High-Level Synthesis: *Introduction to the Scheduling Problem*

Robert A. Walker[†][‡] **and Samit Chaudhuri**[‡]

Computer Science Department[†]
Electrical, Computer, and Systems Engineering Department[‡]
Rensselaer Polytechnic Institute
Troy, NY 12180

ABSTRACT

The *scheduling* problem — one of the central tasks in high-level synthesis — is the problem of determining the order in which the operations in the behavioral description will execute. This tutorial introduces the scheduling problem, and describes four scheduling algorithms commonly used today to solve those problems.

1. INTRODUCTION

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 [Gajski94], high level synthesis has the potential to greatly improve both designer productivity and design space exploration.

As defined in that introductory article, the three central synthesis tasks in a typical high-level synthesis system are the following:

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

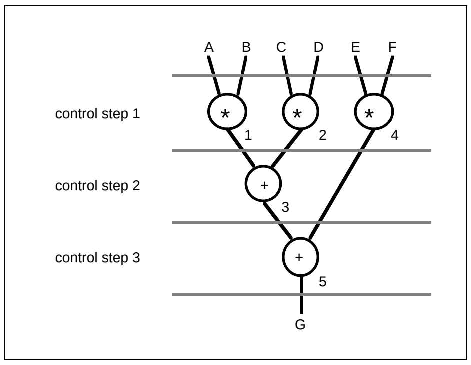

Figure 1: Schedule for the Expression "G=AB+CD+EF"

In most high-level synthesis systems, scheduling and functional unit allocation are performed 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.

More specifically, this tutorial defines the basic scheduling problem, and three common variations on that problem necessary to meet real-world design constraints. It then describes four scheduling algorithms commonly used today to solve those problems: three simple constructive heuristics, and an optimal solution technique based on Integer Linear Programming (ILP).

2. THE BASIC SCHEDULING PROBLEMS

One of the first steps in a typical high level synthesis system is to convert 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*). Operations in the behavioral description, such as additions and multiplications, are represented as nodes in the cdfg, and values (inputs to the expression, temporary results, and the output of the expression) are represented as edges. In more complex behaviors, the cdfg can also represent conditional branches, loops, etc., hence the name "control / data flow graph."

Consider the arithmetic expression "G=AB+CD+EF". This expression might be part of a larger description, for example, part of a digital signal processor description, but for purposes of simplicity we will consider only that one expression here. We can parse this expression to build a cdfg, shown in Figure 1, which will serve as the internal representation for a high-level synthesis system. The goal of scheduling, then, is to determine the order in which these operations will execute, i.e., to schedule each operation into an appropriate control step.

2.1. Basic Concepts

Suppose we try to 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 that same control step, since the two operations do not depend on each other in any way. However, operation 3 cannot be scheduled into control step 1, since it depends upon the results of operation 1 and operation 2, which are not available until the end of control step 1. Thus operation 3 must be delayed 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, operation 5 must be scheduled into control step 3 or later, since it depends upon the result of operation 3.

Let O be the set of all operations in the cdfg. If the result of operation $o_i \in O$ is used by operation $o_j \in O$, then operation o_i must finish its execution before operation o_j can begin, and we say that there is a *data dependency* between the two operations. We would also say that o_i is an *immediate predecessor* of o_j and that o_j is an *immediate successor* of o_i. This data dependency is represented during the synthesis process as a *precedence constraint* between the two operations, which must be satisfied by the control step schedule.

In generating the control step schedule for a particular cdfg, we have two options — we may be satisfied with generating a *feasible schedule* (any "legal" schedule), or we may want to find a schedule 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, so that 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 *type* of the functional unit indicates its functionality (e.g., addition, multiplication, or addition / multiplication). Let K be the set of types that are available, and let a_k and m_k be the area and number of functional units, respectively, of type $k \in K$.

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

We can now use these concepts to define the *Unconstrained Scheduling* (UCS) problem — the basic scheduling problem in high-level synthesis:

Unconstrained Scheduling (UCS)

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.

2.2. Time-Constrained Scheduling (TCS)

Although the UCS problem defined above captures the basic elements of the scheduling problem in high-level synthesis, in practice it may be necessary to add additional constraints to meet particular design goals. For example, we could limit the execution time of the design by constraining the overall length of the control step schedule — we refer to this process as adding a *time constraint* on the overall schedule length (i.e., a *deadline*), which must be satisfied by the control step schedule.

Adding this time constraint on the overall length of the schedule to the UCS problem, we can define the *Time-Constrained Scheduling* (TCS) problem as follows:

Time-Constrained Scheduling (TCS)

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.

Investigating further, we can see by examining Figure 1 that if we do impose an overall time constraint on the schedule, some operations may be forced 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 must be scheduled into control step 2, and operation 5 must be scheduled into control step 3. Since we have no freedom in scheduling these operations (without violating the time constraint), we say that they are on the *critical path*.

In general, we say that there is a continuous range S_i of control steps, called the *schedule interval*, over which an operation o_i can be scheduled. 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.

2.3. Resource-Constrained Scheduling (RCS)

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, 2 multipliers may fit within the available chip area, while 8 multipliers may not — in this case it may be necessary to impose a *resource* (*functional unit*) *constraint* on the design, limiting the number of multipliers to 2.

Consider the effect of adding resource constraints to the example of Figure 1. The schedule shown in that figure was generated without resource constraints, and requires at least 3 multipliers and 1 adder. However, if we constrain the number of multipliers to 1, we might obtain the schedule shown in Figure 2, which would require substantially less functional unit area. (Notice that although this schedule uses two less functional units, the schedule length has increased by one, illustrating the classical *serial-parallel tradeoff* that often occurs in scheduling problems — 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:

Resource-Constrained Scheduling (RCS)

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 that meets the resource constraints for each functional unit type.

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

2.4. Time- and Resource-Constrained Scheduling (TRCS)

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:

Time- and Resource-Constrained Scheduling (TRCS)

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 meets the resource constraints for each functional unit type.

3. ADVANCED SCHEDULING TOPICS

In order to provide a concise introduction to the various scheduling problems, the previous section made a number of overly simplistic assumptions. This section will briefly introduce several advanced topics, all of which must be considered by any practical scheduling algorithm.

3.1. Chaining and Multicycling

In the previous section, the assumption was made that each operation type requires the same amount of time to execute, and that the control step length (i.e., the clock period) is equal to that execution time. In practice, different operation types may have different execution times (i.e., the functional unit types onto which each is mapped may have different propagation delays), and an overly restrictive scheduling model coupled with a poor choice for the control step length can result in poorly-utilized functional units and an overly long schedule.

Consider the cdfg of Figure 3a, where the multiplication and addition operations are mapped onto a multiplier and adder with a 100ns and 50ns propagation delay, respectively (for simplicity, we will assume that these functional unit propagation delays include any necessary register setup times). If the control step length is set to the 100ns (the longest of the two propagation delays), and each operation is scheduled into exactly one control step, the overall length of the schedule will be 200ns, and the multiplier will be utilized only half of the time.

However, the multiplier utilization can be increased, and the schedule length can be decreased, by packing the two additions into a single control step, as shown in Figure 3b. We refer to these two additions as *chained* operations, and implement them at the register-transfer level by connecting the output of the first adder directly to the input of the second adder (i.e., without the intervening register that would otherwise latch the result of the first adder at the control step boundary). In this example, chaining can increase the multiplier utilization and decrease the length of the schedule to 100ns, but at the cost of an additional adder.

Another alternative to improve the schedule of Figure 3a is to set the clock length to 50ns (the shortest of the two propagation delays), and to execute the multiplication over two control steps, as shown in Figure 3c. We refer to an operation such as this multiplication as a *multicycle* operation — it will have to execute continuously throughout its entire execution time, and its input values

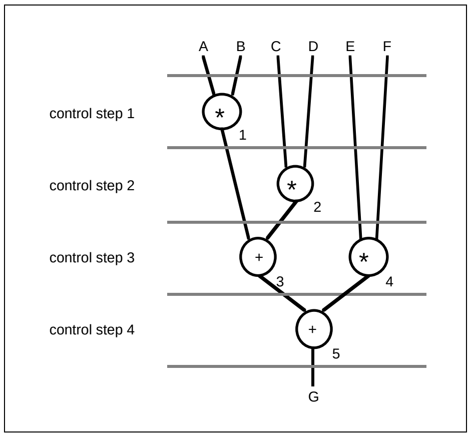

Figure 2: Schedule with a Resource Constraint of 1 Multiplier

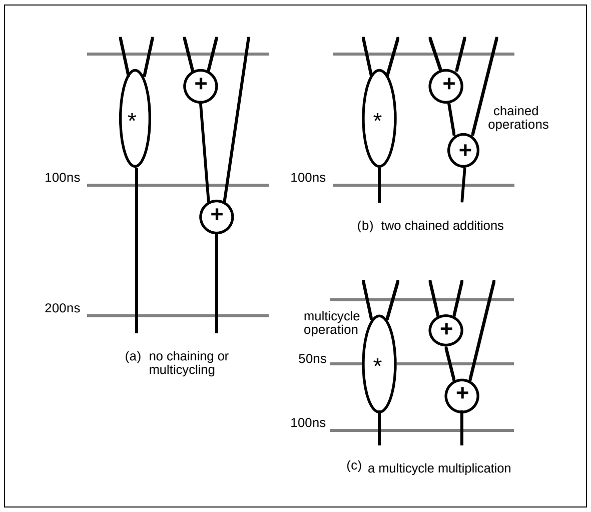
100ns
200ns
(a) no chaining or multicycling
chained operations
100ns
(b) two chained additions
multicycle operation
50ns
100ns
(c) a multicycle multiplication

Figure 3: Chained and Multicycle Operations

must be latched throughout that period as well. In this example, multicycling can decrease the length of the schedule to 100ns, 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.

3.2. Scheduling in the Presence of Control Constructs

The basic scheduling problems presented in the last section were all defined for a single *basic block* — one section of straight-line code with only one entry point and one exit point. Since most hardware description languages support conditionals, loops, and other control constructs, those constructs must be considered during the scheduling process. When scheduling conditional branches, the scheduler should exploit any potential parallelism by sharing functional units between mutually exclusive branches (e.g., the same adder can be used 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.

3.3. Scheduling with Timing Constraints

The basic scheduling problems presented in the last section 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 there may also be a need to specify more detailed *timing constraints* on certain operations: *minimum* timing constraints, which specify that one operation must be executed at least a specified amount of time after another operation, and *maximum* timing constraints, which specify that one operation must be executed 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, and then treating those additional edges in much the same manner as other constraints.

4. SOME COMMON SCHEDULING ALGORITHMS

This tutorial has defined the four basic scheduling problems in high-level synthesis. To solve those scheduling problems, both *heuristic algorithms*, which find feasible (possibly suboptimal) solutions, and *exact algorithms*, which find optimal solutions, have been used.

```
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 (ASAP) Scheduling

This section will highlight four scheduling algorithms commonly used today by high-level synthesis systems to solve those problems.

The first three scheduling algorithms described here (ASAP / ALAP scheduling, list scheduling, and force-directed scheduling) are *constructive heuristic algorithms*, which 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 look-ahead, they may miss the globally optimal solution. However, they do produce results quickly, and those results may be sufficiently close to optimal to be acceptable in practice.

The fourth scheduling algorithm considered here, based on solving an Integer Linear Programming (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 algorithms, which schedule one operation at a time, this algorithm produces a schedule for all operations simultaneously.

4.1. ASAP / ALAP Scheduling — A Constructive Heuristic for the UCS Problem

As-Soon-As-Possible (ASAP) scheduling and *As-Late-As-Possible (ALAP) scheduling* are the two simplest scheduling algorithms used in high-level synthesis. 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.

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

4.2. List Scheduling — A Constructive Heuristic for the RCS Problem

A common choice for solving the RCS problem is *list scheduling* [Pangrle87] (see Figure 5), a venerable algorithm based on work done over 30 years ago by Hu [Hu61], 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 — those unscheduled operations that can be scheduled into the current control step without violating the precedence constraints (i.e., those operations whose immediate predecessors have been scheduled into earlier control steps). As long as there are data-ready operations that meet the resource constraints in the ready list, operations are chosen from that list and scheduled into the current control step.

To make the choice of which operation in the ready list to schedule, those operations are sorted according to some *priority function*, and the operation with the highest priority is always chosen to be scheduled into the current control step. One common priority function is based on *mobility*, which was defined earlier as the length of an operation's schedule interval. Operations with smaller mobility are given higher priority, since there are fewer possible control steps into which those operations can be scheduled, and since 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 are said to be on the *critical path*.

The results of the list scheduling algorithm are clearly biased by the priority function chosen. Some systems give higher priority to those operations with lower mobility, as described above. Others give higher priority to those operations with more immediate successors, arguing that scheduling them in the current control step would make the largest number of operations data-ready, and thereby allow each operation to be considered as early as possible. Unfortunately, there is no agreement on which priority function is the "best", and the choice often depends on the structure of the cdfg.

The list scheduling algorithm may also vary in its treatment of the ready list. As presented in Figure 5, the ready list is constructed only once per control step, but the ready list could instead be constructed every time a data-ready operation is chosen, allowing an operation to be chosen 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 to its simple yet intuitive priority function, list scheduling remains a common choice for solving the RCS problem.

4.3. Force-Directed Scheduling — A Constructive Heuristic for the TCS Problem

Force-directed scheduling (see Figure 8), originally developed as part of Carleton University's HAL system [Paulin89], 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 utilization, and thus minimizes the number of functional units of each type.

```
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

Since force-directed scheduling solves the TCS problem, the first step in the algorithm must be to determine the schedule length (the overall time constraint). A good approximation of the schedule length can be determined by constructing an ASAP schedule, and measuring the length of that schedule. Force-directed scheduling also considers the schedule interval of each operation, so an ALAP schedule is constructed as well, and the two schedules are used to determine the schedule intervals.

Now consider a particular operation o_i, that can theoretically be scheduled into any control step s in its schedule interval S_i. If we denote its ASAP control step as $ASAP_i$ and its ALAP control step as $ALAP_i$, and we assume that it has a uniform probability of being scheduled into any control step in the range $[ASAP_i, ALAP_i]$, then the probability $P_{i,j}$ of scheduling operation o_i into a particular control step $s_j \in S_i$ is:

$$P_{i,j} = \frac{1}{ALAP_i - ASAP_i + 1}$$

These probabilities are illustrated on the left side of Figure 7, where the width of each operation box represents the probability of scheduling the corresponding operation in the data flow graph in Figure 6 into that control step. Operations 1, 2, 5, 7, and 8 each have a mobility of 1, so have a probability of 1 of being scheduled into a particular control step; in Figure 7, each is shown wholly within that control step, represented by a box of width 1. Operation 3, however, has a schedule interval of [1,2], and thus a probability of 0.5 of being scheduled into either control step in that range; in Figure 7, operation 3 is represented by 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 probability of 0.33 of being scheduled into a particular control step in their schedule interval.

Given these probabilities, a *histogram* can be constructed 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:

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

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.

The right side of Figure 7 shows the histogram for multiplication operations, assuming a unit multiplier cost. This histogram is constructed as follows:

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

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

$$\begin{aligned} FCost_{\text{mult},3} &= 0 + 0 + 0 + 0.33 + 0 + 0.5 \\ &= 0.83 \end{aligned}$$

$$FCost_{\text{mult},4} = 0$$

The expected number m_k of functional units of type k can now be computed as the maximum number of functional units of that type in any control step:

$$m_k = \left\lceil \max_{j \in S} (FCost_{k,j}) \right\rceil$$

where S is the index set of all control steps. Thus this example is expected to require $\lceil 2.83 \rceil = 3$ multipliers.

The goal of force-directed scheduling, as stated previously, is to minimize the number of functional units of each type by uniformly distributing the operations of that type across the schedule (i.e., to balance the histogram). Consider the effect of scheduling operation 3 into either control step 1 or 2. If operation 3 is scheduled into control step 1, the maximum expected multiplier cost will be 3.33, and therefore the design will require 4 multipliers. However, if operation 3 is scheduled 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 3 multipliers. Of these two choices, the latter would be preferable, as it tends to reduce the expected multiplier cost and more uniformly distribute the multiplications across the schedule.

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

In force-directed scheduling, the "increase" in expected functional unit cost that will result if an operation is assigned to a particular control step is computed as the sum of a set of *forces* (hence the name). For an operation o_i with schedule interval S_i, the *direct force* of its being scheduled into control step $s_j \in S_i$ is:

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

In other words, for a particular operation (and therefore a particular functional unit type) the direct force 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 (e.g., operation 6 as described above), those "indirect" costs must be considered as well. Thus the total force associated with an operation being scheduled into a particular control step must be computed as the sum of: (1) its direct force, and (2) the indirect force on any other operation whose schedule interval is affected by that scheduling.

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

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

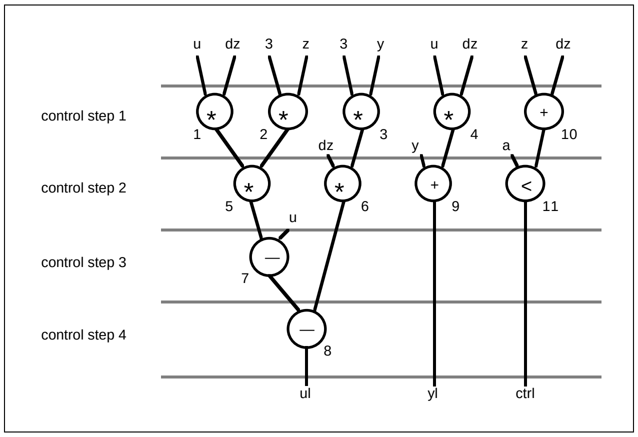

Figure 6: Data Flow Graph and ASAP Schedule for the Differential Equation (DiffEq) Example [Paulin89]

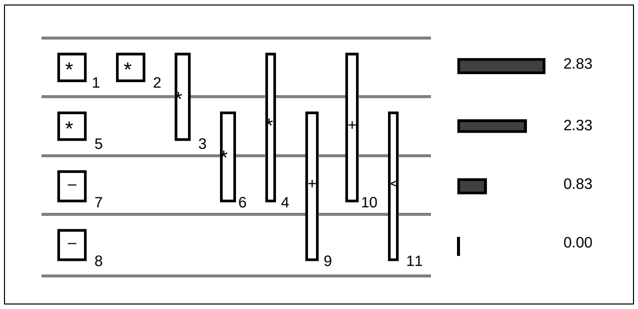

Figure 7: Initial Schedule Intervals and Multiplication Histogram

```
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
   $\Delta C_{best} = \infty$
   **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** $\Delta C_{i,j} < \Delta C_{best}$
            $\Delta C_{best} = \Delta C_{i,j}$
            $bestop = i$; $beststep = j$
   $cstep(bestop) = beststep$
   update histograms
```

Figure 8: Force-Directed Scheduling

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:

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

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.

Although less efficient computationally than either ASAP scheduling 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.

4.4. Integer Linear Programming (ILP) Formulations

Mathematical programming formulations, among them *Integer Linear Programming* (ILP), have been used to solve a wide range of problems in high-level synthesis, beginning with Hafer's early scheduling formulation [Hafer83]. This section introduces an ILP formulation for optimally solving the TCS, RCS, and TRCS problems.

The biggest advantage of these formulations is the quality of the solution — unlike the constructive heuristics described earlier, a commercial ILP solver is guaranteed to find an optimal schedule from these formulations. Unfortunately, this guarantee of quality comes at a price — ILPs can not, in general, be solved in polynomial time. Thus the tradeoff is between the a guarantee of solution quality and a guarantee of quickly finding a solution. Fortunately, however, a carefully-designed ILP formulation can produce results in acceptable time for small and medium-sized problems, and ongoing research on bounding techniques may soon allow larger problems to be solved as well.

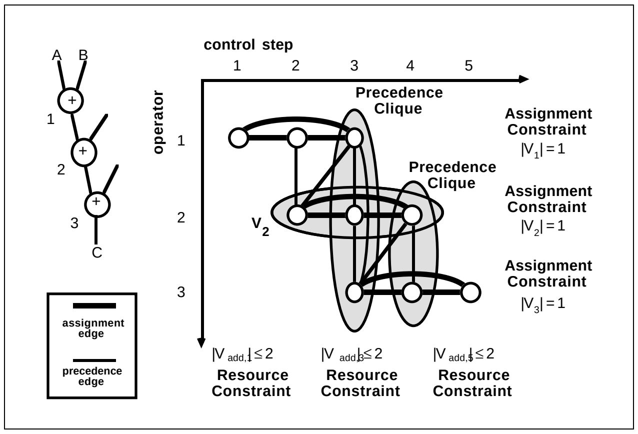
A
B
1
2
3
C
control step
1 2 3 4 5
operator
1
2
3
Precedence Clique
Precedence Clique
V2
Assignment Constraint |V1| = 1
Assignment Constraint |V2| = 1
Assignment Constraint |V3| = 1
|V add,1| ≤ 2 Resource Constraint
|V add,3| ≤ 2 Resource Constraint
|V add,5| ≤ 2 Resource Constraint
assignment edge
precedence edge

Figure 9: A CDFG and Its Constraint Graph, Assuming a Schedule of Length 5, and a Resource Constraint of 2 Adders

Integer Linear Programming (ILP) problems [Nemhauser88] are those problems that either maximize or minimize some *objective function* of many variables, subject to: (1) linear equality and inequality constraints, and (2) integrality restrictions on all of the variables. It is also common to use linear objective functions, and to require that the variables be non-negative. An *integer linear programming* (ILP) formulation is written as:

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

$$\text{where } P_F = \left\{Ax \le b, x \in \mathbb{R}_+^n\right\}$$

where $\mathbb{R}_+^n$ is the set of non-negative real $(n \times 1)$ vectors, c is a $(n \times 1)$ real vector, b is a $(m \times 1)$ integer vector, and A is a $(m \times n)$ integer matrix.

The Set of Feasible Schedules

Consider the set of nodes $V = \left\{(i,s) \mid i \in I; s \in S_i\right\}$ as illustrated in Figure 9, 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 (see Section 1.2).

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 $V_i = \left\{(i,s) \mid s \in S_i\right\}$. For example, in Figure 9, assuming a deadline of 5 control steps, operation 2 can conceivably be scheduled 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 $V_{k,s} = \left\{(i,s) \mid s \in S_i; \tau(i) = k\right\}$, which can be mapped onto that functional unit type during that control step. For example, in Figure 9, assuming all three operations are mapped onto adders, operations 1, 2, and 3 might all be scheduled onto adders during control step 3, as shown by the vertical shaded oval in the third column.

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, in order 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 from there construct an integer linear programming (ILP) formulation of the problem, which will give us an optimal solution to the problem for a specified objective function.

The 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, which will ensure that a feasible schedule has exactly one node per operation:

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

In Figure 9, 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 in this tutorial we defined precedence constraints between two operators; a *precedence clique* C_p is defined as a clique[1] in G_c that has at least one precedence edge (constraint) connecting two of its nodes. We can use this concept to define *precedence constraints* on the scheduling problem, which will prevent two nodes that are in precedence conflict from being in the same feasible schedule:

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

In Figure 9, 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 can not be scheduled into that control step as well.

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

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

In Figure 9, 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:

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

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.

The ILP Formulation

We can now use these constraints to construct ILP formulations that represent the various scheduling problems. Formulations of the TCS and RCS problems are shown below, and the formulation of the TRCS problem can easily be constructed by combining those two formulations. Given these formulations, a commercial ILP solver can be used to find an optimal solution.

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

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

$$P_F(\mathcal{Q}) = \left\{ x \in \mathbb{R}_+^{|V|} \mid M_a x = 1;\ M_p x \leq 1;\ M_r x \leq m \right\}$$

[1]A *clique* is a fully-connected subgraph — a subgraph where each node is connected to all other nodes of the subgraph.

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:

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

where $P_F(\mathcal{Q})$ is defined above, and S_d is the schedule interval of operation o_d.

Rensselaer Polytechnic Institute's RPI-ILP system [Chaudhuri94] uses the formulation described in this section to solve the TRCS directly, and the TCS and RCS problems indirectly, quickly producing guaranteed optimal solutions to each. Similar formulations are used in Tsing Hua University's THEDA System [Hwang91] and the University of Waterloo's OASIC System [Gebotys92].

5. Summary

This paper has attempted to define the more common variations on the scheduling problem in high-level synthesis, and to describe several scheduling algorithms commonly used today in high-level synthesis. 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 have an understanding of the basic scheduling problem. In the future, we will continue to try to better understand the relationship between scheduling, allocation, and binding, and will begin to explore the relationship between scheduling and clock determination, type mapping, and time and resource bounding.

References

[Chaudhuri94]	Samit Chaudhuri and Robert A. Walker, "Analyzing and Exploiting the Structure of the Constraints in the ILP Approach to the Scheduling Problem", *IEEE Trans. on VLSI Systems*, pages 456–471, December 1994.
[Gajski94]	Daniel D. Gajski and Loganath Ramachandran, "Introduction to High-Level Synthesis", *IEEE Design & Test*, pages 44–54, Winter 1994.
[Gebotys92]	Catherine H. Gebotys, "Optimal Scheduling and Allocation of Embedded VLSI Chips", *Proc. of the 29th DAC*, pages 116–119, June 1992.
[Hafer83]	Louis J. Hafer and Alice C. Parker, "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic", *IEEE Trans. on CAD*, pages 4–18, January 1983.
[Hu61]	T.C. Hu, "Parallel Sequencing and Assembly Line Problems", *Operations Research*, pages 841–848, Volume 9, 1961.
[Hwang91]	Cheng-Tsung Hwang, Jiahn-Hurng Lee, and Yu-Chin Hsu, "A Formal Approach to the Scheduling Problem in High-Level

Synthesis”, *IEEE Trans. on CAD*, pages 464–475, April 1991.

[Nemhauser88]	G.L. Nemhauser and L.A. Wolsey, *Integer and Combinatorial Optimization*, Wiley, 1988.
[Pangrle87]	Barry Michael Pangrle and Daniel D. Gajski, “Design Tools for Intelligent Silicon Compilation”, *IEEE Trans. on CAD*, pages 1098–1112, November 1987.
[Paulin89]	Pierre G. Paulin and John P. Knight, “Algorithms for High-Level Synthesis”, *IEEE Design and Test*, pages 18–31, December 1989.