Introduction to the Scheduling Problem

Robert A. Walker

Samit Chaudhuri
Rensselaer Polytechnic Institute

Scheduling—a central task in high-level synthesis—involves determining the execution order of operations in a behavioral description. After introducing the scheduling problem, this tutorial describes four scheduling algorithms commonly used to solve it.

High-level synthesis (sometimes called behavioral synthesis) is the design task of mapping an abstract behavioral description of a digital system onto a register-transfer-level design to implement that behavior. Introduced in the first article in this series by Daniel Gajski and Loganath Ramachandran,[1] high-level synthesis can greatly improve designer productivity and design-space exploration. As that introductory article defined them, the three central synthesis tasks in a typical high-level synthesis system are

- *scheduling*—determining the sequence in which operations execute to produce a control step schedule that specifies the operations executing in each control step, or state
- *allocation*—setting aside the appropriate number of functional, storage, and interconnection units
- *binding*—assigning operations to functional units and values to storage units, and interconnecting those components to form a complete data path.

In most high-level synthesis systems, scheduling and functional unit allocation occur simultaneously, followed by the remaining allocation tasks and binding. This tutorial discusses the scheduling problem, and the next article in this series will discuss the remaining allocation problems and the binding problem.

Basic scheduling problems

Early on, a typical high-level synthesis system converts the input behavioral description of the desired digital system, written in a hardware description language such as VHDL or Verilog, into a control/data-flow graph (CDFG). In the CDFG, nodes represent operations in the behavioral description, such as additions and multiplications. Edges represent values—inputs to the expression, temporary results, and the output of the expression. In more complex behaviors, the CDFG can also represent conditional branches, loops, and so forth—hence the name control/data-flow graph.

Consider the arithmetic expression $G=AB+CD+EF$. This expression might be part of a larger description—such as a digital signal processor description—but for simplicity, we consider only that one expression here. We can parse this expression to build a CDFG, shown in Figure 1, that internally represents this behavior in a high-level synthesis system. Scheduling, then, determines the order of execution for these operations—scheduling each operation into an appropriate control step.

Basic concepts. Suppose we schedule the CDFG of the arithmetic expression $G=AB+CD+EF$, as shown in Figure 1. If we begin by scheduling operation 1 into control step 1, we can also schedule operation 2 into

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

that same control step, since the two operations do not depend on each other. However, we cannot schedule operation 3 into control step 1: Operation 3 depends upon the results of operations 1 and 2, which are not available until the end of control step 1. Thus operation 3 must delay until control step 2 or later (let's assume that it's scheduled into control step 2). As we did with operation 2, we can schedule operation 4 into control step 1 as well, since it does not depend upon the result of either operation 1 or 2. Finally, we must schedule operation 5 into control step 3 or later, because that operation depends upon the result of operation 3.

Let O be the set of all operations in the CDFG. If operation $o_j \in O$ uses the result of operation $o_i \in O$, operation o_i must execute before o_j can begin. Thus there is a data dependency between the two operations. We would also say that o_i is an immediate predecessor of o_j, and o_j is an immediate successor of o_i. We represent this data dependency during the synthesis process as a precedence constraint between the two operations, which the control step schedule must satisfy.

In generating the control step schedule for a particular CDFG, we may generate a feasible schedule (any "legal" schedule), or we may want to find one that is optimal with respect to some objective function (for example, the schedule with the smallest functional unit area). In this latter case, we need to know the type of the functional unit allocated to each operation. With that information, we can compute the overall functional unit area as the sum of the areas of the maximum number of functional units of each type used in any one control step.

For a given functional unit in a particular module library, the functional unit type indicates its functionality (such as addition, multiplication, or addition/multiplication). Let K be the set of available types, and a_k and m_k be the area and number of functional units of type $k \in$ K

For a given operation, a type function $\tau: O \rightarrow K$ determines the type of operation, where $\tau(i) = k$ means that operation $o_i \in O$ executes on a type $k \in K$ functional unit. We call the problem of determining this type function, and thus the type of functional unit on which each operation will execute, the type-mapping problem. Since we need to know the execution delay of each operation to solve the scheduling problem, we must solve this type-mapping problem before (or simultaneously with) the scheduling problem. In this tutorial, we assume that the type mapping for each operation is known prior to scheduling.

These concepts let us define the unconstrained scheduling (UCS) problem—the basic scheduling problem in high-level synthesis:

Given: a set O of operations; a set K of functional unit types; a type function $\tau: O \rightarrow K$; and a partial order on O determined by the precedence constraints.

Find: a feasible (or optimal) schedule for O that obeys the precedence constraints.

Time-constrained scheduling. Although the UCS problem captures the basic elements of the scheduling problem in high-level synthesis, particular design goals may in practice require additional constraints. For example, we could limit the design's execution time by constraining the overall length of the control step schedule. This process adds a time constraint, or deadline, on the overall schedule length that the control step schedule must satisfy.

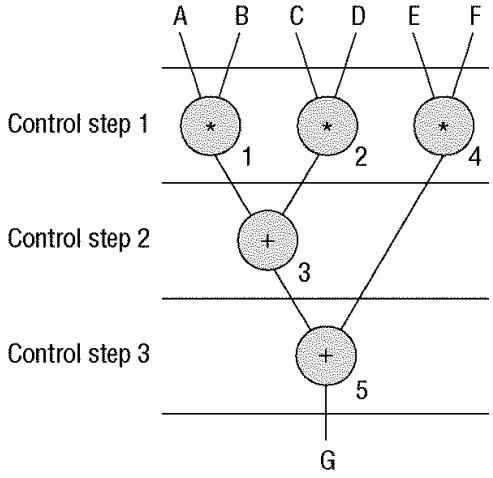

Figure 1. Schedule for the expression *G=AB+CD+EF*.

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

By adding this time constraint to the UCS problem, we can define the time-constrained scheduling (TCS) problem as follows:

Given: a set O of operations; a set K of functional unit types; a type function $\tau:O\rightarrow K$; a partial order on O determined by the precedence constraints; and a time constraint (deadline) D on the overall schedule length.

Find: a feasible (or optimal) schedule for O that obeys the precedence constraints and that meets the deadline D.

Further, Figure 1 showed that imposing an overall time constraint on the schedule may force some operations into specific control steps. For example, if we constrain the schedule to a total length of three control steps, operations 1 and 2 must be scheduled into control step 1, operation 3 into control step 2, and 5 into 3. Since we have no freedom in scheduling these operations (without violating the time constraint), we say they are on the critical path.

In general, there is a continuous range S_i of control steps, called the schedule interval, over which we can schedule an operation o_i. The length of this interval is the mobility of the operation. In Figure 1, the schedule interval for operations 1 and 2 is [1,1], and their mobility is 1; the schedule interval for operation 3 is [2,2], and its mobility is also 1; and the schedule interval for operation 4 is [1,2], and its mobility is 2. As we will see later, this information can be of great benefit during the scheduling process.

Resource-constrained scheduling. Another set of constraints commonly added to the UCS problem, reflecting the design goal of limiting the chip area, are constraints on the number of functional units of each type. For example, two multipliers may fit within the available chip area, while eight multipliers may not. In this case we may need to impose a resource (functional unit) constraint on the design, limiting the number of multipliers to two.

Consider the effect of adding resource constraints to the example of Figure 1. Generated without resource constraints, that schedule requires at least three multipliers and one adder. However, if we constrain the number of multipliers to one, the schedule shown in Figure 2 might result. That schedule would require substantially less functional unit area. (Notice that although this schedule uses two fewer functional units, the schedule length increases by one, illustrating the classic serial-parallel trade-off that often arises in scheduling problems when trading execution time for the number of resources.)

Adding these resource constraints to the UCS problem, we can define the resource-constrained scheduling (RCS) problem as follows:

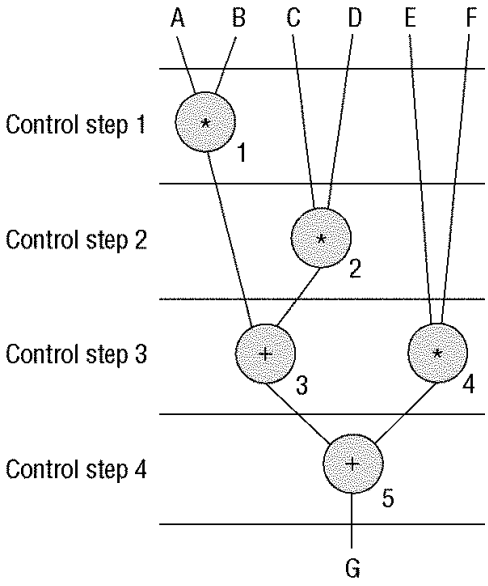

Figure 2. Schedule with a resource constraint of one multiplier.

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

Given: a set O of operations; a set K of functional unit types; a type function $\tau: O \rightarrow K$; resource constraints m_k, $1 \leq k \leq K$ for each functional unit type; and a partial order on O determined by the precedence constraints.

Find: a feasible (or optimal) schedule for O that obeys the precedence constraints and satisfies the resource constraints for each functional unit type.

Note that we can impose a separate resource constraint on each functional unit
type.

Time- and resource-constrained scheduling. Finally, we can combine the TCS and RCS problems to define the time- and resource-constrained scheduling (TRCS) problem, constraining both the overall schedule length and the number of functional units of each type:

Given: a set O of operations; a set K of functional unit types; a type function $\tau: O \rightarrow K$; resource constraints m_k, $1 \leq k \leq K$ for each functional unit type; a partial order on O determined by the precedence constraints; and a time constraint (deadline) D on the overall schedule length.

Find: a feasible (or optimal) schedule for O that obeys the precedence constraints, meets the deadline D, and satisfies the resource constraints for each functional unit type.

Advanced scheduling topics

To introduce the various scheduling problems concisely, we have made a number of overly simplistic assumptions. Now we turn to several advanced topics, all of which must be considered by any practical scheduling algorithm.

Chaining and multicycling. Until now, we have assumed that each operation type requires the same amount of time to execute, and that the control step length—the clock period—equals that execution time. In practice, different operation types may have different execution times, because the functional unit types onto which each is mapped may have different propagation delays. Thus, an overly restrictive scheduling model coupled with a poor choice for the control step length can result in poorly used functional units and an overly long schedule.

Consider the CDFG of Figure 3a, which maps the multiplication and addition operations onto a multiplier and adder with 100- and 50-ns propagation delays. (For simplicity, assume that these functional unit propagation delays include any necessary register setup times.) If we set the control step length to 100 ns, and schedule each operation into exactly one control step, the overall length of the schedule will be 200 ns, and the multiplier will be used only half of the time.

However, as Figure 3b shows, packing the two additions into a single control step will increase the multiplier usage and decrease the schedule length. These two additions are chained operations; we implement these at the register-transfer level by connecting the output of the first adder directly to the input of the second adder (that is, without the intervening register that would otherwise latch the result of the first adder at the control step boundary). In this example, chaining increases the multiplier usage and decreases the schedule length to 100 ns, but at the cost of an additional adder.

As Figure 3c shows, setting the clock length to 50 ns (the shorter propagation delay) and executing the multiplication over two control steps will also improve the schedule of Figure 3a. Such a multiplication is a multicycle operation: It must execute continuously throughout its entire execution time, and its input values must be latched throughout that period. In this example, multicycling decreases the schedule length to 100 ns, just like chaining, but without the cost of another adder. However, multicycling does use twice as many control steps as chaining, which may result in a larger controller.

Control constructs. The basic scheduling problems presented just now were all defined for a single basic block—one section of straight-line code with only one entry and one exit point. Since most hardware description languages support conditionals, loops, and other control constructs, the scheduler must consider those constructs during the scheduling process. When scheduling conditional branches, the scheduler should exploit any potential parallelism by sharing functional units between mutually exclusive branches; for example, the same adder can serve in both the then and else clauses of an if statement. When scheduling loops, the scheduler should exploit any potential parallelism by loop folding—overlapping the loop executions in a pipelined fashion.

Timing constraints. The basic scheduling problems also capture a variety of constraints on the execution time of the schedule: the length of the schedule, the schedule intervals of the operations, and the precedence constraints between them. However, few digital systems work in isolation, so designers may also need to specify more detailed timing constraints on certain operations. There are minimum timing constraints, which

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

specify that one operation must execute at least a specified amount of time after another operation, and maximum timing constraints, which specify that one operation must execute no more than a specified amount of time after another operation. Most schedulers handle these timing constraints by adding additional constraint edges to the CDFG, then treating those additional edges in much the same manner as other constraints.

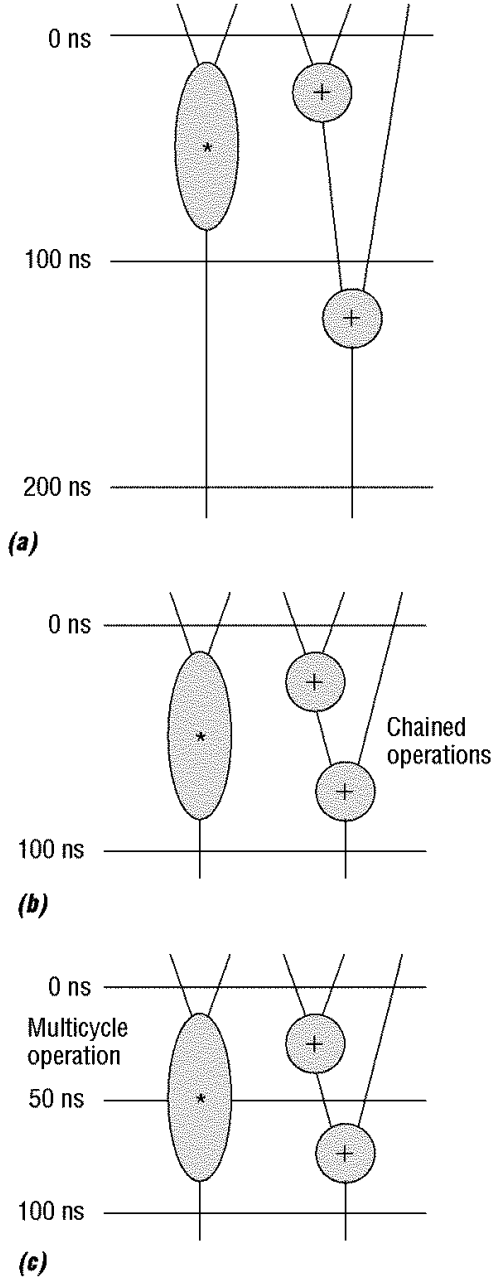

Figure 3. Chained and multicycle operations: no chaining or multicycling (a); two chained additions (b); and a multicycle multiplication (c).

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

Common scheduling algorithms

This tutorial has defined the four basic scheduling problems in high-level synthesis. To solve those problems, systems use either heuristic algorithms, which find feasible (possibly suboptimal) solutions, or exact algorithms, which find optimal solutions. Four scheduling algorithms commonly used by high-level synthesis systems are

- as-soon-as-possible/as-late-as-possible (ASAP/ALAP) scheduling
- list scheduling
- force-directed scheduling
- integer linear programming

The first three are constructive heuristic algorithms that iteratively select and schedule one operation at a time into an appropriate control step. Since these greedy strategies make a series of local decisions, selecting at each point the single "best" operation/control step pairing without backtracking or lookahead, they may miss the globally optimal solution. However, they do produce results quickly, and those results may be sufficient for practical application.

The fourth scheduling algorithm, based on solving an ILP formulation, is an exact algorithm, guaranteed to find the globally optimal schedule, although at the cost of more processing time. In contrast to the first three, which schedule one operation at a time, this algorithm produces a schedule for all operations simultaneously.

ASAP/ALAP scheduling. ASAP and ALAP scheduling are the two simplest scheduling algorithms used in high-level synthesis to solve the UCS problem. ASAP scheduling (see Figure 4) schedules each operation, one at a time, into the earliest possible control step. ALAP scheduling is similar, but schedules each operation into the latest possible control step.

```
for each operation o_i
        if o_i has no immediate
        predecessors
                cstep(o_i) = 1 /* cstep (o_i)
                indicates control step
                into which operation o_i is
                scheduled */
        else
                cstep(o_i) = maximum cstep
                of any of o_i's immediate
                predecessors + 1
```

Figure 4. As-soon-as-possible scheduling.

Although limited by their greedy nature, these algorithms quickly solve the UCS problem. However, to find acceptable solutions to the RCS and TCS problems, we need more sophisticated algorithms.

List scheduling. A common choice for solving the RCS problem is list scheduling (see Figure 5),[2] a venerable algorithm based on work done over 30 years ago by Hu,[3] and long used in project management and microcode compaction. Unlike ASAP/ALAP scheduling, which processes each operation in a fixed order, list scheduling processes each control step sequentially, choosing in each iteration the best operation from all appropriate operations to place into the control step, subject to resource constraints.

During the scheduling process, list scheduling uses a ready list (hence the name) to keep track of data-ready operations. Data-ready operations are unscheduled operations the algorithm can schedule into the current control step without violating the precedence constraints (those operations whose immediate predecessors have been scheduled into earlier control steps). As long as the ready list contains data-ready operations that meet the resource constraints, the algorithm chooses operations from that list and schedules them into the current control step.

```
current-cstep = 0

while there are unscheduled operations

        current-cstep = current-cstep + 1

        place data-ready operations
        into the ready list, evaluate
        the priority of each operation,
        and sort the ready list
        in order of priority

        while there are data-ready
        operations in the ready list
        that meet the resource constraints

                choose the highest priority data-ready
                operation o_i from the
                ready list

                cstep(o_i) = current-cstep
```

Figure 5. List scheduling.

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

In choosing which ready-list operation to schedule, the algorithm sorts the ready list according to some priority function, always choosing the highest priority operation for scheduling into the current control step. One common priority function is based on mobility, defined earlier as the length of an operation's schedule interval. Operations with smaller mobility rate a higher priority, since there are fewer possible control steps into which those operations can be scheduled. Also, delaying them to a later control step would more likely increase the overall length of the schedule; this is especially true for those operations with a mobility of 1, which we regard as on the critical path.

The priority function selected clearly biases the results of the list-scheduling algorithm. Some systems give higher priority to operations with lower mobility. Others give higher priority to operations with more immediate successors, arguing that scheduling them in the current control step would make the largest number of operations data ready, thereby allowing the earliest possible consideration of each operation. Unfortunately, there is no agreement on which priority function is best, and furthermore, the choice often depends on the CDFG structure.

The list-scheduling algorithm may also vary in its treatment of the ready list. As Figure 5 showed, this algorithm constructs the ready list only once per control step. It could instead construct the ready list every time it chooses a data-ready operation, thereby choosing an operation from a more up-to-date list at the cost of additional computation. Another variation is to maintain a separate ready list for each functional unit type $k \in K$, thus making it easier to consider only those operations that meet the resource constraints.

Although less efficient computationally than ASAP scheduling, due to its more global selection of the next operation to schedule and its simple yet intuitive priority function, list scheduling remains a common choice for solving the RCS problem.

Force-directed scheduling. Force-directed scheduling (see Figure 6), originally developed as part of Carleton University's HAL system,[4] is a popular constructive algorithm that solves the TCS problem by uniformly distributing the operations of each type across a time-constrained schedule. Balancing the operations in this manner results in higher functional unit usage, and thus minimizes the number of functional units of each type.

Since force-directed scheduling solves the TCS problem, the algorithm must first determine the schedule length (the overall time constraint). Constructing an ASAP schedule and measuring its length will provide a good approximation of the schedule length. Force-directed scheduling also considers the schedule interval of each operation, so this algorithm also constructs an ALAP schedule, using the two schedules to determine the schedule intervals.

Now consider a particular operation o_i that the algorithm can theoretically schedule into any control step s in its schedule interval S_i. We denote its ASAP control step as $ASAP_i$ and its ALAP control step as $ALAP_i$. If we assume that it has a uniform probability of being scheduled into any control step in the range $[ASAP_i, ALAP_i]$, the probability P_{ij} of scheduling operation o_i into a particular control step $s_j \in S_i$ is $P_{ij} = 1/(ALAP_i - ASAP_i + 1)$.

construct an ASAP and an ALAP schedule, determine the schedule length, and compute the schedule interval of each operation

construct a histogram for each operation type k

while there are unscheduled operations

    $\Delta Cbest = \infty$

    for each operation o_i

        for each cstep $s_j \in S_i$

            compute increase in cost $\Delta C_{i,j}$ if operation o_i is scheduled into *cstep* s_j

        if $\Delta C_{i,j} < \Delta C_{best}$

            $\Delta Cbest = \Delta C_{i,j}$

            $bestop = i$; $beststep = j$

    $cstep(bestop) = beststep$

    update histograms

Figure 6. Force-directed scheduling.

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

Figure 7a illustrates these probabilities, where the width of each operation box represents the probability of scheduling the corresponding operation in the data-flow graph in Figure 8 into that control step. Operations 1, 2, 5, 7, and 8 each have a mobility of 1, so their probability of being scheduled into a particular control step is 1. Figure 7a shows each operation wholly within that control step, represented by a box of width of 1. Operation 3, however, has a schedule interval of [1,2]; thus its probability of being scheduled into either control step in that range is 0.5. Figure 7 represents operation 3 with a box of width 0.5 spanning control steps 1 and 2. Operation 6 is similar, and operations 4, 9, 10, and 11 each have a 0.33 probability of being scheduled into a particular control step in their schedule interval.

Given these probabilities, we can construct a histogram for each functional unit type k, showing the expected cost of performing all operations of that type in each control step. For a functional unit type k, the expected functional unit cost in control step $s_j \in S_i$ is

$$FCost_{kj} = c_k \sum_{i \in I_k} P_{i,j}$$

where c_k is the cost of a functional unit of type k and I_k is the index set of all operations of type k.

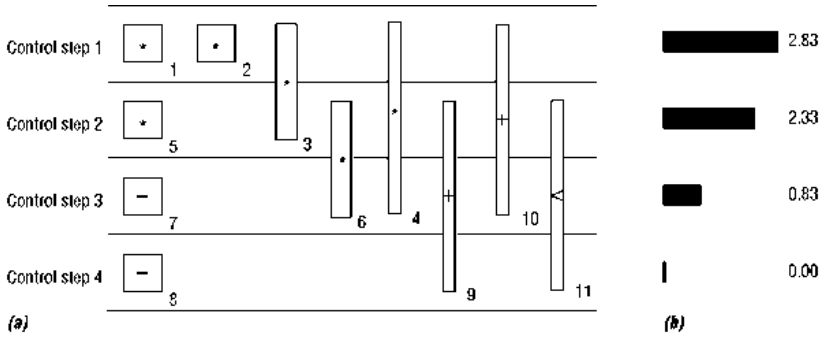

Figure 7. Initial schedule intervals <B>(a)</B> and multiplication histogram (b).

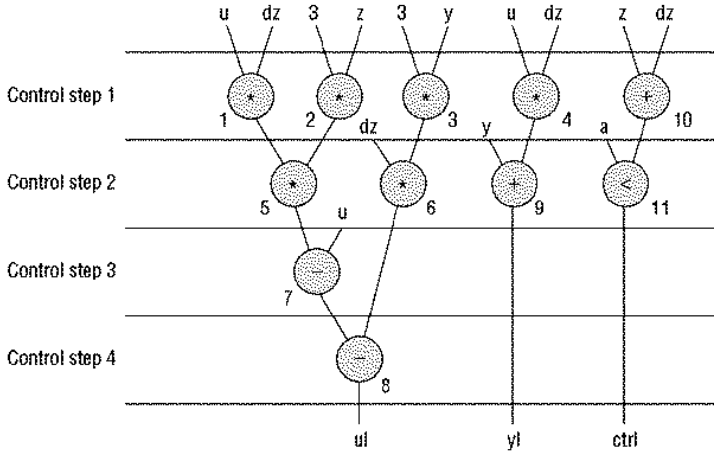

Figure 8. Data-flow graph and ASAP schedule for the differential equation (DiffEq) example.[4]

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

Figure 7b shows the histogram for multiplication operations, assuming a unit multiplier cost. This histogram takes the following form:

$$
\begin{aligned}
FCost_{\text{mult},1} &= P_{1,1} + P_{2,1} + P_{3,1} \\
&\quad + P_{4,1} + P_{5,1} + P_{6,1} \\
&= 1 + 1 + 0.5 \\
&\quad + 0.33 + 0 + 0 \\
&= 2.83 \\
FCost_{\text{mult},2} &= P_{1,2} + P_{2,2} + P_{3,2} \\
&\quad + P_{4,2} + P_{5,2} + P_{6,2} \\
&= 0 + 0 + 0.5 \\
&\quad + 0.33 + 1 + 0.5 \\
&= 2.33 \\
FCost_{\text{mult},3} &= 0 + 0 + 0 \\
&\quad + 0.33 + 0 + 0.5 \\
&= 0.83 \\
FCost_{\text{mult},4} &= 0
\end{aligned}
$$

The algorithm can now compute the expected number m_k of functional units of type k as the maximum number of functional units of that type in any control step

$$m_k = \left\lceil \max_{j \in s}(FCost_{kj}) \right\rceil$$

where S is the index set of all control steps. Thus this example should require [2.83] = 3 multipliers.

Force-directed scheduling minimizes the number of functional units of each type by uniformly distributing the operations of that type across the schedule (that is, to balance the histogram). Consider the effect of scheduling operation 3 into either control step 1 or 2. If the algorithm schedules operation 3 into control step 1, the maximum expected multiplier cost will be 3.33. Therefore, the design will require four multipliers. However, if it schedules operation 3 into control step 2 (changing operation 6's schedule interval to [3,3]), the maximum expected multiplier cost will be 2.33, and the design will require only three multipliers. The latter choice is preferable, as it tends to reduce the expected multiplier cost and more uniformly distributes multiplications across the schedule.

Force-directed scheduling iteratively builds a control step schedule, keeping the schedule balanced as follows. First, it creates the initial histograms. Then it computes the expected functional unit cost of scheduling each unscheduled operation into each control step in its schedule interval and makes the operation/control step scheduling that results in the smallest increase (or largest decrease) in cost. It then updates the histograms, and the process continues until there are no more unscheduled operations.

In force-directed scheduling, we compute the increase in expected functional unit cost that results from assigning an operation to a particular control step as the sum of a set of forces (hence the name). The direct force of an operation o_i with schedule interval S_i being scheduled into control step $s_j \in S_i$ is

$$\text{Force}_{i,k,j} = FCost_{k,j} - \sum_{s=ASAP_i}^{ALAP_i} \frac{FCost_{k,s}}{ALAP_i - ASAP_i + 1}$$

For a particular operation (and therefore a particular functional unit type), the direct force thus is the difference between the expected functional unit cost in that control step and the average expected functional unit cost over that operation's schedule interval.

Since scheduling an operation into a particular control step may affect the schedule intervals of other operations—for example, operation 6 described earlier—we must consider those indirect costs as well. Thus we must compute the total force associated with an operation being scheduled into a particular control step as the sum of 1) its direct force, and 2) the indirect force on any other operation whose schedule interval that scheduling affects.

In our example, the total force associated with scheduling operation 3 into control step 1 (see Figure 7) is only that direct force, since no other schedule intervals are affected.

$$
\begin{aligned}
&Total\text{-}Force_{3,\text{mult},1} \\
&= Force_{3,\text{mult},1} \\
&= 2.83 - (2.83 + 2.33)/2 \\
&= +0.25
\end{aligned}
$$

However, the total force associated with scheduling operation 3 into control step 2 is that direct force plus the indirect force of scheduling operation 6 into control step 3

$$
\begin{aligned}
&Total\text{-}Force_{3,\text{mult},2} \\
&= Force_{3,\text{mult},2} + Force_{6,\text{mult},3} \\
&= 2.33 - (2.83 + 2.33)/2 \\
&\quad + 0.83 - (2.33 + 0.83)/2 \\
&= -1.0
\end{aligned}
$$

Given these two choices, the algorithm would choose the operation/control step scheduling that results in the largest decrease in cost (force). In this example, it would schedule operation 3 into control step 2, as we conjectured earlier.

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

Although less efficient computationally than either ASAP or list scheduling, due to its global selection of the next operation to schedule and to its effectiveness in uniformly distributing the operations across the schedule, force-directed scheduling is a common choice for solving the TCS problem.

Integer linear programming formulations. Mathematical programming formulations, among them integer linear programming (ILP), have solved a wide range of problems in high-level synthesis, beginning with Hafer's early scheduling formulation.[5] Here we discuss ILP formulations for optimally solving the TCS, RCS, and TRCS problems.

The biggest advantage of these formulations is the solution quality. Unlike the constructive heuristics described earlier, a commercial ILP solver is guaranteed to find an optimal schedule from these formulations. Unfortunately, this guarantee comes at a price: ILPs cannot, in general, be solved in polynomial time. Thus the trade-off is between guarantees of solution quality and algorithm runtime. Fortunately, a carefully designed ILP formulation produces results acceptably quickly for small and medium-sized problems; ongoing research on bounding techniques may soon allow larger problems to be solved as well.

ILP problems[6] are problems that either maximize or minimize some objective function of many variables, subject to linear equality and inequality constraints, and integrality restrictions on all of the variables. These problems also commonly use linear objective functions and require nonnegative variables. We write an ILP as

$$Z_{IP} = \min\{c^T x | x \in P_F; x \text{integer}\}$$

where $P_F = \{ Ax \le b, x \in R_+^n \}$, and where R_+^n is the set of nonnegative real (nx1) vectors, c is a (nx) real vector, b is a (mx1) integer vector, and A is an (mxn) integer matrix.

Set of feasible schedules. Consider the set of nodes $V = \{(i,s) \mid i \in I;\ s \in S_i\}$ as illustrated in Figure 9b, where a node (i,s) corresponds to operation o_i being scheduled in control step s, I is the index set of all operations, and S_i is the schedule interval over which an operation o_i can be scheduled.

Each operation o_i can conceivably be scheduled anywhere in its schedule interval S_i, so corresponding to each operation o_i is a set of nodes $V_i = \{(i,s) \mid s \in S_i\}$. For example, in Figure 9b, assuming a deadline of five control steps, it can conceivably schedule operation 2 into either control step 2, 3, or 4, as shown by the horizontal shaded oval labeled V_2.

Furthermore, for each control step s, each functional unit type k corresponds to a set of nodes $V_{k,s} = \{(i,s) \mid s \in S_i;\ \tau(i)=k\}$, which can map onto that functional unit type during that control step. For example, in Figure 9, assuming all three operations map onto adders, operations 1, 2, and 3 might all be scheduled onto adders during control step 3, as the vertical shaded oval in the third column shows.

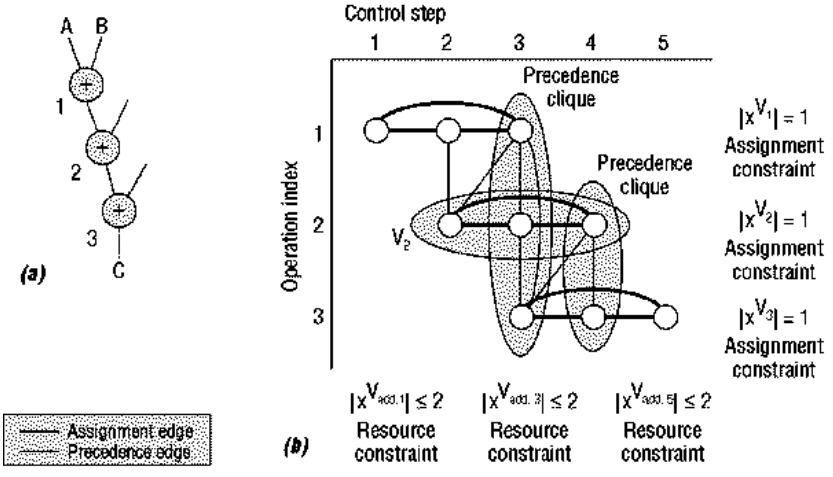

Figure 9. A CDFG (a) and its constraint graph (b), assuming a schedule of length 5 and a resource constraint of two adders.

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

Each feasible schedule contains exactly one node from each set of nodes V_i, satisfies all the precedence constraints between operations, and uses no more than the available number of functional units of each type. Clearly, to find a feasible schedule, we need some way to determine which $x_{i,s}$ variables are 1 and which are 0. We will determine these values by specifying a set of equality and inequality constraints on the scheduling problem and then construct an ILP formulation of the problem. Solving the ILP formulation using an ILP solver will give us an optimal solution for a specified objective function.

Constraints on the scheduling problem. To characterize the constraints on the scheduling problem, we need to construct a constraint graph G_c as follows (see Figure 9). The nodes of G_c are the nodes V that we have already seen.

We can now define assignment constraints on the scheduling problem, thus ensuring that a feasible schedule has exactly one node per operation:

$$\sum_{v \in V_i} x_v = 1, \forall i \in I$$

In Figure 9b, the horizontal shaded oval labeled V_2 represents the assignment constraint for operation 2, constraining it to be scheduled into exactly one control step in the range [2, 4].

Earlier, we defined precedence constraints between two operators; a precedence clique C_p is a clique in G_c with at least one precedence edge (constraint) connecting two of its nodes. (A clique is a fully connected subgraph—a graph in which each node connects to all other nodes of the subgraph.) This concept enables us to define precedence constraints on the scheduling problem, thus preventing two nodes in precedence conflict from being in the same feasible schedule:

$$\sum_{v \in C_p} x_v \le 1, \forall C_p \in I$$

In Figure 9b, the shaded vertical oval in column 3 represents a precedence clique stating that if either operation 1, 2, or 3 is scheduled into control step 3, the two remaining operations cannot be scheduled into that control step as well.

Finally, we can define resource constraints on the scheduling problem, ensuring that in each control step the number of operations of each type do not exceed the available number of functional units of that type:

$$\sum_{v \in k,s} x_v \le m_k, s \in S, \forall k$$

In Figure 9b, the resource constraint at the bottom of column 3 states that no more than two addition operations can be scheduled into control step 3.

We can represent these constraints succinctly in the following form:

$$M_a x = 1;\ M_p x \le 1;\ M_r x \le m$$

where M_a is the coefficient matrix due to the assignment constraints, M_p is the coefficient matrix due to the precedence constraints, and M_r is the coefficient matrix due to the resource constraints.

ILP formulation. We can now use these constraints to construct ILP formulations representing the various scheduling problems. We can easily construct the formulation of the TRCS problem by combining formulations of the TCS and RCS problems. Given these formulations, a commercial ILP solver can produce an optimal solution.

For the TCS problem, we can minimize a function calculating the number of functional units of each type.

$$Z_{IP} = \min\left\{ \sum_{k \in K} a_k m_k \,\middle|\, x \in P_F(Q); x \text{ integer} \right\}$$

$$P_F(Q) = \left\{ x \in R_+^{|V|} \,\middle|\, M_a x = 1;\ M_p x \le 1;\ M_r x \le m \right\}$$

where, for functional units of type $k \in K$, a_k is a weight (usually based on area), and m_k is the number of functional units of that type.

For the RCS problem, we can minimize the number of control steps by introducing a dummy operation o_d, adding edges to ensure that o_d is scheduled after all other operations, and scheduling o_d as early as possible.

$$Z_{IP} = \min\left\{ \sum_{s \in S_d} SX_{d,s} \,\middle|\, x \in P_F(Q); x \text{ integer} \right\}$$

where $P_F(Q)$ was defined earlier, and S_d is the schedule interval of operation o_d.

Rensselaer Polytechnic Institute's RPI-ILP system[7] uses this formulation to solve the TRCS directly, and the TCS and RCS problems indirectly, quickly producing guaranteed optimal solutions to each. Tsing Hua University's THEDA System[8] and the University of Waterloo's OASIC System[9] use similar formulations.

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.

Conclusion

This tutorial attempts to define the more common variations on the scheduling problem in high-level synthesis and describes several commonly used scheduling techniques. The scheduling problem will undoubtedly remain an area of research for years to come, as we begin to explore various related problems now that we understand the basic scheduling problem. In the future, we will continue to improve our understanding of the relationship between scheduling, allocation, and binding, and will explore the relationships between scheduling and clock determination, type mapping, and time and resource bounding.

References

1. D.D. Gajski and L. Ramachandran, "Introduction to High-Level Synthesis," *IEEE Design & Test of Computers*, Vol. 11, No. 4, Winter 1994, pp. 44-54.
2. B.M. Pangrle and D.D. Gajski, "Design Tools for Intelligent Silicon Compilation," *IEEE Trans. Computer-Aided Design*, Vol. CAD-6, No. 6, Nov. 1987, pp. 1098-1112.
3. T.C. Hu, "Parallel Sequencing and Assembly Line Problems," *Operations Research*, Vol. 9, No. 6, Nov. 1961, pp. 841-848.
4. P.G. Paulin and J.P. Knight, "Algorithms for High-Level Synthesis," *IEEE Design & Test of Computers*, Vol. 6, No. 4, Dec. 1989, pp. 18-31.
5. L.J. Hafer and A.C. Parker, "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic," *IEEE Trans. Computer-Aided Design*, Vol. CAD-2, No. 1, Jan. 1983, pp. 4-18.
6. G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, John Wiley & Sons, New York, 1988.
7. S. Chaudhuri and R.A. Walker, "Analyzing and Exploiting the Structure of the Constraints in the ILP Approach to the Scheduling Problem," *IEEE Trans. VLSI Systems*, Vol. 2, No. 4, Dec. 1994, pp. 456-471.
8. C.-T. Hwang, J.-H. Lee, and Y.-C. Hsu, "A Formal Approach to the Scheduling Problem in High-Level Synthesis," I*EEE Trans. Computer-Aided Design*, Vol. 10, No. 4, Apr. 1991, pp. 464-475.
9. C.H. Gebotys, "Optimal Scheduling and Allocation of Embedded VLSI Chips," *Proc. 29th Design Automation Conf.*, IEEE Computer Society Press, Los Alamitos, Calif., 1992, pp. 116-119.

Robert A. Walker is an assistant professor of computer science at Rensselaer Polytechnic Institute. He is the coauthor of two books on high-level synthesis and is particularly interested in those problems connected with scheduling. He has served on the Advisory Board of the ACM SIGDA, has been actively involved with its university booth program, and also serves as SIGDA secretary-treasurer. Walker received the MS and PhD degrees from Carnegie Mellon University. He is a senior member of the IEEE, and a member of the Computer Society, the ACM, ACM SIGDA, and Sigma Xi.

Samit Chaudhuri is working on his PhD at Rensselaer Polytechnic Institute. His research interests include design automation, high-level synthesis, and combinatorial optimization. He received the BE degree in electronics and telecommunications engineering from the University of Calcutta and the Mtech degree in electrical engineering from the Indian Institute of Technology, Kanpur, India.

Address correspondence about this tutorial to Robert A. Walker at Rensselaer Polytechnic Institute, Computer Science Department, Troy, NY 12180; walkerb@cs.rpi.edu.

Originally published in *IEEE Design and Test of Computers*, Summer 1995, pp. 60-69.