Swing Modulo Scheduling: A Lifetime-Sensitive Approach

Josep Llosa, Antonio González, Eduard Ayguadé, and Mateo Valero

Universitat Politècnica de Catalunya
Departament d'Arquitectura de Computadors, Barcelona (Spain)
Email: {josepll,antonio,eduard,mateo}@ac.upc.es

Abstract

This paper presents a novel software pipelining approach, which is called *Swing Modulo Scheduling* (SMS). It generates schedules that are near optimal in terms of initiation interval, register requirements and stage count. *Swing Modulo Scheduling* is an heuristic approach that has a low computational cost. The paper describes the technique and evaluates it for the Perfect Club benchmark suite. SMS is compared with other heuristic methods showing that it outperforms them in terms of the quality of the obtained schedules and compilation time. SMS is also compared with an integer linear programming approach that generates optimum schedules but with a huge computational cost, which makes it feasible only for very small loops. For a set of small loops, SMS obtained the optimum initiation interval in all the cases and its schedules required only 5% more registers and a 1% higher stage count than the optimum.

Keywords: Fine Grain Parallelism, Instruction Scheduling, Loop Scheduling, Software Pipelining, Register Requirements, VLIW and Superscalar Architectures.

1. Introduction

Software pipelining [5] is an instruction scheduling technique that exploits instruction level parallelism out of loops by overlapping successive iterations of the loop and executing them in parallel. The key idea is to find a pattern of operations (named the kernel code) so that when repeatedly iterating over this pattern, it produces the effect that an iteration is initiated before the previous ones have completed.

The drawback of aggressive scheduling techniques, such as software pipelining, is their high register pressure. The register requirements increase as the concurrency increases [18,16], due to either machines with deeper pipelines, or wider issue, or a combination of both. Registers, like functional units, are a limited resource. Therefore, if a schedule requires more registers than available, some actions, such as adding spill code, have to be performed. The addition of spill code can degrade performance [16] due to additional cycles in the schedule, or due to memory interferences.

Some research groups have targeted their work towards exact methods based on integer linear programming. For instance, the proposal in [11] search the entire scheduling space to find the optimal resource-constrained schedule with minimum buffer requirements, while the proposals in [10,6] find schedules with the actual minimum register requirements. The task of generating an optimal (in terms of throughput and register requirements) resource-constrained schedule for loops is known to be NP-hard. All these exact approaches require a prohibitive time to construct the schedules and therefore their applicability is restricted to very small loops. Therefore, any practical algorithm must use some heuristics to guide the scheduling process. Some of the proposals in the literature only care about achieving high throughput [21,14,13,24,8,20] while other proposals have also been targeted towards minimizing the register requirements [9,12,17], which result in more effective schedules.

Stage Scheduling [9] is not a whole modulo scheduler by itself but a set of heuristics targeted to reduce the register requirements of any given modulo schedule. This objective is achieved by moving operations in the schedule. The resulting schedule has the same throughput but lower register requirements. Unfortunately there are constraints in the movement of operations that might yield to suboptimal reductions of the register requirements.

Slack Scheduling [12] is a heuristic technique that simultaneously schedules some operations late and other operations early with the aim of reducing the register requirements and achieving maximum execution rate. The algorithm integrates recurrence constraints and critical-path considerations in order to decide when each operation is scheduled. The algorithm is based on Iterative Modulo Scheduling [8,20] in the sense that it may result in ejecting operations already scheduled to give place to a new one (sort of controlled backtracking).

1089-795X/96 $5.00 © 1996 IEEE
Proceedings of PACT '96

Hypernode Reduction Modulo Scheduling (HRMS) [17] is a heuristic strategy that tries to shorten loop variant lifetimes, without sacrificing performance. The main part of HRMS is the ordering strategy. The ordering phase orders the nodes before scheduling them, so that only predecessors or successors of a node can be scheduled before it is scheduled (except for recurrences). During the scheduling step the nodes are scheduled as soon/late as possible, if predecessors/successors have been previously scheduled. The effectiveness of their proposal is compared in terms of achieved throughput and compilation time against other heuristic methods [12,24] showing a better performance. The main drawback of the HRMS heuristic proposed to order the nodes is that it does not take into account that nodes are more critical in the scheduling process if they belong to a more critical path of the graph.

In this paper we present a novel ordering strategy, *Swing Modulo Scheduling* (SMS), that considers latencies to decide how critical the nodes are. It is an heuristic technique that has a low computational cost (e.g., compiling all the innermost loops without conditional exits and procedure calls of the Perfect Club takes less than half a minute) while it produces schedules very close to those generated by optimal approaches based on exhaustive search which have a prohibitive computational cost for real programs.

The rest of the paper is organized as follows. Section 2 overviews the main concepts related with software pipelining. Section 3 discusses an example to motivate our proposal, which is formalized in Section 4. Section 5 shows the main results of our experimental evaluation of the schedules generated by SMS. It is also compared with the schedules generated by other heuristic approaches and the optimal ones. The main concluding remarks are given in Section 6.

2. Overview of Software Pipelining

In a software pipelined loop, the schedule for an iteration is divided into stages so that the execution of consecutive iterations which are in distinct stages is overlapped. The number of stages in one iteration is termed stage count (*SC*). The number of cycles between the initiation of successive iterations (i.e. the number of cycles per stage) in a software pipelined loop is termed the Initiation Interval (*II*) [21].

The Initiation Interval *II* between two successive iterations is bounded by both recurrence circuits in the graph (*RecMII*) and resource constraints of the architecture (*ResMII*). This lower bound on the *II* is termed the Minimum Initiation Interval (*MII=max(RecMII, ResMII)*). The reader is referred to [8,20] for an extensive dissertation on how to calculate *ResMII* and *RecMII*.

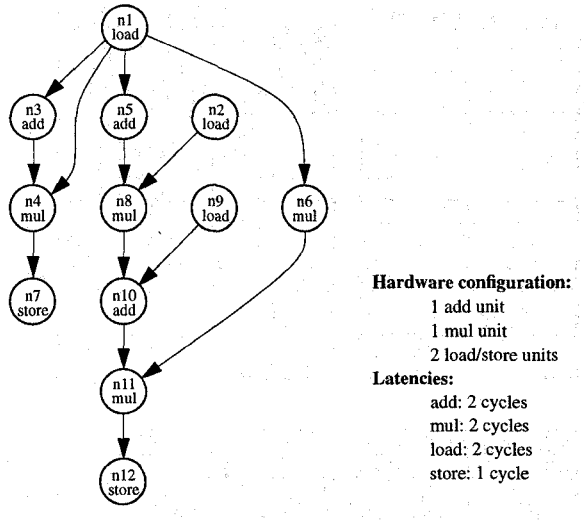


Figure 1: Dependence graph for the motivating example.

Values used in a loop correspond either to loop-invariant variables or to loop-variant variables. Loop-invariants are repeatedly used but never defined during loop execution. Loop-invariants have only one value for all iterations of the loop, therefore each one requires one register for all the execution of the loop regardless of the schedule and the machine configuration.

For loop-variants, a value is generated in each iteration of the loop and, therefore, there is a different lifetime corresponding to each iteration. Because of the nature of software pipelining, lifetimes of values defined in an iteration can overlap with lifetimes of values defined in subsequent iterations. This is the main reason why the register requirements are increased. In addition, for values with a lifetime larger than the *II* new values are generated before the previous ones are used. To fix this problem, either software solutions (modulo variable expansion [15]) and hardware solutions (rotating register files [7]) have been proposed.

Some of the software pipelining approaches can be regarded as the sequencing of two independent steps: node ordering and node scheduling. These two steps are performed assuming *MII* as the initial value for *II*. If it is not possible to obtain a schedule with this *II*, the scheduling step is performed again with an increased *II*. Next section shows how the ordering step influences on the register requirements of the loop.

3. Motivating example

Consider the dependence graph in Figure 1, and an architecture configuration with the pipelined functional units and latencies specified in the same figure. Since the graph in Figure 1 has no recurrence circuits, its initiation interval is constrained only by the available resources; in

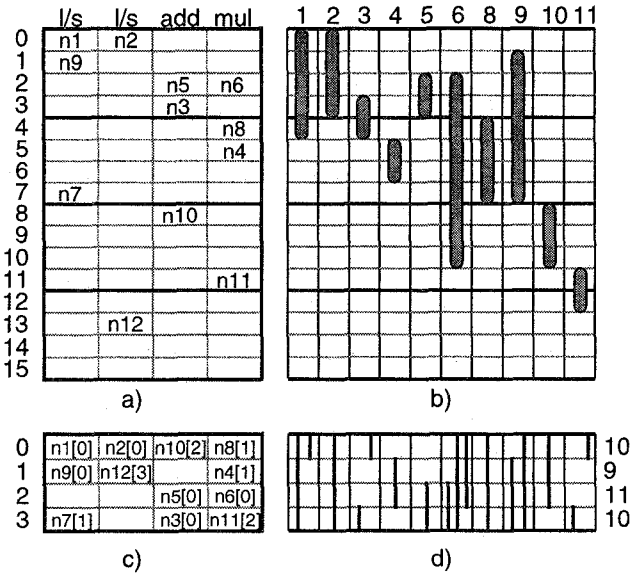


Figure 2: Top-Down scheduling: a) Schedule of one iteration, b) Lifetimes of variables, c) Kernel of the scheduling, and d) Register requirements.

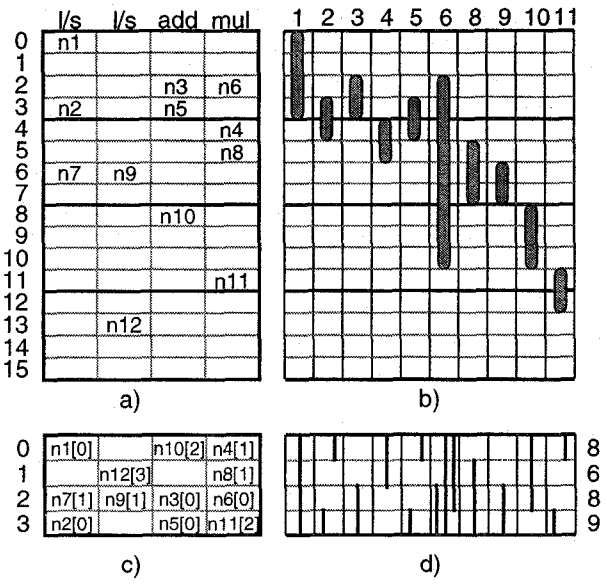


Figure 3: HRMS scheduling: a) Schedule of one iteration, b) Lifetimes of variables, c) Kernel of the scheduling, and d) Register requirements.

this case, the resource that limits the *MII* is the multiplier and the value is *MII = 4/1 = 4.*

A possible approach to order the operations to be scheduled would be to use a top-down strategy that gives priority to operations in the critical path; with this ordering, nodes would be scheduled in the following order: *<n1, n2, n5, n8, n9, n3, n10, n6, n4, n11, n12, n7>*. Figure 2.a shows the top-down schedule for one iteration and Figure 2.c the kernel code (numbers in brackets represent the stage to which the operation belongs). Figure 2.b shows the lifetimes of loop variants. The lifetime of a loop variant starts when the producer is issued and ends when the last consumer is issued. Figure 2.d shows the register requirements for this schedule; for each cycle it shows the number of live values required by the schedule. The number of registers required can be approximated by the maximum number of simultaneously live values at any cycle, which is called *MaxLive* (in [22] it is shown that register allocation never requires more than *MaxLive*+1 registers). In Figure 2.d, *MaxLive=11*. Notice that with this approach, variables generated by nodes *n2* and *n9* have an unnecessary large lifetime due to the early placement of the corresponding operations in the schedule; as a consequence, the register requirements for the loop increase.

In the strategy presented in [17] the ordering is done with the aim that all operations (except for the first one) have a previously scheduled reference operation. For instance, for the previous example, they would suggest the following order to schedule operations *<n1, n3, n5, n6, n4, n7, n8, n10, n11, n9, n2, n12>*. Notice that with this scheduling order, both *n2* and *n9* (the two conflicting operations in the top-down strategy) have a reference operation (*n8* and *n10*, respectively) already scheduled when they are going to be placed in the partial schedule.

Figure 3.a shows the final schedule for one iteration. For instance, when we schedule operation *n9*, operation *n10* has already been placed in the schedule (at cycle 8) so it will be scheduled as close as possible to it (at cycle 6), thus reducing the lifetime of the value generated by *n9*. Something similar happens with operation *n2*, which is placed in the schedule once its successor is scheduled. Figure 3.b shows the lifetimes of loop variants and Figure 3.d shows the register requirements for this schedule. In this case, *MaxLive=9*.

The ordering suggested by HRMS does not give preference to operations in the critical path. For instance, operation *n5* should be scheduled 2 cycles after the initiation of operation *n1*; however this is not possible since during this cycle the adder is busy executing operation *n3*, which has been scheduled before. Due to that, an operation in a more critical path (*n5*) is delayed in front of another operation that belongs to a less critical path (*n3*). Something similar happens with operation *n11* that conflicts with the placement of operation *n6*, which again belongs to a less critical path but the ordering has selected it before. Figures 4.a and 4.c show the schedule obtained by our proposal and Figures 4.b and 4.d the lifetime of variables and register requirements for this schedule. *MaxLive* for this schedule is 8. The schedule is obtained using the following ordering *<n12, n11, n10, n8, n5, n6, n1, n2, n9, n3, n4, n7>*. Notice that nodes in the critical

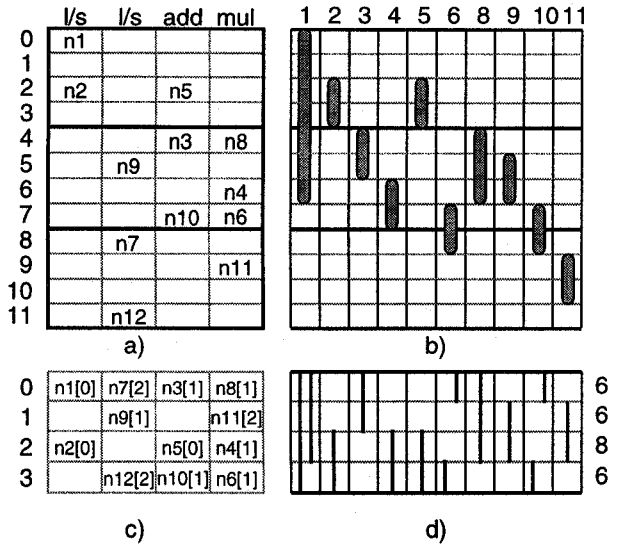


Figure 4: SMS scheduling: a) Schedule of one iteration, b) Lifetimes of variables, c) Kernel of the scheduling, and d) Register requirements.

path are scheduled with a certain preference with respect to the others. The following section details the algorithm that orders the nodes having in mind these ideas, and the scheduling step.

4. Swing Modulo Scheduling (SMS)

Most modulo scheduling approaches consists of two steps. First, they compute an schedule trying to minimize the *II* but without caring about register and then variables are allocated to registers. The execution time of a software pipelined loop depends on the *II*, the maximum number of live values of the schedule (*MaxLive*) and the stage count. The *II* determines the issue rate of loop iterations. Regarding the second factor, if *MaxLive* is not higher than the number of available registers then the computed schedule is feasible and then it does not influence the execution time. Otherwise, some action should be taken in order to reduce the register pressure. Some possible solutions outlined in [20] and evaluated in [16] are:

- Reschedule the loop with an increased *II*. In general, increasing the *II* will reduce *MaxLive* but it decreases the issue rate.
- Add spill code. This again has a negative effect since it increases the required memory bandwidth and it will result in more memory penalizations (e.g. cache misses). In addition, memory may become the most saturated resource and therefore adding spill code may require to increase the *II*.

Finally, the stage count determines the number of iterations of the epilogue part of the loop (it is exactly equal to the stage count minus one).

Swing Modulo Scheduling (SMS) is a modulo scheduling technique that tries to achieve a minimum *II*, reduce *MaxLive* and minimize the stage count. It is an heuristic technique that has a low computational cost while it produces schedules very close to those generated by optimal approaches based on exhaustive search, which have a computational cost prohibitive for real programs.

In order to achieve a minimum *II* and to reduce the stage count, SMS schedules the nodes in an order that takes into account the *RecMII* of the recurrence to which each node belongs (if any) and as a secondary factor it considers how critical is the path to which the node belongs.

To reduce *MaxLive*, SMS tries to minimize the lifetime of all the values of the loop. To achieve that, it tries to keep every operation as close as possible to both its predecessors and successors. When an operation is to be scheduled, if the partial schedule has only predecessors, it is scheduled as soon as possible. If the partial schedule contains only successors, it is scheduled as late as possible. The situation in which the partial schedule contains both predecessors and successors of the operation to be scheduled is undesirable since in this case, if the lifetime from the predecessors to the operation is minimized, the lifetime from the operation to its successors is increased. Some techniques like [9] deal with this situation by rescheduling the predecessors and the successors. SMS does not perform this type of backtracking but schedules the operations in such an order that this situation happens very rarely. In fact it happens only once for each recurrence and it is avoided completely if the loop does not contain any recurrence.

The algorithm followed by SMS consists of the following three steps that are described in detail below:

- Computation and analysis of the dependence graph.
- Ordering of the nodes.
- Scheduling.

SMS can be applied to generate code for innermost loops without subroutine calls. Loops containing IF statements can be handled after applying *if-conversion* [1] and provided that the processor supports predicated execution [7].

4.1. Computation and analysis of the dependence graph

The *dependence graph* of an innermost loop consists of a set of four elements ($DG=\{V, E, \delta, \lambda\}$):

- V is the set of nodes (vertices) of the graph, where each node $v \in V$ corresponds to an operation of the loop.
- E is the set of edges, where each edge $(u,v) \in E$ represents a dependence from operation u to operation v. Only data dependences (flow, anti and output-dependences) are included since the type of loops that SMS can handle only include one branch instruction at

the end that is associated to the iteration count. Other branches have been previously eliminated by the if-conversion phase.

- $\delta_{u,v}$ is called the distance function. It assigns a nonnegative integer to each edge $(u,v) \in E$. This value indicates that operation v of iteration I depends on operation u of iteration $I-\delta_{u,v}$.
- λ_u is called the latency function. For each node of the graph, it indicates the number of cycles that the corresponding operation takes.

Given a node $v \in V$ of the graph, $Pred(v)$ is the set of all the predecessors of v. That is, $Pred(v) = \{u \mid u \in V \text{ and } (u,v) \in E\}$. In a similar way, $Suc(v)$ is the set of all the successors of v. That is, $Suc(v) = \{u \mid u \in V \text{ and } (v,u) \in E\}$.

Once the dependence graph has been computed, some additional functions that will be used by the scheduler are calculated. In order to avoid cycles, one backward edge of each recurrence is ignored for performing these computations. These functions are the following:

- $ASAP_u$ is a function that assigns an integer to each node of the graph. It indicates the earliest time at which the corresponding operation could be scheduled. It is computed as follows:

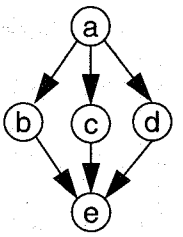
  $If\ Pred(u) = \varnothing\ then\ ASAP_u = 0$
  $else\ ASAP_u = max\ (ASAP_v + \lambda_v - \delta_{v,u} \times MII) \forall\ v \in Pred(u)$
- $ALAP_u$ is a function that assigns an integer to each node of the graph. It indicates the latest time at which the corresponding operation could be scheduled. It is computed as follows:

  $If\ Suc(u) = \varnothing\ then\ ALAP_u = max\ ASAP_v\ \forall\ v \in V$
  $else\ ALAP_u = min\ (ALAP_v - \lambda_u + \delta_{u,v} \times MII)\ \forall\ v \in Suc(u)$
- MOV_u is called the mobility function. For each node of the graph, it denotes the number of time slots at which the corresponding operation could be scheduled. Nodes in the most critical path have a mobility equal to zero and the mobility will increase as the path in which the operation is located is less critical. It is computed as follows:

  $$MOV_u = ALAP_u - ASAP_u$$
- D_u is called the depth of each node. For each node of the graph, it is defined as the maximum number of predecessors weighted by their latency. It is computed as follows:

  $If\ Pred(u) = \varnothing\ then\ D_u = 0$
  $else\ D_u = max\ (D_v + \lambda_v)\ \forall\ v \in Pred(u)$
- H_u is called the height of each node. For each node of the graph, it is defined as the maximum number of successors weighted by their latency. It is computed as follows:

  $If\ Suc(u) = \varnothing\ then\ H_u = 0$
  $else\ H_u = max\ (H_v + \lambda_u)\ \forall\ v \in Suc(u)$



$\lambda_a = \lambda_b = \lambda_c = \lambda_d = \lambda_e = 1$
$\delta_{a,b} = \delta_{a,c} = \delta_{a,d} = \delta_{b,e} = \delta_{c,e} = \delta_{d,e} = 0$
$ASAP_a = 0;\ ASAP_b = ASAP_c = ASAP_d = 1;\ ASAP_e = 2$
$ALAP_a = 0;\ ALAP_b = ALAP_c = ALAP_d = 1;\ ALAP_e = 2$
$MOV_a = MOV_b = MOV_c = MOV_d = MOV_e = 0$
$D_a = 0;\ D_b = D_c = D_d = 1;\ D_e = 2$
$H_a = 2;\ H_b = H_c = H_d = 1;\ H_e = 0$

Figure 5: A sample dependence graph.

4.2. Ordering the nodes

The ordering phase takes as input the dependence graph previously calculated and produces an ordered list containing all the nodes of the graph. This list indicates the order in which the nodes of the graph will be analyzed by the scheduling phase. That is, the scheduling phase (see next section) first allocates a time slot for the first node of the list; then, it looks for a suitable time slot for the second node of the list and so on. Notice that, as the number of nodes already placed in the partial schedule increases, there are more constraints to be met by the remaining nodes and therefore it is more difficult to find a suitable location for them.

As previously outlined, the target of the ordering phase is twofold:

- Give priority to the operations that are located in the most critical paths. In this way, the fact that the last operations to be scheduled should meet more constraints is offset by their higher mobility (MOV_u). This approach tends to reduce the II and the stage count.
- Try to reduce *MaxLive*. In order to achieve this, the scheduler will place each node as close as possible to both its predecessors and successors. However, the order in which the nodes are scheduled has a severe impact on the final result. For instance, assume the sample dependence graph of Figure 5 and a dual-issue processor.
  If node a is scheduled at cycle 0 and then node e is scheduled at cycle 2 (that is, they are scheduled based on their *ASAP* or *ALAP* values), it is not possible to find a suitable placement for nodes b, c and d since there are not enough slots between a and e. On the other hand, if nodes a and e are scheduled too far away, there are many possible locations for the remaining nodes. However, *MaxLive* will be too high no matter which possible schedule is chosen. For

instance, if we try to reduce the lifetime from *a* to *b*, we are increasing by the same amount the lifetime from *b* to *e*. In general, having scheduled both predecessors and successors of a node before scheduling it may result in a poor schedule. Because of this, the ordering of the nodes will try to avoid this situation whenever possible (notice that in the case of a recurrence, it can be avoided for all the nodes excepting one).

If the graph has no recurrences, the intuitive idea to achieve these two objectives is to compute an ordering based on a traversing of the dependence graph. The traversing starts by the node at the bottom of the most critical path and moves upwards, visiting all the ancestors. The order in which the ancestors are visited depends on their depth. In case of equal depth, nodes are ordered from less to more mobility. Once all the ancestors have been visited all the descendants of the already ordered nodes are visited but now moving downwards and in the order given by their height. Successive upwards and downwards sweeps of the graph are performed alternatively until all the graph has been traversed.

If the graph has recurrences, the graph traversing starts at the recurrence with the highest *RecMII* and applies the previous algorithm considering only the nodes of the recurrence. Once this subgraph has been traversed, the nodes of the recurrence with the second highest *RecMII* are traversed. At this step, the nodes located at any path between the previous and the current recurrence are also considered in order to avoid having scheduled both predecessors and successors of a node before scheduling it. When all the nodes belonging to recurrences or any path among them have been traversed, then the remaining nodes are traversed in a similar way.

Concretely, the ordering phase is a two-level algorithm. First a partial order is computed. This partial order consists of an ordered list of sets. The sets are ordered from the most to the least priority set but there is not any order inside each set. Each node of the graph belongs to just one set.

The most priority set consists of all the nodes of the recurrence with the highest *RecMII*. In general, the i^{th} set consists of the nodes of the recurrence with the i^{th} highest *RecMII*, eliminating those nodes that belong to any previous set (if any) and adding all the nodes located in any path that joins the nodes in any previous set and the recurrence of this set. Finally, the remaining nodes are grouped into sets of the same priority but this priority is lower than that of the sets containing recurrences. Each one of these sets consists of the nodes of a connected component of the graph that do not belong to any previous set.

Once this partial order has been computed, then the nodes of each set are ordered to produce the final and complete order. This step takes as input the previous list of sets and the whole dependence graph. The sets are handled in the order previously computed. For each recurrence of the graph, a backward edge is ignored in order to obtain a graph without cycles. The final result of the ordering phase is a list of ordered nodes *O* containing all the nodes of the graph.

```
O := Empty_list
For each set of nodes S in decreasing priority do
  if Pred_L(O) ≠ ∅ and Pred_L(O) ⊆ S then
    R := Pred_L(O) ∩ S
    order := bottom-up
  else if Suc_L(O) ≠ ∅ and Suc_L(O) ⊆ S then
    R := Suc_L(O) ∩ S
    order := top-down
  else
    R := {node with the highest ASAP value in S} ;
         if more than one, choose anyone
    order := bottom-up
  end if

  Repeat
    if order = top-down
      while R ≠ ∅ do
        v := Element of R with the highest H_v ;
             if more than one, choose node with lowest MOV_u
        O := O | <v>
        R := R - {v} ∪ (Suc (v) ∩ S)
      endwhile
      order := bottom-up
      R := Pred_L(O) ∩ S
    else
      while R ≠ ∅ do
        v := Element of R with the highest D_v ;
             if more than one, choose node with lowest MOV_u
        O := O | <v>
        R := R - {v} ∪ (Pred(v) ∩ S)
      endwhile
      order := top-down
      R := Suc_L(O) ∩ S
    endif
  until R = ∅
endfor
```

Figure 6: Ordering algorithm.

The ordering algorithm is shown in Figure 6, where | denotes the list append operation and *Suc_L(O)* and *Pred_L(O)* are the sets of predecessors and successors of a list of nodes respectively, which are defined as follows:

$$Pred\_L(O) = \{v \mid \exists\, u \in O \text{ such that } v \in Pred(u) \text{ and } v \notin O\}$$

$$Suc\_L(O) = \{v \mid \exists\, u \in O \text{ such that } v \in Suc(u) \text{ and } v \notin O\}$$

4.3. Scheduling

The scheduling step analyses the operations in the order given by the ordering step. The scheduling tries to schedule the operations as close as possible to the neighbors that have already been scheduled. When an operation is to be

scheduled, it is scheduled in different ways depending on the neighbors of these operations that are in the partial schedule.

- If an operation u has only predecessors in the partial schedule, then u is scheduled as soon as possible. In this case the scheduler computes the *Early_Start* of u as:

  $Early\_Start_u = max_{v \in PSP(u)} (t_v + \lambda_v - \delta_{v,u} \times II)$

  Where t_v is the cycle where v has been scheduled, λ_v is the latency of v, $\delta_{v,u}$ is the dependence distance from v to u, and $PSP(u)$ is the set of predecessors of u that have been previously scheduled. Then the scheduler scans the partial schedule for a free slot for the node u starting at cycle $Early\_Start_u$ until the cycle $Early\_Start_u + II - 1$. Notice that, due to the modulo constraint, it makes no sense to scan more than II cycles.
- If an operation u has only successors in the partial schedule, then u is scheduled as late as possible. In this case the scheduler computes the *Late_Start* of u as:

  $Late\_Start_u = min_{v \in PSS(u)} (t_v - \lambda_u + \delta_{u,v} \times II) =$

  Where $PSS(u)$ is the set of successors of u that have been previously scheduled. Then the scheduler scans the partial schedule for a free slot for the node u starting at cycle $Late\_Start_u$ until the cycle $Late\_Start_u - II + 1$.
- If an operation u has both predecessors and successors, then the scheduler computes $Early\_Start_u$ and $Late\_Start_u$ as described above and scans the partial schedule starting at cycle $Early\_Start_u$ until the cycle $min(Late\_Start_u, Early\_Start_u + II - 1)$. This situation will only happen for exactly one node of each recurrence circuit.
- Finally, if an operation u has neither predecessors nor successors, the scheduler computes the *Early_Start* of u as:

  $Early\_Start_u = ASAP_u$

  and scans the partial schedule for a free slot for the node u from cycle $Early\_Start_u$ to cycle $Early\_Start_u + II - 1$.

If no free slots are found for a node, then the II is increased by 1. The scheduling step is repeated with the increased II, which will provide more opportunities for finding free slots. One of the advantages of our proposal is that the nodes are ordered only once, even if the scheduling step has to do several trials.

4.4. Examples

This section illustrates the performance of the SMS by means of two examples. The first example is a small loop without recurrences and the second example uses a dependence graph with recurrences.

Assume that the dependence graph of the body of the innermost loop to be scheduled is that of Figure 1 (page 2), where all the edges represent dependences of distance zero. Assume also a four-issue processor with four functional units (1 adder, 1 multiplier and 2 load/store units) fully pipelined with the latencies listed in Figure 1.

The first step of the scheduling is to compute the *MII* and the *ASAP*, *ALAP*, mobility, depth and height of each node of the graph. *MII* is equal to 4. Table 1 shows the remaining values for each node.

Node	ASAP	ALAP	M	D	H
n1	0	0	0	0	10
n2	0	2	2	0	8
n3	2	6	4	2	4
n4	4	8	4	4	2
n5	2	2	0	2	8
n6	2	6	4	2	4
n7	6	10	4	6	0
n8	4	4	0	4	6
n9	0	4	4	0	6
n10	6	6	0	6	4
n11	8	8	0	8	2
n12	10	10	0	10	0

Table 1: ASAP, ALAP, mobility (M), depth (D) and height (H) of nodes of Figure 1.

Then, the nodes are ordered. The first level of the ordering algorithm groups all the nodes into the same set since there are not recurrences. Then, the elements of this set are ordered as follows:

- Initially $R=\{n12\}$ and *order* = *bottom-up*.
- Then, all the ancestors of *n12* are ordered depending on their depth and their mobility as a secondary factor. This gives the partial order O = <*n12, n11, n10, n8, n5, n6, n1, n2, n9*>.
- Then, the order shifts to *top-down* and all the descendants are ordered based on their height and mobility. This gives the final ordering O = <*n12, n11, n10, n8, n5, n6, n1, n2, n9, n3, n4, n7*>.

The next step is to schedule the operations following the previous order. II is initialized to *MII* and the operations are scheduled as shown in Figure 4 (page 4):

- The first node of the list, *n12*, is scheduled at cycle 10 (given by its *ASAP*) since there are neither predecessors nor successors in the partial schedule[1]. Once the schedule is folded this will become cycle 3 of stage 2.

1. In fact the resulting schedule stretches from cycles -1 to 10 but in all the figures we have normalized the representation starting always at cycle 0, so *n12* is in cycle 11 of Figure 4.