UNIVERSITAT POLITÈCNICA DE CATALUNYA

LOOP PIPELINING WITH RESOURCE AND TIMING CONSTRAINTS

Autor: Fermín Sánchez

October, 1995

5

SCHEDULING A Π-GRAPH

5.1 INTRODUCTION

According to Gajski [GDP86], "scheduling is the most important step during the architecture synthesis". This chapter presents the scheduling algorithm used by the methodologies proposed in this work. Henceforth, the algorithm will also be called *scheduler*.

A π-graph constituted by all the instructions (nodes) in the loop, but only those dependences that constrain the scheduling, contains all the information required by the scheduler. This π-graph is called the *scheduling graph*. Section 5.2 describes it in detail.

Several techniques [PK89a, Gof76, PK91, KD92] schedule an iteration of the loop as a basic block, i.e. the execution of a new iteration cannot start until the former iteration is completely executed. Therefore, the overlapping in the execution of consecutive iterations (functional pipelining) is not considered.

The execution of two consecutive instances of the schedule can be overlapped when the scheduler knows the target initiation interval in advance, in the same way as the execution of the iterations from the original loop are overlapped by software pipelining the loop. This produces the same effect as a functionally-pipelined data path. Section 5.3 shows how an overlapped schedule may be found by using a basic block.

Among all the scheduling algorithms described in the literature, we have selected *list scheduling* [Gof76] for its low computational complexity and easy implementation. The priority functions used by the algorithm are mainly based on the positive and the negative depth of each node. Section 5.5 provides an exhaustive description of the criteria used to select an instruction to be scheduled. Finally, Section 5.6 presents the algorithm used by the methodologies proposed in this work.

5.2 SCHEDULING GRAPH

Chapter 4 showed that only *positive scheduling dependences* (PSDs) and *negative restrictive dependences* (NRDs) constrain the scheduling process. Therefore, from the point of view of scheduling, it is sufficient for the π-graph to contain only the PSDs and the NRDs instead of all the dependences in the loop. Such a π-graph will be called the *scheduling graph*.

Definition 5.1 E^S **: Set of scheduling dependences**

For an expected initiation interval II, the set of dependences which constrain the scheduling of a π-graph, E^S, is $E^S = E^+ \cup E^-$.

$$E^S = \{(u,v) \in E \mid L_u - II \cdot \delta(u,v) > 0 \vee L_u - II \cdot \delta(u,v) > D^+(u) - II \}$$

Definition 5.2 π^S **: Scheduling Graph of** π

For a given II, the scheduling graph *of $\pi = G(V, E, \lambda, \delta)$, π^S, is the directed graph $\pi^S = G(V, E^S)$.*

5.3 OVERLAPPED SCHEDULE

This section presents how instructions are assigned to cycles. An instruction may start at the time of a given iteration and finish at the time of a different (following) iteration.

We represent the resource utilization performed by the instructions in a reservation table (RT), similar to the *modulo reservation table* used in [Lam88]. The RT has II rows and T columns, where T is the number of different types of resources of the architecture. Every row of the RT represents the number of resources of a given type available at a cycle; i.e. the element (i, j) identifies the number of resources of type j available at cycle i. Initially, all the elements in each column j are initialized with the number of resources of type j available in the architecture. Every time an instruction u is scheduled at cycle i, all the resources used during the execution of u are discounted in the RT from cycle i, according to the execution pattern of u. In order to achieve a schedule which overlaps with the schedule of the next iteration, the mapping between the execution pattern of the instructions and the RT must be modulo II. Thus, the assignation of resources at cycle c is actually performed at cycle $c \bmod II$.

Figure 5.1 shows an example of assignment of instructions to cycles in an RT with a length of four cycles. We assume the architecture has 2 FUs of type R_0, 3 FUs of type R_1 and 1 FU of type

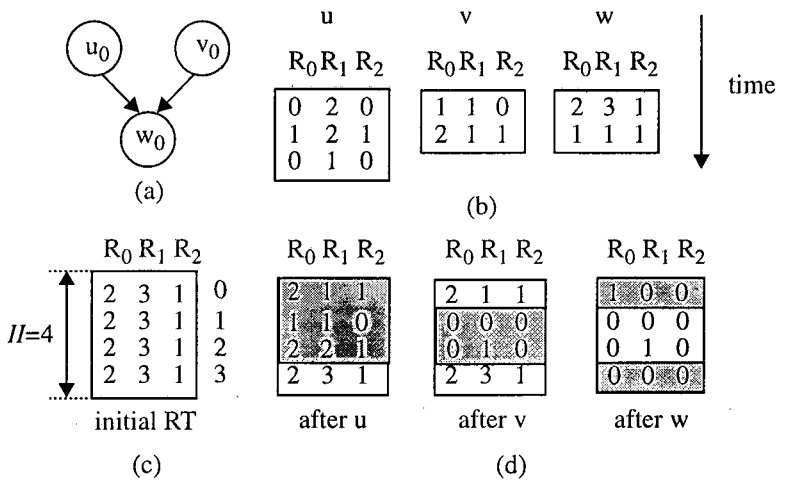

Figure 5.1 Reservation table example
(a) π-graph example
(b) Description of the execution pattern of instructions
(c) Initial contents of the reservation table
(d) Contents of the reservation table during the scheduling process

R_2. The shaded rectangles in the RT from Figure 5.1(d) represent the values updated during the scheduling of each instruction. Note that w is modulo-assigned in the RT. As a result, it starts at cycle 3 and ends at cycle 0 of the following iteration.

5.4 LIST SCHEDULING OVERVIEW

From among all the algorithms described in the literature, *list scheduling* [Gof76] has been selected for its low computational complexity and easy implementation. A *list scheduling* algorithm selects instructions in a topological order, guided by a user-defined *priority function*. Instructions are stored in a list, and the priority function is used to select an instruction from this list [DLSM81]. Different types of *list scheduling* algorithms can be found in the literature. A good description of them can be found in [Rim93]. We will use the algorithm 5.1.

Several *local priority functions* have been described in the literature, most of them related to the ASAP and the ALAP time of a node. RECALS [JMSW91] uses the IALAP time of a node as priority function. RECALS II [Rim93] uses a value called PALAP (*pseudo* ALAP time). The PALAP time of an instruction is the IALAP time when a subset of resource constraints is taken into account (note that the IALAP time is obtained by considering unlimited resources). The non-pipelined scheduling algorithm of SEHWA [PP88] uses *urgency* as priority function. The *urgency* of a node is similar to the positive depth. ELF [GK84] and the algorithm described in [Gof76] also use a priority function based on *urgency*. Slicer [PG87] proposes the *mobility* of the instructions as priority function. *Mobility* of u is defined as $ALAP(u) - ASAP(u)$. ATOMICS [GRVD87] also uses *mobility* as priority function, and MAHA [PPM86] uses *freedom*, a criterion

function List Scheduling;
LIST :=list of nodes ready to be scheduled;
while *LIST* not empty **do**
(1) Select a node u from *LIST*;
(2) Assign u to the earliest cycle satisfying precedence and resource constraints;
(3) Remove u from LIST;
(4) Append the successors of u ready to be scheduled to LIST (if they were not in LIST);

Algorithm 5.1 List Scheduling Algorithm

similar to *mobility*. Other priority functions mentioned in the literature are the *latest starting time* [ACD74, DLSM81] (the *latest starting time* of a node is closely related to the IALAP time of the node), the *violation of time constraints* (BSI)[NT86] (whether placing the instruction in the current cycle will violate a minimum time constraint and whether placing an instruction in a later cycle will violate a maximum time constraint) and the *self force* of an instruction [PK89a, PK89b] (the *self force* of an instruction reflects the effect of attempting a cycle assignment on the overall instruction concurrency, by taking resource constraints into account).

The reservation table guarantees that there are sufficient FUs in the architecture to execute the loop as it has been scheduled. The process of assigning concrete FUs to execute each instruction is known as *allocation* or *binding* [TS86, PK89b, ST91, HLH90, BC93], and it is beyond the scope of this paper. Some approaches perform resource allocation before scheduling [PK89a].

5.5 SCHEDULING PRIORITY FUNCTIONS

We have done experiments with several priority functions. Some of them have previously been proposed by other authors, but others are new in scheduling literature. The criteria to select nodes for scheduling are the following (in priority order):

1. The 0-mobility
2. The positive depth
3. The negative depth
4. The number of successors in the scheduling graph not yet scheduled
5. The resource utilization of each instruction

The order of consulting the criteria has been selected according to the influence of each one in the final length of the schedule. Next sections study in detail the motivations for each priority function.

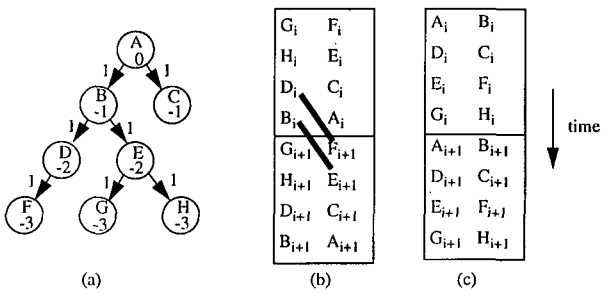

Figure 5.2 List scheduling when the priority function is the negative depth
(a) Example of π-graph composed only of INRDs
(b) Schedule obtained by assigning more priority to nodes with the highest negative depth
(c) Schedule obtained by assigning more priority to nodes with the lowest negative depth

5.5.1 The 0-mobility of a node

0-mobility is a particular case of mobility. Mobility is defined as the difference between the ALAP time and the ASAP time of a node. A node u has no mobility or has 0-mobility when $ALAP(u) - ASAP(u) = 0$. A node u which has 0-mobility is a node that can be scheduled only at cycle $ASAP(u)$. Therefore, a node with 0-mobility must be immediately scheduled; otherwise, finding a schedule would be impossible. Since 0-mobility nothing says about nodes with mobility greater than zero, it must be used in combination with other priority function(s).

5.5.2 The positive depth of a node

Positive depth gives an idea of the *urgency* for scheduling the node. Some algorithms [GK84, Gof76, PP88] use this priority function as main criterion to select which node must be scheduled. This criterion is sufficient for finding an optimal schedule when the number of resources is not constrained. When this priority function is used, the nodes with the highest positive depth have the greatest priority.

5.5.3 The negative depth of a node

Motivation and example

Figure 5.2(a) shows an example of π-graph representing a loop in which all dependences are INRDs (*initial negative restrictive dependences*). The numbers under labels of nodes are the negative depth assigned to each node. We assume that all instructions are multiplications that can be executed in a fully pipelined multiplier in 3 cycles (the result latency is 3 and the issue latency is 1). We also assume that two multipliers are available. With this assumption, $MII(\pi) = 4$.

Figure 5.2(b) shows a schedule obtained by assigning first the nodes with the lowest negative depth (the most negative). The schedule is incorrect, since dependences (D, F) and (B, E) are

violated. Such dependences are represented in the figure as bold lines between the schedule of two consecutive iterations. Intuitively, it is easy to see that the topology of the π-graph must guide the scheduling process. Thus, it seems better to schedule first the instructions with the highest negative depth (the least negative), as shown in Figure 5.2(c).

The negative depth initially assigned to each node assumes the node is ALAP scheduled (see definition of INRD). Let us consider an INRD (u, v). The negative depth of v would decrease from its initial value as much as $S(u)$ differs from the initial value of $ALAP(u)$, reducing its priority for scheduling with regard to other instructions. Moreover, the decreasing must be extended across all the NRDs. The new value in the negative depth must reflect the same priority among instructions that recomputing the negative depth for all nodes after scheduling u. The advantage of extending the decrease is that the computational complexity is lower than the computational complexity of recomputing the overall negative depth[1]. Summarizing:

- An initial negative depth must be assigned to every node by using the INRDs from the scheduling graph.
- Negative depth may change for some nodes during the scheduling process.

Changing negative depth of successors

Given an NRD (u, v), v is called a *negative successor* of u. Let k be an integer number greater than zero. If u is scheduled at cycle $ALAP(u) - k$, the negative depth of v must decrease in k units. Moreover, the decrease must be extended towards all the negative successors of v, and so on. The negative depth of a node is modified when it has not yet been scheduled, and once as maximum (to avoid negative cycles). A simple Breadth-first search algorithm may update the negative depth of the negative successors of a node in $O(V + E)$ time [CLR90]. The algorithm is invoked only when a node is scheduled before its ALAP time.

Changing negative depth of predecessors

Let us consider an NRD (u, v), and let us assume that v has already been scheduled and u has not yet been scheduled. The priority to select u must increase as much as $ALAP(u)$ differs from $IALAP(u)$. Since Equation (4.6) gives $IALAP(u) = II - D^+(u)$, we conclude that the negative depth of u must increase in $k = II - D^+(u) - ALAP(u)$ units after scheduling v (note that $k > 0$). Moreover, the increment must be extended towards the predecessors of u not yet scheduled, and so on. Updating the negative predecessors of v always increments their negative depth, and thus their priority for scheduling. In order to avoid negative cycles, all nodes are visited once as maximum. Negative depth is only modified if the node has not yet been scheduled.

We will consider NSDs (*negative scheduling dependences*) instead of NRDs when negative depth of predecessors is dynamically changed. This is because NRDs are a subset of the NSDs, which

[1] $O(V + E)$ if a node is modified only once, face to $O(V^2 \lg V + VE)$.

express constraints only towards the successors. Therefore, we define that u is a negative predecessor of v if (u, v) is an NSD. As for the negative successors, a simple Breadth-first search algorithm may update the negative depth of the negative predecessors of a node in $O(V + E)$ time [CLR90].

Dynamic changes on negative depth

For the sake of simplicity in the notation, we will henceforth shorten the negative predecessors not yet scheduled by NPNS, and the negative successors not yet scheduled by NSNS. When u is scheduled before $ALAP(u)$, the negative depth of its NPNS increases while the negative depth of its NSNS decreases (in general). Thus:

- The priority for scheduling the NPNS of u increases face to the remainder nodes not yet scheduled.
- The priority for scheduling the NSNS of u decreases face to the remainder nodes not yet scheduled.

Experiments have shown that taking NPNS before NSNS into account gives better results. Therefore, the negative depth of the NPNS is updated before the negative depth of the NSNS. Note that since the negative depth of a node is updated only once (as maximum) for each scheduled instruction, the result obtained is different according to whether NPNS are modified before or after NSNS.

Example of dynamic changes on negative depth

Figure 5.3 shows the effectiveness of considering dynamic changes on negative depth during the scheduling process. We assume that all instructions in the π-graph from Figure 5.3(a) are multiplications, and the architecture has two fully pipelined multipliers able to multiply in 2 cycles. The objective is finding a schedule in 4 cycles (the *MII*).

Figures 5.3(b) and 5.3(c) show respectively the positive and initial negative depth under labels of nodes, as well as the PSDs and INRDs. Dependences (B, E), (B, F), (C, G) and (D, G) are NSDs but are not INRDs. Figure 5.3(g) shows an incorrect schedule obtained by using the initial value of the negative depth as priority function. According to the priority function (1st) positive depth and (2nd) negative depth, nodes have been scheduled in the following order: E, F, G, H, A, D, C and B.

Note that instruction B cannot be scheduled at the last cycle because dependence (B, E) is violated. A bold line in Figure 5.3(g) represents such a violation. On the other hand, if B is the last scheduled instruction, it can only be assigned at cycle 3 because all the resources are busy at cycles 0, 1 and 2. However, the scheduling of B is advanced to the C and D when dynamic changes on negative depth are taken into account, as shown in Figure 5.3(h). Nodes have now been scheduled in the following order: E, F, G, A, H, B, D and C.

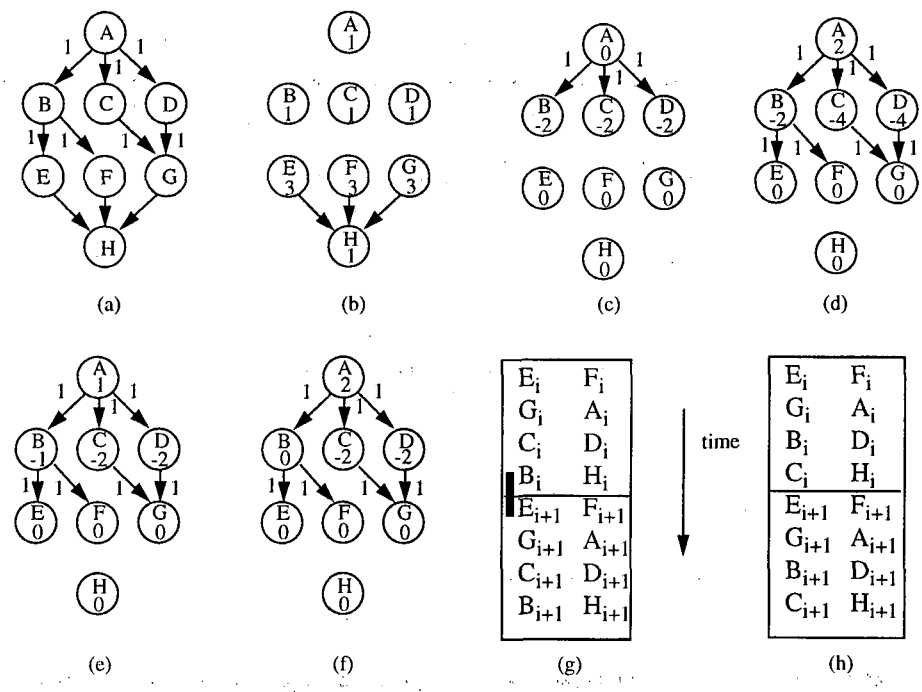

Figure 5.3 Scheduling with (1st) Positive depth (2nd) Negative depth
(a) Example of π-graph
(b) PSDs and positive depth
(c) INRDs and initial negative depth
(d) NSDs and final negative depth for the schedule of π-graph (h)
(e) Negative depth of nodes after scheduling instruction E
(f) Negative depth of nodes after scheduling instruction F
(g) Wrong schedule found without considering dynamic changes on negative depth
(h) Schedule found by taking dynamic changes on negative depth into account

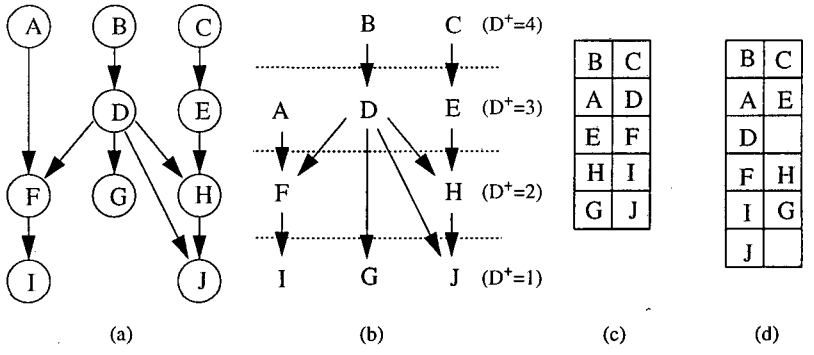

Figure 5.4 Scheduling with: (1st) Positive depth (2nd) Number of successors
(a) Example of π-graph
(b) Scheduling without considering resource constraints, dependences belonging to the scheduling graph and positive depth of nodes. Instructions are aligned according to their positive depth
(c) Schedule obtained by using two adders with the priority function (1st) Positive depth (2nd) Number of successors
(d) A possible schedule for two adders obtained by using as priority function (1st) Positive depth (2nd) *random*

5.5.4 The number of successors (not yet scheduled) of a node in the scheduling graph

The scheduler has greater possibilities of success if it can choose among the highest number of nodes. Thus, assigning more priority to nodes with the highest number of successors connected by PSDs or NRDs is a good priority function[2]. This priority function is a refinement of a more general one, consisting in assigning more priority to the node with the highest number of successors. This priority function has previously been used by systems as [CLS93], which uses it in the loop pipelining algorithm as the only priority function.

Figure 5.4 shows an example of the effectiveness of considering this priority function. Note that the schedule found by using the *number of successors not yet scheduled* as criterion is shorter than that obtained by using a random criterion.

5.5.5 The use of resources performed by an instruction

Since instructions are assigned at its ASAP time if possible, the resources are in general more used at the first cycles of a schedule. Therefore, scheduling an instruction among consecutive iterations (by modulo-assignment) will become difficult as the resource utilization performed by the instruction increases. In order to avoid this mishap, we first schedule instructions performing the highest use of the resources.

Figure 5.5 shows an example of the effectiveness of using this priority function in an overlapped schedule. Note that w uses fewer resources than v. Figure 5.5(c) shows a schedule in 8 cycles,

[2]Note that a successor connected by an FSD is already in the list of nodes ready to be scheduled.

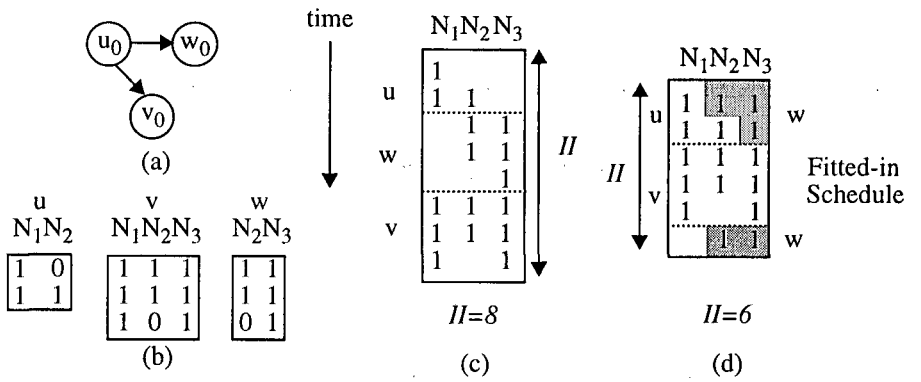

Figure 5.5 List scheduling by using as priority function the resource utilization performed by an instruction
(a) π-graph example
(b) Execution pattern of instructions
(c) A possible schedule requiring 8 cycles
(d) Overlapped schedule obtained by considering resource utilization

achieved by scheduling w before v. Figure 5.5(d) shows an overlapped schedule of 6 cycles length, achieved by scheduling v before w. Shadow parts in the schedule show the resource utilization carried out by w.

5.5.6 Complexity of selecting a node for scheduling

The running time of function selecting a node for scheduling is the highest running time of all the priority functions.

- By assuming that the ASAP time and the ALAP time of a node can be computed in $O(1)$ time (they can be held as information in the node), deciding which nodes have 0-mobility is computed in $O(V)$ time.
- Computing the node with the highest positive and negative depth also executes in $O(V)$ time, since positive and negative depth are data included in node information.
- The running time of computing the number of successors for each node is, at most, $O(E)$.
- Finally, by assuming that the resource utilization of a node is also a datum included in node information, computing the node that uses more resources executes in $O(V)$ time.

Therefore, the running time of selecting a node for scheduling is $O(V + E)$.

```
function scheduling (π, II,);
    if MPP(π) > II then return false; {scheduling impossible}
    π_p := G(V, E+, λ, δ);
    READY := ∅;
    for each u ∈ V do
        if predecessors(u, π_p)=0 then append(u, READY);
    while READY is not empty do
        node:=select_node(READY);
        delete_node(node, READY);
        c:=ASAP_scheduling(node);
        if node cannot be scheduled then return false;
        update_negative_predecessors(π, node, II );
        if c ≠ ALAP(node) then update_negative_successors(π, node, II );
        append_successors(node, π_p, READY);
    return true;
```

Figure 5.6 Scheduling algorithm

5.6 SCHEDULING ALGORITHM

The overall scheduling algorithm is sketched in Figure 5.6. The scheduling is impossible if $MPP(\pi) > II$. $READY$ is the list of the nodes ready to be scheduled. Function *predecessors* returns the number of predecessors of a node. Function *ASAP_scheduling* assigns a node *as soon as possible* and returns the cycle at which the node has been scheduled. Function append_successors($u, \pi_p, READY$) appends the successors of u in π_p to the list $READY$.

Computational complexity

The running time of the loop is as follows:

- The running time of function *select_node* is $O(V + E)$.
- Updating negative successors and predecessors executes in $O(V + E)$ time.
- Appending successors of a node to the list $READY$ executes in $O(E)$ time.

Since the loop iterates $|V|$ times, the running time of the loop is $O(V^2 + VE)$.

5.7 SUMMARY AND CONCLUSIONS

In this chapter, we present a list scheduling algorithm which executes in polynomial time. We describe how the schedules of successive iterations may be overlapped by modulo-assigning instruc-

tions in the reservation table. Although modulo scheduling is a well-known family of algorithms, most of the algorithms in HLS literature do not use the modulo-assignment feature in the reservation table. This is because such algorithms try to find as short a schedule of the loop as possible. We show that systems which know in advance the expected length of the schedule may efficiently modulo-assign instructions, obtaining shorter schedules than other approaches that do not make use of this knowledge. We call them overlapped schedules.

We also present the priority functions used to select an instruction for scheduling. Some of the priority functions are well-known, while others are presented here for the first time.

This chapter does not report results since the scheduling algorithm has been devised to be integrated in a loop pipelining system. Chapters 6 and 8 show the results obtained by the loop pipelining approach.

The main contributions of this chapter are the following:

- Although maximum time constraints have previously been studied by other authors, this is the first time negative depth is proposed as priority function in a list scheduling algorithm. In fact, to our knowledge, this is the first time negative depth is proposed to represent maximum time constraints.
- The use of resources of each node must be taken into account by the algorithm since the schedule may be overlapped.

Some algorithms previously proposed by other authors could benefit from negative depth. For example, the algorithms that simultaneously schedule and transform the loop, instruction by instruction. These algorithms retime the loop while scheduling is in process, transforming positive scheduling dependences into negative scheduling dependences. Therefore, taking negative scheduling dependences into account is very important for these algorithms. Modulo scheduling [RG81] and the algorithm described in [MD90] are examples of these kind of algorithms.

Dynamic changes on negative depth are shown to be an important key for scheduling retimed loops. Retimed loops usually contain a lot of NSDs, and therefore this kind of dependence plays an important role in the scheduling process. It is therefore crucial for some loops to take the variations produced in such dependences during the scheduling process fully into account. Moreover, the combination between positive and dynamic negative depth behaves is often similar to human behavior when schedules are performed by hand (the other priority functions take part in only ten per cent of the decisions).