Force-Directed Scheduling for the Behavioral Synthesis of ASIC's

PIERRE G. PAULIN, STUDENT MEMBER, IEEE, AND JOHN P. KNIGHT, MEMBER, IEEE

Abstract—**The HAL system described performs behavior synthesis using a global scheduling and allocation scheme that proceeds by stepwise refinement. The *force-directed* scheduling algorithm at the heart of this scheme reduces the number of functional units, storage units, and buses required by balancing the concurrency of operations assigned to them. The algorithm supports a comprehensive set of constraint types and scheduling modes. These include:**

- **multicycle and chained operations;**
- **mutually exclusive operations;**
- **scheduling under *fixed global timing constraints* with:**
  - **minimization of functional unit costs,**
  - **minimization of register costs,**
  - **minimization of global interconnect requirements;**
- **scheduling with local time constraints (on operation pairs);**
- **scheduling under *fixed hardware resource constraints*;**
- **functional pipelining;**
- **structural pipelining (use of pipelined functional units).**

Examples from current literature, one of which was chosen as a benchmark for the 1988 High-Level Synthesis Workshop, are used to illustrate the effectiveness of the approach.

I. Introduction

As LOGIC and RTL-level synthesis tools gain a stable foothold in industry, the automatic synthesis of a digital system from a behavioral description—*behavioral or high-level synthesis*—is the next step on the ladder of the design automation hierarchy. As demonstrated by the recent flurry of activity in this area [1]–[7], [9], [11], [13]–[15], [17]–[33], [35]–[47], behavioral synthesis is becoming an increasingly popular research topic. The interest is a natural consequence of the shift of the IC designer's involvement away from device-level considerations and toward architectural ones.

Behavioral synthesis is commonly achieved by dividing the task into a data path design and a control path design. Scheduling data path operations into the best control steps is a task whose importance has been recognized in many systems [1]–[4], [7], [22], [25], [41]. According to Gajski [1], it is "perhaps the most important step during the architecture synthesis."

Manuscript received December 18, 1987; revised July 22, 1988, and December 16, 1988. This work was supported in part by the Natural Sciences and Engineering Research Council of Canada and by Bell-Northern Research. It was realized as part of a cooperative Ph.D. research agreement between P. Paulin, Carleton University, and Bell-Northern Research. The review of this paper was arranged by Associate Editor M. R. Lightner.

P. G. Paulin is with Bell-Northern Research, P.O. Box 3511, Stn. C. Ottawa, Ont., K1Y 4H7, Canada.

J. P. Knight is with the Electronics Department, Carleton University, Ottawa, Ont. K1S 5B6, Canada.

IEEE Log Number 8926971.

Operation scheduling determines the serial/parallel trade-offs of the design, which approximately determines the cost–speed trade-offs [5]. If the design is subjected to a speed constraint, the scheduling algorithm will attempt to make sufficient operations run in parallel to meet the constraint. Conversely, if there is a limit on chip area, the scheduler can be asked to serialize operations to give the maximum speed consistent with the constraint.

The major purpose of this paper is to present a general scheduling methodology that *can be integrated into specialized or general-purpose high-level synthesis systems*. In [7], we presented an initial version of the *force-directed* scheduling algorithm at the heart of this methodology. This algorithm has been taken up and reimplemented by other research groups, both in academia [8], [9] and in industry [10]. In this paper, we will present the latest implementation of the algorithm, which includes a more computationally efficient formulation of the force metric and supports the following new scheduling problems:

- minimization of global storage and interconnect requirements;
- scheduling under fixed hardware resource constraints;
- two forms of pipeline scheduling.

We will start by describing the scheduling task in the wider context of behavior synthesis. This will be followed by a review of existing scheduling techniques. We will then present the *force-directed* scheduling algorithm, which is the main emphasis of this paper. We will show how the scheduling can be optimized for either a speed constraint or a constraint on hardware resources. Extensions for two simple forms of pipelining will also be described. Finally, we present experimental results for design examples taken from current literature.

II. Scheduling in the Context of Behavioral Synthesis

There are several major tasks in the automatic synthesis of digital systems [6]. The first is the *definition of the circuit function* in a high-level hardware description language (HDL). Fig. 1(a) depicts a simple behavioral description that will be used to illustrate the synthesis process. This step is usually followed by a *translation to a graph-based representation* derived from the control and

data flow (the *control-data flow graph*, or CDFG) as shown in Fig. 1(b).

The next step (Fig. 1(c)) is the *operation scheduling*, where operations are assigned a propagation delay value and partitioned into specific control steps (c-steps). This step can be preceded by, followed by, or performed simultaneously with the *allocation of hardware modules*. The latter consists of allocating functional units (FU's) to execute the operations (nodes) of the CDFG and storage units (registers, memories) to store the values (edges). The number of each type of unit is determined, but they are not yet bound to a specific node or edge in the CDFG. Although scheduling and allocation can be performed separately, many systems attempt to link them in order to improve the overall process [2], [4], [7], [26], [28].

The next step is the *data path synthesis*, where the allocated functional units and storage units are bound to specific operations. These structural units are then interconnected by using the data transfer information derived from the CDFG. The synthesis process usually includes an optimization phase which improves the register and interconnect bindings.

The final realization of the data path is done either by assigning the high-level modules to predesigned templates that represent their gate-level structure or by synthesizing them directly with module generators.

The schedule and the list of data path operations are used to create a state graph which specifies the required control signals in each control step. This state graph can be used in turn to generate a controller for the circuit. This controller is often microprogrammed or is a PLA-based finite state machine.

A possible register-transfer circuit for the example is shown in Fig. 1(d). Here, it is assumed that the values A, B, C, D, and E must be preserved and, therefore, the corresponding registers cannot be shared.

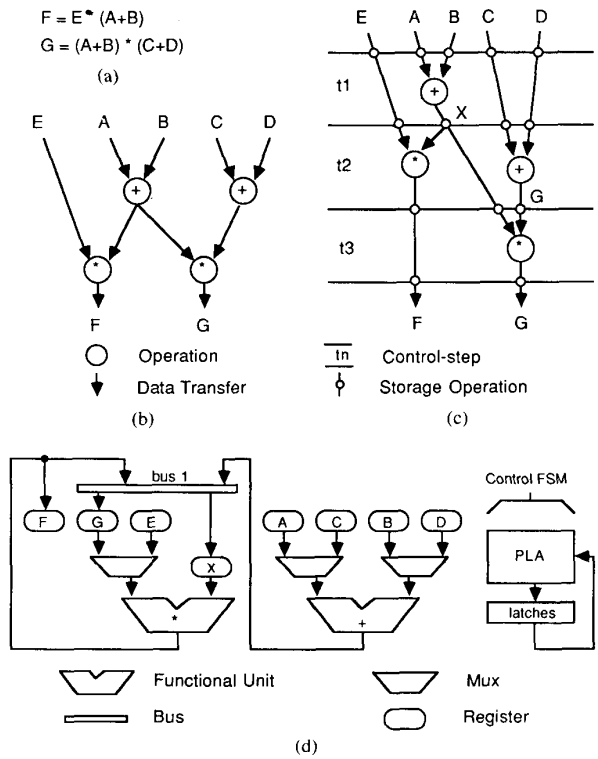


Fig. 1. Synthesis subtasks for a simple example. (a) HDL description. (b) Control data flow graph (CDFG). (c) Scheduled CDFG. (d) RTL architecture.

III. Related Scheduling Approaches for Synthesis

In this section, we describe the scheduling used in many recent synthesis systems. The intention is only to give an overview of the different approaches, not to be exhaustive. Furthermore, space does not permit a review of scheduling algorithms for other areas. These are covered in an excellent survey by Lawler *et al.* [12].

The simplest way to perform scheduling is to relegate the task to the user. This is the approach favored by the Silc system [13] under the assumption that the user should explicitly define the parallelism of the design.

A. Independent Scheduling/Allocation Schemes

The following systems have been classified as independent scheduling and allocation schemes; strictly speaking, however, there is usually some form of interaction. For example, many of them derive estimates for the operation propagation delays based on the most probable allocation of functional units.

The first and simplest scheme is to schedule operations "as soon as possible" (ASAP), as is done in Carnegie Mellon University's (CMU) Emerald/Facet system [14] and in the CATREE system [15] from the University of Waterloo. An example of ASAP scheduling is given in Fig. 3, which will be discussed in Section IV. This technique has proved useful in the past for near-optimal microcode compaction [16].

A refinement of this concept is ASAP scheduling with conditional postponement of operations. In the MIMOLA system [17], postponement occurs whenever the operation concurrency is higher than the number of available functional units. The Flamel system [18], developed at Stanford and AT&T, uses the same idea. The scheme used in the book by Kung, Whitehouse, and Kailath on digital signal processing [19] is similar, except that an operation is postponed when it blocks a later one with a lower as late as possible (ALAP) level. Fig. 4 illustrates the meaning of ASAP and ALAP.

The next, more complex class of algorithms, makes use of the *list scheduling* techniques described in [16]. The EMUCS system, developed at CMU by Hitchcock and Thomas [20]; the behavior synthesis of interfaces (BSI) system, developed by Nestor and Thomas, again at CMU [21]; Pangrle and Gajski's SLICER system, developed at the University of Illinois [22]; and IMEC's Cathedral-II system [23], [24], [25] all use this type of scheduling algorithm.

In list scheduling, operations are sorted in topological order (top to bottom) using the precedences dictated by

data and control dependencies in the CDFG. The sorted operations are then iteratively scheduled into control steps. When a resource conflict occurs due to insufficient hardware, one or more operations are deferred by one control step. The selection of the deferred operation(s) is determined by a *local priority function* that is applied to all operations that could be scheduled in the current control step.

In the BSI system, the priority function reflects whether placing the operation in the current step will violate a minimum time constraint and whether placing an operation in a later step will violate a maximum time constraint. In the SLICER system, the priority function is based on operation *mobilities*. The mobility of an operation is defined as the difference between its ASAP time and ALAP time. Operations on the critical path should therefore be scheduled first. The Atomics scheduler [25] of the Cathedral-II system uses a similar priority function and also includes a heuristic method for scheduling loops in a CDFG.

B. Interdependent Scheduling/Allocation Schemes

In the next two systems, the operation scheduling is done concurrently with the functionl unit allocation. Elf [2] uses a variation of the list scheduling algorithm adapted to fixed timing constraints. Its priority function is based on operation *weights* and *urgencies*. An operation's weight is the number of control steps needed to execute the operation, plus the sum of the weights of all its successor nodes along the "heaviest" path ending in a timing constraint. An operation's urgency is the ratio of its weight divided by the number of control steps left until its latest completion time. When an operation is delayed, its urgency increases, which raises its priority for scheduling in the next control step.

The scheme used in the MAHA system [4], from the University of Southern California, relies on critical path determination and the concept of *freedom* to guide scheduling. The freedom of an operation is identical to the mobility calculated in the SLICER system; in this case, however, list scheduling is not used. The MAHA system first invokes the clocking scheme synthesis package (CSSP), written by Park [3]. Here, the critical path is determined and divided optimally into n steps, one per clock cycle. MAHA then allocates functional units for the critical path in a first-come first-served fashion. The notion of freedom is used to guide the scheduling of nodes not on the critical path. The node with the smallest freedom is chosen for allocation.

C. Scheduling/Allocation by Stepwise Refinement

The BUD-DAA system [26], [27], from AT&T, uses a stepwise refinement approach to the scheduling task. In this system, operations are first partitioned into clusters, using a metric that takes into account potential functional unit sharing, interconnect, and parallelism. The functional units are then assigned to each cluster and the scheduling is done. BUD uses a list scheduling algorithm with *a priori* function that is similar to the one used in the SLICER system.

Camposano's work [28] on the IBM Yorktown Silicon Compiler (YSC) separates scheduling into two steps. In the first step, scheduling is performed by assigning the minimum number of control steps possible. Therefore, single states are created for loops, for module calls, and as required to avoid conflicts over register and memory usage. An initial design is then produced using logic synthesis tools to obtain speed and area estimates. Following this, *state splitting* is used to reduce the cycle time by splitting long control steps. This also allows increased sharing of hardware resources to minimize area.

The HAL system described in this paper also makes use of a stepwise refinement approach, as depicted in Fig. 2. The system does a preliminary allocation and uses that information to establish a schedule estimate. The allocation is repeated using the schedule to perform a much more detailed analysis and an improved selection of FU's based on operation concurrency. The scheduler is then reinvoked and the final schedule is established by optimizing the use of the preselected FU's. Details are given in [7].

The force-directed scheduling algorithm at the heart of this scheme allows specific information, such as the area and speed of a functional unit, to be fed back to optimize the scheduling process.

This algorithm is different in several ways from the ones described earlier:

- It is a more global approach that considers the side effects of control-step (c-step) assignments on three types of operations:
  1) arithmetic and logic operations executed by functional units;
  2) storage operations realized by registers;
  3) data transfer operations realized by muxes and buses.
- Unscheduled operations also contribute to the determination of the operation concurrency; however their contribution is probabilistic rather than deterministic.
- Finally, tradeoffs in functional unit, register, and interconnect requirements are resolved using a built-in cost-weighing mechanism. Using this mechanism, operation scheduling priorities are set according to their associated hardware area costs.

D. Pipeline Scheduling

The optimized scheduling of pipelined behavioral descriptions is a recent development in the field of behavioral synthesis. Two simple types of pipelining have been attempted: *structural pipelining* [22] and *functional pipelining* [29], [30].

In functional pipelining, the algorithm description is subdivided into sequences of operation stages that will be performed concurrently. Successive stages are streamed into the pipe so that *different algorithm instances are executed in an overlapping fashion* on a single data path.

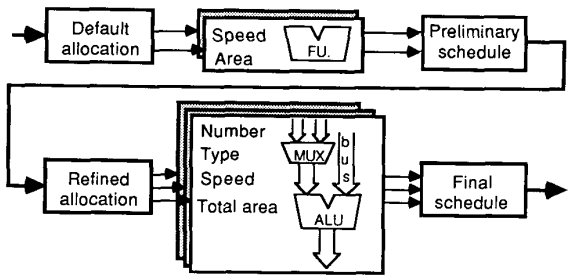


Fig. 2. Scheduling and allocation by stepwise refinement.

This is illustrated in Fig. 12, where two algorithm instances are depicted.

Park and Parker's Sehwa system is one of the first synthesis programs described in the available literature to perform functional pipelining [29]. This system makes use of two polynomial-time pipeline scheduling algorithms:

1) *Feasible scheduling:* Schedule with constraints on the total implementation cost.
2) *Maximal scheduling:* Schedule for maximum performance, that is assuming there is no cost constraint.

Sehwa also incorporates an exhaustive algorithm for optimal scheduling. Here the search time is reduced by using the feasible or maximal schedule as an upper bound. The algorithms are invoked iteratively, and each scheduling cycle is guided by the performance and cost estimation for the previous schedule.

Pipeline scheduling has also been demonstrated in the current version of the Elf system [30], developed at Audesyn Inc. Here, *loop winding* is used to realize an alternate form of functional pipelining. Loop winding partitions each loop iteration according to path length and then winds these in parallel to form a shorter loop executing at a higher frequency. User-supplied throughput and latency constraints are used to determine the length of the shorter loop.

Scheduling is performed by partitioning the critical path operations into equal length stages. The operation *weights* described earlier are used to assign noncritical operations to these stages. This is followed by a refinement phase, where *load balancing* [11] is used to reduce the concurrency of similar operations.

In *structural pipelining*, temporal parallelism is obtained through the use of pipelined functional units, e.g., a two-stage pipelined multiplier. In this case, *the operation instances are executed in an overlapping fashion.*

The Slicer system [22], mentioned in the previous subsection, was one of the first systems to perform scheduling optimizations with pipelined functional units. As in regular scheduling, the Slicer systems uses *mobilities* to guide the pipeline scheduling process.

Functional and structural pipeline scheduling are supported in HAL using simple modifications of the force-directed scheduling algorithm that will be described in Section VII.

E. Summary of Scheduling Approaches

The scheduling approaches can be summarized as follows:

- Independent scheduling/allocation:
  - ASAP scheduling approaches:
    - Direct (Facet-Emerald, CATREE).
    - Conditional deferment (MIMOLA, Flamel, [19]).
  - List scheduling approaches:
    - Priority function: time constraints (BSI).
    - Priority function: mobility (Slicer).
- Interdependent scheduling/allocation:
  - User-defined schedule (Silc).
  - List scheduling, priority function: urgency (Elf).
  - Freedom and critical path scheduling (MAHA).
- Stepwise refinement approaches:
  - List scheduling after design partitioning (BUD-DAA).
  - State splitting after initial logic synthesis (YSC).
  - Force-directed scheduling (HAL).

The pipeline scheduling approaches can be distinguished using the following classification:

- Functional pipelining:
  - Feasible, maximal or exhaustive scheduling (Sehwa).
  - Loop winding using operation weights (Elf).
  - Modified force-directed scheduling (HAL).
- Structural pipelining:
  - List scheduling using mobilities (Slicer).
  - Modified force-directed scheduling (HAL).

IV. Basic Force-Directed Scheduling Algorithm

The intent of the force-directed scheduling algorithm is to reduce the number of functional units, registers, and buses required by *balancing the concurrency of the operations assigned to them* but without lengthening the total execution time. Concurrency balancing helps to achieve high utilization—or low idle time—of structural units, which in turn minimizes the number of units required. The algorithm is iterative, with one operation scheduled in each iteration. The selection of the control step in which it will be placed is based on achieving a balanced distribution of operations in each control step.

In this section, we will describe a simplified version of the algorithm that attempts to balance the concurrency of arithmetic and logic operations only. Furthermore, we will temporarily assume that all operations execute in one control step (we will use the term *c-step* from now on). Chaining (placing two data-dependent operations in the same c-step) and multiple c-step operations are not considered. The complete algorithm, which does not have these restrictions, will be presented in Section V.

A. Determination of Time Frames

The first step involves the determination of both an ASAP (as soon as possible) schedule and an ALAP (as

late as possible) schedule. Combining results for both schedules will determine the *time frame* of each operation.

To illustrate this, we will use the example first presented in [11] and subsequently used in [15], [22], [31], and [32]. The algorithmic description and the corresponding control data flow graph (CDFG) for this example are given in Fig. 3.

The raw CDFG would not have operations scheduled in time (control steps). Fig. 3 shows ASAP scheduling. ASAP versus ALAP scheduling are shown in simplified form in Fig. 4(a) and (b) respectively.

By noting that an operation can be assigned to any c-step between its ASAP and ALAP c-steps, one can draw a time frame diagram as shown in Fig. 5. Here, the width of the box containing a particular operation represents the probability that the operation will eventually be placed in a given time slot. A useful heuristic is to assume uniform probability of assigning an operation to any feasible c-step. The area assigned to each operation is always one, but it is stretched along its time frame.

This method of assigning probabilities is identical to the one used in Nagle's *attraction* algorithm [33]. The resemblance ends there, however, as the attraction algorithm is used for a very different application, i.e., the synthesis of a control path for a predefined data path. As we will see in the next subsections, probabilities are used in the HAL system in a different way and for a different purpose—namely the minimization of data path costs.

B. Creation of Distribution Graphs

The next step is to take the summation of the probabilities of each type of operation for each c-step of the CDFG. The resulting *distribution graphs* (DG's) indicate the concurrency of similar operations. For each DG, the distribution in c-step i is given by

$$\mathrm{DG}(i) = \sum_{Opntype} \mathrm{Prob}(Opn, i) \qquad (1)$$

where the sum is taken over all operations of a single type. $\mathrm{Prob}(Opn, i)$ is the probability of an operation in c-step i. The distribution graphs derived from Fig. 5 are shown in Fig. 6.

The first graph represents the distribution of the multiply operations, and the next one combines the distributions of the add, subtract, and compare operations. The latter three operations are actually assigned to separate DG's but are grouped here for brevity. Each horizontal bar of the DG's corresponds to a distinct c-step. The boxed c-steps delineate the time frames where operations can occur. The unboxed c-steps in the *Multiply DG* indicate that multiply operations could never take place in the fourth c-step without extending the critical path of the CDFG.

1) Conditional Statements: If-then-else and *case* statements cause forks in the CDFG as shown in Fig. 7. Operations in the different branches of a fork are mutually exclusive. When operations in different branches can be executed on the same type of FU, they can be scheduled into the same c-step without increasing the required number of FU's. The same FU is simply shared by those operations as they will never execute concurrently.

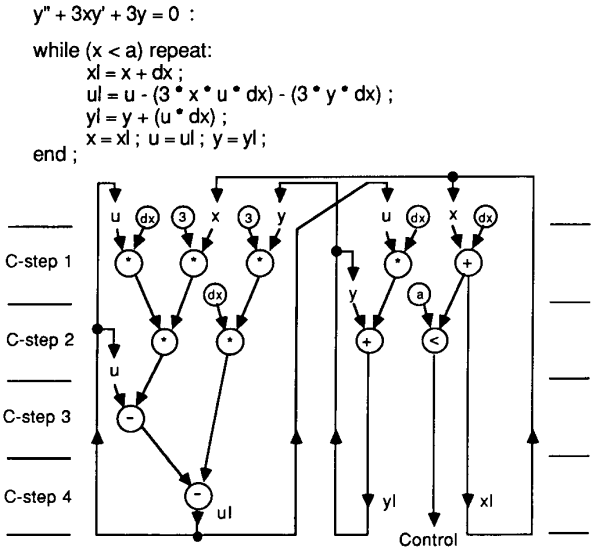


Fig. 3. Control data flow graph (CDFG) for first example.

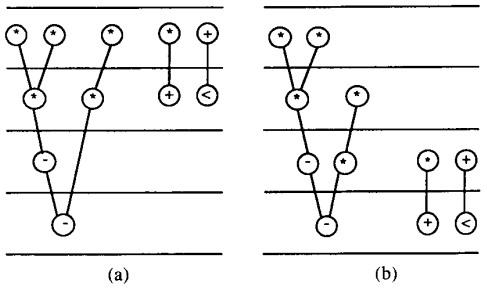


Fig. 4. (a) ASAP and (b) ALAP schedules.

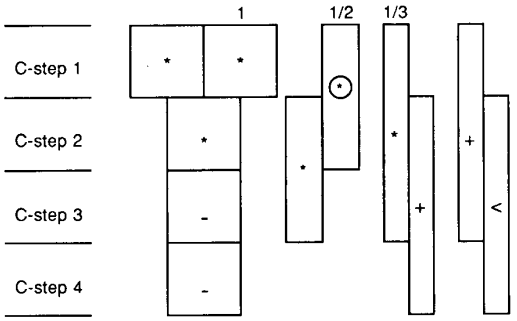


Fig. 5. Time frames of operations (initial state).

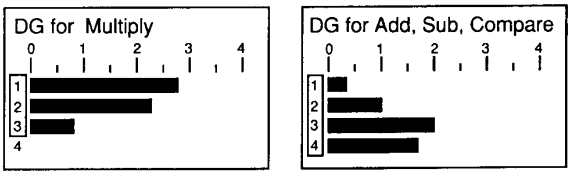


Fig. 6. Distribution graphs (initial state).

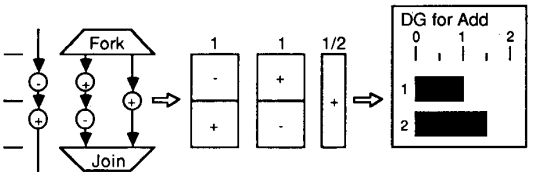


Fig. 7. Distribution evaluation for CDFG with conditional statement. (a) DFG). (b) Time frames. (c) DG.

We use the following strategy to take advantage of this observation. For each c-step in which the time frames of the mutually exclusive operations intersect, the probability of only one of these is added to the corresponding DG. The operation selected is the one with the highest probability. This is illustrated for the CDFG fragment of Fig. 7.

Without special treatment of the mutually exclusive additions, the total distribution would be 1.5 in both c-steps. The unscheduled addition would then have an equal probability of being assigned to either c-step. It is obviously preferable to schedule it in the first c-step, as in this case only one adder will be required. That is exactly what will happen using the strategy just described due to the reduced distribution in the first c-step.

2) Trace Scheduling Transformations: In [34], Fisher described a series of transformations used in *trace scheduling*. These transformations involve the displacement of operations in and out of control blocks to improve performance. For example, an operation that is not explicitly defined as part of a block may still be scheduled in the same c-step as an operation in the block. This may be done to shorten the critical path. In this case, the operation is implicitly duplicated in all branches of the case statement and becomes part of these blocks.

This is illustrated in Fig. 7, where the *add* and *subtract* operations are implicitly duplicated in both branches of the case construct. In the HAL system, *if-then else* and *case* control blocks do not have rigid boundaries so that operations can be freely moved in and out of them (as long as data dependencies are still respected). The side effects of these moves are taken into account by selective addition of the operation probabilities to the DG as described above. Further details are given in [35].

C. Calculation of "Self" Forces

Each operation of the CDFG will have a *self force* associated with each c-step i of its time frame. This is a quantity which reflects the effect of an attempted control step assignment on the overall operation concurrency. It is positive if the assignment causes an increase of operation concurrency, and is negative for a decrease.

The force is much like that exerted by a *spring* that obeys Hooke's law: $F = Kx$. K represents the spring's constant (rigidity), x the displacement, and F the force that caused the displacement.

More precisely, each DG can be represented as a series of springs (one for each c-step) that will exert forces on all operations. The constant of each spring K is given by the value of $DG(i)$, where i is the c-step number for which the force is calculated. The displacement of the spring x is given by the increase (or decrease) of the probability of the operation in the c-step due to a rescheduling of the operation. For a given operation whose initial time frame spans c-steps t to b ($t \leq b$), the force in c-step i is given by

$$\text{Force}(i) = \text{DG}(i) * x(i) \tag{2a}$$

where $DG(i)$ is the current distribution value (i.e., the spring's constant) and $x(i)$ is the change in the operation's probability (i.e., the spring's displacement).

The total *self force* associated with the assignment of an operation to c-step j ($t \leq j \leq b$) is simply

$$\text{self Force}(j) = \sum_{i=t}^{b} [\text{Force}(i)] \tag{2b}$$

We will illustrate this equation by using the partial time frame diagram of Fig. 8. The constant of each of the three springs corresponds to the value of the multiplication DG given in Fig. 6 for the first three c-steps. We will attempt to schedule the circled multiply operation in c-step 1 as depicted in Fig. 8(b). The probability of the operation will change from 1/2 to 1 in c-step 1 and from 1/2 to 0 in c-step 2. These probability shifts correspond to the displacement x of each of the springs. The resulting force associated with the assignment to c-step 1 is the sum of forces caused by the spring displacement in both c-steps of the time frame and is given by

$$\begin{aligned}\text{self Force}(1) &= \text{Force}(1) + \text{Force}(2) \\ &= (\text{DG}(1) * x(1)) + (\text{DG}(2) * x(2)).\end{aligned}$$

Using the values from Figs. 6 and 8, we obtain

$$\begin{aligned}\text{self Force}(1) &= (2.833 * +0.5) + (2.333 * -0.5) \\ &= +0.25.\end{aligned}$$

The force is positive, as expected, because the concurrency in c-step 1 is higher than in c-step 2. Scheduling the multiplication in that c-step will have an adverse effect on the overall distribution.

Alternative Interpretation of Force: An alternative interpretation of the meaning of *force* follows from the evaluation of (2b). The total *self force* associated with the assignment of an operation to c-step j is simply

$$\text{self Force}(j) = \text{DG}(j) * x(j) + \sum_{\substack{i=t \\ i \neq j}}^{b} [\text{DG}(i) * x(i)]. \tag{2c}$$

Evaluating the displacements $x(j)$ and $x(i)$ yields

$$x(j) = (h - 1)/h$$

and

$$x(i) = -1/h$$

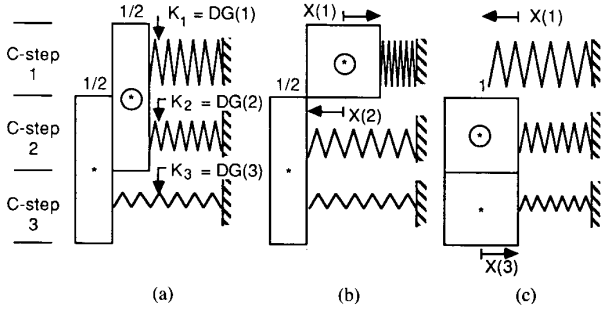


Fig. 8. Time frame modifications for force calculations.

where

$$h = (b - t + 1)$$

is the time frame height. After rearrangements, we obtain

$$\text{self Force}(j) = \text{DG}(j) - \sum_{i=t}^{b} [\text{DG}(i)/h]. \quad (2d)$$

In other words, the force associated with the tentative assignment of an operation to c-step j is equal to the difference between the distribution value in that c-step and the average of the distribution values for the c-steps bounded by the operation's initial time frame. In Fig. 8(b), the use of equation (2d) yields

$$\text{self Force}(1) = \text{DG}(1) - \sum_{i=1}^{2} [\text{DG}(i)/2]$$
$$= 2.833 - (2.833 + 2.333)/2 = +0.25$$

which of course is identical to the result obtained earlier.

For the more general case, namely the force associated with the reduction of an initial time frame (bounded by c-step t and b) to a new time frame (bounded by c-steps nt and nb), we may use the following equation:

$$\text{Force}(nt, nb) = \sum_{i=nt}^{nb} [\text{DG}(i)/(nb - nt + 1)] - \sum_{i=t}^{b} [\text{DG}(i)/(b - t + 1)]. \quad (2e)$$

Each sum represents the average of the distribution values for the c-steps bounded by the time frame. The force is therefore equal to the difference between the average distribution for the c-steps bounded by the new time frame and the average for the c-steps of the initial one.

This formulation of the force equation is more computationally efficient for two reasons. The first is that the initial average distribution need be calculated only once when evaluating the force associated with all c-steps for the time frame. The second is that it involves mostly subtractions. On the other hand, due to its more intuitive physical interpretation, we will use the former equations ((2a) and (2b)) in the remainder of the text.

D. Calculation of Predecessor and Successor Forces

Assigning an operation to a specific c-step will often affect the time frames of linked operations in the CDFG. This is because scheduling an operation is equivalent to reducing its time frame to one c-step. This modification will be propagated to the time frames of the predecessor and/or successor operations. In turn, this will create additional forces that can reduce or even counter the originally intended improvement, *so it is imperative that they be accounted for.*

This is achieved by calculating the extra forces due to the implicit modifications of the time frames of linked operations. They shall then be added to the *self* force. There will be two extra force contributions: the *predecessor* forces and the *successor* forces.

For example, if the circled *multiply* operation in Fig. 8(a) was tentatively assigned to c-step two, as in Fig. 8(c), the succeeding *multiply* operation would implicitly be assigned to c-step 3. The time frames of the other operations would not be affected. The resulting force associated with that assignment would then be the sum of the two individual forces. The *self* force of the first multiply would be given by:

$$\text{self Force}(2) = (\text{DG}(1) * x(1)) + (\text{DG}(2) * x(2))$$
$$= (2.833 * -0.5) + (2.333 * + 0.5)$$
$$= -0.25.$$

The *successor* force due to the implicit assignment of the second multiplication to c-step 3 would be given by

$$\text{succ Force}(3) = (\text{DG}(2) * x(2)) + (\text{DG}(3) * x(3))$$
$$= (2.333 * -0.5) + (0.833 * +0.5)$$
$$= -0.75.$$

The resulting force due to the c-step assignment is

$$\text{Total Force}(2) = \text{self Force}(2) + \text{succ Force}(3)$$
$$= -1.$$

This c-step assignment is more effective than the one attempted earlier, and this is reflected clearly by the force calculations.

Algorithm Summary: The previous steps can be summarized as follows:

Repeat until (all operations scheduled):
- Step 1: Evaluate time frames:
  - 1.1. Find ASAP schedule.
  - 1.2. Find ALAP schedule.
- Step 2: Update *distribution graphs*, using equation (1).
- Step 3: Calculate *self* Forces for every feasible control-step, using equation (2).
- Step 4: Add *predecessor* and *successor* forces to *self* forces.
- Step 5: Schedule operation with lowest force; set its time frame equal to the selected c-step.

End Repeat.

E. Look-Ahead

Look-ahead is a useful but sometimes costly method of improving an algorithm's effectiveness. In the case of force-directed scheduling, the extra computing costs could not be justified by the small improvements obtained using even a simple one-stage look-ahead. However, the simple modification presented here can be applied to approximate a simple look-ahead scheme, without increasing the algorithm's complexity. This modification has proved to be extremely important as it has *improved the algorithm's effectiveness considerably*.

The *self* force calculation (2a) given earlier can be modified to partially reflect the effect of an assignment on the future value of the DG. The idea is to *temporarily* modify the spring constant $DG(i)$ so that it takes on a value somewhere between the current one and the value that would be obtained *after* the current iteration. Applying this reasoning, the following improved force calculation was obtained after extensive experimentation:

$$\begin{aligned} \text{Force}(i) &= \text{temp\_DG}(i) * x(i) \\ &= \big(\text{DG}(i) + x(i)/3\big) * x(i) \qquad (2a') \end{aligned}$$

which replaces (2a) presented earlier. The force calculation for the example of Fig. 8(b) is now given by

$$\begin{aligned} &\text{Total Force}(1) \\ &\quad = \text{self Force}(1) \\ &\text{self Force}(1) \\ &\quad = \text{Force}(1) + \text{Force}(2) \\ &\quad = \text{temp\_DG}(1) * x(1) + \text{temp\_DG}(2) * x(2) \\ &\quad = \big(\text{DG}(1) + x(1)/3\big) * x(1) + \big(\text{DG}(2) \\ &\qquad + x(2)/3\big) * x(2) \\ &\quad = (2.833 + 0.5/3) * +0.5 \\ &\qquad + (2.333 - 0.5/3) * -0.5 \\ &\quad = +0.41667 \text{ (the previous value was } + 0.25). \end{aligned}$$

This higher value will further reduce the likelihood that the circled multiply operation will be scheduled in c-step 1.

Fig. 9 depicts the final schedule and the distributions obtained for the example. It is easily seen that the operation distributions are optimal. The result was obtained in just three iterations, although there were twice as many operations to schedule. This is due to the global nature of the algorithm and the use of the look-ahead factor, which acts much like *overrelaxation* in the iterative solution of linear equations.

F. Complexity Considerations

The algorithm described above is $O(cn^3)$ when implemented in a straightforward fashion. Here, c is the time constraint expressed in c-steps, and n is the total number of operations. This complexity can be derived as follows:

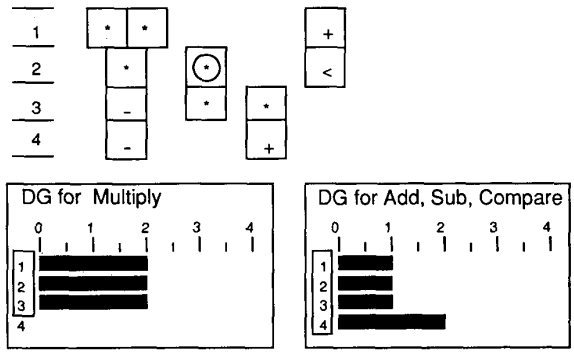


Fig. 9. Final time frames and DG's.

1) Each iteration of the algorithm schedules at least one operation. This implies there can be at most n iterations. However, in most iterations many other operations are scheduled implicitly, so that can be considered a very conservative upper bound.
2) Within each iteration, there are n operations for which forces must be calculated.
3) For each of these operations, the force must be calculated for h control steps, where h is the height of the operation's time frame. In the worst case, where all operations are totally independent, the maximum possible time frame height is equal to c, the global time constraint. Typically, though, operations are interdependent so that the average time frame height is a fraction of c.
4) Finally, for each tentative operation to c-step assignment, there may be at most $n - 1$ predecessor/successor operations affected, and their force must also be calculated.

The combined effect of these four considerations yields the $O(cn^3)$ complexity stated above.

Reducing the Complexity: Fortunately, we can apply two methods to reduce the complexity substantially. The first is to perform a preliminary reduction of all time frames which exceed a constant maximum allowable height H (for example, $H = 10$ c-steps). Forces are calculated in the usual fashion, and all long time frames are reduced simply by removing from the time frame the c-steps with the highest forces. The time frame reduction step is $O(cn)$ and the scheduling phase that follows is now reduced to $O(Hn^3)$, where H is a predefined constant.

The second method is to evaluate the forces of predecessor and successor operations differently. In this method, the *self*, *predecessor*, and *successor* forces are calculated in three separate phases. The first phase consists of *calculating and storing the self* force of all operations. In the second phase, the graph is traversed from bottom to top and each operation "queries" its *immediate successors* for their stored force. In turn, their stored force is made equal to the sum of their own *self* force and the stored force of their immediate successors. We are effec-

tively performing a running sum of all the successor forces in linear time. In the third and final phase, the process is repeated by traversing the graph from top to bottom and performing a running sum of the predecessor forces.

This second method reduces the complexity to $O(cn^2)$. Therefore, for problems in which scheduling with a maximum H is feasible, one obtains a complexity of $O(n^2)$. These methods and the complexity calculations are described in further detail in [35].

V. Refined Scheduling Algorithm

The basic algorithm just presented is a powerful means of solving a relatively narrow class of problems. To make it usable for the wide range of constraints that a good synthesis tool should address, it was necessary to incorporate a few important refinements. However, we will see that the basic framework was extended in a straightforward fashion to include every refinement presented here. The refinements can be divided into four broad classes:

1) Refinements that extend the algorithm's scope:
   - minimization of bus costs,
   - minimization of register costs,
   - local timing constraints,
   - loops.
2) Refinements that ease the inclusion of allocation information into the scheduling process:
   - incorporation of structural unit area costs,
   - multiclass DG's (for ALU's),
   - chained and multicycle operations.
3) Refinements to allow scheduling under fixed area constraints.
4) Refinements for pipeline scheduling:
   - Extension for functional pipelining,
   - Extension for structural pipelining.

This section will deal with the first two classes, while the area constraints and pipeline extensions will be presented in Sections VI and VII.

A. Minimization of Bus Costs

As we mentioned earlier, scheduling has an effect on the interconnect cost of the final circuit. For example, the minimum number of buses required is directly related to the number of concurrent data transfers in a given c-step. Scheduling an operation implies that at least two transfers will occur in the c-step to which it is assigned. The first transfer is from the output of the FU to the register storing the result of the operation it performed. The others are from the registers that supply the FU with its operands. The minimum number of buses required for a given schedule can be found by taking the maximum over all c-step of the number of simultaneous data transfers.

To minimize the number of concurrent transfers and the associated bus costs, we create a special DG that contains the distributions of the data transfers. We will refer to this as the *transfer DG*. Since transfers are directly related to operations, the transfer DG is simply the sum of every operation distribution multiplied by the combined number of *distinct*[1] inputs and outputs:

$$\text{Trans DG}(i) = \sum_{Opn\,type} [\text{Prob}(Opn, i) * Opn\_\text{No\_InOuts}] \quad (3)$$

where Prob (Opn, i) is the probability of an operation in c-step i, and Opn_No_InOuts is the combined number of distinct inputs and outputs for Opn. The additional forces due to these new DG's will be calculated in the same manner as for the regular operations.

B. Minimization of Register Costs

The minimum number of registers required for the implementation of a scheduled CDFG is given by the largest number of data arcs traversing a control step boundary. For a CDFG in which many or all of the operations are unscheduled, it is more difficult to establish this lower bound.

In this subsection, we will present a technique that reduces register costs and also determines a lower bound on the register costs, at any stage of the scheduling process.

To achieve register minimization, we must create a new class of operations that we call *storage operations*. A storage operation is created at the output of every source operation that transfers a value to one or more destination operations in a later control step. We will also need another special DG that will be referred to as a *storage DG*.

When both the source and the destinations of a storage operation are scheduled, then the storage operation's distribution coincides with its *lifetime*. In other words, the distribution is bounded by the c-step of the source operation and the c-step of the last destination operation. The distribution value is equal to one, indicating that it must exist with probability *one* for the entire lifetime.

If one or more of the source or destination operations are not scheduled, then we have to determine probabilistic distributions. This can be done in much the same way as for regular operations. The only complication is that, as opposed to regular operations, the length of the storage operation, i.e., the length of its lifetime, is dependent on the final schedule.

For example, in the simple data flow graph of Fig. 10 there are three possible lifetimes for storage operation **S**, as depicted in parts (a), (b), and (c) of the figure. This operation stores the value of the source operation **s** and transfers it to the destination operation **d**.

The following simple but effective heuristic can be used to quickly estimate the resulting storage distribution. The first step is to determine the lifetimes associated with the ASAP and ALAP schedules (*ASAP life* and *ALAP life*) as well as the longest possible lifetime (*max life*). An estimate of the average length of all possible lifetimes is given

[1]The word *distinct* is emphasized here because several operations may have common inputs. When this occurs, each common input should be counted only once. For example, in the first c-step of Fig. 3 there are only five distinct inputs (i.e., u, x, y, dx, and 3) for the five operations. A common input should preferably be linked to the operation with the highest probability.

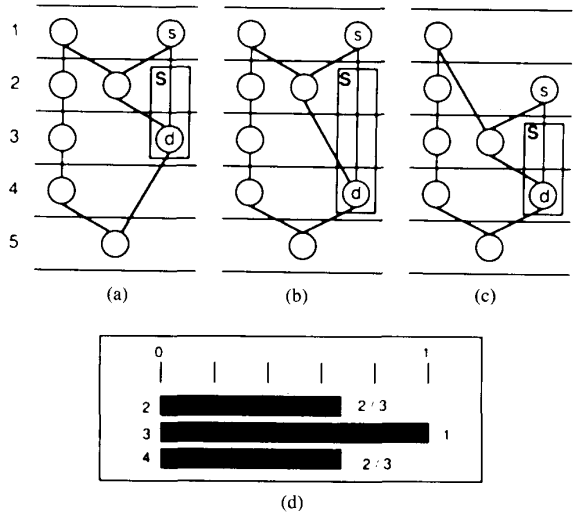


Fig. 10. Storage lifetimes for all schedules. (a) ASAP lifetime. (b) Maximum lifetime. (c) ALAP lifetime. (d) Storage distribution (for storage S).

by

$$[\text{avg life}] = \frac{[\text{ASAP life}] + [\text{ALAP life}] + [\text{max life}]}{3} \tag{4}$$

where

$[\text{ASAP\_life}]$ = length of lifetime for ASAP schedule

$[\text{ALAP\_life}]$ = length of lifetime for ALAP schedule

$[\text{max\_life}]$ = length of the longest possible lifetime

$= (\text{ASAP\_life\_begin} - \text{ALAP\_life\_end} + 1).$

For our example (Fig. 10), the ASAP, ALAP, and maximum lifetimes of the storage operation S are

$$\text{ASAP\_life} = (2, 3) \Rightarrow [\text{ASAP\_life}] = 2$$

$$\text{ALAP\_life} = (3, 4) \Rightarrow [\text{ALAP\_life}] = 2$$

$$\text{max\_life} = (2, 3, 4) \Rightarrow [\text{max\_life}] = 3.$$

Therefore, (4) yields

$$[\text{avg life}] = (2 + 2 + 3)/3 = 7/3.$$

At this point, there are two possible situations. The first and simplest is when there is no overlap between the ASAP and ALAP lifetimes. In this case the storage distribution in c-step i is given by

$$\text{storage DG}(i) = \frac{[\text{avg life}]}{[\text{max life}]} \tag{5}$$

where

$$(\text{ASAP\_life\_begin} \leq i \leq \text{ALAP\_life\_end}).$$

The second situation is when the ASAP and ALAP lifetimes overlap. This is the case in Fig. 10, where

$$\text{overlap} = (3, 3) \Rightarrow [\text{overlap}] = 1.$$

The overlap represents the c-steps where it is certain that a storage operation will be required. Therefore, for each c-step of the overlap, the distribution must be set equal to one. These overlaps can be determined at any intermediate stage of the scheduling process. This is extremely useful as they can also be used to *set a lower bound on the number of registers required* to implement the CDFG, even if any or all of the operations have not been scheduled.

When taking overlaps into account, the storage distribution in c-step i is now given by

$$\text{storage DG}(i) = \frac{[\text{avg life}] - [\text{overlap}]}{[\text{max life}] - [\text{overlap}]} \qquad (i \text{ outside of overlap})$$

or

$$\text{storage DG}(i) = 1 \qquad (\text{for } i \text{ inside overlap}). \tag{6}$$

For our example, using the lifetimes and the overlap calculated earlier, (6) yields

$$\text{storage DG}(2) = \frac{7/3 - 1}{3 - 1} = 2/3$$

$$\text{storage DG}(3) = 1 \text{ (due to overlap)}$$

$$\text{storage DG}(4) = 2/3.$$

The resulting storage distribution is depicted in Fig. 10(d).

This process is repeated for all storage operations and the separate distributions are added to a single *storage DG*. Using the scheduling process described in Section IV, the additional force exerted by these new DG's can be calculated in much the same way as that presented for regular forces, using (2). This force is added to an operation's *self* force by applying a mechanism similar to the one used for predecessor and successor forces.

In Fig. 11, we compare the register requirements for our differential equation example using a simple ASAP schedule and the force-directed schedule. For the ASAP scheduling at least seven registers are required as there are seven values to be stored at the end of c-step one. For the force-directed schedule, this number is reduced to five.

The final assignment of storage operations to registers is performed separately. The HAL system makes use of the approach described in [35] and [36]. Alternative approaches [37], [38] could also be used.

C. Local Timing Constraints

In certain classes of applications, it is useful to specify minimum and/or maximum timing constraints between operation pairs. This has been shown to be particularly useful in the behavioral synthesis of interfaces [21].

This is implemented in the HAL system as a dummy *timing operation* that links two operations via control edges similar to the ones used in case and loop constructs.

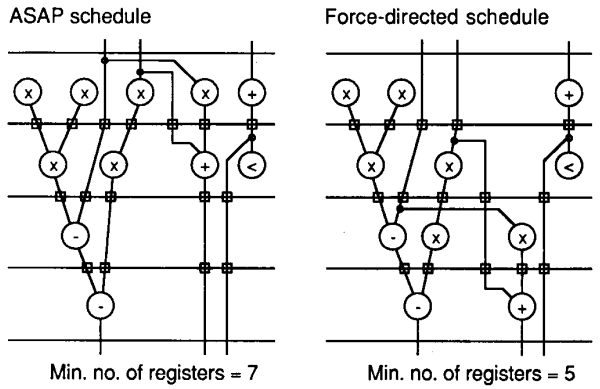


Fig. 11. Register requirements for ASAP and force-directed schedules.

These timing operations can be inserted between any two regular operations; they specify that one must be scheduled before, after, or concurrently with the other. A pair of minimum and/or maximum timing constraints can be associated with each timing operation. They are taken into account during the time frame determination phase of the scheduling algorithm

D. Scheduling with Loops

Whenever a loop is specified in a behavioral description, the user must also specify a constraint on the loop iteration time or alternatively, a constraint on the number of structural units available. A loop timing constraint is implemented in the CDFG by asserting a local maximum timing constraint between the *Begin* and *End* operations that delimit the loop. The operations bounded by the loop are scheduled within the constraint using the regular force-directed algorithm.

Rather than specifying a time constraint, the user may instead specify a constraint on the number of structural units available. In this case, the scheduling extension described further in Section VI will be exploited to generate a schedule, restricted to the given number of structural units, with a minimal delay through the loop body.

For multiple embedded loops, the operations of the innermost loop are scheduled first, relative to the local time constraint (or hardware constraint). When this is done, the entire loop is treated as a single operation with an execution time that is equal to the loop's local time constraint. Operations external to the loop cannot be scheduled concurrently with loop operations. This process is repeated for all loops until the outermost loop is scheduled.

To force pipelining, the user can place a local time constraint on a loop which is shorter than the total execution time of the operations on the longest path through the loop. When this occurs, the scheduling is done using the extension that will be described in Section VII. Here, the latency is set equal to the time constraint. In other words, any operation that is scheduled into a c-step that is greater than the latency will actually be performed concurrently with some of the operations in the next iteration.

This entails that operations from the current iteration will affect the time frames of those in the next one. This is taken into account in the time frame determination process. This form of pipelining is similar to *loop winding* as presented by Girczyc [30] as part of the current version of the Elf system.

E. Incorporation of Structural Unit Area Costs

Different structural units have different realization costs. Operation types associated with high-cost units should be given a higher priority in the scheduling process. An easy way to do this is to multiply the constant of the spring associated with the DG by a *cost factor* that reflects the unit's area. For example, the cost factor of the multiplication DG would be roughly[2] equal to the allocated multiplier area, while that of the storage DG is equal to the register area. This is a very simple mechanism that allows the algorithm to *automatically perform trade-offs between structural units of different costs*. This capability is particularly useful in the early evaluation phase of a design.

F. Multiclass DG's

In many cases we want to make effective use of multifunction units (e.g., ALU's). To deal with this, the concept of distribution graphs (DG's) was extended so that the distribution of one or more operation types can be stored in a single DG. For example, if the allocator assigns additions and subtractions to an ALU, then a two-class DG will be created in lieu of the two single-class ones. Therefore, these two types of operations will tend to be scheduled in different c-steps to make best use of the ALU.

For each multiclass DG, the distribution in c-step i is given by

$$\text{Multi DG}(i) = \sum_{type} \sum_{Opn} \text{Prob}\,(Opn, i) \qquad (7)$$

where the sum is taken over all operations of all specified types and where Prob (Opn, i) is the probability of an operation in c-step i. For example, if an ALU were allocated to perform the add, subtract, and compare operations of our previous example, then we would obtain the multiple DG shown earlier in Fig. 6 of the previous section.

G. Chained Operations

The possibility of scheduling consecutive data-dependent operations in a single c-step (chaining) has also been incorporated into the system. This feature is implemented in a straightforward fashion by extending the time frames of fast combinatorial operations into the previous and/or next c-steps (when the total propagation delay in those c-steps is less than the clock cycle).

[2]In [35], an alternate cost factor is used for FU's. This cost factor is also a function of the associated mux, bus, and interconnect area. For example, simple logical operations (AND, OR, etc.) that make use of low-area FU's (relative to interconnect) will have a near-zero cost factor. Therefore, their scheduling priority will be nearly nil.

To determine if operations can be chained, a *propagation delay* value must be stored with each one. When establishing the ASAP schedule, this value represents the running sum of propagation delay values of all the predecessor operations that may be scheduled in the same c-step. Conversely, for the ALAP schedule it represents the sum of delays for all the successor operations in the same c-step.

H. Multiple Control Step Operations

Operations that require multiple c-steps for execution are also supported. They are implemented with a simple extension of the single c-step methodology. This extension takes into account the effect on the time frames evaluation, and requires a new method for calculating the distributions of multiple c-step operations.

1) Determination of Time Frames: The time frame determination is modified in a straightforward fashion by extending the approach used for chained operations. For example, let us assume a multiply operation (with a propagation delay of 120 ns) that is linked to an addition (40 ns) in a CDFG where the clock cycle is 100 ns and latch delays are 10 ns. If the multiply operation is scheduled in c-step 1, then the addition cannot be chained in c-step 1. Furthermore, it can only be scheduled in c-step 2 if the sum of the combined propagation delays plus one latch delay is less than two clock cycles (200 ns). Here this sum is equal to 170 ns so the time frame of the addition will start in c-step 2.

2) Evaluation of Operation Distributions: The distributions of multiple c-step operations must be calculated differently since each c-step (or stage) of the operation must be taken into account. For example, if a two c-step multiply operation is scheduled in c-step 1, its distribution (or probability) is equal to one in *both* c-step 1 and c-step 2.

On the other hand, if the multiplication is not scheduled and may begin in either c-step 1 or c-step 2 (i.e., its time frame is equal to two), then the distribution contribution of the operation's first stage is equal to one half in each of those two c-steps. Following the same reasoning, the contribution of the second stage is equal to one half in both c-step 2 and c-step 3. The combined distribution of both stages of the operation is therefore equal to $1/2$, 1, and $1/2$ in c-steps 1, 2, and 3 respectively. The value 1 in the second c-step indicates that one of the two stages *must* be scheduled in that c-step.

VI. Scheduling Under Fixed Hardware Constraints

The scheduling approach just described supports the synthesis of near-minimum cost data paths under fixed timing constraints. The *force-directed list scheduling* (FDLS) algorithm [36] summarized here solves the dual problem: the determination of a schedule with a near-minimal number of c-steps, given fixed hardware constraints. It is based on the well-known *list scheduling* (LS) algorithm [16] as well as on the *force-directed* scheduling (FDS) algorithm presented in the two previous sections.

Recall that in list scheduling, operations are sorted in topological order by using control and data dependencies. The set of operations that may be placed in a c-step may then be evaluated; we call these the *ready* operations. If the number of ready operations of a single type exceeds the number of hardware modules available to perform them, then one or more operations must be deferred. In previous list scheduling algorithms, the selection of the deferred operations is determined by a local priority function such as *urgency* [2] or *mobility* [22].

In force-directed list scheduling, the approach is similar except that *force* is used as the priority function. More precisely, whenever a hardware constraint is exceeded in the course of regular scheduling, force calculations are used to select the best operation(s) to defer. Here, a defferal does not necessarily mean that the operation will be scheduled in the next c-step; only that its time frame has been reduced so that it excludes the current c-step. The deferral that produces the lowest force, namely, the best balancing of concurrency in the graph, is chosen. This is repeated until the hardware constraint is met.

Forces are calculated using the method described in the previous sections. However, as these calculations depend on the existence of time frames, a global time constraint must be temporarily specified. Here, it is simply set to the length of the *current* critical path. This length is increased when the only way of resolving a resource conflict is to defer a critical operation. The FDLS algorithm can be summarized as follows:

1) Initialize time constraint to length of critical path.
2) **for** c-step **from** 1 **to** time constraint **do**:
   - 2.1) Determine time frames.
   - 2.2) Determine *ready* operations in c-step (i.e. opns. whose time frame intersects the current c-step).
   - 2.3) **while** (no. of *ready* opns. > no. of FU's) **do**:
     - **if** all opns. on critical path **then**
       - extend time constraint by 1 c-step,
       - reevaluate time frames.
     - Calculate forces for possible deferrals.
     - Defer operation with lowest force.
     - Remove it from *ready* opns.

     end;
   - 2.4) Schedule remaining *ready* opns. in current c-step.

   end;

Using this approach, the advantages of both types of scheduling are maintained:

1) High utilization of functional units, as this feature is intrinsic to list scheduling.
2) Low computational complexity; the FDLS algorithm has a worst case complexity of $O(n^2)$—as opposed to $O(cn^2)$ for the regular FDS algorithm—and typically exhibits linear behavior.

3) Global evaluation of all the side effects of an attempted operation to c-step assignment. This characteristic is the basis of force-directed scheduling.

A New Design Space Exploration Technique

Taken alone, the new FDLS algorithm allows the user to partially specify a target architecture by setting the number and type of functional units, as well as limits on the total register and bus counts [36]. This flexibility, added to the algorithm's effectiveness, justifies the relatively small effort required to implement it.

However, from these and other experiments, we have found that the most powerful method of exploring the design space is by *making use of both the FDS and FDLS algorithms*. In a first phase, the designer sets a maximum time constraint and uses the FDS algorithm to arrive at a near-optimal allocation. This makes it possible to take advantage of the fact that the FDS algorithm automatically performs trade-offs between functional units of different types and costs by assigning scheduling priorities, as described in Section V-E.

In a second phase, the designer can then focus on that area of the design space by using the FDLS algorithm with the resulting allocation to determine if a faster schedule can be obtained. The reason for this improvement lies in the fact that the scheduler starts out with more information about the design.

Regardless of the method chosen, we have given the designer an added level of flexibility with an integrated scheduling methodology that makes it possible to explore the design space from two dimensions: area or time. This also fulfills our original intention: to devise general algorithms that can be tailored to specific applications. Another advantage—from the implementer's point of view at least—is that most of the subroutines are common to both the FDS and FDLS algorithms.

VII. Extensions for Pipeline Scheduling

In this section, we will address the two types of pipelining described in Section III, namely *functional* and *structural* pipelining.

A. Extension for Functional Pipelining

The task of scheduling a pipelined algorithm is resolved here with a simple and straightforward modification of the regular force-directed scheduling algorithm. For a given latency L, the operations scheduled into c-steps $i + kL (k = 0, 1, 2, \cdots)$ run concurrently. So now we must balance the distribution across all *groups* of concurrent c-steps, as opposed to the previous balancing across all *individual* c-steps. For our previous example, for a latency of two c-steps (i.e. $L = 2$), the concurrent c-step groups are $(1', 3)$ and $(2', 4)$, as shown in Fig. 12.

Our previous schedule (Fig. 9) would now require the allocation of at least four multipliers instead of two. The adder, subtractor, and comparator allocations remains unchanged. This simple example illustrates that the regular scheduling does not guarantee an even distribution of operations across groups of concurrent c-steps.

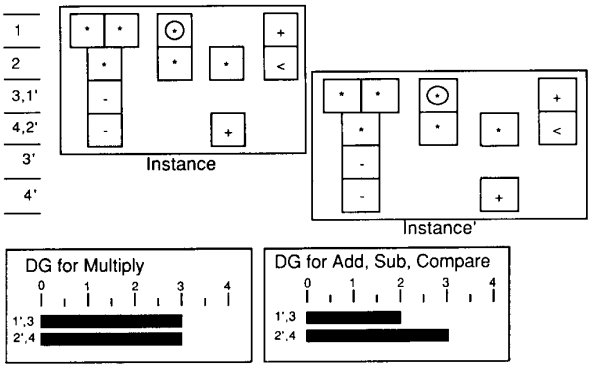


Fig. 12. Schedule obtained using functional pipelining extension.

Fortunately, this can be improved by cutting the distribution graphs horizontally and superimposing the slices. The cut boundary is determined by the value of the latency. In the previous example, the DG would be sliced in the middle and both halves superimposed so that the DG would be reduced to two rows. The first corresponds to the combined distribution in c-steps 1 and 3 (as they are now concurrent and the second to that of c-steps 2 and 4. By performing regular force-directed scheduling with these modified DG's, the operation distribution will be balanced while taking into account the additional level of parallelism due to functional pipelining.

Fig. 12 illustrates the new schedule obtained by using this method. This solution requires three multipliers instead of four and all three are fully utilized. We therefore obtain a twofold increase in throughput while the number of multipliers is increased by a factor of only 1.5 and the adder, subtractor, and comparator allocations are unchanged.

This technique can be used for straight-line code as well as for conditionals and loops. The example presented here is actually a loop, so we have in effect performed *loop winding* as described in [30]. Special considerations for conditionals will be discussed in Section VIII-B.

B. Extension for Structural Pipelining

The problem of scheduling operations assigned to pipelined functional units can be solved with another very simple extension. This extension is based on the fact that once the first stage of a pipelined functional unit is empty, it is considered available.

For example, let us make the realistic assumption that the multiplication in our previous example require two c-steps for execution. Furthermore, if a two-stage pipelined multiplier is selected in the allocation phase, then independent multiplications such as **u*dx** and **3*x** (refer to Fig. 3 in Section IV) could be scheduled in successive c-steps, and executed by a single multiplier. On the other hand, data-dependent multiplications (e.g. **3*y** and **dx*(3*y)**) can ony be scheduled in nonconsecutive c-steps, the result of the multiplication being available only after two c-steps.

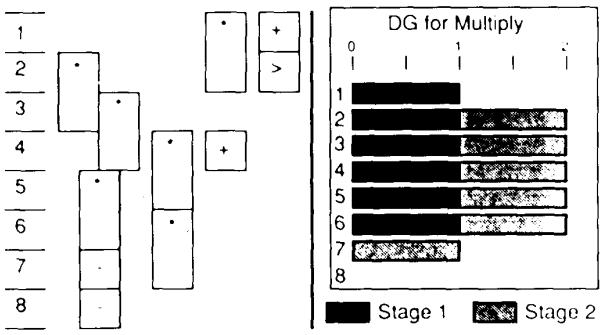


Fig. 13. Schedule and distributions using pipelined multipliers.

These considerations translate to the following simple extension of the scheduling algorithm: when calculating distributions for pipelined operations we need to *sum only the probability of the c-step(s) corresponding to the first stage*. The remaining ones are given an artificial probability of zero. The scheduler will then be effectively balancing the distribution of the first stage only, which accounts for the fact that the functional unit is available after the stage is empty. The time frame calculations are unchanged, however, as this ensures that data-dependent multicycle operations are not scheduled in successive c-steps.

To illustrate the algorithm extension, we will use our previous example with a new clock cycle of 50 ns (it was previously 100 ns). Multiplications, which have a propagation delay of 80 ns, now require two c-steps for execution. The remaining operations can still be assigned to a single c-step. Using the modified scheduling algorithm and a time constraint of eight c-steps, we obtain the schedule shown in Fig. 13. Here, the scheduler obtained a perfect balancing of the first stage of the multiplications, so the use of a single pipelined multipler will significantly reduce the overall costs. The adder, subtractor, and comparator allocations are the same in both cases. The data path resulting from this schedule is given in Fig. 16(b) in the next section.

VIII. Design Examples

The examples presented in this section were used in some of the systems described in the literature survey. They were chosen to illustrate the flexibility of the HAL system to different constraints and to allow comparison with the results obtained from these systems.

Experimental Procedure:

For each of the examples presented, synthesis was performed using the same assumptions[3] as the original reference. They are listed at the beginning of each subsection. They varied considerably, which put the flexibility of the HAL system to test. Except where explicitly stated, the synthesis was always performed using the same parameters. The results were obtained without any fine tuning of the algorithm to the examples.

The difficulty of evaluating designs produced by high-level synthesis systems was discussed in [39]. The results presented here make use of the register-transfer level cost metrics suggested in [39], which include number of registers, multiplexers, functional units, and control steps. Furthermore, the interconnect costs presented are obtained using the simple cost function described in [40]. The costs are used here to indicate cost trends, and not absolute cost values as these depend on layout and technology. When applicable, we also include the number of mux inputs, a measure that is often used to indicate relative interconnect costs.

The CPU execution times given are for the scheduling phase only, using a Xerox 1108 Lisp machine. This is a single-user workstation in the medium–low performance range by today's standards, yet the system allows nearly interactive scheduling (usually under two minutes of CPU time). The remaining allocation and binding tasks ran typically in a minute or less (ten minutes for the largest example presented).

A. Differential Equation Example from HAL

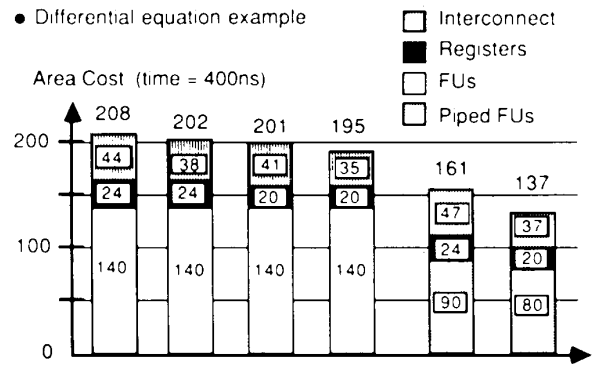
This example corresponds to the CDFG of Fig. 3 in Section IV. It was first presented in [11] and subsequently used in [15], [22], [31], [32], and [47]. The summary of costs for these and the HAL system is given in Fig. 14. In the first four columns, the following conditions are assumed:

- The clock cycle is 100 ns.
- The maximum time constraint is 400 ns (4 cycles).
- Additions are performed by adders in 40 ns.
- Multiplications are performed by multipliers in 80 ns.
- The latch delays are 10 ns.
- Multicycle and chained operations are allowed.
- Registers are preset with their original values (u, x, and y).
- Constants (e.g. dX, a, 3) are implemented as direct connections to power and ground lines.

The first column in Fig. 14 corresponds to an early version of the HAL system [11]. This result was improved on by the Splicer [47] and CATREE [15] systems, as indicated in the second and third columns. The fourth-column corresponds to the data path of Fig. 15 obtained using the current version of the HAL system. This version uses the new register and mux binding algorithms described in [35] and [36]. Interconnect, register and FU cost percentages (given below the graph) are given relative to the initial result of the early HAL system.

As both the CATREE and Splicer systems made use of the HAL schedule, the minimum FU and register requirements will be the same. The only possible cost reduction is obtained through the register and interconnect assignments.

[3] *Disclaimer*: The assumptions were duplicated as accurately as possible. However, as it is always possible that some constraints were left unspecified in the original references, these comparisons may or may not be absolute.



System	HAL '86	Splicer '88	Catree '87	HAL '88	Splicer '87	HAL '88
CPU	40 s	n/a	n/a	50 s	n.a	120 s
%	100	86	93	79	107	84
%	100	100	83	83	100	83
%	100	100	100	100	64	57

Fig. 14. Cost summary for HAL example.

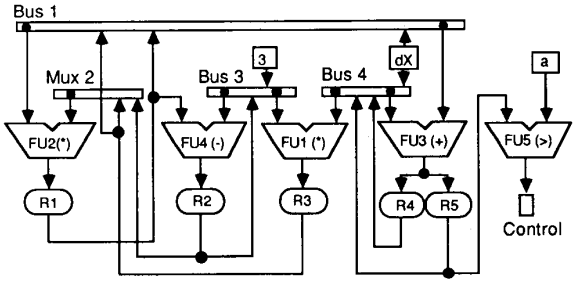


Fig. 15. HAL data path (using regular FU's).

One way of obtaining a more significant reduction is through structural pipelining. This was achieved in the Splicer system [32] by using a timing constraint of eight clock cycles, dividing the cycle time by 2, and making use of a two-stage pipelined multiplier. In this way, the original time constraint is still respected. The fifth column of Fig. 14 corresponds to the Splicer data path shown in Fig. 16(a). Note that for this data path, the constants are stored in registers.

The last result, the sixth column, is for the HAL system where the schedule (shown in Fig. 13) was obtained with the structural pipelining extension described earlier. The data path obtained is depicted in Fig. 16(b). The use of force-directed scheduling, which minimizes global storage and interconnect costs, and a different approach to register and bus allocation are the two main reasons for the efficiency of the design obtained.

B. Pipelined FIR Filter from Sehwa

The results presented here are for the pipelined 16-point digital FIR filter example borrowed from [29]. In this example, *functional pipelining* is used to increase throughput. The following assumptions are stated:

- The maximum stage time limit is 100 ns.

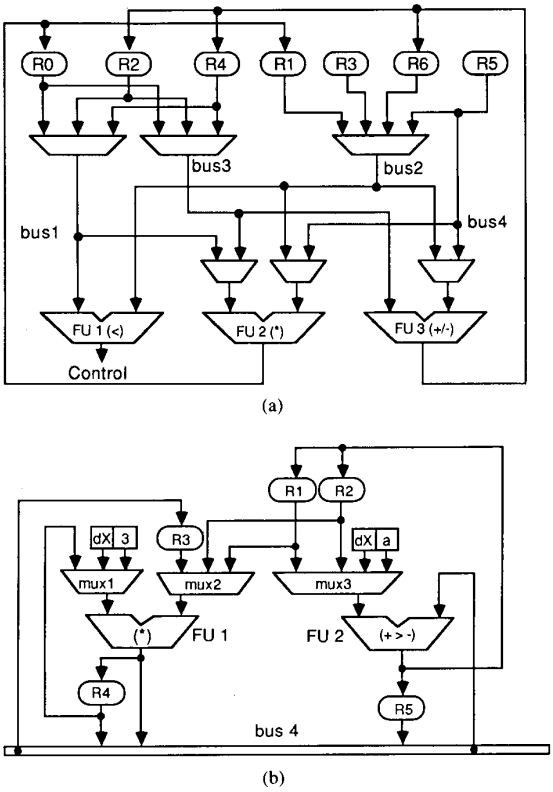


Fig. 16. Splicer and HAL data paths (using pipelined multipliers). (a) Splicer data path. (b) HAL data path.

- The latency is equal to 300 ns.
- Additions are performed by adders in 40 ns.
- Multiplications are performed by multipliers in 80 ns.
- The latch delays are 20 ns.
- Chained operations are allowed.

In Fig. 17 we present the results for three scheduling methods:

- SEHWA: Backward feasible scheduling.
- SEHWA: Exhaustive feasible scheduling.
- HAL: Force-directed pipeline scheduling.

These results include the algorithm complexity, the number of cycles, the FU allocation, and CPU time. In the Sehwa system, *the feasible schedule is used to establish an upper cost bound for the exhaustive scheduling phase that follows*, thereby reducing the search time.

The force-directed algorithm generated a schedule unlike the ones obtained by the feasible or exhaustive algorithms. However, the allocation cost is identical to that obtained by the exhaustive algorithm (i.e., an optimal cost). The data path realization was not given in [29].

We also ran the second pipeline example from the same article [29]. This example contained mutually exclusive operations that require special consideration in pipelined data-flow graphs. In this case, our system generated a schedule identical to the one obtained from Sehwa's exhaustive algorithm.

Pipelined 16 point FIR filter

System	SEHWA		HAL
Algorithm Complexity	Feasible $O(n^2 \log n)$	Exhaustive	Force-directed $O(cn^2)$
No. of cycles	7	6	6
No. of adders	6	5	5
No. of multipliers	3	3	3
CPU time	n/a	n/a	30 sec

Fig. 17. FIR filter results for SEHWA example.

For this type of example, special care must be taken not to assign mutually exclusive operations from different stages to the same functional unit. Doing so could produce a "twisted pair" [41] and cause deadlock. In our system, this cannot occur since the *initial* distributions are calculated without considering the pipeline nature of the problem. These distributions are calculated using the mechanism described in subsection IV-B, where only the *mutually exclusive operations* which can be scheduled in the *same control step* can (implicitly) share a functional unit. Later, when the distributions graphs are folded using the mechanism described in subsection VII-A, all the distributions will be added, without regard to mutual exclusion. The scheduling is then performed with these modified DG's, which guarantees that mutually exclusive operations of different pipe stages are never assigned to the same FU.

C. Fifth-Order Digital Elliptical Wave Filter

The CDFG used in this example is borrowed from Kung, Whitehouse, and Kailath's book on signal processing [19] and implements a fifth-order wave digital elliptic filter. This is a more substantial behavioral description that contains 43 operations (additions and multiplications) submitted to over 60 precedence constraints. This example was chosen as a benchmark for the 1988 High-Level Synthesis Workshop [42].

It is stated in [19] that multiplications require an execution time that is twice as long as that for additions. Here, they were assigned two and one c-steps respectively. The critical path is therefore 17 c-steps long.[4] It is also assumed that the filter coefficients (the left operand of the multiplications) are user defined and not necessarily equal to a multiple of 2. Therefore, shift registers are not used to implement multiplications.

In the first row of Table I below, we summarize the adder and multiplier allocations for different timing constraints as obtained from the regular force-directed scheduling (FDS) algorithm. In this row, we assume nonpipelined functional units where the multipliers require two c-steps for execution and the adders only one. CPU times varied between two and six minutes.

[4]Retiming makes it possible to reduce this path to 16 c-steps. However, to allow easier comparison with other systems, we have not exploited this transformation.

TABLE I
WAVE FILTER FU ALLOCATIONS FOR DIFFERENT EXECUTION TIMES

Algorithm	17	18	19	21
FDS	3 + 3 ×	3 + 2 ×	2 + 2 ×	2 + 1 ×
FDLS	3 + 3 ×	2 + 2 ×		2 + 1 ×
ASAP	4 + 4 ×			
LS			2 + 2 ×	
FDS, FDLS	3 + 2 ⊗	3 + 1 ⊗	2 + 1 ⊗	

+ Adder × Multiplier ⊗ Pipelined multiplier

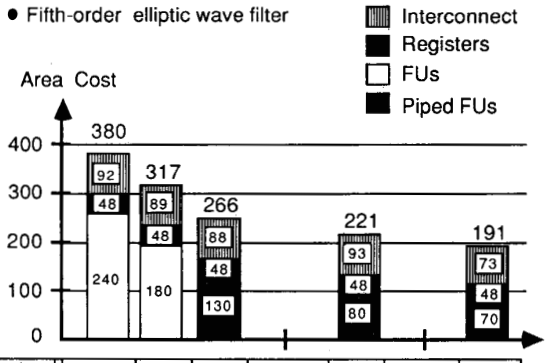
The second row was obtained by taking the FDS allocations for 17, 19, and 21 c-steps, setting these as a maximum FU limit and running the force-directed list scheduling (FDLS) algorithm to obtain the shortest execution time. For the 17 and 21 c-step allocations, the results were already optimal so the time could not be reduced. However, for the 19 c-step allocation (two adders and two multipliers), the FDLS algorithm produced a schedule requiring one c-step less. This is also an optimal result with respect to functional unit cost. CPU times were significantly faster than those for the FDS algorithm and varied between one and two minutes.

The improvement of the 19 c-step result by the FDLS algorithm is mostly due to the fact that we have given more information about the design, i.e., the number and type of FU's than in the case of the FDS algorithm, where only a time constraint is given. The number of force calculations in FDLS is also inferior, which explains the reduced CPU times.

The optimal 18 c-step schedule was also obtained by researchers at the University of Eindhoven [9] and Karlsruhe [43]. The former use a slightly modified version of the original FDS algorithm [7] where only a subset of the force calculations are performed. The recent scheduling algorithm [44] of Karlsruhe's CADDY system uses time crames and distibutions graphs as in FDS, but incorporates a different force function.

The third row represents the schedule obtained in [19] using an as soon as possible (ASAP) algorithm. In comparison with the FDS and FDLS algorithms, it required one extra adder and multiplier. The fourth row represents the result obtained from the System Architect's Workbench (SAW) at CMU [45] using a list scheduling (LS) algorithm. It requires one extra c-step with respect to the force-directed list scheduling approach.

Finally, the fifth row represents the allocations obtained using a two-stage pipelined multiplier. The struc-



Cycles	17	17	17		18		19
System	Kailath	HAL	HAL		HAL		HAL
+	4	3	3		3		2
*	4	3	2		1		1
Registers	12	12	12		12		12
Mux inp.	32	31	31		34		26
CPU	n/a	2min	2min		4min		6min

Fig. 18. Cost summary for wave filter example.

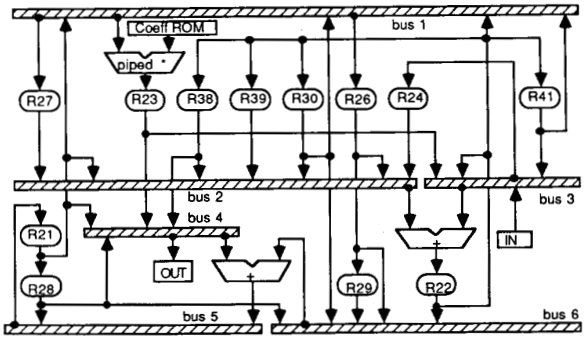


Fig. 19. Data path for wave filter example (time constraint = 19 c-steps).

tural pipelining extension described in Section V is used to take this type of multiplier into account. The FDS and FDLS algorithms both obtained optimal results with respect to FU costs. As shown in [36], register and interconnect costs also compare favorably with the results of other systems [45]–[47]. For the examples presented, the HAL system typically obtains 15–20 percent lower interconnect costs—as expressed by the total number of mux inputs—than these systems, with identical register costs.

The graph shown in Fig. 18 summarizes the FU, register, and allocation costs obtained from the HAL system for a subset of the possible constraints. The figure also gives the number of adders, multipliers, registers, and mux inputs[5] that were used in each design. The first column was obtained by using the schedule given in [19].

The data path for the result of the last column is given in Fig. 19. The right operand of the pipelined multiplier is a small constant ROM that contains the filter coefficients. This result (one pipelined multiplier and two adders) is a good example of a substantial area cost reduction (approximately 50 percent) against a small loss in speed (close to 10 percent) as compared to the 17 c-step result given in [19].

[5]This value is actually the *combined number of inputs to mixes and buses*, where a bus is considered equivalent to a mux with multiple outputs.

IX. System Limitations

Although a large number of the scheduling problems encountered in behavioral synthesis can be solved using the force-directed algorithm, there remain some difficult problems which are not addressed.

A. Control Costs

The associated cost of the control path is largely ignored by the algorithm. This is particularly true for pipelined data paths (functional pipelining), where control path costs may outweight the savings realized in the data path. The integration in our framework of Nagle's work on the minimization of control path costs [33], which, as mentioned, has some similarities with the force-directed algorithm, would be a step in the right direction.

B. Functional Unit Allocation

Another difficult problem is the assignment of n operations to m functional units, some of them multifunction, some of them not. A clique partitioning approach to allocation, such as the one presented in the Facet/Emerald system [14], could possibly be used as a starting point. Operations would be partitioned into different sets corresponding to a single ALU type (single or multifunction). Each of these sets could then be assigned to a single or multiclass DG. As described in subsection V-F, the forces exerted by operations of a set would be calculated based on the force associated with the assigned DG.

C. Physical Partitioning

The effect of scheduling on interconnect costs is another difficult issue. Balancing the concurrency of data transfers, thereby reducing the total number of buses required (as described in subsection V-A), only partially solves the problem. The interaction of scheduling and floorplanning also needs to be taken into account. In the near future, we hope to apply the work of McFarland [26], Dirkes [45], and Gebotys [46] to help resolve this limitation.

D. Heuristics

The presence of empirical heuristics throughout the algorithm, for example the use of *look-ahead* and the simple estimation of probabilities, precludes the systematic determination of optimal solutions. Furthermore, even though the force-directed algorithm is global, it is nevertheless a *greedy* one. The system can therefore get caught in a local optimum.

X. Conclusion

We have shown the importance of scheduling for the synthesis of ASICs. Ultimately, the schedule will determine:

- the area of the design given a speed constraint;
- the speed of the design given an area cost constraint.

The methodology presented features a novel *force-directed* scheduling algorithm that is invoked in a scheduling/allocation scheme that proceeds by stepwise refinement. In spite of its relatively low complexity—$O(cn^2)$ for the FDS algorithm, $O(n^2)$ for FDLS—the scheduling algorithm at the heart of this process explores the search space in a global fashion and produced optimal or near-optimal results for all of the examples attempted to date.

Furthermore, the flexibility of the system was highlighted by the variety of constraints and requirements it can deal with. These include:

- chained and multicycle operations,
- mutually exclusive operations,
- functional and structural pipelining,
- scheduling under local and global time constraints,
- scheduling with limited hardware resources.

Some of the more difficult problems still need to be solved. For example:

- the consideration of control costs during scheduling,
- the allocation of multifunction units,
- the interaction of scheduling and floorplanning.

However, a reasonable subset of the typical scheduling problems encountered in synthesis can be solved efficiently using the force-directed scheduling approach. Furthermore, we have shown that the force and distribution graph framework can be cleanly and effectively applied to include additional cost terms. The consideration of bus and storage costs was achieved in this way. Two simple subclasses of the general pipelining problem were also taken into consideration. In the near future, we hope to extend the approach to take control costs into consideration, particularly for pipelined applications, and to integrate physical partitioning information into the scheduling process.

Acknowledgment

The authors would like to thank D. Agnew and R. Peacocke of the CAE and AI Departments of BNR, as well as the Electronics Department of Carleton University for their active involvement in this cooperative research project. Thanks are also due to the reviewers for their helpful comments, and especially to Dr. E. Girczyc of the University of Alberta, who suggested some of the initial concepts of this work.

References

[1] D. D. Gajski, N. D. Dutt, and B. M. Pangrle, "Silicon compilation (tutorial)," in *Proc. IEEE 1986 Custom Integrated Circuits Conf.* (Rochester NY), May 1986, pp. 102-110.

[2] E. F. Girczyc and J. P. Knight, "An ADA to standard cell hardware compiler based on graph grammars and scheduling," in *Proc. IEEE Int. Conf. Computer Design*, October 1984, pp. 726-731.

[3] N. Park and A. C. Parker, "Synthesis of optimal clocking schemes," in *Proc. 22nd Design Automat. Conf.*, July 1985, pp. 489-495.

[4] A. C. Parker *et al.*, "MAHA: A program for datapath synthesis," in *Proc. 23rd Design Automat. Conf.* (Las Vegas, NV), July 1986, pp. 461-466.

[5] P. G. Paulin and J. P. Knight, "Extended design-space exploration in automatic data path synthesis," in *Proc. 1986 Canadian Conf. VLSI* (Montreal), Oct. 1986, pp. 221-226.

[6] M. C. McFarland, A. C. Parker, and R. Camposano, "Tutorial on high-level synthesis," in *Proc. 25th Design Automat. Conf.*, July 1988, pp. 330-336.

[7] P. G. Paulin and J. P. Knight, "Force-directed scheduling in automatic data path synthesis," in *Proc. 24th Design Automat. Conf.* (Miami Beach, FL), July 1987, pp. 195-202.

[8] R. Cloutier, (CMU), private communication, Nov. 1987.

[9] L. Stok and R. Van den Born, "EASY: Multiprocessor architecture optimization," in *Proc. Int. Workshop Logic and Architecture Synthesis for Silicon Compilers* (Grenoble, France), May 1988.

[10] T. Fuhrman (GM-Delco), private communication, Jan. 1988.

[11] P. G. Paulin, J. P. Knight, and E. F. Girczyc, "HAL: A multi-paradigm approach to automatic data path synthesis," in *Proc. 23rd Design Automat. Conf.* July 1986, pp. 263-270.

[12] L. Lawler *et al.*, "Deterministic and stochastic scheduling," *Proc. NATO Adv. Study*, 1982.

[13] T. Blackman *et al.*, "The Silc compiler: Language and features," in *Proc. 22nd Design Automat. Conf.*, pp. 232-237, June 1985.

[14] C. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Trans. Computer-Aided Design*, vol. CAD-5, pp. 379-395, July 1986.

[15] C. H. Gebotys and M. I. Elmasry, "A VLSI methodology with testability constraints," in *Proc. 1987 Canadian Conf. VLSI* (Winnipeg), Oct. 1987.

[16] S. Davidson *et al.*, "Some experiments in local microcode compaction for horizontal machines," *IEEE Trans. Comput.*, pp. 460-477, July 1981.

[17] P. Marwedel, "A new synthesis algorithm for the MIMOLA software system," in *Proc. 23rd Design Automat. Conf.* (Las Vegas, NV), July 1986, pp. 271-277.

[18] H. Trickey, "Flamel: A high-level hardware compiler," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 259-269, Mar. 1987.

[19] S. Y. Kung, H. J. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*. Englewood Cliffs, NJ: Prentice Hall, 1985, pp. 258-264.

[20] C. Y. Hitchcock and D. E. Thomas, "A method of automatic data path synthesis," in *Proc. 20th Design Automat. Conf.*, July 1983, pp. 484-489.

[21] J. Nestor and D. E. Thomas, "Behavioral synthesis with interfaces," in *Proc. IEEE Int. Conf. CAD*, Nov. 1986, pp. 112-115.

[22] B. M. Pangrle and D. D. Gajski, "Slicer: A state synethesizer for intelligent silicon compilation," in *Proc. IEEE Int. Conf. Computer Design*, Oct. 1987.

[23] H. De Man *et al.* "Cathedral-II: A silicon compiler for digital signal processing," *IEEE Design & Test Magazine*, pp. 13-25, Dec. 1986.

[24] J. Vanhoof, J. Rabaey, and H. De Man, "A knowledge-based CAD system for synthesis of multi-processor digital signal processing chips," in *VLSI '87*, C. H. Séquin, Ed. New York: Elsevier, 1988, pp. 73-88.

[25] G. Goosens *et al.*, "An efficient microcode compiler for custom multiprocessor DSP systems," in *Proc. Int. Conf. CAD*, Nov. 1987, pp. 24-27.

[26] M. C. McFarland, "BUD: Bottom-up design of digital systems," in *Proc. 23rd Design Automat. Conf.* (Las Vegas, NV), July 1986, pp. 474-479.

[27] T. J. Kowalski *et al.*, "The VLSI design automation assistant: From algorithms to silicon," *IEEE Design & Test*, pp. 33-43, Aug. 1985.

[28] R. Camposano, "Structural synthesis in the Yorktown Silicon Compiler," in *VLSI '87*, C. H. Séquin, Ed. New York: Elsevier, 1988, pp. 61-72.

[29] N. Park and A. C. Parker, "SEHWA: A program for synthesis of pipelines," in *Proc. 23rd Design Automat. Conf.* (Las Vegas, NV), July 1986, pp. 454-460.

[30] E. F. Girczyc, "Loop winding—A data flow approach to functional pipelining," in *Proc. Int. Symp. Circuits Syst.* (Philadelphia PA), May 1987, pp. 382-385.

[31] B. M. Pangrle and D. D. Gajski, "Design tools for intelligent silicon compilation," *IEEE Trans. Computer-Aided Design*, vol. CAD-6, pp. 1098-1112, Nov. 1987.

[32] F. D. Brewer and D. D. Gajski, "Knowledge-based control in microarchitecture design," in *Proc. 24th Design Automat. Conf.* (Miami Beach, FL), July 1987, pp. 203-209.
[33] A. W. Nagle and A. C. Parker, "Algorithms for multiple-criterion design of microprogrammed control hardware," in *Proc. 18th Design Automat. Conf.*, June 1981, pp. 486-493.
[34] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Trans. Comput.* vol. C-30, pp. 478-490, July 1981.
[35] P. G. Paulin, "High-level synthesis of digital circuits using global scheduling and binding algorithms," Ph.D. thesis, Carleton University, Ottawa, Canada, Feb. 1988.
[36] P. G. Paulin and J. P. Knight, "Scheduling and binding algorithms for high-level synthesis," submitted to the *26th Design Automat. Conf.*, Las Vegas, NV, June 1989.
[37] J. Midwinter and J. P. Knight, " A weight-directed approach to register and interconnect allocation for digital circuit systhesis," in *Proc. 1986 Canadian Conf. VLSI*, (Halifax N.S.), Oct. 1988, pp. 379-384.
[38] M. Balakrishnan *et al.*, "Allocation of multi-port memories in data path synthesis," *IEEE Trans. Computer-Aided Design*, vol. 7, pp. 536-540, Apr. 1988.
[39] G. Borriello and E. Detjiens, "High-level synthesis: Current status and future directions," in *Proc. 25th Design Automat. Conf.* June 1988, pp. 477-482.
[40] J. Midwinter, "Improving interconnect for the behavior synthesis of ASICs," M.Sc. thesis, Carleton University, Ont., Canada, Feb. 1988.
[41] K. S. Hwang *et al.*, "Constrained conditional resource sharing in pipeline synthesis," in *Proc. IEEE Int. Conf. CAD*, Nov. 1988, pp. 52-55.
[42] K. S. Hwang *et al.*, "Workshop on high-level synthesis," Orcas Island, WA, Jan. 1988.
[43] W. Rosenstiel, "CADDY: The Karlsruhe behavioral synthesis system," presented at the ACM/IEEE High-Level Synthesis Workshop, Orcas Island, WA, Jan. 1988.
[44] H. Kramer *et al.*, "Data path and control synthesis in the CADDY system," in *Proc. Int. Workshop Logic and Architecture Synthesis for Silicon Compilers*, (Grenoble, France), May 1988, paper V(3).
[45] D. E. Thomas *et al.*, "The system architect's workbench," in *Proc. 25th Design Automat. Conf.* July 1988, pp. 337-343.
[46] C. H. Gebotys and M. I. Elmasry, "VLSI design synthesis with testability," in *Proc. 25th Design Automat. Conf.* June 1988, pp. 16-21.
[47] B. M. Pangrle, "Splicer: A heuristic approach to connectivity binding," in *Proc. 25th Design Automat. Conf.* July 1988, pp. 536-541.

Pierre Paulin (S'83) received the B.Sc. degree in engineering physics in 1982 and the M.Sc. degree in electrical engineering in 1984 from the Université Laval, Québec City, Canada. He obtained the Ph.D. degree in electronics engineering in 1988 from Carleton University, Ottawa, Canada, as part of a cooperative research project with Bell-Northern Research (BNR), Ottawa.

In 1982, he worked for Renault (France) in robotics, and in 1988 he took on a postdoctoral research fellowship at the INPG/CSI in Grenoble, France. He is presently a member of the scientific staff at BNR. His research interests include CAD for VLSI and high-level synthesis, as well as FSM and logic synthesis.

Dr. Paulin received the best presentation award at the 23rd Design Automation Conference in 1986, the best paper award at the 1986 Canadian Conference on VLSI, and was nominated for the best paper award at the 24th Design Automation Conference in 1987. He was also corecipient of the best paper award at the International Workshop on Logic and Architecture Synthesis for Silicon Compilers, in Grenoble, 1988.

*

John P. Knight is with the Electronics Department at Carleton University in Ottawa, Canada, where he teaches in the areas of microprocessors, digital design and computer-aided design. His research interests center around circuit synthesis, with his present work heading toward controller and mixed analog-digital synthesis. He has been active locally in the IEEE, in the Ottawa Section and the student chapters, and has received a few awards for such work.