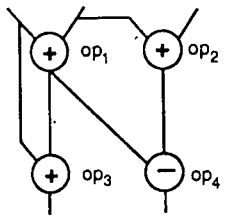


Figure 6.14: A sample data-flow graph.

13. Explain the difference between partitioning of behavioral descriptions and CDFGs.

14. Define several different quality measures for high-level partitioning.

15. *Outline an algorithm using a partitioning technique to guide the scheduling and allocation procedures. Hint: [McKo90].

16. Explain the differences between the five-clustering stages described in Section 6.3.1 and give an example of the loop relationship between clusters of operators. Hint: [LaTh91].

17. *Formulate a multiple-chip set partitioning into a hypergraph partitioning problem, define the objective functions and outline an algorithm. Hint: [GuDe90].

18. *Formulate a multiple-chip-set partitioning problem and outline an algorithm for chip partitioning as illustrated in Figures 6.11 and 6.12.

19. Explain the importance of estimation and interface synthesis in high-level partitioning. Hint: [NeTh86, Borr88, KuPa91].

20. *Discuss the interactions between estimation, interface synthesis and high-level partitioning.

21. **Develop a multiple-chip partitioning scheme including interface synthesis.

Chapter 7

Scheduling

7.1 Problem Definition

A behavioral description specifies the sequence of operations to be performed by the synthesized hardware. We normally compile such a description into an internal data representation such as the control/data flow graph (CDFG), which captures all the control and data-flow dependencies of the given behavioral description. Scheduling algorithms then partition this 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 in the FSMD model defined in Chapter 2. In the CDFG of the shift-and-add multiplier (Figure 5.3) dashed lines define the control step boundaries, and operations between two adjacent dashed lines are performed in the same control step.

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 directly 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 tradeoff between design cost and performance.

Scheduling algorithms have to be tailored to suit the different target architectures discussed in Chapter 2. 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, sophisticated scheduling techniques must be used when a description has multidimensional arrays with complex indices.

The tasks of scheduling and unit allocation are closely related. It is difficult to characterize the quality of a given scheduling algorithm without considering the algorithms that perform allocation (see Chapter 8). Two different schedules with the same number of control steps and requiring the same number of functional units may result in designs with substantially different quality metrics after allocation is performed.

In this chapter we discuss issues related to scheduling and present some well known 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. We then describe extensions to the basic algorithms for handling more sophisticated description models and realistic libraries of functional units.

7.2 Basic Scheduling Algorithms

In order to simplify the discussion of the basic scheduling algorithms, let us begin by assuming the following restrictions:

(a) behavioral descriptions do not contain conditional or loop constructs,

(b) each operation takes exactly one control step to execute,

(c) each type of operation can be performed by one and only one type of functional unit.

These assumptions will be relaxed later when more realistic models are considered.

We can define two different goals for the scheduling problem, given a library of functional units with known characteristics (e.g., size, delay, power) and the length of a control step. First, we can minimize the number of functional units for a fixed number of control steps. We call this fixed-control-step approach, 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 two-input NAND gates, or the chip-layout area. This cost-minimizing approach is called resource-constrained scheduling.

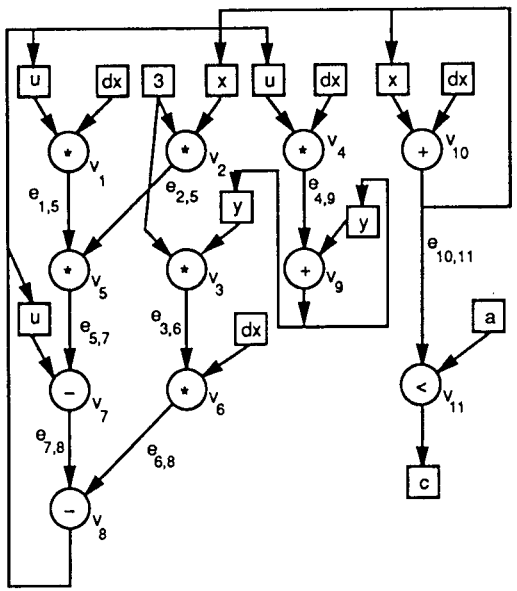
In order to understand the scheduling algorithms discussed in this chapter, we require a few simple definitions. A data-flow graph (DFG) is a directed acyclic graph G(V, E), where V is a set of nodes and E a set of edges. Each $v_i \in V$ represents an operation (o_i) in the behavioral description. A directed edge e_{ij} from $v_i \in V$ to $v_j \in V$ exists in E if the data produced by operation o_i (represented by v_i) is consumed by operation o_j (represented by v_j). In this case we say that v_i is an immediate predecessor of v_j. The set of all immediate predecessors of v_i is denoted by $Pred_{v_i}$. Similarly v_j is an immediate successor of v_i. $Succ_{v_i}$ denotes the set of all immediate successors of v_i. Each operation o_i can be executed in D_i control steps. Since we assume that each operation takes exactly one control step, the value of D_i is 1 for every o_i.

We can illustrate the above definitions using the simple HAL example [PaKn89]. Figure 7.1(a) shows HAL's textual description, and Figure 7.1(b) shows its DFG, which consists of eleven vertices, V = {v_1, v_2, ..., v_{11}} and eight edges, E = {$e_{1,5}$, $e_{2,5}$, $e_{5,7}$, $e_{7,8}$, $e_{3,6}$, $e_{6,8}$, $e_{4,9}$, $e_{10,11}$}.

DFGs expose parallelism in the design. Consequently, 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

```
while (x < a)  do
  x1 := x + dx;
  u1 := u – (3 * x * u * dx) – (3 * y * dx);
  y1 := y + (u*dx);
  x  := x1; u := u1; y := y1;
endwhile
```

(a)



(b)

Figure 7.1: HAL example: (a) textual description, (b) DFG.

Algorithm 7.1: ASAP Scheduling.

```
for each node v_i ∈ V do
  if Pred_{v_i} = 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_{v_i}, E) then
      E_i = MAX(Pred_{v_i}, E) + 1;
      V = V - { v_i };
    endif
  endfor
endwhile
```

by the simple ASAP scheduling algorithm outlined in Algorithm 7.1.

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 operation o_i into the earliest possible control step s_{E_i}. $Pred_{v_i}$ denotes all the nodes in the DFG that are immediate predecessors of the node v_i. The function ALL_NODES_SCHED($Pred_{v_i}$, E) returns true if all the nodes in set $Pred_{v_i}$ are scheduled (i.e., all immediate predecessors of v_i have a non-zero E label). The function MAX($Pred_{v_i}$, E) returns the index of the node with the maximum E value from the set of predecessor nodes for v_i.

The *for* loop in Algorithm 7.1 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($Pred_{v_i}$, E) + 1.

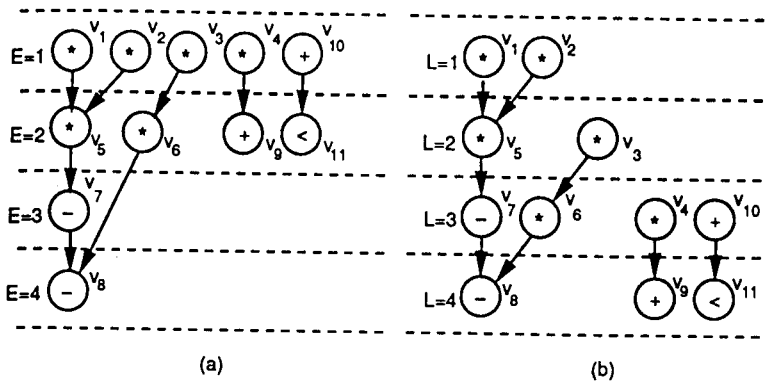


Figure 7.2: HAL example: (a) ASAP schedule, (b) ALAP schedule.

Figure 7.2(a) shows the results of the ASAP scheduling algorithm on the HAL example shown in Figure 7.1. 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.

The ALAP value for a node defines the latest state to which a node can possibly be scheduled. We can determine the ALAP value using the scheduling algorithm detailed in Algorithm 7.2. 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 operation o_i in the latest possible control step s_{L_i}. The function ALL_NODES_SCHED($Succ_{v_i}$, L) returns true if all the nodes denoted by $Succ_{v_i}$ are scheduled (i.e., all immediate successors of v_i have a non-zero L label). The function MIN($Succ_{v_i}$, L) returns the index of the node with the minimum L value from the set of successor nodes for v_i.

The *for* loop in Algorithm 7.2 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 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.

Algorithm 7.2: ALAP Scheduling.

```
for each node v_i ∈ V do
  if Succ_{v_i} = 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(Pred_{v_i}, L) then
      L_i = MIN(Succ_{v_i}, L) - 1;
      V = V - { v_i };
    endif
  endfor
endwhile
```

Figure 7.2(b) shows the results of the ALAP scheduling algorithm (where $T = 4$) on the HAL example shown in Figure 7.1. 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 s_3 and operations o_5 and o_3 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 (Figure 7.2(a)) 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 (Figure 7.2(b)) 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.

7.2.1 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 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. In this section we discuss the integer linear programming method (an example of mathematical programming), the force-directed scheduling method (an example of a constructive heuristic methodology) and an iterative rescheduling technique.

Integer Linear Programming Method

The integer linear programming (ILP) method [LeHL89] finds an optimal schedule using a branch-and-bound search algorithm that involves backtracking, i.e., some of the decisions made in the initial stages of the algorithm are revisited as the search progresses. Let s_{E_k} and s_{L_k} be the control steps into which operation o_k is scheduled by the ASAP and ALAP algorithms. Clearly, $E_k \leq L_k$. In a feasible schedule, o_k must begin its execution in a control step no sooner than s_{E_k} and no later than s_{L_k}.

The number of control steps between s_{E_k} and s_{L_k} is called the mobility range of operation o_k (i.e., mrange $(o_k) = \{s_j | E_k \leq j \leq L_k\}$). Figure 7.3(a) shows the range of every operation in the DFG for the HAL example, computed from the ASAP and ALAP labels in Figure 7.2. For example, the range of o_4 is $\{s_1, s_2, s_3\}$, since its ASAP and ALAP labels are $E_4 = 1$ and $L_4 = 3$.

We can now use the ASAP, ALAP and range values of the operations to formulate the scheduling problem using ILP. We use the following notations for the ILP formulation:

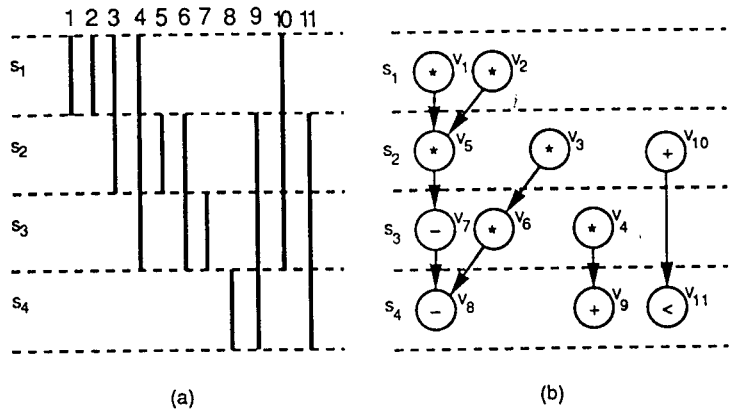


Figure 7.3: An ILP scheduling example: (a) ranges for operations, (b) final schedule.

Let OP = $\{o_i \mid 1 \leq i \leq n\}$ be the set of operations in the flowgraph and t_i = type(o_i) be the type of each operation o_i. Let T = $\{t_k \mid 1 \leq k \leq m\}$ be the set of possible operation types. The set OP_{t_k} consists of operations from the set OP which are of type t_k (i.e., OP_{t_k} = $\{ o_i \mid o_i \in$ OP $\wedge$ type$(o_i) = t_k\}$). Let $INDEX_{t_k}$ define the set of all operation indices in OP_{t_k}. (i.e., $INDEX_{t_k} = \{ i \mid o_i \in OP_k \}$). Let N_{t_k} be the number of units performing operation t_k and let C_{t_k} be the cost of such an unit. Let S = $\{s_j \mid 1 \leq j \leq r\}$ be the set of control steps available for scheduling the operations. Let x_{ij} be the 0-1 integer variables, which assume a value of 1 if operation o_i is scheduled into step s_j and 0 otherwise.

The scheduling problem can be formulated as follows:

$$minimize \quad \sum_{k=1}^{m} (C_{t_k} \times N_{t_k}) \tag{7.1}$$

under the assumptions,

$$\forall i, 1 \leq i \leq n, (\sum_{E_i \leq j \leq L_i} x_{i,j} = 1), \tag{7.2}$$

$$\forall j, 1 \leq j \leq r, \forall k,\ 1 \leq k \leq m, (\sum_{i \in INDEX_{t_k}} x_{i,j} \leq N_{t_k}), \tag{7.3}$$

and

$$\forall i, j, o_i \in Pred_{o_j}(\sum_{E_i \leq k \leq L_i} (k \times x_{i,k}) - \sum_{E_j \leq l \leq L_j} (l \times x_{j,l}) \leq -1) \tag{7.4}$$

The objective function (7.1) minimizes the total cost of the required functional units. Condition 7.2 requires that every operation o_i be scheduled into one and exactly one control step no sooner than E_i and no later than L_i. Condition 7.3 ensures that no control step contains more than N_{t_k} operations of type t_k. Finally, Condition 7.4 guarantees that for an operations o_j, all its predecessors (i.e., $Pred_{o_j}$)) are scheduled in an earlier control step. In other words, if $x_{i,k} = x_{j,l} = 1$ then $k < l$.

Let us now use the DFG from Figure 7.1(b) to illustrate the ILP formulation for scheduling the flowgraph into four control steps. Since the DFG contains four different types of operations (i.e., multiplication, addition, subtraction and comparison), we need four types of functional units from the library. Let C_m, C_a, C_s and C_c be the costs of a multiplier, an adder, a subtracter and a comparator, respectively. Let N_m, N_a, N_s and N_c be the number of multipliers, adders, subtracters and comparators needed in the final schedule. The ILP formulation for scheduling the DFG of Figure 7.1(b) into four control steps is

$$minimize \quad C_m \times N_m + C_a \times N_a + C_s \times N_s + C_c \times N_c \tag{7.5}$$

subject to

$$\begin{aligned}
x_{1,1} &= 1 \\
x_{2,1} &= 1 \\
x_{3,1} + x_{3,2} &= 1 \\
x_{4,1} + x_{4,2} + x_{4,3} &= 1 \\
x_{5,2} &= 1 \\
x_{6,2} + x_{6,3} &= 1 \\
x_{7,3} &= 1 \\
x_{8,4} &= 1 \\
x_{9,2} + x_{9,3} + x_{9,4} &= 1 \\
x_{10,1} + x_{10,2} + x_{10,3} &= 1 \\
x_{11,2} + x_{11,3} + x_{11,4} &= 1 \\
x_{1,1} + x_{2,1} + x_{3,1} + x_{4,1} &\leq N_m \\
x_{3,2} + x_{4,2} + x_{5,2} + x_{6,2} &\leq N_m \\
x_{4,3} + x_{6,3} &\leq N_m \\
x_{7,3} &\leq N_s \\
x_{8,4} &\leq N_s \\
x_{10,1} &\leq N_a \\
x_{9,2} + x_{10,2} &\leq N_a \\
x_{9,3} + x_{10,3} &\leq N_a \\
x_{9,4} &\leq N_a \\
x_{11,2} &\leq N_c \\
x_{11,3} &\leq N_c \\
x_{11,4} &\leq N_c \\
1x_{3,1} + 2x_{3,2} - 2x_{6,2} - 3x_{6,3} &\leq -1 \\
1x_{4,1} + 2x_{4,2} + 3x_{4,3} - 2x_{9,2} - 3x_{9,3} - 4x_{9,4} &\leq -1 \\
1x_{10,1} + 2x_{10,2} + 3x_{10,3} - 2x_{11,2} - 3x_{11,3} - 4x_{11,4} &\leq -1.
\end{aligned}$$

If we assume that $C_m = 2$ and $C_a = C_b = C_c = 1$, the cost function is minimized and all inequalities are satisfied when the values for the variables are set as follows: $N_m = 2$, $N_a = N_s = N_c = 1$, $x_{1,1} = x_{2,1} = x_{3,2} = x_{4,3} = x_{5,2} = x_{6,3} = x_{7,3} = x_{8,4} = x_{9,4} = x_{10,2} = x_{11,4} = 1$, and

other xs = 0. This solution to the ILP formulation of the scheduling problem is shown in Figure 7.3(b). In the figure, an operation o_i is scheduled into control step s_j, if and only if $x_{i,j}$ is set to 1.

The size of the ILP formulation increases rapidly with the number of control steps. For example, if we increase the number of control steps by 1, we will have n additional x variables in the inequalities since we have to consider an extra control step for every operation. Moreover, the number of inequalities due to Condition 7.3 will increase by an amount that depends on the structure of the given DFG. Since the execution time of the algorithm grows rapidly with the number of variables or the number of inequalities, in practice the ILP approach is applicable only to very small problems.

Since the ILP method is impractical for large descriptions, heuristic methods that run efficiently at the expense of the design optimality have been developed. The improved efficiency of the heuristic methods is obtained by eliminating the backtracking in the ILP method. Backtracking can be eliminated by scheduling the operations one at a time, based on a criterion that produces the right decisions most of the time. In these heuristic approaches, the cost of the scheduled design depends largely on the selection of the next operation to be scheduled and the assignment of that operation to the best control step.

Force-Directed Heuristic Method

The force-directed scheduling (FDS) heuristic [PaKn89] is a well known heuristic for scheduling with a given timing constraint. In this section 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. The algorithm achieves its objective by uniformly distributing operations of the same type (e.g., multiplication) 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.

Like the ILP formulation, the FDS algorithm relies on both the ASAP and the ALAP scheduling algorithms to determine the range of control steps for every operation (i.e., mrange (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 step. 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 $p_j(o_i) = 1/(L_i - E_i + 1)$.

We can illustrate these probability calculations using the HAL example from Figure 7.1, with the ASAP (E_i) and ALAP (L_i) values from Figure 7.2 used in the calculations. The operation probabilities for the example are shown in Figure 7.4(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 operation getting scheduled 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 $p_1(o_3) = p_2(o_3) = 0.5$.

A set of probability distribution graphs (i.e., 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 $EOC_{j,k} = c_k * \sum_{i,s_j \in mrange(o_i)} p_j(o_i)$, 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 7.4(b) is a bar graph of expected operation costs for the multiplication operation in each control step. We can calculate the value for $EOC_{1,multiplication}$ as $c_{multiplication} \times (p_1(o_1) + p_1(o_2) + p_1(o_3) + p_1(o_4))$, which is $c_k \times (1.0 + 1.0 + 0.5 + 0.33)$ or $2.83 \times c_k$. Consequently, the bar graph in state s_1 for the multiplication operation shows an EOC value of 2.83.

The bar graph in Figure 7.4(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 7.4(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

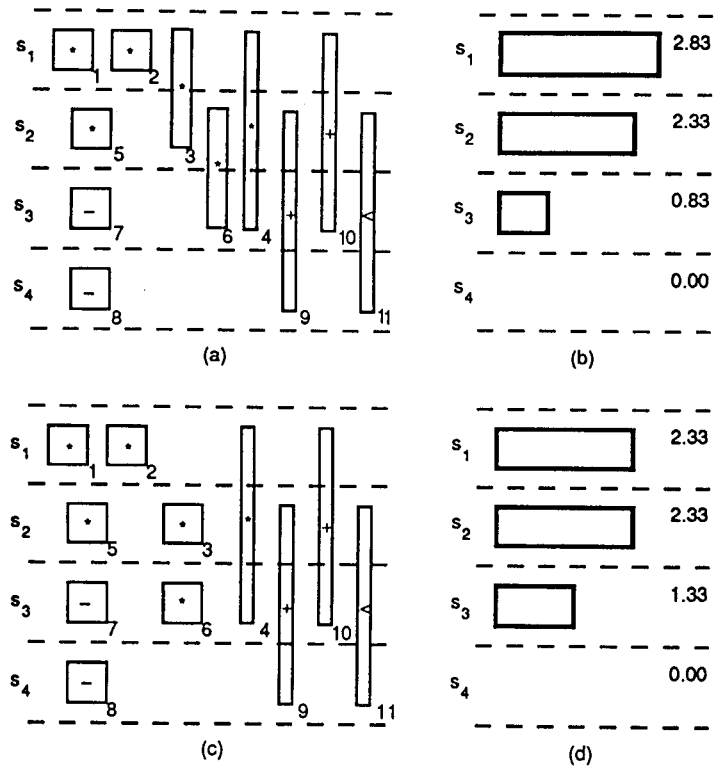


Figure 7.4: Force-directed scheduling example: (a) probability of scheduling operations into control steps, (b) operator cost for multiplications in a, (c) probability of scheduling operations into control steps after operation o_3 is scheduled to step s_2, (d) operator cost for multiplications in c.

Algorithm 7.3: Force-Directed Scheduling.

```
Call ASAP (V);
Call 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
```

for each operation type. Algorithm 7.3 (Force-Directed Scheduling) 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 the *BestOp* and *BestStep* are determined 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. The FDS the algorithm is shown in Algorithm 7.3.

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 as shown in Equation 7.6.

$$COST(S) = \sum_{1 \le k \le m} \max_{1 \le j \le s} EOC_{j,k}. \qquad (7.6)$$

This cost is calculated using the ASAP (E_i) and the 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., mrange(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.

Returning to our example, from the initial probability distribution for the multiplication operation shown in Figure 7.4(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 7.4(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. Alternatively, we can invert the strategy by pruning only one control step from an operation's possible destinations during each iteration [VAKL91]. By doing so, we effectively postpone the decision of choosing one of the many feasible control step assignments to a later stage when the feasible assignment set is smaller.

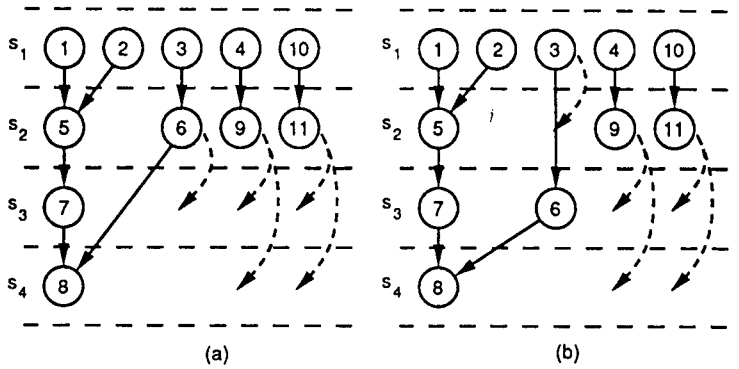


Figure 7.5: Rescheduling: (a) initial schedule where three operations are movable to five control steps (dashed arrows), (b) after operation 6 is moved and locked.

Iterative Refinement Method

We call algorithms similar to FDS "constructive" because they construct a solution 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. We can cope with this weakness by rescheduling some of the operations in the given schedule.

The essential issues in rescheduling are: the choice of a candidate for rescheduling, the rescheduling procedure and the control of the improvement process. We describe a method [PaKy91] based on the paradigm originally proposed for the graph-bisection problem by Kernighan and Lin (KL) [KeLi70]. Chapter 6 gives a detailed description of the KL algorithm.

Any scheduling algorithm can be used to compute the initial schedule. New schedules can be obtained by simply rescheduling one operation at a time. An operation can be rescheduled into an earlier or a later step, as long as it does not violate the data dependence constraints. For

...edule. For
...sult in the new schedule
...next moves are indicated by dashed
... an operation that has been moved once, (i.e.,
...e 7.5(b)) is locked and not allowed to move again. After a sequence of $m \leq n$ move and locks, all operations become locked. We can find a subsequence of $k \leq m$ move and locks that maximizes the cumulative gain, where gain is defined as the decrease in the cost of implementing the new schedule. If the gain is non-negative (i.e., the cost decreases), we change the initial schedule according to the first k moves, unlock all operations and iterate the refinement process. If no non-negative cumulative gain is attainable, the process stops.

In the iterative rescheduling algorithm (Algorithm 7.4), the variable $S_{current}$ maintains the most recently updated copy of the schedule. S_{move} and S_{work} are temporary versions of the schedule used to evaluate various rescheduling options. The function SCHEDULE_OP(S_{work}, o_i, s_j) returns a new schedule, after scheduling operation o_i into state s_j in S_{work}. The function COST(S) calculates the cost of a particular schedule. This cost can be determined from equation 7.6. The set UnlockOps is a set of nodes that remains unlocked in a particular iteration and hence can be rescheduled in future iterations. Array $O[m]$ maintains the sequence of operations that are moved in each iteration and array $S[m]$ maintains the states into which each operation from $O[m]$ is to be moved. The function MAX_CUM_GAIN scans through the array G[m] and returns the maximum possible cumulative gain. The variable MaxGainIndex stores the number of moves required to attain the maximum cumulative gain.

The *while* loop in the algorithm determines the sequence of best possible moves until all the operations become locked. The largest gain for each move is stored in the array G[m]. The two functions MAX_CUM_GAIN and MAX_CUM_GAIN_INDEX scan this array G[m] and extract the sub-sequence of moves that results in the largest cumulative gain. The schedule $S_{current}$ is changed to reflect the modifications induced by these moves.

The KL algorithm has been studied extensively and has proven to ...

Algorithm 7.4: Iterative Rescheduling.

```
repeat
  S_work = S_current; m = 0;
  UnlockOps = {All operations in S_work};
  while UnlockOps ≠ φ do
    m = m + 1; G[m] = −∞;
    for each operation o_i ∈ S_work do
      for each possible destination s_j of o_i do
        S_move = SCHEDULE_OP(S_work, o_i, s_j);
        gain = COST(S_work) - COST(S_move);
        if gain > G[m] then
          O[m] = i; S[m] = j; G[m] = gain;
        endif
      endfor
    endfor
    S_work = SCHEDULE_OP(S_work, o_O[m], s_S[m]);
    UnlockOps = UnlockOps - {o_O[m]};
  endwhile

  MaxGain = MAX_CUM_GAIN(G);
  MaxGain_index = MAX_CUM_GAIN_INDEX(G);
  if MaxGain ≥ 0 then
    for j = 1 to MaxGain_index do
      S_current = SCHEDULE_OP(S_current, o_O[j], s_S[j]);
    endfor
  endif
until MaxGain < 0.
```

useful in the physical design domain. Its usefulness for scheduling can be increased by incorporating some enhancements proposed for the graph bipartitioning problem. Two of them are outlined below.

1. The first enhancement is probabilistic in nature and exploits the fact that the quality of a result produced by the KL method depends greatly on the initial solution. Since the algorithm is computationally efficient, we can run the algorithm many times, each with a different initial solution, and then pick the best solution.

2. The second enhancement, called the look-ahead scheme [Kris84], relies on a more sophisticated strategy of move selection. Instead of just evaluating the gain of a move, the algorithm looks ahead and evaluates the present move on the basis of possible future moves. The depth of the look-ahead determines the tradeoff between the computation time and the design quality: the deeper the look ahead, the more accurate the move selection and more expensive the computation.

7.2.2 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. When total area is given as a constraint, the scheduling algorithm determines the type of functional units used in the design. The goal of such an algorithm is to produce a design with the best possible performance but still meeting the given area constraint.

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 area constraints are satisfied by ensuring that the area of all units or the floorplan area does not exceed the constraints. A unit constraint can be checked for each control step, but the total area constraint can only be checked for the whole design. 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 two resource-constrained scheduling algorithms: the list-based scheduling method and the static-list scheduling method.

List-Based Scheduling Method

List scheduling is one of the most popular methods for scheduling operations under resource constraints. Basically, it is a generalization of the ASAP scheduling technique since it produces the same result in the absence of resource constraints.

The list based scheduling algorithm 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 till all the resources get used in that state. The priority list is always sorted with respect to a priority function. Thus 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. Operations with lower priority will be deferred to the next or later control steps. 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.

Algorithm 7.5 outlines the list scheduling method. The algorithm 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 (i.e., all its predecessors are scheduled), deletes each ready node from the set V and appends it to one of the priority lists based on its operation type. The function

Algorithm 7.5: List Scheduling.

```
INSERT_READY_OPS(V, PList_t1, PList_t2, ... PList_tm);
Cstep = 0;
while((PList_t1 ≠ φ) or ... or (PList_tm ≠ φ)) do
   Cstep = Cstep + 1;
   for k = 1 to m do
      for funit = 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_tm );
endwhile
```

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.

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 till all the priority lists are empty.

We will illustrate the list scheduling process with an example (Figure 7.6). Suppose the available resources are two multipliers, one adder, one subtracter and one comparator. (Figure 7.6(c)). Each operation o_i in the DFG in Figure 7.6(a) is labeled with its mobility range (i.e., mrange(o_i)). 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 (Figure 7.6(b)) 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.

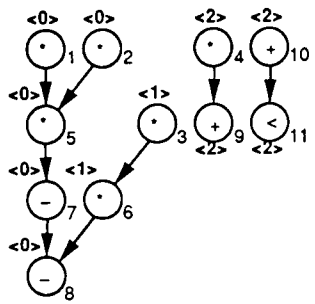
During the first iteration, when ready operations are scheduled into control step s_1, there are five ready operations: o_1, o_2, o_3, o_4 and o_{10}. Operation o_{10}, which is the only addition operation, is scheduled into control step s_1 without considering any other factors. Since only two multipliers are available, only two out of the four ready multiplications can be scheduled into control step s_1; o_1 and o_2 are chosen for scheduling since they have lower mobilities than o_3 or o_4.

After the first iteration, operations o_1, o_2 and o_{10} are scheduled into control step s_1. For the second iteration, operations o_5 and o_{11} are added to the ready list because these two operations have all their input nodes scheduled. The ready list during the second iteration consists of o_5, o_{11}, o_3 and o_4. This ready list is sorted by mobilities and the whole process is repeated again. After four iterations, all the operations are scheduled into the appropriate control steps (Figure 7.6(d)).

As we stated previously, the success of a list-scheduler depends mainly on its priority function. Mobility is a good priority function because a smaller value of mobility indicates a higher urgency for scheduling an operation since the schedule will run out of alternative control steps earlier. Mobility is just one of the many priority functions that have been proposed as a priority function for list-scheduling. 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 to 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. Yet 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.

Static-List Scheduling Method

Instead of dynamically building a priority list while scheduling every control step, we can create a single large list before starting scheduling [JMSW91]. This approach is different from the ordinary list-based



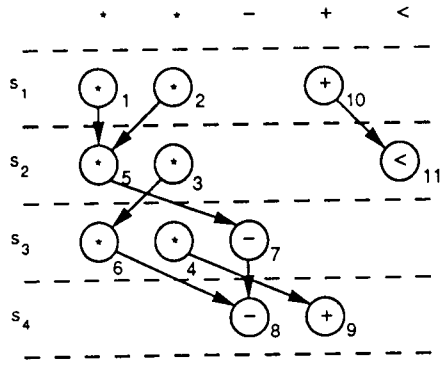
(a)

$PList_*$: 1<0>, 2<0>, 3<1>, 4<2>
$PList_+$: 10<2>
$PList_-$: NIL
$PList_<$: NIL

(b)

$Resources_*$: 2
$Resources_+$: 1
$Resources_-$: 1
$Resources_<$: 1

(c)



(d)

Figure 7.6: List scheduling: (a) a DFG with mobility labeling (inside <>), (b) the list of ready operations for control state s_1, (c) the resource constraints, (d) the scheduled DFG.

scheduling not only in the selection of candidate operations but also in the assignment of the target steps for the candidates. The algorithm sorts all operations using the ALAP labels in ascending order as the primary key and the ASAP labels in descending order as the secondary key. If both the keys have the same value, an arbitrary ordering is used. This sorted list is maintained as the priority list that determines the order in which operations are scheduled.

Figure 7.7(a) shows the HAL dataflow graph, and Figure 7.7(b) shows the ASAP and ALAP values for each node and the complete priority list for the example. Operations o_8, o_9 and o_{11} have the lowest ALAP value of 1 and so fall into the first three slots in the priority list. Among these three nodes, o_8 has a higher ASAP labeling than both o_9 and o_{11}. Therefore it falls in the lowest priority slot. Operations o_9 and o_{11} have the same ASAP and ALAP values and tie for the second and third slots on the list. Operation o_9 is selected randomly to precede operation o_{11}. The rest of the list is formed in a similar manner.

Once the priority list is created, the operations are scheduled sequentially starting with the last operation (i.e., the highest priority) in the list. An operation is scheduled as early as possible, subject only to available resources and operator dependencies. Thus, operation o_2 is the first one to be scheduled. It is scheduled into the first control step, since both the multipliers are available. The next operation to be scheduled is o_1 and it is assigned to the second multiplier in the same control step. Operation o_3 cannot be scheduled into state s_1 since there are no available multipliers, so it is scheduled into state s_2. Operation o_5 cannot be scheduled into state s_1 because its input data is available only in state s_2, so it is scheduled into state s_2. Although operation o_{10} has a lower priority compared to operations o_3 and o_5, it is scheduled into control step s_1 because an adder is available in state s_1 and it does not depend on any other previously scheduled operations. The final schedule for the example is shown in Figure 7.7(d).

7.3 Scheduling with Relaxed Assumptions

In Section 7.2 we outlined some basic scheduling algorithms that used a set of simplifying assumptions. We now extend the scheduling algo-

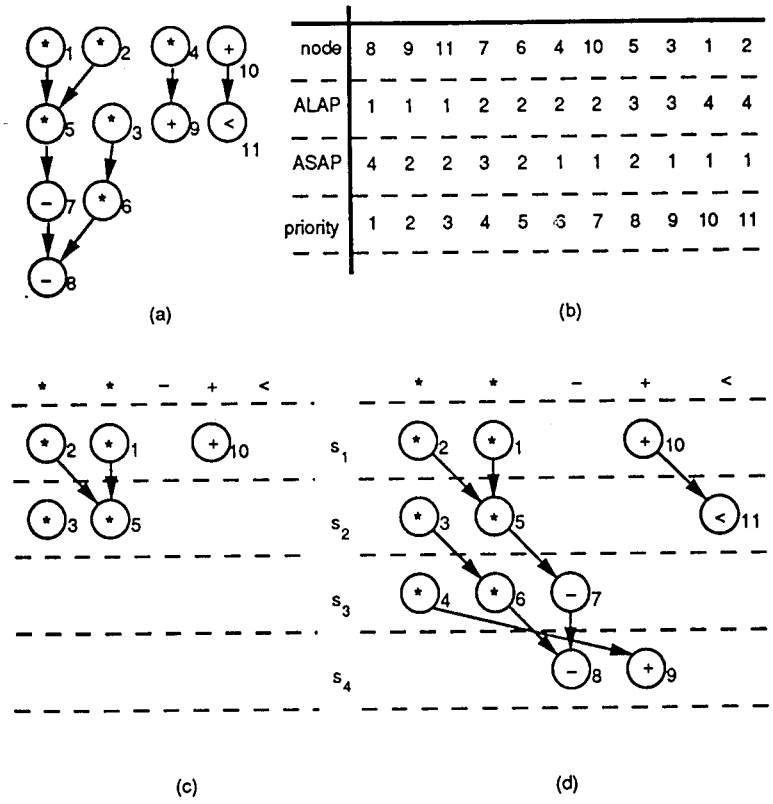


Figure 7.7: Static-list scheduling: (a) a DFG, (b) priority list, (c) partial schedule of five nodes, (d) the final schedule.

rithms to handle more realistic design models, such as functional units with varying execution times, multi-function units, and behavioral descriptions that are not limited to straight-line code.

7.3.1 Functional Units with Varying Delays

Real functional units have different propagation delays based on their design. For example, a 32-bit floating-point multiplier is much slower than a fixed-point adder of the same word length. Therefore we should not assume that every operation finishes in one control step. This assumption would lead to a clock cycle that is unusually lengthened to accommodate the slowest unit in the design, as shown in Figure 7.8(a), where the multiplier takes approximately three times longer than an adder or a subtracter. As a result, units with delay shorter than the clock cycle remain idle during part of the cycle and so are underutilized. Allowing functional units with arbitrary delays will improve the utilization of the functional units. Since functional units with arbitrary delays may no longer all execute within a single clock cycle, we have to generalize the execution model of operation to include multicycling, chaining and pipelining.

If the clock cycle period is shortened to allow the fast operations to execute in one clock cycle, then the slower operations take multiple clock cycles to complete execution. These slower operations, called multicycle operations, have to be scheduled across two or more control steps (Figure 7.8(b)). This increases the utilization of the faster functional units since two or more operations can be scheduled on them during a single operation on the multicycle units. However, input latches are needed in front of the multicycle functional units to hold its operands until the result is available a number of steps later. Also, a shorter clock cycle results in a larger number of control steps to execute the CDFG, which in turn increases the size of the control logic.

Another method of increasing the functional unit utilization is to allow two or more operations to be performed serially within one step, i.e., chaining. We can feed the result of one functional unit directly to the input of some other functional units (Figure 7.8(c)). This method requires direct connections between functional units in addition to the connections between the functional and storage units.

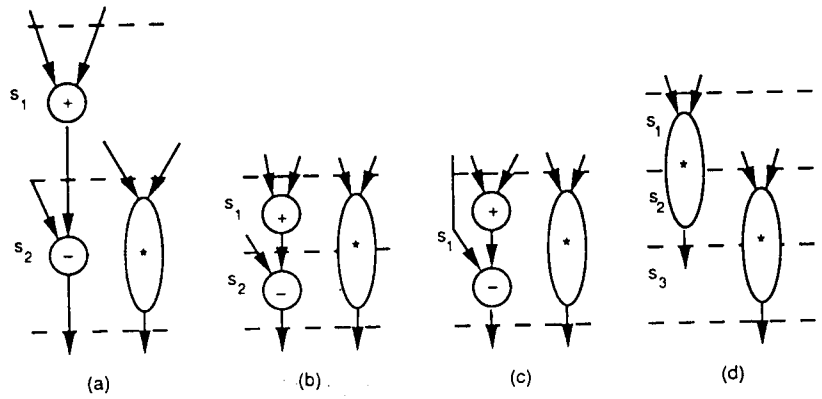


Figure 7.8: Scheduling with arbitrary-delay units: (a) schedule in which every operation is executed in one step, (b) schedule with a 2-cycle multiplier, (c) schedule with a chained adder and subtracter, (d) schedule with a 2-stage pipelined multiplier.

As described in Chapter 2, pipelining is a simple yet effective technique for increasing parallelism. When a pipelined functional unit is used, the scheduler has to calculate the resource requirements differently. For example, in Figure 7.8(d), the two multiplications can share the same two-stage pipelined multiplier despite the fact that the two operations are executing concurrently. This sharing is possible because each multiplication operation uses a different stage of the pipelined multiplier. Therefore, only one pipelined multiplier is needed instead of two non-pipelined multipliers.

7.3.2 Multi-functional Units

In the previous section, we extended the scheduling problem to functional units with non-uniform propagation delays and different number of pipeline stages. However, we have still assumed that each functional unit performs exactly one operation. This assumption is unrealistic since in practice multi-function units cost less than a set of uni-functional units

that perform the same operations. For example, although an adder-subtracter component costs twenty percent more than an adder or subtracter separately, it is considerably cheaper than using one adder and one subtracter. Therefore, it is reasonable to expect that designers will use as many multi-function units as possible. The scheduling algorithms have to be reformulated to handle these multi-function units.

We have also been assuming single physical implementations for each functional unit. In reality, the component library has multiple implementations of the same component, with each implementation having a different area/delay characteristic. For example an addition can be done either quickly with a large (hence, costly) carry-look-ahead adder or slowly with a small (hence, inexpensive) ripple-carry adder. Given a library with multiple implementations of the same unit, the scheduling algorithms must simultaneously perform two important tasks : operation scheduling, in which operations are assigned to control steps, and component selection, in which the algorithm selects the most efficient implementation of the functional unit for each operation in the library.

We can exploit the range of component implementations in a library by using a technology-based scheduling algorithm [RaGa91]. Given a library that has multiple implementations of functional units, the main goal of the algorithm is to implement the design within the specified time constraint, with minimal costs. For each operation in the flowgraph an efficient component is selected so that the operations on the critical path are implemented with faster components, and the operations not on the critical path are implemented with slower components. Simultaneously, the algorithm ensures that the operations are scheduled into appropriate control steps that enable sharing the functional units across the various states.

7.3.3 Realistic Design Descriptions

In addition to blocks of straight line code, behavioral descriptions generally contain both conditional and loop constructs. Thus, a realistic scheduling algorithm must be able to deal with both of these modeling constructs.

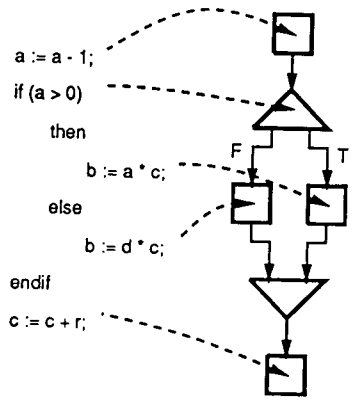


Figure 7.9: CDFG for a behavioral description with a conditional construct.

Conditional Constructs

A conditional construct is analogous to an "if" or "case" statement in programming languages. It results in several branches that are mutually exclusive. During one execution path of the design, only one branch gets executed based on the outcome of an evaluated condition. Figure 7.9 shows a segment of a behavioral description that contains a conditional *if* statement. It is not possible to represent this behavior with a pure data-flow graph (see Chapter 5). Instead, a flow graph with control and data dependencies (e.g., a CDFG) is needed. In this example, the control flow determines the execution of the four DFGs represented by boxes.

An effective scheduling algorithm shares the resources among mutually exclusive operations. For example, only one of the two multiplications in Figure 7.9 gets executed during any execution instance of the given behavior. Therefore, the scheduling algorithm can schedule both the multiplication operations in the same control step, even if only one multiplier is available in each step.

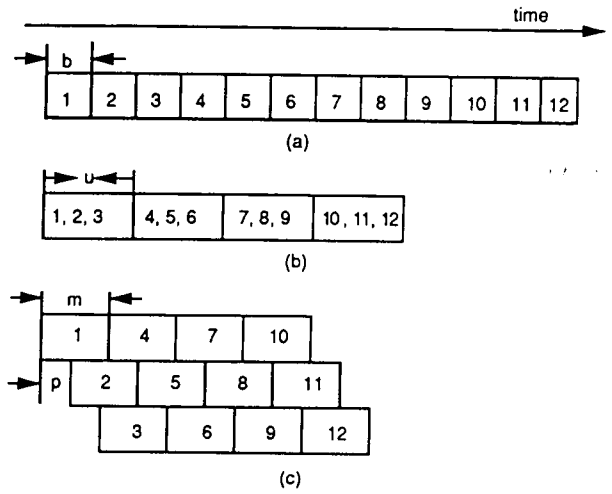


Figure 7.10: Loop scheduling: (a) sequential execution, (b) partial loop unrolling, (c) loop folding.

Loop Constructs

A behavioral description sometimes contains loop constructs. For example, in a filter for digital signal processing, a set of actions is repeatedly executed for every sample of the input data stream. This repeated action is modeled using a loop construct. In these descriptions, optimization of the loop body improves the performance of the design.

Loop constructs exhibit potential parallelism between different iterations of the loop. This parallelism can be exploited by executing several iterations of the same loop concurrently. Scheduling a loop body differs from scheduling a pure data flow graph, in that we need to consider potential parallelism across different loop iterations.

We use Figure 7.10 to illustrate three different ways of scheduling a loop. Suppose the loop is finite and consists of n iterations (n = 12 in Figure 7.10). The first and most simple approach assumes sequential execution of the loop and schedules each loop iteration into b control

steps. If the twelve iterations are executed sequentially, the total execution time is $12b$ control steps as shown in Figure 7.10(a). The other two approaches exploit parallelism across loop iterations.

The second technique is called loop unrolling since a certain number of loop iterations are unrolled. This action results in a loop with a larger loop body but with fewer iterations. The larger loop body provides a greater flexibility for compacting the schedule. In Figure 7.10(b) three iterations are unrolled into a super-iteration resulting in four super-iterations. Each super-iteration is scheduled over u control steps. Therefore, if $u < 3b$, the total execution time is less than $12b$ control steps.

The third method exploits intraloop parallelism using loop folding, by which successive iterations of the loop are overlapped in a pipelined fashion. Figure 7.10(c) shows a loop with a body that takes m control steps to execute. In loop folding, we initiate a new iteration every p control steps, where $p < m$; that is, we overlap successive iterations. The total execution time is $m+(n-1)\times p$ control steps (Figure 7.10(c)). Loop unrolling is applicable only when the loop count is known in advance, but loop folding is applicable to both fixed and unbounded loops.

To illustrate these three methods of loop execution, we introduce an example DFG that contains multiple iterations. Figure 7.11(a) shows the DFG of a loop body [Cytr84] that contains 17 identical operations, with each taking one control step to execute. The dashed arcs indicate data dependencies across loop boundaries. For instance, the arc from node P to node J indicates that the result produced by operation P during one iteration is used by operation J of the next iteration. We schedule this DFG using the three loop scheduling techniques. In order to evaluate the quality of the resulting schedules, we use two performance measures: functional unit utilization, i.e., the percentage of states in which the functional units are used to perform some operation, and control cost, which is measured by the number of unique control words in the control unit.

In Figure 7.11(b) we show a schedule with six control steps using three functional units. Since a datapath with three functional units needs at least six control steps to execute seventeen operations and the length of the critical path of the DFG is six, this schedule is optimal. The functional unit utilization rate is $17/(3 \times 6) = 17/18$. The control cost is six words assuming that one word is needed in the control unit

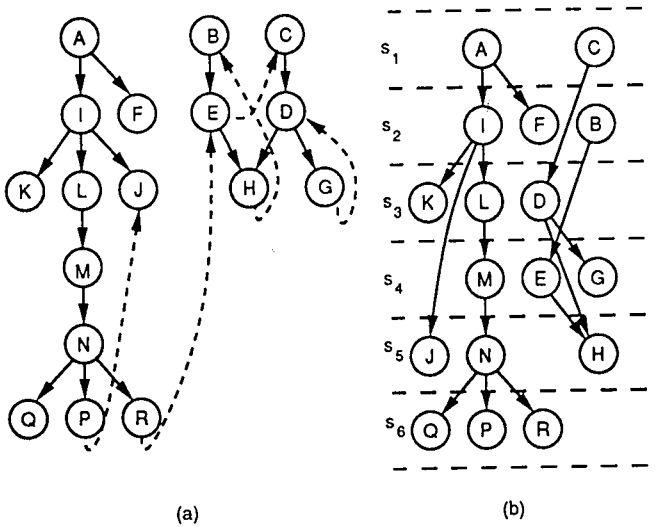


Figure 7.11: Standard loop scheduling: (a) DFG with dependencies across iterations, (b) sequential schedule using three functional units.

for each scheduled control step. We can increase the performance of this design only by exploiting parallelism across loop boundaries.

Figure 7.12(a) shows a schedule of the example in Figure 7.11 using loop unrolling. Two iterations, i.e., two copies of the DFG, are scheduled into nine control steps using four functional units. The hardware utilization rate remains the same: $(17\times2)/(4\times9) = 17/18$. However, the average time required to execute an iteration is reduced from 6 to 9/2 = 4.5 control steps. The control cost increases from six to nine control words.

Figure 7.12(b) shows a schedule of the same example using loop folding. Successive iterations begin their execution three control steps apart. Six functional units are used and their utilization rate is $(5+6+6)/(6\times3)$ = 17/18, which is same as in the two previous schedules. But the average time for executing an iteration is approximately three control steps

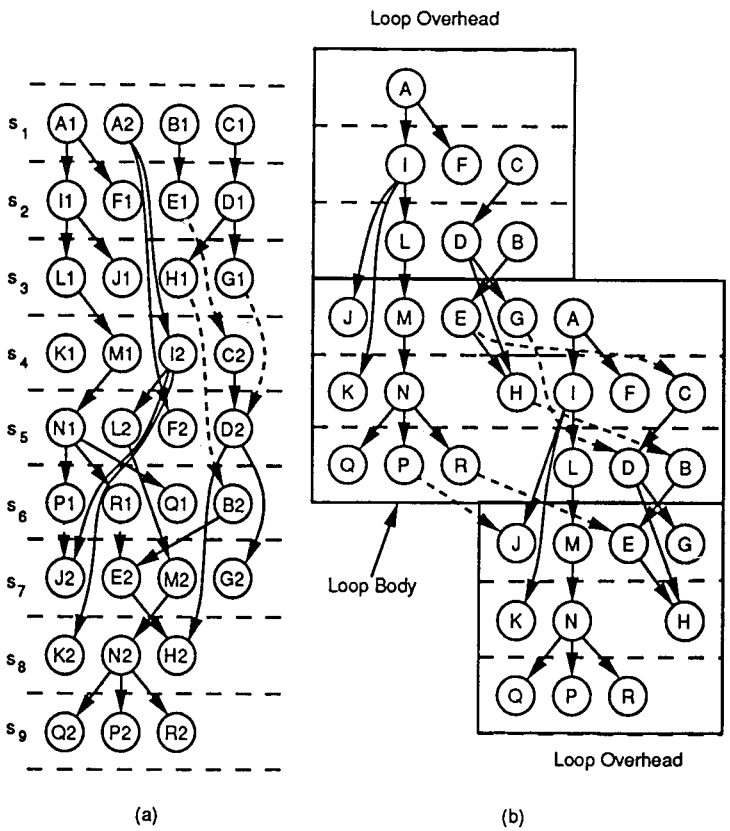


Figure 7.12: Scheduling with iteration overlapping: (a) loop unrolling, (b) loop folding.

when the number of iterations is very large. The control cost is nine (e.g., three each for the head, body and tail of the loop).

In the context of high-level synthesis, we must consider the change in control costs when performing these optimizations. Both loop unrolling and loop folding increase the control cost. When unrolling a loop, the scheduler can explore more parallelism by expanding more iterations, but the control cost increases as well. Therefore, it performs tradeoffs between the execution speed and the control cost. When folding a loop, there are two important constituents to the control cost: the loop overhead and the loop body. The loop overhead constituent is proportional to the execution time of an iteration. The loop body constituent is proportional to the product of the execution time of an iteration and the number of control steps between successive iterations, i.e., the latency.

Scheduling an unrolled loop is fairly easy since the basic scheduling algorithms described in Section 7.2 can be applied to the unrolled DFG. Problem size is the only concern. Some algorithms that have high computational complexity (e.g., the ILP method) will perform poorly on deep unrolling of loops. On the other hand, scheduling a folded loop is more complicated. For a fixed latency k, we can extend the list-scheduling algorithm to schedule operations under a resource constraint. In a control-step q where $q > k$, the operations from various iterations of the loop are executed concurrently. Consequently, the resources required to execute different operations from all the concurrent iterations would have to be provided.

7.4 Other Scheduling Formulations

In addition to the basic scheduling algorithms described in Section 7.2, a variety of other techniques can be adopted. We discuss three approaches: simulated annealing, path-based scheduling and DFG restructuring. We will use the simplifying assumptions discussed in Section 7.2 in order to explain these algorithms clearly.

7.4.1 Simulated Annealing

Simulated annealing (described in Algorithm 6.4) can also be used to schedule a dataflow graph [DeNe89]. The schedules are represented using a two-dimensional table where one dimension corresponds to the control steps and the other dimension corresponds to the the available functional units. We view scheduling as a placement problem in which operations are placed in table entries. By placing the operation nodes on the table, we define the control step in which the operation is executed and the functional unit that will carry out the operation. Since a functional unit can perform only a single operation in a given control step, no table entry can be occupied by more than one operation. For example, Figure 7.13(a) shows a schedule of five operations into three control steps using three ALUs.

Beginning with an initial schedule, the annealing procedure improves upon the schedule by iteratively modifying it and evaluating the modification. In [DeNe89] the quality measure for a given schedule is computed as the weighted sum of the number of functional units, registers, busses and the number of control steps.

The simulated annealing algorithm modifies a schedule by displacing an arbitrary operation from one table entry to another or by swapping two arbitrary operations. For example, Figure 7.13(b) shows the schedule resulting from swapping two operations of the schedule depicted in Figure 7.13(a): ($v_1 = v_2 + v_3$) and ($v_4 = v_2 * v_3$). Figure 7.13(c) shows the schedule resulting from displacing an operation (i.e., $v_2 = v_4 * v_5$). A modification is accepted for the next iteration if it results in a better schedule. If the modification does not result in a better schedule, its acceptance is based on a randomized function of the quality improvement and the annealing temperature.

Although the simulated annealing approach is robust and easy to implement, it suffers from long computation times and requires complicated tuning of the annealing parameters.

7.4.2 Path-Based Scheduling

Path-based scheduling algorithm [Camp91] minimizes the number of con-

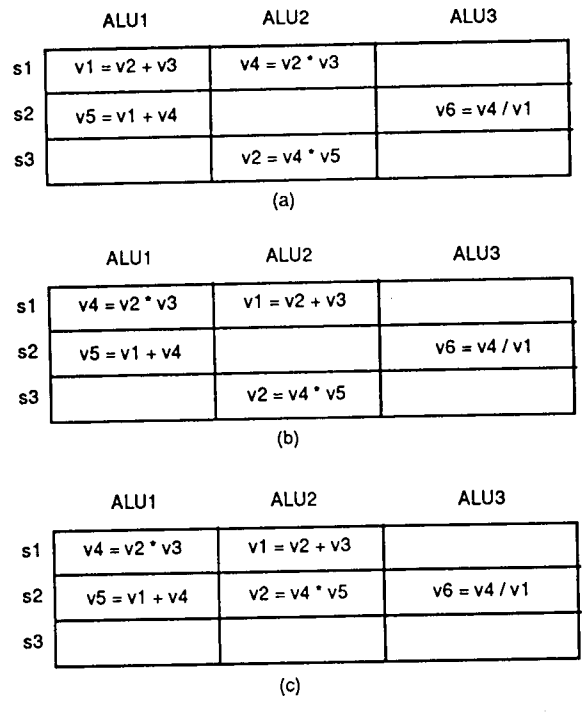


Figure 7.13: Scheduling using simulated annealing: (a) an initial schedule, (b) after swapping two operations, (c) after displacing an operation.

tracts all possible execution paths from a given CDFG and schedules them independently. The schedules for the different paths are then combined to generate the final schedule for the design.

Consider the CDFG shown in Figure 7.14(a). In order to extract the execution paths, the CDFG is made acyclic by deleting all feedback edges of loops. A path starts with either the first node of the CDFG or the first node of a loop, and ends at a node with no successors. Figure 7.14(b) shows a path of the original CDFG in Figure 7.14(a).

We then partition each path in the CDFG into control steps in such a way that:

(a) no variable is assigned more than once in each control step,

(b) no I/O port is accessed more than once in each control step,

(c) no functional unit is used more than once in each control step,

(d) the total delay of operations in each control step is not greater than the given control-step length,

(e) all designer imposed constraints for scheduling particular operations in different control steps are satisfied.

In order to satisfy any of the above conditions, the path-based scheduling algorithm generates constraints between two nodes that must be scheduled into two different control steps. Such a constraint is represented by an interval starting and ending at the two conflicting nodes. For example, operations indicated by the nodes 5 and 9 in Figure 7.14(b), cannot be in the same control step since both the operations assign values to the same variable. This constraint is represented by the interval i_1 in Figure 7.14(b). Similarly operations indicated by nodes 3 and 9 use the same adder and must occur in separate control steps. This constraint is represented by interval i_2 in Figure 7.14(b). If two constraints have overlapping intervals, both the constraints can be simultaneously satisfied by introducing a control step between any two nodes that belong to the overlapping part of the two intervals. For example, introducing a new control step between nodes 5 and 9 will satisfy both constraints i_1 and i_2.

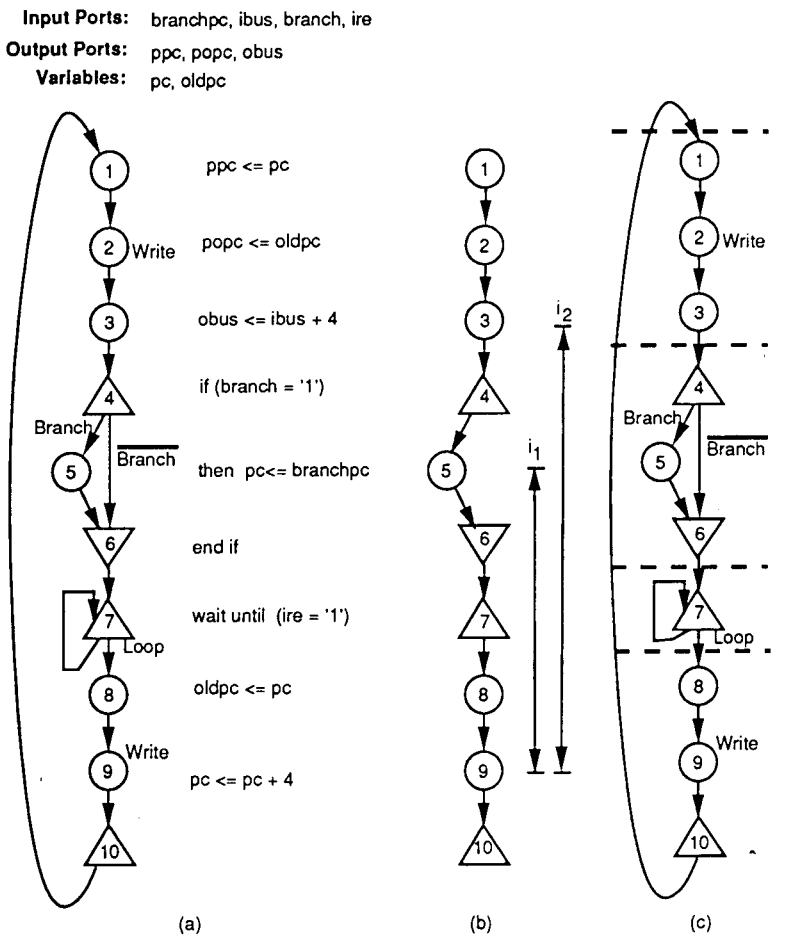


Figure 7.14: Path-based scheduling: (a) an example CDFG, (b) a path in the CDFG with constraint intervals, (c) scheduled CDFG.

In order to schedule the entire path efficiently, it is important to introduce the minimum number of control steps. The problem of introducing the minimum number of control steps that satisfy all constraints can be transformed into a clique-partitioning problem for a graph. A graph is constructed in which each node represents a constraint interval. An edge exists between two nodes if the two corresponding intervals overlap with each other.

The clique-partitioning solution indicates a set of minimum non-overlapping intervals for the given path. Similar intervals are obtained for each path in the graph. The clique partitioning technique is used again to combine the intervals generated from different paths. The results of this clique partitioning represent a final set of intervals for the entire CDFG. Introducing a new control step between any two nodes in this final set of intervals generates an unique schedule for the original CDFG. Figure 7.14(c) shows the final schedule for the example in Figure 7.14(a).

7.4.3 DFG Restructuring

The performance (in terms of the number of control steps) attainable by the scheduling techniques described so far is bounded by the length of the critical path of the DFG. In Figure 7.15(a), the critical-path length is four (i.e., operations 1, 2, 3 and 4). Therefore, none of the techniques we described can do better than scheduling into four control steps. However, we could lower this bound by modifying the structure of the DFG. Two techniques, tree-height reduction [NiPo91] and redundant-operation introduction [LoPa91], can modify the structure of the DFG while preserving its behavior.

Tree-height reduction, also discussed in Chapter 5, restructures a DFG represented as a tree by exploiting the associativity of some operations, e.g., addition. For example, in Figure 7.15(a), the DFG, which represents the computation (((a + b) + c) + d) + (e + f), has a critical-path length of four; the DFG in Figure 7.15(b), which represents the computation ((a + b) + c) + (d + (e + f)), has a critical-path length of three. Both DFGs have the same behavior since they compute $a + b + c + d + e + f$. However, the second DFG has a shorter tree height, resulting in a shorter critical path, and therefore a possibly

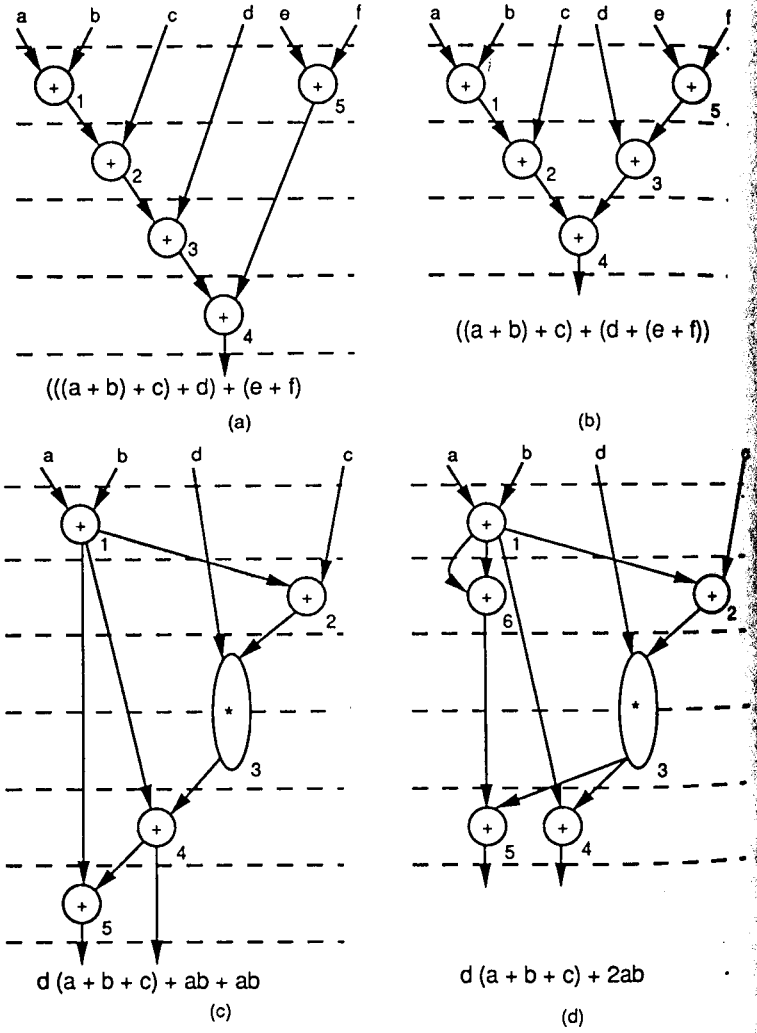


Figure 7.15: DFG restructuring: (a) DFG1, (b) DFG1 after tree-height reduction, (c) DFG2, (d) DFG2 after redundant operation insertion.

better schedule.

Redundant-operation insertion also tries to decrease the critical-path length by introducing additional operators to offload computations on the critical path. For example, Figure 7.15(c) shows a DFG with a critical-path length of six (operation 3 takes two control steps). After introducing a new operation 6, the last operation, 5, on the critical path can be scheduled earlier. Therefore, the critical-path length is reduced from six to five (Figure 7.15(d)).

7.5 Summary and Future Directions

In this chapter, we described several algorithms for scheduling operations into control steps. We first discussed two types of scheduling problems: time-constrained and resource-constrained scheduling. The integer linear programming approach solves the time-constrained problem optimally but suffers from long execution times. On the other hand, the force-directed heuristic produces schedules quickly, but the optimality of the solution cannot be guaranteed. The iterative refinement approach improves the design quality of an initial schedule generated by any scheduling algorithm. We described list scheduling with various priority functions for the resource-constrained formulation of the scheduling problem. We initially used restrictive models for input descriptions and target architectures to explain basic algorithms. We later discussed realistic extensions to the basic scheduling formulations. Finally, we discussed techniques for improving performance by exploiting parallelism beyond the loop boundaries and by restructuring input descriptions.

In scheduling, work is needed on utilizing more realistic libraries, target architectures and cost functions. Given a library of functional units, the scheduling algorithm should be able to select the functional units to optimize both the schedule and the cost. Thus, 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.

Furthermore, the cost functions used in scheduling must be realistic. Simple quality measures, such as the number of operators or the number of RT components will not be sufficient. The cost functions must include layout parameters, particularly information about possible floorplans and routing delays for performance-driven scheduling.

Many scheduling algorithms are intended for straight-line code. Few scheduling algorithms exist for scheduling arbitrary descriptions with conditional and loop constructs. Techniques used in optimizing compilers, such as loop-pipelining and tree-height reduction, can be integrated into scheduling for datapath design. Scheduling algorithms must also be extended to include arrays and other data structures in the input description as well as memories with multiclock access times in the target architecture.

Scheduling algorithms should be expanded to incorporate different target architectures, for example a RISC architecture with a large register file and a few functional units, for which minimization of load-and-store instructions from the main memory is the primary goal. Such an architecture could be extended to very large instruction word (VLIW) architectures, in which several RISC datapaths execute in parallel. Similarly, we need scheduling algorithms for other specific architectures, such as those found in DSP applications.

7.6 Exercises

1. Write the constraint inequalities for scheduling the DFG shown in Figure 7.16 into four control steps.

2. Derive a formula for the number of inequalities in the ILP formulation for the DFG in Figure 7.16 given a time constraint of five control steps. Assume that there are two types of functional units: a multiplier for multiplication and an ALU for all other operations.

3. Extend the ILP formulation for multicycled and chained operations.

4. Extend the ILP formulation to minimize a linear cost function of storage, bus and functional units. Estimate the growth in the size of your formulation as compared to the original formulation in Section 7.2.1.

5. **Extend the ILP formulation to handle resource constrained scheduling and module selection simultaneously. Hint: For each type of operation, the component library can contain a number of functional units with different speed/cost attributes. Moreover, the units may be multi-functional.

6. Schedule the DFG in Figure 7.16 into five control steps using the FDS algorithm.

7. Demonstrate that the FDS algorithm can generate non-optimal schedules by using an example with less than ten operations.

8. Calculate the operation costs of the read-data transfers for the DFG in Figure 7.16 for a five control-step schedule.

9. *Extend the concept of balancing operation cost to that of balancing the data-transfer cost, resulting in a high utilization of the bus units. Hint: Read [PaKn89].

10. Improve the effectiveness of the iterative rescheduling method by implementing a look-ahead scheme. Write a pseudo-code description of your algorithm.

11. Given one multiplier and two ALUs, schedule the DFG shown in Figure 7.16 using:
(a) static-list scheduling, and
(b) list scheduling based on the critical-path length.
Can you decrease the number of control steps with two multipliers and two ALUs?

12. Is there a difference between static-list scheduling and list scheduling that uses the same priority function as static-list scheduling?

13. Suppose the DFG in Figure 7.16 is to be executed endlessly where variables r and s carry the data dependence across iterations. Fold the loop using two multipliers and two ALUs. Calculate the average execution time per iteration, the functional unit utilization rate and the control cost.

14. Fold the loop in Figure 7.11(a) using no more than five functional units. Calculate the average execution time per iteration, the functional unit utilization rate and the control cost.

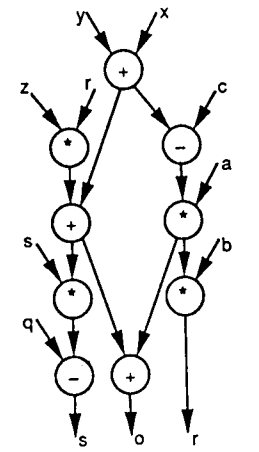


Figure 7.16: A sample DFG.

15. Can you fold the loop in Figure 7.11 so that a new iteration can start its execution every two control steps? Is there a lower bound on the achievable latency (i.e., initiation interval) when folding a loop?

16. Under what conditions can a loop be folded to fully (i.e., 100%) to utilize the functional units?

17. Derive a formula for the control cost in terms of the execution time of an iteration and the number of control steps between the initiation of successive iterations.

18. **Design a folding algorithm for a loop that contains branch constructs. Hint: Your algorithm should be able to explore the parallelism achievable by moving a subset of the branch's subconstructs across the loop boundary.

19. Extract all possible execution paths for the CDFG shown in Figure 5.3.

20. Define a set of rules for adding operations to a DFG in order to increase the degree of parallelism and decrease the number of control