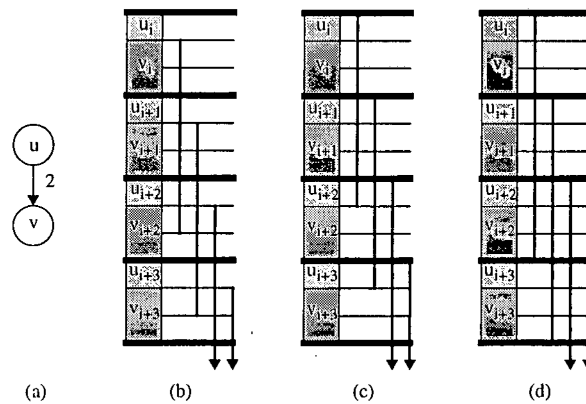

Figure 7.4 (a) Example of π-graph
(b) Overlapping of variable lifetimes in a HLS system
(c) Overlapping of variable lifetimes in superscalar processors
(d) Overlapping of variable lifetimes in VLIW processors

requires are available, and in general it is not necessary for the values used by v to be alive during the execution of v). Figure 7.3(b) shows the previous argumentation. The variable lifetime related to a dependence $e = (u, v)$ is:

$$S(v) - S(u)$$

- In a VLIW processor, all resources used by an instruction must be available from the beginning because the execution pattern cannot be dynamically changed. Moreover, since a data input may be read in any moment during the execution, the data must be available during all the execution of the instruction. Therefore, the variable lifetime related to a dependence $e = (u, v)$ spreads from the starting of u to the ending of v, as it is shown in Figure 7.3(c).

$$S(v) - S(u) + L_u$$

Other types of architectures may be considered, in which the variable lifetime might be calculated in a different way. In this work, we will only consider the previous models.

7.3.2 Registers required for a dependence

Software pipelining allows the execution of the iterations of a loop to be overlapped. Therefore, for a given instruction u, the results produced by u_i and u_{i+1} (and, in general, u_{i+k} with $k > 0$) may be alive simultaneously, as shown in Figure 7.4. Let us assume that u executes in a single cycle and v executes in two cycles in a fully pipelined FU. Figures 7.4(b), 7.4(c) and 7.4(d) show the overlapped variable lifetime when the architecture is an HLS system, a superscalar processor or a VLIW processor respectively. Note that the register requirements depend on the architecture and the distance of the dependence (u, v).

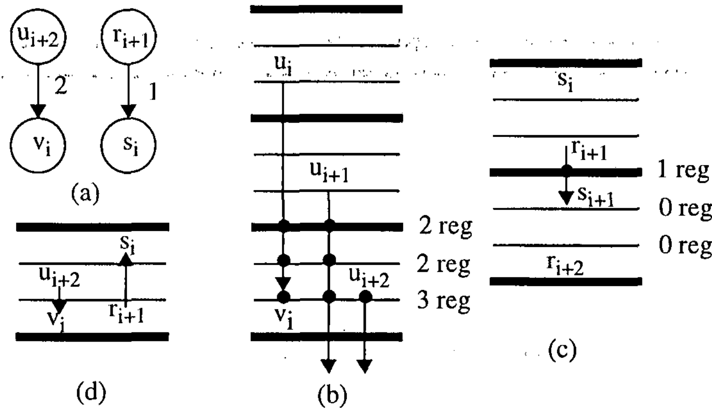

Figure 7.5 Register requirements for a dependence
(a) Example of π-graph
(b) Variable lifetimes and registers required by (u, v) when u is scheduled at cycle 2 and v at cycle 3
(c) Variable lifetimes and registers required by (r, s) when r is scheduled at cycle 3 and s at cycle 1
(d) Cycles traversed by dependences

An edge $e = (u, v)$ with $\delta(e) = d$ spreads the variable lifetime across d iterations. For any cycle c of the schedule, edge e requires $d - 1, d$, or $d + 1$ registers depending on where u and v have been scheduled. Three cases can be distinguished, as Figure 7.5 shows:

- $d + 1$ registers are required when $u \xrightarrow{d} v$ crosses cycle c forward (See Figure 7.5(b)).
- d registers are required when $u \xrightarrow{d} v$ does not cross cycle c (See Figure 7.5(b)).
- $d - 1$ registers are required when $u \xrightarrow{d} v$ crosses cycle c backwards (See Figure 7.5(c)).

Since the number of registers required to store a variable depends on the distance of the edge which represents the variable, reducing the distance of such an edge implicitly reduces the number of registers required to store the variable.

7.3.3 Register pressure

The *register pressure* is the ratio between the number of registers required by the schedule and the number of registers available in the architecture. Reducing the number of registers required by the schedule also reduces the *register pressure*. The number of registers required by a schedule is only known after doing *register allocation* [RLTS92]. *Allocating registers* for a software-pipelined loop is beyond the scope of this work. An extensive discussion including heuristic solutions and empirical results can be found in [RLTS92]. 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 [RLTS92, RF93, EDA94]. Such a lower bound may be calculated by using an integer linear

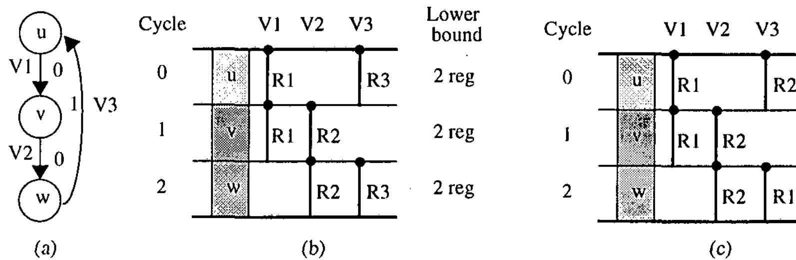

Figure 7.6 Register assignment and lower bound
(a) Example of π-graph
(b) Register assignment and number of registers required at each cycle
(c) Register assignment by using *chameleon intervals*

approach [MSAD92, LB94]. However, the lower bound may not be reached, since a register assignment with such requirements may not exist. Figure 7.6 shows an example for a VLIW processor in which all the instructions are sums which execute in a single cycle. Variable lifetimes are represented as vertical lines. A point in the intersection between a line and a cycle indicates that a register is required at this cycle. Without loss of generality, we will assume that, for a dependence $e = (u, v)$, registers are used from the first cycle of u until the last cycle of v minus one[1]. Note that the lower bound on the number of registers required is 2. However, no register assignment exists so that the code may be written by using only two registers. The lifetimes of $V1$ and $V2$ overlap, and therefore both variables must be stored in different registers. Since the lifetime of $V3$ overlaps with both $V1$ and $V2$, it requires a different register to be stored.

Hendren et al. propose in [HGAM92] the concept of *chameleon intervals*. A variable lifetime is represented as an *interval*, and an interval is always related to a single register. A *Chameleon interval* represents a variable that is stored into different registers during its lifetime. The value of the variable may be moved from one register to another, instead of the traditional *register spills* (storing the spilled value in memory). Note that both special instructions of movements among registers and special buses interconnecting registers are required to implement *chameleon intervals*. Figure 7.6(c) shows how *chameleon intervals* allow the use of only two registers in the schedule from Figure 7.6(a).

Instead of considering the maximum of variables whose lifetime overlap at any cycle, other authors consider the sum of all the registers required by all instructions as the number of registers required by the schedule [NG93, WKEE94, GAG94]. In [WKEE94] a methodology is proposed based on the *register requirement graph*, which is a directed graph with a topology similar to that of the initial π-graph. However, each edge $e = (u, v)$ is labelled with a number representing the estimated difference between $\lambda(u)$ and $\lambda(v)$ in the worst case. [NG93] presents an integer linear technique in which the register pressure is calculated by using the same approach. [GAG94] uses the same technique as [NG93] to compute the register requirements.

[1] Similar results are obtained when assuming registers are used from the first cycle of u plus one until the last cycle of v.

7.3.4 Lower bounds on registers

Minimum variable lifetime

Henceforth, we will use the term variable lifetime to refer to the lifetime of the variable associated to a given dependence. For a dependence $e = (u, v)$, the minimum variable lifetime occurs when u is ALAP scheduled and v is ASAP scheduled. As in [EDA94], we reserve a virtual register for each variable during all its lifetime. The number of registers required for a given dependence e is $\left\lceil \frac{lifetime(e)}{II} \right\rceil$. A variable with the lowest lifetime requires the minimum number of registers.

A lower bound on the number of registers required by any schedule is the number of registers required when:

- All variables have the minimum lifetime
- The instructions are scheduled in a way that minimizes the overlapping of variable lifetimes.

A lower bound on the number of registers required by any schedule of a π-graph can be computed by adding the minimum variable lifetimes for all dependences, and dividing it by II [Huf93].

$$\left\lceil \frac{\sum_{e \in E} min_var_lifetime(e)}{II} \right\rceil \tag{7.1}$$

Note that this lower bound does not require to know how the instructions have been scheduled.

Absolute and relative lower bounds

As the minimum lifetime of a variable depends on the type of architecture, the minimum number of registers required for a π-graph also depends on the architecture. The IASAP and IALAP values are related to the schedule of a given iteration. However, for a given dependence $e = (u, v)$, u and v may be scheduled at different iterations. Therefore, the distance $\delta(e)$ must be considered to calculate the variable lifetime of e. By using equations from Sections 4.5 and 4.4, we have that $IALAP(u_i) = II - D^+(u)$, and $IASAP(v_{i+\delta(e)}) = H(v) + II \cdot \delta(e)$.

Two different types of dependences must be considered when calculating the minimum variable lifetime:

- *Cyclic dependences*: a *cyclic dependence* is a dependence $e = (u, u)$ which forms a cycle. Since all iterations are scheduled in the same way, the distance between u_i and $u_{i+\delta(e)}$ is exactly $II \cdot \delta(e)$ cycles. Moreover, the value read by u must be alive during the issue latency of u (see definition 3.4 of IL_u). Therefore, the minimum lifetime for a variable representing a *self-recurrence* is:

– **High-level synthesis** (see Figure 7.3(a)):

$$min_var_lifetime(u,u) = II \cdot \delta(e) - L_u + IL_u$$

– **Superscalar Processors** (see Figure 7.3(b)):

$$min_var_lifetime(u,u) = II \cdot \delta(e)$$

– **VLIW processors** (see Figure 7.3(c)):

$$min_var_lifetime(u,u) = II \cdot \delta(e) + L_u$$

Note that *IASAP* and *IALAP* values do not have to be calculated for *cyclic dependences*.

- *Linear dependences*: A dependence is *linear* if it is not *cyclic*. We will use here $IASAP(v) = H(v) + II \cdot \delta(e)$ and $IALAP(u) = II - D^{+}(u)$.

– **High-level synthesis** (see Figure 7.3(a)):

$$\begin{aligned} min_var_lifetime(u,v) &= IASAP(v) - IALAP(u) - L_u + IL_v \\ min_var_lifetime(u,v) &= H(v) + D^{+}(u) + IL_v - L_u + II \cdot (\delta(e) - 1) \end{aligned}$$

– **Superscalar Processors** (see Figure 7.3(b)):

$$\begin{aligned} min_var_lifetime(u,v) &= IASAP(v) - IALAP(u) \\ min_var_lifetime(u,v) &= H(v) + D^{+}(u) + II \cdot (\delta(e) - 1) \end{aligned}$$

– **VLIW processors** (see Figure 7.3(c)):

$$\begin{aligned} min_var_lifetime(u,v) &= IASAP(v) - IALAP(u) + L_v \\ min_var_lifetime(u,v) &= H(v) + D^{+}(u) + L_v + II \cdot (\delta(e) - 1) \end{aligned}$$

The minimum number of registers required for any schedule of a π-graph is given by Equation (7.1). Two different π-graphs are distinguished by *UNRET*, (1) the initial π-graph and (2) the π-graph which produces a feasible schedule. Therefore, two different lower bounds on the number of registers can be defined, according to the π-graph considered.

Definition 7.1 : Absolute Lower Bound on Register Pressure

The absolute lower bound on register pressure *of a loop is the lower bound computed for the initial π-graph.*

Definition 7.2 : Relative Lower Bound on Register Pressure

The relative lower bound on register pressure *of a schedule is the lower bound computed for the π-graph which produces the final the schedule.*

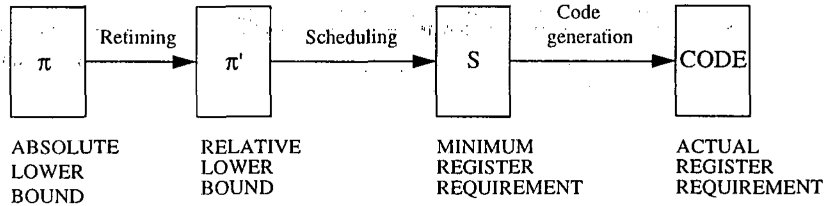

Figure 7.7 Lower bounds on registers

In general, the initial π-graph contains the greatest number of ILDs of all the equivalent π-graphs representing the loop. Therefore, variable lifetimes are the shortest possible, and thus the *absolute lower bound* is a lower bound for any schedule of the loop. The *relative lower* bound is a lower bound for the schedules of a given π-graph. Therefore, a good scheduling algorithm is the one that obtains similar results when comparing the *relative lower bound* with the minimum number of registers required by the schedule.

Figure 7.7 shows in which part of the loop pipelining process each lower bound on the number of registers is computed. The results presented in Appendix C show that the schedule proposed at Chapter 5 is a good scheduling algorithm.

7.4 SPAN REDUCTION

7.4.1 Introduction

The *SPAN* of a π-graph is defined as $\lambda_{max} - \lambda_{min} + 1$, where λ_{max} and λ_{min} are the maximum and minimum values for λ respectively. After finding a schedule, *UNRET* tries to reduce the *SPAN* while maintaining the initiation interval. In general, a reduction of the *SPAN* leads to:

- a reduction in the variable lifetimes
- a reduction in the number of required registers to store partial results across iterations
- a reduction in the iteration time
- a reduction in the size of the prologue and the epilogue
- a reduction in the number of times the loop must be unrolled to generate code when *modulo variable expansion* [Lam88] is used

Figure 7.8 shows an example of the effectiveness of reducing the *SPAN* in an architecture with 2 FUs.

The idea of the algorithm to reduce the *SPAN* is as follows: First, the maximum value for λ (λ_{max}) is computed by exploring all nodes in the π-graph. Then, this value is iteratively decreased until

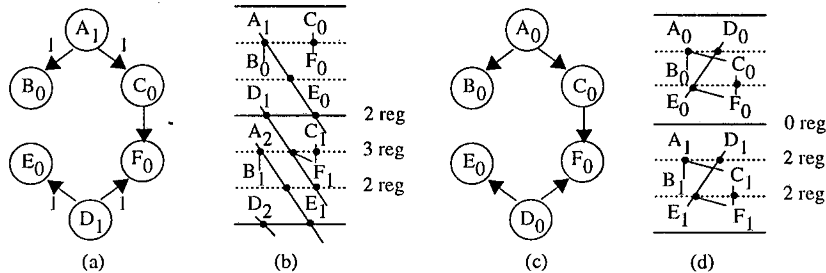

Figure 7.8 Example of *SPAN* reduction
(a) π-graph example before *SPAN* reduction
(b) Scheduling of (a), requiring 3 registers
(c) π-graph after *SPAN* reduction
(d) Scheduling of (c), requiring 2 registers

a π-graph with minimum *SPAN* is found (minimum *SPAN*=loop unrolling degree) or the *maximal positive path* (MPP) of the current π-graph is longer than the expected initiation interval.

Figure 7.9 shows a flow diagram of the algorithm to reduce the *SPAN*. The heuristics used to select a node are explained in Section 7.4.2. Section 7.4.3 shows the algorithm to reduce the index of the selected node. In Section 7.4.4 the algorithm to transform the π-graph into an equivalent one with the same *SPAN* but less positive and negative scheduling dependences (PSDs and NSDs) is described. When the *SPAN* of the π-graph cannot be further reduced, the variable lifetimes of nodes whose *SPAN* is not maximal are reduced. The function to do so is explained in Section 7.4.5. The scheduling algorithm used by *SPAN* reduction is somewhat different from the one used by *UNRET*. The differences are presented in Section 7.4.6. The overall algorithm is detailed in Section 7.4.7.

7.4.2 Heuristics to select a node to reduce the SPAN

In order to reduce the *SPAN*, two different approaches may be used: try to reduce λ_{max} of π or try to increase λ_{min}. *Dependence retiming*, as defined in Section 3.5.1, transforms the loop by increasing the distance of the dependence and the index of the source node. Although we also use a similar transformation to increase the index of the target node, experiments have shown that such a transformation is used less frequently than in the first one. Thus, it seems more logical to reduce the maximum index than to increase the minimum index. Consequently, a node is selected from among those nodes u so that:

- $\lambda(u) = \lambda_{max}$.
- No dependence (u, v) exists such that $\delta(u, v) = 0$

Among all nodes verifying the former conditions, the node which will produce the π-graph with the shortest MPP is selected. Selecting a node takes $O(VE)$ time.

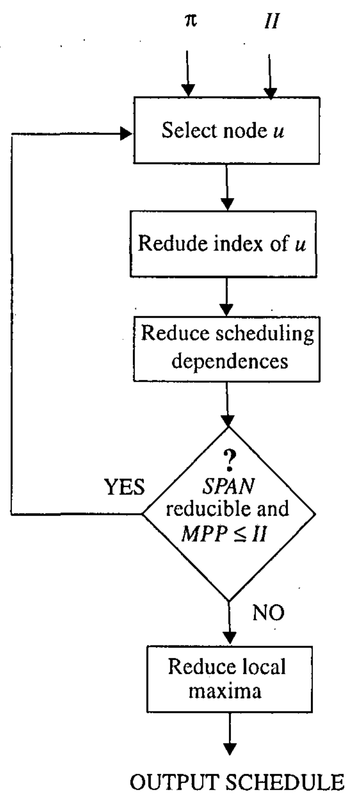

Figure 7.9 Flow diagram of *SPAN* reduction

```
function reduce_scheduling_dependences(π, II);
    π1 := π;
    Repeat
        sched := scheduling(π, II);
        if no schedule has been found then
            e :=select_edge(π, λmax);
            if edge_selected then
                π :=dependence_retiming(π, e);
                if (quality(π) > quality(π1)) then  π1 := π;
    until (no edge selected or schedule found);
    return true if a schedule has been found;
```

Algorithm 7.1 Function reduce_scheduling_dependences

7.4.3 Reduce index transformation

A transformation called *reduce_index* is used to reduce λ_{max}. *Reduce_index(u)* is based on *dependence retiming*, and it is only applied to nodes so that the transformed π-graph has non-negative dependences. *Reduce_index(u)* decreases $\lambda(u)$ by also transforming $\delta(e)$ for the incoming and outgoing edges of u as follows:

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

7.4.4 Reducing the number of scheduling dependences

In order to make the scheduling task easier, the algorithm attempts to improve the quality of the π-graph without increasing the *SPAN*. This is done by reducing the number of PSDs and NRDs (negative restrictive dependences). Function *reduce_scheduling_dependences* performs such a task (Algorithm 7.1) by means of dependence retiming.

An edge is selected to be retimed only once without improving the quality of the π-graph. We try to reduce PSDs before NRDs because PSDs constrain the scheduling process more. Therefore, the heuristics used to select an edge for retiming (in order of priority) are as follows:

1. the head-edge of a critical path
2. the tail-edge of a critical path
3. the head-edge or the tail-edge of a positive path not maximal
4. the negative restrictive dependence with the shortest distance

In order to reduce the complexity, scheduling is called once after α transformations. The parameter α has been tuned by doing experiments with a wide set of loops. The best behavior (the same result that for $\alpha = 1$ but in the lowest time) has been for $\alpha = \frac{\text{number of edges}}{2}$

We will use the following observations to calculate the computational complexity of function *reduce_shifting_dependences*:

1. Experiments have shown that an edge is selected only a few times (K times, with $K \ll |E|$). The repeat loop executes $K \cdot |E|$ times as maximum.
2. Scheduling is performed once every α transformations. As will be shown in Section 7.4.6, the complexity of the scheduling algorithm used by *SPAN* reduction is the same as that used by *unrolling_and_retiming* shown in Chapter 5. That is, it executes in $O(V^2 + VE)$ time.
3. Functions *select_edge* and *quality* have a lower execution time than *scheduling*.

Therefore, the execution time of *reduce_scheduling_dependences* is $O(V^2E + E^2)$.

7.4.5 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 lower indices. However, the index of such nodes may also be reduced, also reducing the variable lifetimes and thus the register requirements.

On the other hand, the function *reduce_scheduling_dependences* increases the distance of some dependences. As a side effect, the indices of some nodes are also incremented. Since such indices never reach the value λ_{max}, and since heuristics are used to decide which dependences must be retimed, the indices of some nodes may be unnecessarily incremented.

Given the previous argumentation, the index of head nodes of local paths whose index is smaller than λ_{max} is reduced after *SPAN* has been reduced. These nodes are called *local maxima*.

The criteria used to select a node for index reduction are as follows:

1. A node u is selected among those that fulfill:
   - $\lambda(u) < \lambda_{max}$
   - $\forall e = (u, v) \in E \ , \ \delta(e) > 0$
2. The node which will produce the shortest positive path is selected.
3. Among nodes producing paths with the same depth, we prefer nodes without predecessors (the distance of the dependences from the predecessors increases if the index of the node is decreased. Therefore, variable lifetimes from the predecessors also increase and new positive paths may be created).

The algorithm to reduce local maxima performs scheduling each step by looking for a schedule requiring fewer registers than the last one found. To do so, the algorithm repeatedly selects a

```
function reduce_local_maxima(π, sched, II);
    π2 := π;
    loop
        u := select_node_to_reduce_local_maxima(π2);
        exit if no node selected;
        π2 :=reduce_index(π2, u);
        new_sched :=scheduling(π2, II);
        if schedule found
            then
                π := π2;
                sched := new_sched;
            else undo_reduce_index(π2, u);
    return sched;
```

Algorithm 7.2 Function reduce_local_maxima

node, reduces its index and performs scheduling. If no schedule is found, it undoes the index reduction and selects another node (if possible). Nodes are selected only once if the number of required registers is not reduced. Algorithm 7.2 shows the approach used to reduce *local maxima*.

In order to calculate the computational complexity of *reduce_local_maxima* we will use the following observations:

1. Function *select_node_to_reduce_local_maxima* explores once every node in the π-graph. Therefore, its execution time is $O(V)$.
2. Since the distance of any edge must be $\delta(e) \leq \lceil \frac{L_u+II-1}{II} \rceil$, we conclude that a node can be reduced as maximum $K = \lceil \frac{L_u+II-1}{II} \rceil$ times. Experiments have proved that $K \ll V$.
3. Functions *reduce_index* and *undo_reduce_index* execute in $O(E)$ time.
4. As will be shown in Section 7.4.6, the execution time of the scheduling algorithm is $O(V^2 + VE)$.

In the worst case, the loop executes $(|V| - 1) \cdot K$ times (at least one node has an index equal to λ_{max}. Since $K \ll V$, the execution time of *reduce_local_maxima* is $O(V^3 + V^2E)$.

7.4.6 Scheduling

The scheduling algorithm used by the *SPAN* reduction algorithm is quite similar to the one used by *retiming_and_scheduling*, and shown in Chapter 6. The only difference between both algorithms is the priority function used to select which instruction must be scheduled at each moment. The priority function of the scheduling algorithm used by *SPAN* tries to minimize variable lifetimes, since the initiation interval has already been minimized by *retiming_and_scheduling*. The criteria used to select an instruction for scheduling are as follows:

1. *0-Mobility*: The nodes that have no mobility are the first nodes scheduled.
2. *The number of incoming edges*: This priority function reduces variable lifetimes, and therefore the register pressure.
3. The remainder criteria are the same used by *retiming_and_scheduling*, in the same order.

7.4.7 *SPAN* Reduction. Final algorithm

Algorithm 7.3 presents the final approach used to reduce the *SPAN*. The input parameters are the π-graph (π), the expected initiation interval (II_K), the unrolling degree (K) and the number of registers required by the schedule found by *retiming_and_scheduling* (*reg*).

A reduction in the *SPAN* does not always produce a reduction in the number of required registers. Therefore, the register requirements are calculated for each schedule found. Function *number_of_registers* perform such calculations. π-graphs consuming fewer registers are successively stored.

In order to calculate the computational complexity of *reduce_span*, we will use the following observations:

1. Function *select_node* executes in $O(VE)$ time. Function *reduce_index* executes in $O(E)$ time. The execution time of function *maximum_index* is $O(V)$.
2. *Reduce_scheduling_dependences* executes in $O(V^2E + E^2)$ time.
3. The lower bound on the number of registers required by the schedule, computed by function *number_of_registers*, can be calculated in $O(E)$ time by exploring all edges in the π-graph and by updating a table of II positions, one for each cycle of the schedule. The number of registers is calculated by using the method described at Section 7.3.2 and illustrated in Figure 7.5.

In the worst case, the loop executes $|V| \cdot K$ times, with $K \ll V$ (as experiments have proved). Therefore, the execution time of *reduce_span* is $O(V^3E + VE^2)$.

7.5 INCREMENTAL SCHEDULING

7.5.1 Overview

After reducing the *SPAN*, we try to reduce the register requirements by rearranging some instructions without changing their iteration indices. Code rearranging strategies have previously been proposed by other authors [SJ93, VVB+93]. Such strategies fine-tune the schedule by moving instructions. These moves are steered by using exact cycle count measurements and more accurate estimators for the area component. *Incremental scheduling* techniques are used not only to reduce register pressure, but also to reduce area cost in general. Figure 7.10 shows an example of the way to reduce register requirements by *incremental scheduling*.

We consider two different movements of instructions in the schedule:

```
function reduce_span (π, II_K, K, reg);
    num_registers_old := reg;
    π2 := π;
    Do
        u :=select_node(π2);
        π2 :=reduce_index(π2, u);
        Sched :=reduce_scheduling_dependences(π2, II_K);
        if schedule found then
            num_registers_new := number_of_registers(Sch);
            if num_registers_old > num_registers_new then
                π := π2;
                num_registers_old := num_registers_new;
    while (K − 1 < λ_max) and (MPP(π2) ≤ II_K );
    Sched:=reduce_local_maxima(π, Sched, II_K);
    return Sched;
```

Algorithm 7.3 Function *reduce_span*

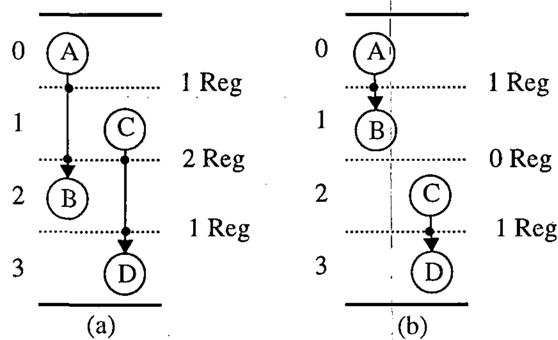

Figure 7.10 Reducing registers by *incremental scheduling*.

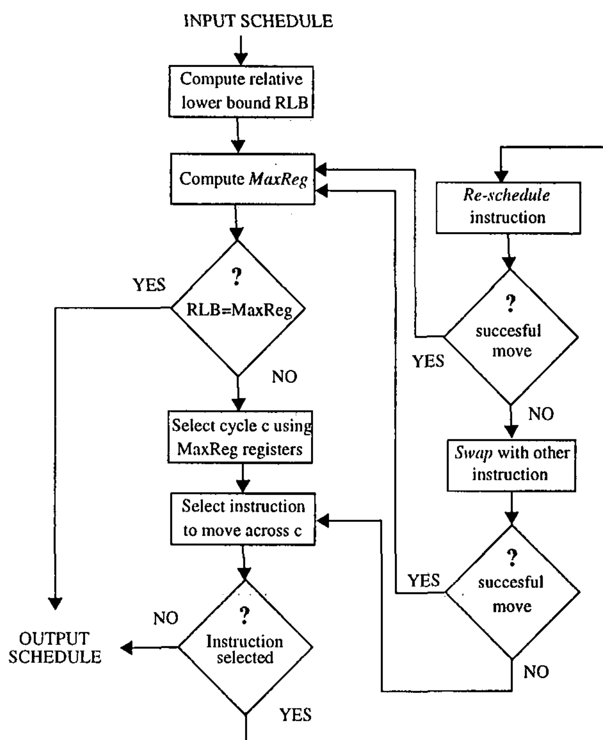

Figure 7.11 Flow diagram of *incremental scheduling*.

- *Re-schedule*: moves an instruction from the current cycle to another cycle if sufficient resources are available.
- *Swap*: swaps the scheduling of two instructions. The swapping is performed when both instructions have a similar execution pattern.

Figure 7.11 shows the flow diagram of the algorithm to rearrange instructions within the schedule of an iteration. It works as follows:

1. Compute the relative lower bound (RLB) on the number of registers required by the schedule. This is done to stop the search when a schedule requiring such resources is found.
2. Compute $MaxReg$, the maximum number of registers required at any cycle. The algorithm ends if $MaxReg = RLB$.
3. Select a cycle c which requires $MaxReg$ 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.
5. Move the selected instruction forward (if the variable is written) or backwards (if the variable is read) until crossing cycle c. This movement decrements the number of registers required

```
function incremental_scheduling;
    Calculate relative lower bound (RLB);
    Calculate MaxReg;
    c:=select_cycle(MaxReg);
    loop
        u:=select_node_to_move(c);
        exit if no node selected;
        success:=move_node(u, c);
        if success
            then
                Calculate MaxReg
                exit if MaxReg = RLB;
                c:=select_cycle(MaxReg);
            else restore schedule; {movement failed}
    return MaxReg;
```

Algorithm 7.4 Function *incremental_scheduling*

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 instruction, select another instruction 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. An instruction can be selected only once while the number of registers of the schedule is not reduced.

Algorithm 7.4 performs the previously described incremental scheduling.

The following sections describe the heuristics used to select an instruction to be moved (Section 7.5.2) and how an instruction is moved within the schedule (Section 7.5.3). Finally, Section 7.5.6 presents the computational complexity of the algorithm.

7.5.2 Selecting an instruction to move

Let u be the instruction to move, and let c be the cycle across which u must be moved. If u has been scheduled before c ($S(u) \leq c$), u will be moved *forward* across cycle c. Otherwise, It will be moved *backwards* across cycle c. The criteria used to select a node to be moved are as follows:

1. First, we try to select nodes to be moved *forward*. This is done because the scheduling algorithm schedules *as soon as possible* the instructions. Since a forward movement delays the instruction, the variable lifetime may be reduced. Nodes are selected among those that have no predecessors scheduled before (or at) cycle c and have at least one successor scheduled after cycle c (otherwise, no register reduction will be produced). The node which produces the longest variable lifetime is selected.

2. If no node with such characteristics exists, we try to select a node to be moved backwards. A node is selected among those that are scheduled after cycle c. The node selected is the one with the largest difference between the number of required input data (written in registers) and the number of data outputs to be written in registers. The number of input data stored in registers is approximated by the number of predecessors of the node. The number of output data is zero for instructions like *store*, and one for the remainder instructions. Among nodes with the same difference, the node which produces the variable with the longest lifetime is selected.

An instruction is selected only once if no movements are carried out within the schedule. The execution time of selecting a node to be moved is $O(VE)$.

7.5.3 Moving an instruction

This Section presents how a movement is performed within the schedule. A movement consists of moving a node (*forward* or *backward* across a given cycle c, attempting to prevent the variable lifetime from being alive at cycle c. The instruction to move is scheduled (if possible) after cycle c if movement is *forward*, and before or at cycle c if movement is *backwards*.

The *swapping* of two instructions is only performed when the instruction selected for movement cannot be *re-scheduled* because of the lack of available resources.

7.5.4 Re-scheduling

An instruction u is *re-scheduled* as follows:

1. Unschedule u (u was scheduled at cycle $S(u)$).
2. If u must be moved forward, try to schedule u from cycle $c+1$ to cycle $ALAP(u)$.
3. If u must be moved backwards, try to schedule u from cycle c to cycle $S(u)$.

Note that both explorations are not as greedy as possible. For example, if u must be moved forward, we could try to schedule u from cycle $ALAP(u)$ to cycle $c+1$ instead of from cycle $c+1$ to cycle $ALAP(u)$. A similar argumentation can be performed for a backwards move. It seems that the algorithm would quickly converge towards the final solution. However, experiments have shown that in some cases this approach obtains worse results.

7.5.5 Swapping

In order to *swap* u with another node, a node v is selected among those that have a similar execution pattern as u (both u and v use the same resources at the same cycle).

Node v is moved across cycle c without increasing the number of registers required at cycle c. The *swapping* is recursively done by following the same algorithm as that used to move u. That is,

first of all we try to *reschedule* v without moving any other node. If v cannot be *rescheduled*, we attempt to put v in the space used by other node x, and then we try to *swap* x for another node which has been scheduled after cycle c.

Although this is a recursive behavior, experiments have shown that the depth of the search is not very great and the recursive exploration is not expensive. In fact, the theoretical complexity is not greater than $O(V)$. In order to calculate this complexity, we have taken into account that a node is moved only once if the number of registers is not reduced. Therefore, no more than $|V|$ nodes can be recursively moved without success. Since $|V|$ nodes can be selected to be *swapped* with u, the computational complexity of *swapping* u with another node is $O(V^2)$. Given that the complexity of moving a node is dominated by the complexity of *swapping*, we conclude that moving a node executes in $O(V^2)$ time.

7.5.6 Computational complexity of incremental scheduling

The following observations are taken into account to calculate the computational complexity of function *incremental_scheduling*:

- Compute the relative lower bound on the number of required registers executes in $O(V + E)$, as can easily be derived from the definitions in Section 7.3.4.
- $MaxReg$ can be computed in $O(E)$ time, as shown in Section 7.4.7.
- The execution time of selecting a node to be moved is $O(VE)$, as shown in Section 7.5.2.
- Moving a node executes in $O(V^2)$ time, as shown in Section 7.5.3.
- The number of iterations of the loop is very difficult to compute. Experiments have shown that it is never greater than the number of instructions. Therefore, we will assume that the loop will seldom iterate more that $|V|$ times.

By taking into account the previous observations, we conclude that the execution time of *incremental_scheduling* is $O(V^3)$.

Since the execution time of reducing *SPAN* is $O(V^3E+VE^2)$, we conclude that the total execution time of *RESIS* is $O(V^3E + VE^2)$.

7.6 EXPERIMENTAL RESULTS

This section presents a summary of the results obtained by *RESIS*. More data can be found in Appendix C.

FUs	MS		after	after	time	time	diff
	II	Reg	SR	IS	SR	IS	
∞	4	6	8	8	0.16	0.01	+2
6	4	6	8	8	0.16	0.01	+2
5	4	7	9	9	0.16	0.01	+2
4	5	8	7	7	0.18	0.01	-1
3	6	8	7	7	0.08	0.01	-1

Table 7.1 Register reduction in a modulo scheduling algorithm for the Cytron example

FUs		MS		after	after	time	time	diff
*	+	II	Reg	SR	IS	SR	IS	
8	15	1	28	28	28	0.01	0.11	
4	8	2	16	16	13	0.01	0.10	-3
3	6	3	12	10	6	0.23	0.01	-6
3	5	3	12	10	6	0.23	0.01	-6
2	4	4	10	9	5	0.30	0.01	-5
2	3	4	11	8	5	0.25	0.01	-6
1	2	8	10	8	3	0.38	0.01	-7

Table 7.2 Register reduction in a modulo scheduling algorithm for the 16-Point Digital FIR Filter

FUs			MS		after	after	time	time	diff
+	-	*	II	Reg	SR	IS	SR	IS	
5	5	6	3	20	20	18	0.55	0.21	-2
4	4	4	4	18	18	17	0.60	0.20	-1
3	3	4	5	13	18	16	0.85	0.01	+3
3	3	3	6	12	18	18	1.06	0.01	+6
2	2	2	8	16	18	18	1.11	0.01	+2
1	1	1	16	13	15	12	1.31	0.01	-1

Table 7.3 Register reduction in a modulo scheduling algorithm for the Fast Discrete Cosine Transform

7.6.1 High-level synthesis

In order to show the efficiency of *RESIS*, we have use *RESIS* to reduce the register pressure in the schedules obtained by a modulo scheduling approach. The heuristics used by the modulo scheduling to select which instruction must be scheduled are based on the positive depth of each node. The node with the greatest positive depth is scheduled at each moment.

Tables 7.1 to 7.3 show the data obtained for the HLS examples. For each benchmark, the first columns show the available resources for each case. The next column shows the initiation interval and the register requirements of the schedule found by the modulo scheduling (MS). The next columns show the registers used by the schedule after each step of *RESIS*, as well as the CPU-time used by each step. The final column (diff) shows the register reduction achieved.

FUs	MS		after	time	diff
	II	Reg	IS	IS	
∞	4	6	6	0.06	
6	4	6	6	0.05	
5	4	7	7	0.08	
4	5	8	8	0.11	
3	6	8	8	0.05	

Table 7.4 Incremental scheduling after modulo scheduling for the Cytron example

FUs		MS		after	time	diff
*	+	II	Reg	IS	IS	
8	15	1	28	28	0.11	
4	8	2	16	13	0.05	-3
3	6	3	12	8	0.08	-4
3	5	3	12	8	0.11	-4
2	4	4	10	6	0.10	-4
2	3	4	11	8	0.13	-3
1	2	8	10	3	0.08	-7

Table 7.5 Incremental scheduling after modulo scheduling for the 16-Point Digital FIR Filter

Tables 7.1 to 7.3 show the registers required by the schedule after performing *SPAN* reduction (see the column labelled *after SR*). We do not consider such schedules if they require more registers than those obtained by *UNRET*. However, in order to show how reducing the *SPAN* may influence the final number of required registers *incremental scheduling* is executed by using such schedules. Note that, in some case, the number of registers increases as the *SPAN* decreases for Tables 7.1 and 7.3. Note also that the great improvement is found in those examples with greater quantity of available parallelism (see Table 7.2).

We have also carried out experiments by only executing incremental scheduling (without previously executing *SPAN* reduction). The results (see tables 7.4 to 7.6) suggest very interesting conclusions. On one hand, the register requirements after execute only incremental scheduling are sometimes better than those obtained after *SPAN* reduction plus incremental scheduling. This suggests that reducing variable lifetimes is not always efficient from the point of view of reducing register pressure. This is probably because the scheduler has less freedom to schedule instructions, since the distance of some dependences is decreased. On the other hand, reducing the *SPAN* before incremental scheduling works better in some cases. Therefore, we conclude that the *SPAN* must be selectively reduced. A way to do so is by giving the ability of moving operations across consecutive schedules to the incremental scheduling function. We believe that this new approach would probably obtain better results.

FUs			MS		after	time	diff
+	-	*	II	Reg	IS	IS	
5	5	6	3	20	18	0.20	-2
4	4	4	4	18	17	0.21	-1
3	3	4	5	13	12	0.18	-1
3	3	3	6	12	11	0.18	-1
2	2	2	8	16	10	0.20	-6
1	1	1	16	13	10	0.18	-3

Table 7.6 Incremental scheduling after modulo scheduling for the Fast Discrete Cosine Transform

7.6.2 Superscalar and VLIW processors

As performed with the HLS examples, we have executed *RESIS* with the schedules obtained by a modulo scheduling approach in order to show the efficiency of *RESIS*. A complete list of results can be found in Appendix C.

As an example, Table 7.7 shows the reduction obtained in the number of registers for a VLIW processor by using 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units. For each benchmark, the first two columns show the initiation interval and the register requirements of the schedule found by the modulo scheduling (MS). The next columns show the registers used by the schedule after each step of *RESIS*, as well as the CPU-time used by each step. The final column (diff) shows the register reduction achieved.

The results obtained show that the number of registers required by superscalar processors and by VLIW processors is quite similar for the same loops. This fact suggests that an instruction which produces a result and another instruction which consumes it are closely scheduled. Since this is one of the purposes of the register reduction phase, we claim that it is very effective. Moreover, the number of required registers is closed to the relative lower bound, also indicating the goodness of the approach. The effectiveness of *UNRET* is manifested by the fact that the relative lower bound is closed to the absolute lower bound, and a schedule with the minimum initiation interval is found in most cases.

7.7 SUMMARY AND CONCLUSIONS

In this chapter we present *RESIS*, a new algorithm for register optimization. *RESIS* is divided into two steps, namely *SPAN reduction* and *incremental scheduling*. *SPAN reduction* is based on reducing the maximum index (λ) for any instruction in the schedule. *Incremental scheduling* is a code reordering technique, oriented to reduce the number of registers required by the schedule.

SPAN reduction reduces the iteration time, variable lifetimes, and the size of the prologue and the epilogue of the loop execution. A reduction in the *SPAN* also reduces the number of times the loop must be unrolled to generate code.

Application Program		MS II	MS Reg	after SR	after IS	time SR	time IS	diff
SPEC-SPICE	Loop1	1	5	5	5	0.06	0.01	
	Loop2	2	15	15	14	0.10	0.01	-1
	Loop3	6	3	3	3	0.10	0.01	
	Loop4	10	11	11	11	0.33	0.01	
	Loop5	2	2	2	2	0.06	0.01	
	Loop6	2	35	34	34	0.08	0.01	-1
	Loop7	2	41	41	41	0.08	0.01	
	Loop8	2	6	6	6	0.06	0.01	
	Loop10	3	3	3	3	0.10	0.01	
SPEC-DODUC	Loop1-f	20	10	10	10	0.18	0.01	
	Loop3	21	8	8	7	0.13	0.01	-1
	Loop7	2	28	28	28	0.08	0.01	
SPEC-FPPPP	Loop1	20	4	4	4	0.13	0.01	
Livermore	Loop1	2	16	16	15	0.11	0.01	-1
	Loop5	3	5	5	5	0.08	0.01	
	Loop23	8	18	15	14	0.33	0.01	-4
Linpack	Loop1	1	13	13	13	0.08	0.01	
Whetstone	Loop1	17	8	8	8	0.26	0.01	
	Loop2	6	9	9	9	0.15	0.01	
	Loop3	5	5	5	5	0.13	0.01	
	Cycle1	4	2	2	2	0.11	0.01	
	Cycle2	2	4	4	4	0.10	0.01	
	Cycle4	2	6	6	6	0.06	0.01	
	Cycle8	2	10	10	10	0.06	0.01	

Table 7.7 Register reduction in a modulo scheduling algorithm by assuming a VLIW processor with 3 FP adders, 2 FP multipliers, 1 FP divisor and 2 load/store units

Incremental scheduling is based on moving instructions within the schedule. It attempts to reduce the maximum number of variables whose lifetime overlaps at any cycle. Two movements are considered: *reschedule* an instruction and *swapping* two instructions.

This chapter also presents different lower bounds on register requirements:

- an *absolute lower bound* for the number of registers required by any schedule of the loop.
- a *relative lower bound* for the number of registers required by any schedule of a given π-graph.
- a *lower bound* on the number of registers required by a given schedule, calculated as the maximum of the register requirements for each cycle. Such a lower bound might not be reached for a register allocation algorithm.

The results obtained by *RESIS* show that it is a good approach for reducing the number of registers required by a schedule. *RESIS* has been used with success for optimizing the schedules found by a modulo scheduling algorithm. However, *RESIS* improves the schedules found by *UNRET* only a few times. Therefore, we claim that *UNRET* is also a good algorithm from the point of view of register utilization. Results also suggest that reducing variable lifetimes (by *SPAN* reduction) is not always efficient from the point of view of reducing register pressure. Therefore, we conclude that *SPAN* reduction must be used selectively.