Scheduling Algorithms for High-Level Synthesis

Zoltan Baruch
Computer Science Department, Technical University of Cluj-Napoca
E-mail: baruch@utcluj.ro

Abstract

The high-level synthesis is a process of translation from a behavioral description into a set of connected storage and functional units. The basic operations executed in high-level synthesis are partitioning, scheduling and allocation. Partitioning algorithms divide a behavioral description or design structure into subdescriptions in order to reduce the size of the problem or to satisfy some external constraints. Scheduling algorithms partition the variable assignments and operations into time intervals, and allocation partition them into storage and functional units.

In this paper we describe several algorithms for scheduling operations into control steps. We present two basic scheduling algorithms, the *ASAP* and the *ALAP* scheduling. We discuss two types of scheduling problems: time-constrained and resource-constrained. Time-constrained scheduling can use three different techniques: mathematical programming, constructive heuristics and iterative refinement. We present a constructive heuristic, called force-directed scheduling. For the resource-constrained scheduling, we describe a list-based scheduling algorithm.

1. Introduction

A behavioral description specifies the sequence of operations to be performed by the synthesized hardware. This description is compiled into an internal data representation such as the control/data flow graph (CDFG). Scheduling algorithms then partition the CDFG into subgraphs so that each subgraph is executed in one control step. Each control step corresponds to one state of the controlling finite-state machine.

Within a control step, a separate functional unit is required to execute each operation assigned to that step. Thus, the total number of functional units required in a control step corresponds to the number of operations scheduled in it. If more operations are scheduled into each control step, more functional units are necessary, which results in fewer control steps for the design implementation. On the other hand, if fewer operations are scheduled into each control step, fewer functional units are sufficient, but more control steps are needed.

Scheduling is an important task in high-level synthesis because it impacts the compromise between design cost and performance. Scheduling algorithms have to be tailored to suit the different target architectures used for implementation. For example, a scheduling algorithm designed for a non-pipelined architecture would have to be reformulated for a target architecture with datapath or control pipelining. The types of functional and storage units and of interconnection topologies used in the architecture also influence the formulation of the scheduling algorithms.

The different language constructs also influence the scheduling algorithms. Behavioral descriptions that contain conditional and loop constructs require more complex

scheduling techniques since dependencies across branch and loop boundaries have to be considered. Similarly, more sophisticated scheduling techniques must be used when a description has multidimensional arrays with complex indices.

In this paper we discuss issues related to scheduling and present solutions to the scheduling problem. We introduce the scheduling problem by discussing some basic scheduling algorithms on a simplified target architecture, using a simple design description.

2. Basic Scheduling Algorithms

In order to simplify the discussion of the basic scheduling algorithms, we assume the following restrictions:

- behavioral descriptions do not contain conditional or loop constructs;
- each operation takes exactly one control step to execute;
- each type of operation can be performed by one and only one type of functional unit.

We can define two different goals for the scheduling problem, given a library of functional units with known characteristics and the length of a control step. First, we can minimize the number of functional units for a fixed number of control steps. This is called *time-constrained scheduling*. Second, we can minimize the number of control steps for a given design cost. The design cost can be measured in terms of the number of functional and storage units, the number of NAND gates, or the chip-layout area. This approach is called *resource-constrained scheduling*.

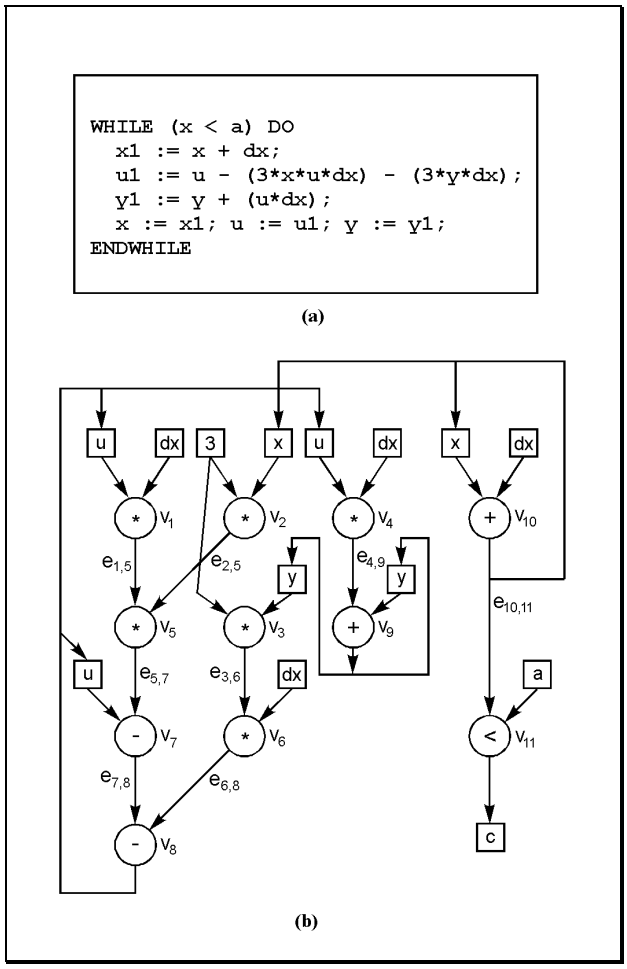

Figure 1. Example for solving a differential equation: (a) textual description; (b) DFG.

For the presentation of the scheduling algorithms, we introduce a few definitions. A data-flow graph (DFG) is a directed acyclic graph *G(V, E)*, where *V* is a set of nodes and *E* is a set of edges. Each $v_i \in V$ represents an operation o_i in the behavioral description. A directed edge $e_{i\,j}$ from $v_i \in V$ to $v_j \in V$ exists in *E* if the data produced by the operation o_i (represented by v_i) is consumed by operation o_j (represented by v_j). In this case v_i is an immediate predecessor of v_j. The set of all immediate predecessors of v_i is denoted by $Pred_{vi}$. Similarly, v_j is an immediate successor of v_i. The set of all immediate successors of v_i is denoted by $Succ_{vi}$.

This definitions are illustrated by an example of a circuit for solving numerically (by means of the *forward Euler method*) the following differential equation:

$$y'' + 3xy' + 3y = 0$$

in the interval [*0, a*], with step-size *dx* and initial values $x(0) = x$; $y(0) = y$; $y'(0) = u$. **Figure 1(a)** shows the textual description, and **Figure 1(b)** shows the DFG, which consists of 11 vertices, $V = \{v_1, v_2, ..., v_{11}\}$ and 8 edges, $E = \{e_{1,5}, e_{2,5}, e_{5,7}, e_{7,8}, e_{3,6}, e_{6,8}, e_{4,9}, e_{10,11}\}$.

Each DFG node has some flexibility about the state into which it can be scheduled. Many scheduling algorithms require the earliest and latest bounds within which operations are to be scheduled. We call the earliest state to which a node can possibly be assigned its *ASAP* value. This value is determined by the simple *ASAP* scheduling algorithm presented in **Figure 2**.

```
for each node v_i ∈ V do
    if Pred_vi = φ then
        E_i = 1;
        V = V - {v_i};
    else
        E_i = 0;
    endif
endfor
while V ≠ φ do
    for each node v_i ∈ V do
        if ALL_NODES_SCHED (Pred_vi, E) then
            E_i = MAX (Pred_vi, E) + 1;
            V = V - {v_i};
        endif
    endfor
endwhile
```

Figure 2. The *ASAP* scheduling algorithm.

The *ASAP* scheduling algorithm assigns an *ASAP* label (i.e., control-step index) E_i, to each node v_i of a DFG, thereby scheduling the operation o_i into the earliest possible control step s_{Ei}. The function *ALL_NODES_SCHED* ($Pred_{vi}$, *E*) returns *TRUE* if all the nodes in set $Pred_{vi}$ are scheduled (i.e., all immediate predecessors of v_i have a non-zero label *E*). The function *MAX* ($Pred_{vi}$, *E*) returns the index of the node with the maximum *E* value from the set of predecessor nodes for v_i.

The **for** loop of the algorithm initializes the *ASAP* value of all the nodes in the DFG. It assigns the nodes which do not have any predecessors to state s_1 and the other nodes to state s_0. In each iteration, the **while** loop determines the nodes that have all their predecessors scheduled and assigns them to the earliest possible state. Since we assume that the delay of all operations is 1 control step, the earliest possible state is calculated using the equation $E_i = MAX\,(Pred_{vi}, E) + 1$.

Figure 3(a) shows the results of the *ASAP* scheduling algorithm for the example shown in **Figure 1**. Operations o_1, o_2, o_3, o_4 and o_{10} are assigned to control step s_1, since they do not have any predecessors. Operations o_5, o_6, o_9 and o_{11} are assigned to control step s_2, and operations o_7 and o_8 are assigned to control steps s_3 and s_4, respectively.

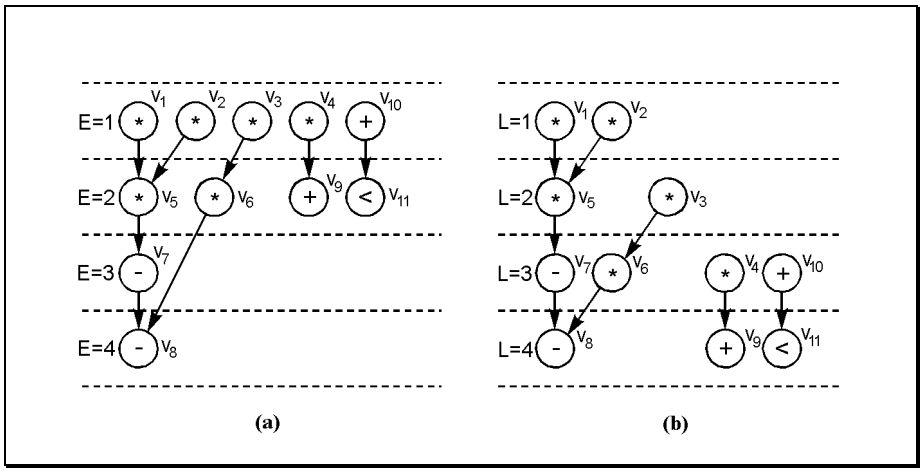

Figure 3. (a) *ASAP* schedule; (b) *ALAP* schedule.

The *ALAP* value for a node defines the latest state to which a node can possibly be scheduled. This value can be determined using the *ALAP* scheduling algorithm detailed in **Figure 4**. Given a time constraint of T control steps, the algorithm determines the latest possible control step in which an operation must begin its execution. The *ALAP* scheduling algorithm assigns an *ALAP* label L_i to each node v_i of a DFG, thereby scheduling the operation o_i in the latest possible control step s_{Li}. The function *ALL_NODES_SCHED* ($Succ_{vi}$, L) returns *TRUE* if all the nodes denoted by $Succ_{vi}$ are scheduled (i.e., all immediate successors of v_i have a non-zero L label). The function *MIN*($Succ_{vi}$, L) returns the index of the node with the minimum L value from the set of successor nodes for v_i.

```
for each node v_i ∈ V do
    if Succ_vi = φ then
        L_i = T;
        V = V - {v_i};
    else
        L_i = 0;
    endif
endfor
while V ≠ φ do
        for each node v_i ∈ V do
                if ALL_NODES_SCHED (Succ_vi, L) then
                        L_i = MIN (Succ_vi, L) - 1;
                        V = V - {v_i};
                endif
        endfor
endwhile
```

Figure 4. The *ALAP* scheduling algorithm.

The **for** loop of the algorithm initializes the *ALAP* value of all the nodes in the DFG. It assigns the nodes which do not have any successors to the last possible state, and the other nodes to the state s_0. In each iteration, the **while** loop determines the nodes that have all their successors scheduled and assigns them to the latest possible state.

Figure 3(b) shows the results of the *ALAP* scheduling algorithm (where $T = 4$) for the example shown in **Figure 1**. Operations o_8, o_9 and o_{11} are assigned to the last control step s_4, since they do not have any successors. Operations o_4, o_6, o_7 and o_{10} are assigned to control step s_3, and operations o_3 and o_5 are assigned to control step s_2. The remaining operations o_1 and o_2 are assigned to control step s_1.

Given a final schedule, we can easily compute the number of functional units that are required to implement the design. The maximum number of operations in any state denotes the number of functional units of that particular operation type. In the *ASAP* schedule, the maximum number of multiplication operations scheduled in any control step is four (state s_1), thus four multipliers are required. In addition, the *ASAP* schedule also requires an adder/subtracter and a comparator. In the *ALAP* schedule, the maximum number of multiplication operations scheduled in any control step is two (states s_1, s_2 and s_3), thus two multipliers are sufficient. In addition, the *ALAP* schedule also requires an adder, a subtracter and a comparator.

3. Time-Constrained Scheduling

Time-constrained scheduling is important for designs targeted towards applications in a real-time system. For example, in many digital signal processing (DSP) systems, the sampling rate of the input data stream dictates the maximum time allowed for carrying out a DSP algorithm on the present data sample before the next sample arrives. Since the sampling rate is fixed, the main objective is to minimize the cost of the hardware. Given the control step length, the sampling rate can be expressed in terms of the number of control steps that are required for executing a DSP algorithm.

Time-constrained scheduling algorithms can use three different techniques: mathematical programming, constructive heuristics and iterative refinement. We will present an example of the constructive heuristic methodology, called the force-directed scheduling method.

The force-directed scheduling (FDS) heuristic is a well known heuristic for scheduling with a given timing constraint. We present a simplified version of the FDS algorithm. The main goal of the algorithm is to reduce the total number of functional units used in the implementation of the design. This objective is achieved by uniformly distributing operations of the same type into all the available states. This uniform distribution ensures that functional units allocated to perform operations in one control step are used efficiently in all other control steps, which leads to a high unit utilization rate.

The FDS algorithm relies on both the *ASAP* and the *ALAP* scheduling algorithms to determine the range of control steps for every operation ($m\_range(o_i)$). It also assumes that each operation o_i has a uniform probability of being scheduled into any of the control steps in the range, and probability zero of being scheduled in any other control steps. Thus, for a given state s_j, such that $E_i \leq j \leq L_i$, the probability that operation o_i will be scheduled in that state is given by $p_j(o_i) = 1/(L_i - E_i + 1)$.

These probability calculations can be illustrated using the example from **Figure 1**, with the *ASAP* (E_i) and *ALAP* (L_i) values from **Figure 3** used in the calculations. The operation probabilities for the example are shown in **Figure 5(a)**. Operations o_1, o_2, o_5, o_7 and o_8 have probability values of 1 for being scheduled in steps s_1, s_1, s_2, s_3, and s_4 respectively, because the s_{E_i} value is equal to the s_{L_i} value for these operations. The width of a rectangle in this figure represents the probability ($1/(L_i - E_i + 1)$) of an opera-

tion getting started in that particular control step. For example, operation o_3 has a probability of 0.5 of being assigned to either s_1 or s_2. Therefore, the value of $p_1(o_3) = p_2(o_3) = 0.5$.

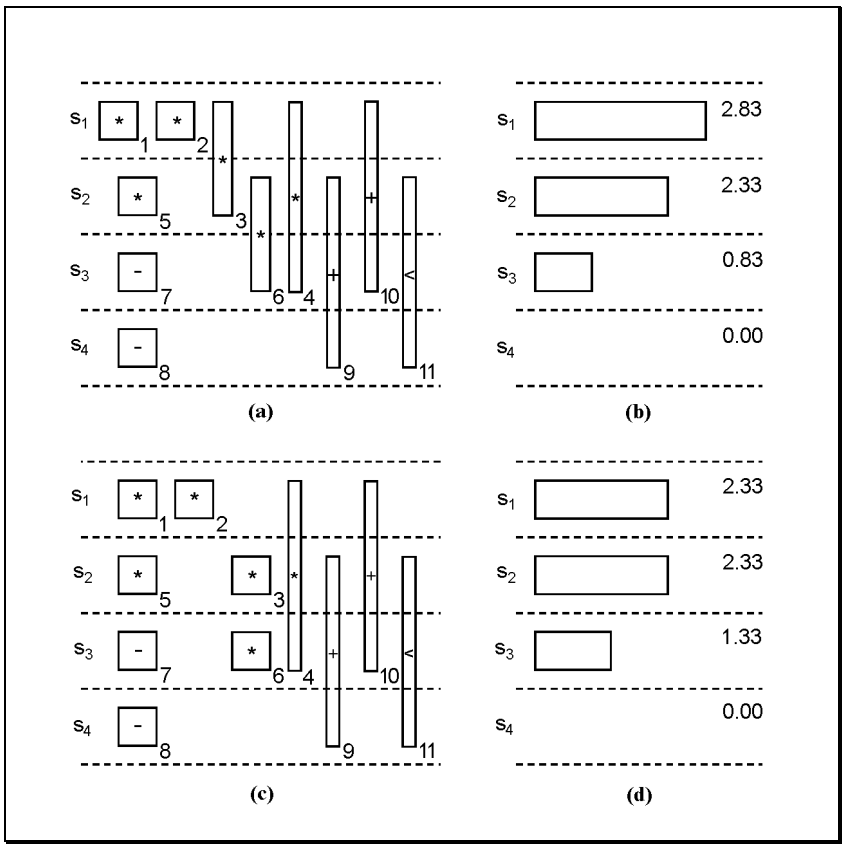

Figure 5. Force-directed scheduling example.

A set of probability distribution graphs (bar graphs) are created from the probability values of each operation, with a separate bar graph being constructed for each operation type. A bar graph for a particular operation type (e.g., multiplication), represents the expected operator cost (*EOC*) in each state. The expected operator cost in state s_j for operation type k is given by:

$$EOC_{j,k} = c_k * \sum_{i, s_j \in m\_range(o_i)} p_j(o_i)$$

where o_i is an operation of type k and c_k is the cost of the functional unit performing the operation of type k.

Figure 5(b) is a bar graph of expected operation costs for the multiplication operation in each control step. We can calculate the value for $EOC_{1,mult}$ as $EOC_{1,mult} = c_m \times (p_1(o_1) + p_1(o_2) + p_1(o_3) + p_1(o_4))$, which is $c_{mult} \times (1.0 + 1.0 + 0.5 + 0.33)$ or $2.83 \times c_{mult}$.

The bar graph in **Figure 5(b)** shows that the *EOC* for multiplication in the four states are 2.83, 2.33, 0.83 and 0.0. Since the functional units can be shared across states, the maximum of the expected operator costs over all states gives a measure of the total cost of implementing all operations of that type. Bar graphs similar to **Figure 5(b)** are constructed for all other operation types.

Since the main goal of the FDS algorithm is efficient sharing of functional units across all states, it attempts to balance the *EOC* value for each operation type. The

algorithm in **Figure 6** describes a method to achieve this uniform value of expected operator costs. During the execution of the algorithm, $S_{current}$ denotes the most recent partial schedule. S_{work} is a copy of the schedule on which temporary scheduling assignments are attempted. In each iteration, the variables *BestOp* and *BestStep* maintain the best operation to schedule and the best control step for scheduling the operation. When *BestOp* and *BestStep* are determined for for a given iteration, the $S_{current}$ schedule is changed appropriately using the function *SCHEDULE_OP* ($S_{current}$, o_i, s_j), which returns a new schedule, after scheduling operation o_i into state s_j on $S_{current}$. Scheduling a particular operation into a control step affects the probability values of other operations because of data dependencies. The function *ADJUST_DISTRIBUTION* scans through the set of vertices and adjusts the probability distributions of the successor and predecessor nodes in the graph.

```
ASAP (V);
ALAP (V);
while there exists o_i such that E_i ≠ L_i do
    MaxGain = – ∞;
    /* Try scheduling all unscheduled operations */
    /* to every state in its range */
    for each o_i , E_i ≠ L_i do
        for each j, E_i ≤ j ≤ L_i do
            S_work = SCHEDULE_OP (S_current, o_i, s_j);
            ADJUST_DISTRIBUTION (S_work, o_i, s_j);
            if COST (S_current) - COST (S_work) > MaxGain then
                MaxGain = COST (S_current) - COST (S_work);
                BestOp = o_i; BestStep = s_j;
            endif
        endfor
    endfor
    S_current = SCHEDULE_OP (S_current, BestOp, BestStep);
    ADJUST_DISTRIBUTION (S_current, BestOp, BestStep);
endwhile
```

Figure 6. The force-directed scheduling algorithm.

The function *COST* (*S*) evaluates the cost of implementing a partial schedule *S* based on any given cost function. A simple cost function could add the *EOC* values for each operation type:

$$COST(S) = \sum_{1 \le k \le m} \max_{1 \le j \le r} EOC_{j,k}$$

This cost is calculated using the *ASAP* (E_i) and *ALAP* (L_i) values for all the nodes.

During each iteration, the cost of assigning each unscheduled operation to possible states within its range (i.e., *m_range*(o_i)) is calculated using S_{work}.The assignment that leads to the minimal cost is accepted and the schedule $S_{current}$ is updated. Therefore, during each iteration an operation o_i gets assigned into control step s_k, where $E_i \le k \le L_i$. The probability distribution for operation o_i is changed to $p_k(o_i)$=1 and $p_j(o_i)$=0 for all j not equal to k. The operation o_i remains fixed and does not get moved in later iterations.

For the example presented, from the initial probability distribution for the multiplication operation shown in **Figure 5(b)**, the costs for assigning each unscheduled

operation into possible control steps are calculated. The assignment of o_3 to control step s_2 results in the minimal expected operator costs for multiplication, because max(p_j) falls from 2.83 to 2.33. This assignment is accepted. When operation o_3 is assigned to control step s_2, the probability values from operation o_6 also change as shown in **Figure 5(c)**. The operation o_3 is never moved while the iterations for scheduling other unscheduled operations continue.

In each iteration of the FDS algorithm, one operation is assigned to its control step based on the minimum expected operator costs. If there are two possible control-step assignments with close or identical operator costs, then the above algorithm cannot estimate the best choice accurately.

The method is called "constructive" because a solution is constructed without performing any backtracking. The decision to schedule an operation into a control step is made on the basis of a partially scheduled DFG; it does not take into account future assignments of operators to the same control step. Most likely, the resulting solution will not be optimal, due to the lack of a look-ahead scheme and the lack of compromises between early and late decisions. The solution can be optimized by rescheduling some of the operations in the given schedule.

4. Resource-Constrained Scheduling

The resource-constrained scheduling problem is encountered in many applications where we are limited by the silicon area. The constraint is usually given in terms of either a number of functional units or the total allocated silicon area.

In resource-constrained scheduling, we gradually construct the schedule, one operation at a time, so that the resource constraints are not exceeded and data dependencies are not violated. The resource constraints are satisfied by ensuring that the total number of operations scheduled in a given control step does not exceed the imposed constraints. The dependence constraints can be satisfied by ensuring that all predecessors of a node are scheduled before the node is scheduled. Thus, when scheduling operation o_i into a control state s_j, we have to ensure that the hardware requirements for o_i and other operations already scheduled in s_j do not exceed the given constraint and that all predecessors of node o_i have already been scheduled.

We describe the list-based scheduling method. The algorithm based on this method maintains a priority list of ready nodes. A ready node represents an operation that has all predecessors already scheduled. During each iteration the operations in the beginning of the ready list are scheduled until all the resources get used in that state. The priority list is always sorted with respect to a priority function. The priority function resolves the resource contention among operations. Whenever there are conflicts over resource usage among the ready operations (e.g., three additions are ready but only two adders are given in the resource constraint), the operation with higher priority gets scheduled. Scheduling an operation may make some other non-ready operations ready. These operations are inserted into the list according to the priority function. The quality of the results produced by a list-based scheduler depends predominantly on its priority function.

The algorithm (**Figure 7**) uses a priority list *PList* for each operation type ($t_k \in T$). These lists are denoted by the variables $PList_{t_1}$, $PList_{t_2}$, ..., $PList_{t_m}$. The operations in these lists are scheduled into control steps based on N_{t_k}, which is the number of functional units performing operation of type t_k. The function *INSERT_READY_OPS* scans

the set of nodes, V, determines if any of the operations in the set are ready, deletes each ready node from the set V and appends it to one of the priority lists based on its operation type. The function *SCHEDULE_OP* ($S_{current}$, o_i, s_j) returns a new schedule after scheduling the operation o_i in control step s_j. The function *DELETE* ($PList_{t_k}$, o_i) deletes the indicated operation o_i from the specified list.

```
INSERT_READY_OPS (V, PList_t1 , PList_t2 , ..., PList_tm );
Cstep = 0;
while ((PList_t1 ≠ φ ) or ... or (PList_tm V ≠ φ )) do
    Cstep = Cstep + 1;
    for k = 1 to m do
        for f_unit = 1 to N_k do
            if PList_tk ≠ φ then
                SCHEDULE_OP (S_current, FIRST (PList_tk ), Cstep);
                PList_tk = DELETE (PList_tk , FIRST (PList_tk ));
            endif
        endfor
    endfor
    INSERT_READY_OPS (V, PList_t1 , PList_t2 , ..., PList_tm );
endwhile
```

Figure 7. The list-based scheduling algorithm.

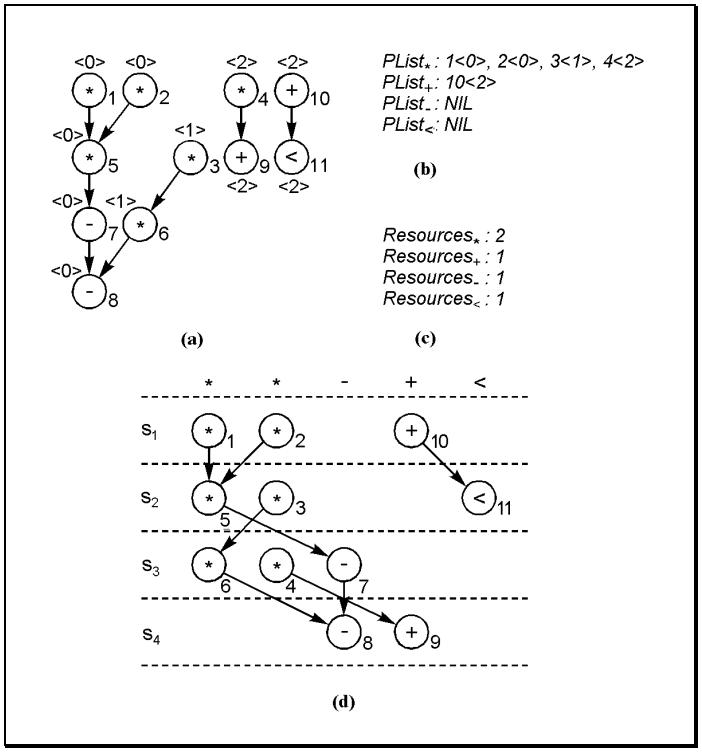

Figure 8. List-based scheduling example.

Initially, all nodes that do not have any predecessors are inserted into the appropriate priority list by the function *INSERT_READY_OPS*, based on the priority function. The **while** loop extracts operations from each priority list and schedules them into the current step until all the resources are exhausted in that step. Scheduling an operator in

the current step makes other successor-operations ready. These ready operations are scheduled during the next iterations of the loop. These iterations continue until all the priority lists are empty.

We illustrate the list scheduling process with an example (**Figure 8**). Suppose the available resources are two multipliers, one adder, one subtracter and one comparator (**Figure 8(c)**). Each operation o_i in the DFG in **Figure 8(a)** is labeled with its mobility range. Nodes with lower mobility must be scheduled first since delaying their assignment to a control step increases the probability of extending the schedule. Consequently, the mobility value is a good priority function. For each operator type, a priority list is constructed, in which priority is given to ready nodes with lower mobility. If two operations have the same mobility, then the one with a smaller index is given a higher priority.

The success of a list-scheduler depends mainly on its priority function. Mobility is just one of the many priority functions that have been proposed as a priority function. An alternative priority function uses the length of the longest path from the operation node to a node with no immediate successor. This longest path is proportional with the number of additional steps needed to complete the schedule if the operation is not scheduled into the current step. Therefore, an operation with a longer path label gets a higher priority. Another scheme uses the number of immediate successor nodes for an operation as a priority function: an operation node with more immediate successors is scheduled earlier because it makes more of these operations ready than a node with fewer successors.

5. Conclusions

We described several algorithms for scheduling operations into control steps. The *ASAP* and the *ALAP* schedules are basic scheduling algorithms, and the *ASAP* and *ALAP* values are used by other scheduling algorithms. The force-directed heuristic method produces schedules quickly, but the optimality of the solution cannot be guaranteed. The design quality of an initial schedule generated by any scheduling algorithm can be improved by the iterative refinement approach.

In scheduling, work is needed on using more realistic libraries, target architectures and cost functions. Scheduling algorithms that combine both scheduling and module selection must be developed. Similarly, scheduling algorithms must be combined with allocation, since scheduling and allocation are interdependent. Scheduling algorithms must be extended to incorporate different target architectures, for example a RISC architecture with a large register file and a few functional units, for which the minimization of load-and-store instructions from the main memory is the primary goal.

References

[1] Giovanni De Micheli: *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[2] D. D. Gajski, N. D. Dutt, C. H. Wu, Y. L. Lin: *High-Level Synthesis. Introduction to Chip and System Design*. Kluwer Academic Publishers, 1992.

[3] A. J. Martin: *Tomorrow's Digital Hardware will be Asynchronous and Verified*. Technical Report, Department of Computer Science, California Institute of Technology, Pasadena CA, 1993.

[4] Yachyang Sun: *Algorithmic Results on Physical Design Problems in VLSI and FPGA*. PhD Thesis, University of Illinois, Urbana, 1994.