ALGORITHMS FOR HIGH-LEVEL SYNTHESIS

PIERRE G. PAULIN
Bell-Northern Research

JOHN P. KNIGHT
Carleton University

Synthesis tools at the logic and register-transfer levels are gaining a foothold in industry. The next step is the automatic synthesis of a digital system from a behavioral description. The synthesis algorithms presented here offer a technique for scheduling operations and allocating registers and buses in light of both timing constraints and available hardware resources. The algorithm enhances current scheduling techniques by using a global priority function that minimizes storage, interconnections, and functional unit cost. Algorithms for allocating registers and buses minimize storage and interconnection costs and take into account the interdependence of both tasks.

The recent flurry of activity in high-level synthesis is further evidence of its increasing popularity as a research topic. In the commercial realm, the success of various logic and register-transfer-level synthesis tools is prompting many companies to extend the scope of their synthesis products.

This increased interest is a natural consequence of the shifting focus in IC design. Designers today are less interested in the details of the device than the architecture around the device. High-level description languages such as VHDL, which allow for behavioral as well as RTL and gate-level descriptions, are becoming more accessible. High-level synthesis[1] fills the gap between these two levels by automatically generating an RTL realization from a behavioral description.

In high-level synthesis, we typically divide the task into data-path design and control-path design. Scheduling data-path operations into control steps is perhaps the most important task. The scheduling strategy must consider both timing and resource constraints as well as storage and interconnection costs. The algorithms we describe here incorporate such a strategy by offering a new way to explore the design space. They are also applicable to more than one method of synthesis. Although first implemented in the HAL system,[2] they have since been integrated into more specialized high-level synthesis systems in use by both academia and industry.[3]

SCHEDULING

Scheduling consists of determining a propagation delay for every operation of the input behavioral description and then assigning each operation to a specific control step (a control step is often equivalent to a single state of a finite-state machine.

One commonly used approach is list scheduling,[4] in which we specify a hardware constraint and use an algorithm to minimize the total execution time. The algorithm uses a local priority function to defer operations when resource conflicts occur. Another approach, called force-directed scheduling, allows us to specify a global time constraint, and the algorithm tries to minimize the resources required to meet that constraint. This formulation of constraints is useful for digital-signal-processing applications in which the system throughput is fixed and the area must be minimized.

0740-7475/89/1200-18$1.00©1989 IEEE

TIME CONSTRAINTS

The force-directed scheduling algorithm reduces the number of functional units, registers, and buses required. The strategy is to place similar operations in different control steps so as to balance the concurrency of the operations assigned to the units without increasing the total execution time. By balancing the concurrency of operations, we ensure that each structural unit has a high utilization, which in turn decreases the total number of units required. This balancing is done in three steps: determine the time frame of each operation, create a distribution graph, and calculate the force associated with each assignment.

Determine time frame. We determine the time frame of each operation by evaluating the ASAP (as soon as possible) and ALAP (as late as possible) schedules. By combining results for both schedules, we can ascertain the time frame of each operation. A simple example, DiffEq,[2] illustrates this process. Figure 1a gives a differential equation that we can solve using the iterative algorithm in Figure 1b. Figures 1c and 1d depict the control- and data-flow graphs for the ASAP and ALAP schedules for the inner loop of the DiffEq example. Nodes represent functional operations, while edges represent data depedencies between these operations.

The resulting time frames are given in Figure 2. The width of the box containing an operation represents the probability that the operation will be eventually placed in some time slot. We assume that the probability distribution for each operation is uniform. We chain operations by extending their time frames into the previous (or next) control step. Before we can extend them, however, their combined propagation delays—added to the latch and estimated interconnection delays—must be less than the clock cycle. We can extend this single-cycle method in a straightforward way to support multicycle operations.[3]

Create distribution graph. The next step is to add the probabilities of each type of operation for each control step, or c-step, of the control-flow or data-flow graph. The resulting distribution graphs indicate the concurrency of similar operations. For each graph, the distribution in c-step i is

$$DG(i) = \sum_{\text{Opn type}} \text{Prob}(\text{Opn}, i) \qquad (1)$$

where the sum is over all operations of a given type. Using Figure 2, we can calculate the values of the multiplication distribution graph, or DG. The result is DG(1) = 2.833, DG(2) = 2.333, DG(3) = 0.833, and DG(4) = 0 as depicted in Figure 3a.

Force calculation. The final step is to calculate the force associated with every feasible c-step assignment of each operation. We temporarily reduce the operation's time frame to the selected c-step. For an operation with an initial time frame that extends from c-steps t to b, the force associated with its assignment to c-step j is

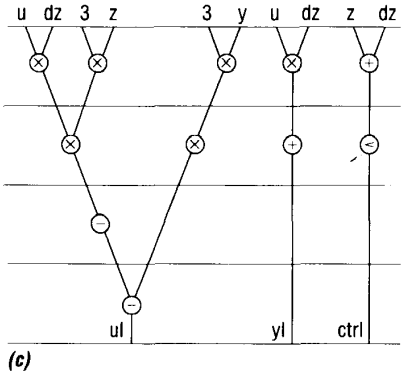
$$\text{Force}(j) = DG(j) - \sum_{i=t}^{b} \left[\frac{DG(i)}{(b - t + 1)} \right] \qquad (2)$$

$y'' + 3zy' + 3y = 0$

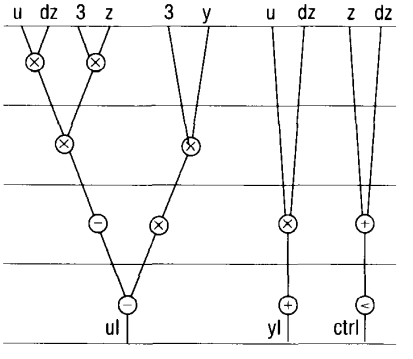
(a)

```
while (z < a) repeat
    zl := z + dz;
    ul := u – (3 • z • u • dz) – (3 • y • dz);
    yl := y + (u • dz);
    z := zl ; u := ul ; y := yl ;
```

(b)



(c)



(d)

Figure 1. *Simple example, DiffEq, to illustrate how to determine the timeframe of an operation: Differential equation to be solved* ***(a)****, iterative solution* ***(b)****, as-soon-as-possible schedule* ***(c)****, and as-late-as-possible schedule* ***(d)****.*

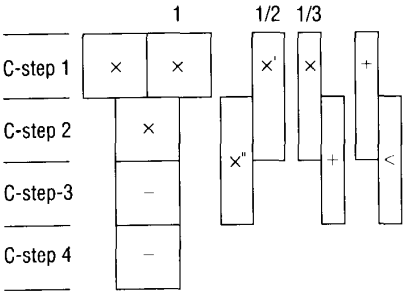
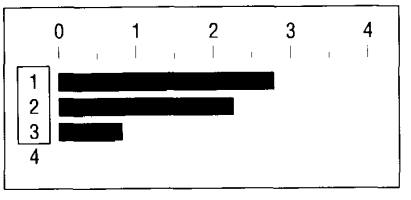
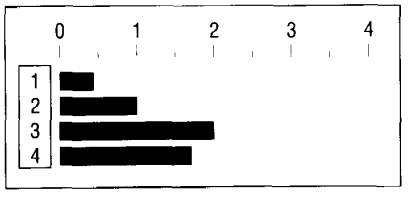


Figure 2. *Time frames for Diffeq example.*

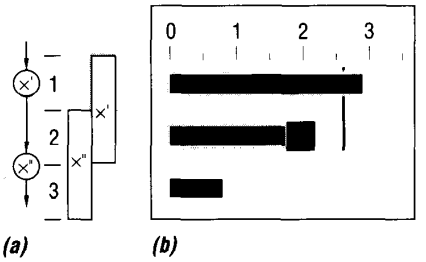


(a)



(b)

Figure 3. *Distribution graphs for multiply* **(a)** *and for add, subtract, and compare* **(b)**.



(a) *(b)*

Figure 4. *Calculating the force of assigning x′ to control step 1: time frames* **(a)** *and the distribution graph for multiply* **(b)**.

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. Figure 4 illustrates this relationship, which we then use to calculate the force associated with the assignment of multiplication x' to c-step 1. Here, Equation 2 yields

$$\text{Force}(1) = \text{DG}(1) - \text{average DG value over time frame}$$

$$= \text{DG}(1) - \sum_{i=1}^{2} \left[\text{DG}\frac{(i)}{2} \right]$$

$$= 2.833 - \frac{(2.833 + 2.333)}{2} = +0.25$$

As the shaded columns in Figure 4b show, if we assign multiplication x' to c-step 1, the distribution is not very well balanced, and multiplier costs will be higher.

We must also calculate the force for all predecessors and successors of the current operation whenever their time frames are affected. These additional forces are called indirect forces. The total force is the sum of the direct and indirect forces. In the force calculation in Figure 4, we did not have any indirect force because the time frame of the successor multiplication operation, x'' was not affected.

However, if we assign x' to c-step 2, we are implicitly forcing x'' into the third control step, as the shaded bars in Figure 5a illustrate. Thus, we are exerting additional force, and the total force becomes

$$\text{Force}(2) = \text{direct force}(x' \textit{ in c-step } 2) + \text{indirect force}(x'' \textit{ in c-step } 3)$$
$$= -0.25 + -0.75 = -1.00$$

As shown by the shaded columns in Figure 5b, this assignment causes a better balancing of the DG. The calculated force is actually a negative value.

After we have calculated the force of all operations, we assign an operation to a c-step in a way that yields the lowest force, that is, balances the concurrency of the operations most effectively. We readjust the time frames accordingly and repeat the entire process until all operations are scheduled.

We consider I/O operations in the same way we would any regular operation. By balancing the concurrency of I/O operations, we minimize the number of required ports. This benefit is particularly significant for designs that limit the number of pins.

A more effective way of calculating force[3] is to calculate DG(j)″, the distribution value we would get if we assigned the operation to control step j. We then replace DG(j) in Equation 2 by DG′(j), which is

$$\text{DG}'(j) = \text{DG}(j) + \frac{\text{DG}''(j) - \text{DG}(j)}{3} \quad (3)$$

In Figure 4, DG″ = 3.333, which yields

$$\text{DG}'(1) = 2.833 + \frac{3.33 - 2.833}{3} = 3.00$$

Therefore,

$$\text{Force}(1) = 3.00 - \frac{2.833 + 2.333}{2} = +0.417$$

This implements a simple form of lookahead that has considerably improved the force-directed scheduling algorithm's effectiveness.

MINIMIZING STORAGE AND INTERCONNECTION COST

Most scheduling algorithms minimize the cost of functional units but ignore the associated storage and data-transfer costs, even though scheduling has a direct effect on them. For example, the fewest buses required for a scheduled control-flow or data-flow graph is the number of concurrent data transfers in a control step. The fewest registers required is the maximum number of data arcs that cross the boundary of a control step. Figure 6 illustrates two simple schedules with different hardware costs. The schedule in Figure 6a appears to be the best of the two, since it requires only one multiplier. However, the schedule in Figure 6b may have a lower global cost because the allocation cost for ports and buses and the storage costs are considerably lower.

Minimizing storage costs. The first step in minimizing the number of registers is to create a new class of operations, called 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 also need a special distribution graph, called a storage DG.

We calculate the force of storage operations in much the same way as we do for regular operations. The only complication is that the length, or lifetime, of a storage operation depends on the final schedule. As an example, consider a simple data-flow graph, in which storage operation S has three possible lifetimes. Figure 7a shows the ASAP life, which spans c-steps 2 and 3. In our approach, we combine the ASAP, ALAP, and maximum lifetimes to calculate a nonuniform probability distribution. The sum of the distributions of all the storage operations yields the storage DG shown in Figure 7b. The gray portion of the graph reflects the contribution of S.

We add the storage forces to an operation's direct force by applying a mechanism similar to that used for indirect forces caused by a predecessor or successors.

Minimizing bus costs. To minimize the number of concurrent transfers and the associated bus costs, we create a special distribution graph, called the transfer DG. The transfer DG contains the distributions of the data transfers. 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 inputs and outputs. For example, we have only four distinct inputs and outputs in c-step 2 of Figure 7a. We calculate the forces from these new DGs in the same manner we use for regular operations.

RESOURCE CONSTRAINTS

The force-directed scheduling, or FDS, approach just described supports the synthesis of data paths that have a near-minimum cost under fixed timing constraints, but does not consider hardware constraints. The FDLS (force-directed list scheduling) approach presented here solves the opposite problem: finding the fastest schedule given fixed hardware constraints. It combines the characteristics and strengths of the well-known list scheduling algorithm[4] as well as the

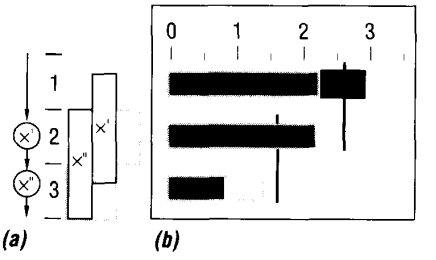


Figure 5. *Force calculations for x′ scheduled in control step 2: time frames* ***(a)*** *and the distribution graph for multiply* ***(b)****.*

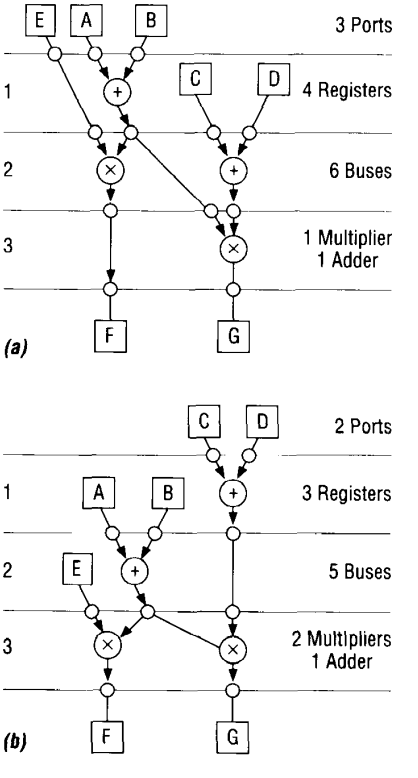


Figure 6. *Hardware cost of a schedule requiring one multiplier* ***(a)*** *and two multipliers* ***(b)****.*

Force-directed list scheduling is similar to list scheduling except force is the priority function, not mobility or urgency.

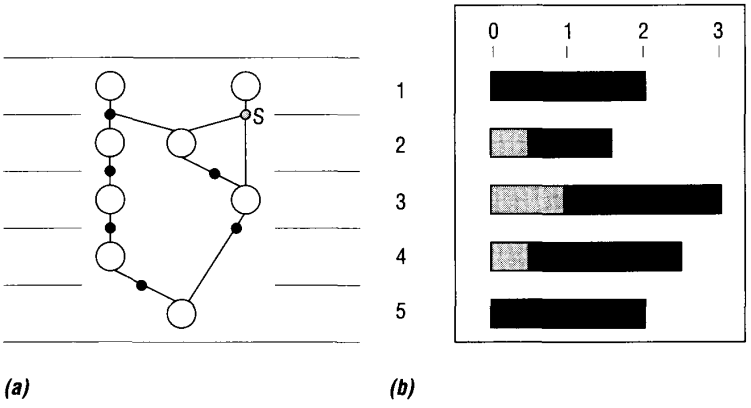


Figure 7. *Storage distribution graph.*

FDS algorithm. FDLS uses a global measure of concurrency throughout the scheduling process just as FDS does.

In list scheduling, we sort operations in topological order using control and data dependencies. Ready operations are those we can assign to the first control step. If there are more ready operations of a single type than there are hardware modules to perform them, then we must defer one or more operations. Which operations to defer often depends on some local priority such as mobility or urgency.[1] In Figure 8, for example, two add operations may be scheduled in the first control step, so we must defer one of them. Since they are both on the critical path, they have equal mobilities and the same urgency, so we could choose either one. In the figure, the left addition is deferred. We repeat this process, which yields the final schedule requiring four control steps.

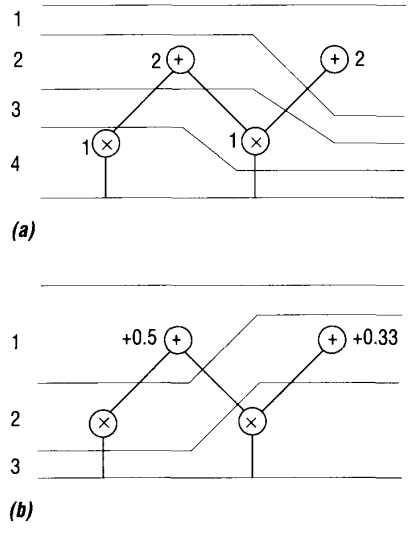


Figure 8. *List schedule* ***(a)*** *and force-directed schedule* ***(b)*** *for a simple example.*

FDLS is similar to list scheduling except force is the priority function, not mobility or urgency. Whenever we exceed a hardware constraint during regular scheduling, we calculate the force of the operation to select the best operation(s) to defer. The deferral that produces the lowest force—that is, the lowest global increase of concurrency in the graph—is the best candidate for deferral. We repeat these calculations and the deferral process until we meet the hardware constraint. Typically, the hardware constraint is given as the maximum number of functional units of each type. However, we can apply this principle to data-transfer and storage operations by using fixed limits on buses and registers.

Force calculations depend on the existence of time frames, so we must temporarily specify some global constraint. In the FDLS algorithm, this constraint is the length of the current critical path. This path gets longer if we have to defer a critical operation to solve a resource conflict.

In Figure 8, the initial time constraint was two control steps. We had to extend it to three control steps immediately to solve a resource conflict between operations on the critical path. The force values in Figure 8b are from the enhanced force formulation given in Equation 3, which has a simple lookahead scheme. In this case, the force values for the two addition operations are not equal. Deferring the

left addition causes a force of +0.5, while deferring the right addition causes only a +0.33 force, so we defer it to c-step 2.

Even for this simple example, FDLS yields a faster schedule than list-scheduling. However, when we take into account the storage and transfer distribution graphs, we could get a different schedule. The final schedule depends the relative values of storage, interconnection, and port costs, as Figure 6 illustrates.

To summarize, the FDLS algorithm, shown in Figure 9, allows us to maintain the advantages of both list scheduling and force-directed scheduling by allowing

- high utilization of functional units, as in list scheduling
- fast computation times, since the FDLS algorithm has a worst case complexity of $O(n^2)$, where n is the number of operations in the control-/data-flow graph, and typically exhibits linear behavior
- global evaluation of all the side effects from assigning an operation to a control step, which is a characteristic of force-directed scheduling

```
Initialize time constraint to length of critical path
for c-step from 1 to time constraint do:
  Determine time frames
  Determine ready operations in c-step
    {operations whose time frame intersects
      current c-step}
  while (number of ready operations > number of
    functional units) do
    if all operations 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 operations
  end;
  Schedule remaining ready operations in current
    c-step
end;
```

Figure 9. *The algorithm for force-directed list scheduling.*

SCHEDULING EXAMPLES

To illustrate and compare scheduling algorithms, we use the fifth-order elliptic wave filter described in Kung et al.'s book on signal processing.[5] In the first row of Table 1, we summarize the adder and multiplier allocations for timing constraints in FDS. In this table, we assume that multipliers require two control steps for execution, while adders require only one. The minimum timing constraint for this example is 17 c-steps. Using retiming, we could reduce that to 16 c-steps, but to ensure a fair comparison, we do not use retiming. CPU times were between two and six minutes on the Xerox 1108, a Lisp machine with medium to low performance.

The second row of the table is the result of taking the FDS allocations for 17, 19, and 21 c-steps, setting these as a limit of the number of functional units allowed, and running the FDLS algorithm to get the shortest execution time. For the 17 and 21 c-step allocations, we already had optimal results and 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 that had one c-step less. This result is optimal with respect to functional unit cost.

Table 1. *Functional unit allocations for different execution times; + = adder, x = multiplier, x^p = pipelined multiplier; FDS = force-directed schduling, FDLS = force-directed list scheduling, ASAP = as soon as possible, LS = list scheduling.*

Algorithm	Number of Control Steps 17	18	19	21
FDS	+++ xxx	+++ xx	++ xx	++ x
FDLS	+++ xxx	++ xx		++ x
ASAP	++++ xxxx			
LS			++ xx	
FDS, FDLS	+++ $x^p x^p$	+++ x^p	++ x^p	

Regardless of the method choosen, the designer has an added level of flexibility with this integrated scheduling methodology.

CPU times were significantly faster than those for FDS, varying between one and two minutes.

One reason for the improved performance of FDLS is that with this algorithm we have more information about the design. We provide the number and type of functional units in FDLS, while FDS uses only a time constraint. FDLS also requires fewer force calculations, which explains the reduced CPU times.

The third row of Table 1 represents the schedule with ASAP scheduling and conditional deferral.[5] The fourth row represents the result from CMU's System Architect's Workbench[6] using list scheduling. Finally, the fifth row shows the HAL results when two-stage pipelined multipliers are used. The use of this type of functional unit involves a simple extension of the force algorithm.[3] Functional unit costs were optimal with both FDS and FDLS. Also, HAL's register and interconnection costs compare favorably with those from other systems.

A NEW EXPLORATION TECHNIQUE

Taken alone, FDLS 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. The flexibility of FDLS justifies the small additional effort to implement it.

We have found, however, that the most powerful method of exploring the design space is to use both FDS and FDLS. The designer sets a maximum time constraint and uses the FDS algorithm to arrive at a near-optimal allocation. In this phase, we can take advantage of FDS's ability to automatically perform cost tradeoffs among functional units of different types.[3]

In the second phase, the designer focuses on one area of the design space by using FDLS with the allocation from the first phase to determine if he can get a faster schedule. The faster schedule may be possible because the scheduler starts out with more information about the design.

Regardless of the method choosen, the designer has an added level of flexibility with this integrated scheduling methodology. He can explore the design space from either the area dimension or time dimension. A simple extension of FDS can also solve two types of pipeline scheduling problems.[3] Thus, FDLS combined with FDS provides general algorithms that we can tailor to specific applications. Better still, from the implementer's point of view at least, most subroutines are common to both algorithms.

ALLOCATING DATA PATHS

Once we have the schedule and have allocated the functional units, we need to allocate the data paths. Two of the most important subtasks are allocating registers and interconnections. This emphasis is justified by McFarland's experience,[7] which shows that multiplexing costs seem to have the most significant effect on the overall cost-speed tradeoff curve.In HAL, we do this allocation by following three transformation steps:[2]

1. Bind operations to functional units.
2. Bind storage operations to registers.
3. Bind data-transfer operations to elements such as multiplexers and buses.

Bind operations to functional units. We bind all arithmetic and logic operations to specific functional units so as to minimize the number of distinct inputs on each one.

Bind storage operations to registers. We create a storage operation for each data transfer that crosses a c-step boundary. We use a novel technique to divide the variable's life into two intervals. The first interval lasts one c-step and is assigned to a local storage operation. The remaining c-steps are assigned to the second storage operation. Typically, we assign the two storage operations to the same register. In many cases, however, we could lower the interconnection cost by assigning them to different registers.

Figure 10 illustrates the benefits of local storage. Figure 10a uses regular storage operations and yields the rather complex data path shown in the bottom of the figure. In Figure 10b, the local storage operation in c-step 1 is assigned to R1 while the rest of the operation's life is assigned to R2. The result is a much simpler data path.

Bind data-transfer operations to interconnections. Here, we temporarily bind data transfers to interconnections by creating multiplexers and connecting them to the input of every register and functional unit. We use these multiplexers to form a transfer path to each of their input source objects. We preserve single-input multiplexers because we may want to merge them with other multiplexers to form a bus.

Multiplexing costs seem to have the most significant effect on the overall cost-speed tradeoff curve.

MERGING REGISTERS

In this important optimization step, we selectively merge registers with disjoint lifetimes. To illustrate the difficulty of this problem, we again refer to the DiffEq example. We determine for each register

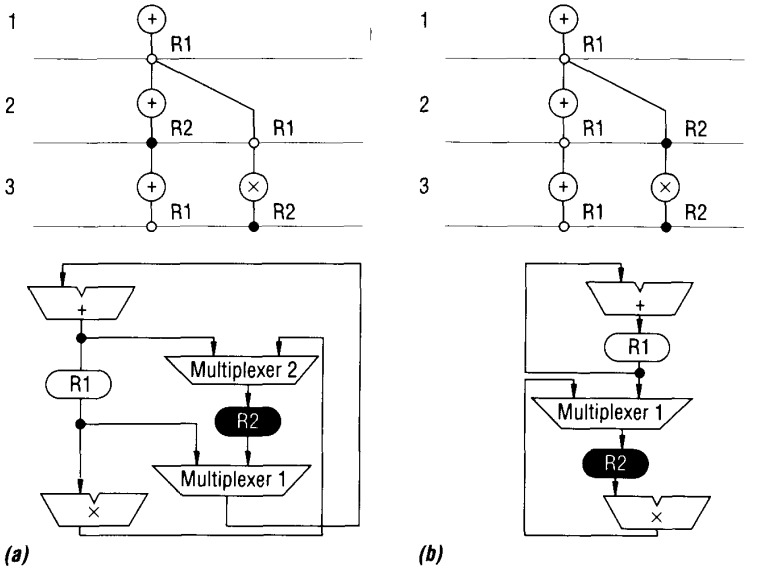


Figure 10. *Comparison of regular storage* **(a)** *and local storage* **(b).**

In this type of partitioning, we reduce the compatibility graph by considering only the registers that have an interconnection affinity above a certain threshold.

defined the set of disjoint registers. This yields a register-compatibility graph in which each edge represents the disjoint usage times of two registers. These registers are candidates for merging. We could use exhaustive clique partitioning to generate all possible register groupings, but this problem is NP-complete.

Researchers at the University of Southern California[1] exploit the left-edge algorithm, which yields an optimal track assignment in channel routing. Here, we can apply it to guarantee the minimum number of registers.

Another alternative to exhaustive clique partitioning is heuristic clique partitioning. The Facet system from Carnegie-Mellon University uses this special form of clique partitioning[1] to determine a near-minimum number of registers. It incorporates heuristics based on the clique graph structure to prune the graph and reduce the number of possible cliques.

Both these approaches, however, ignore the impact of register merging on interconnection costs. An earlier version of HAL[2] tried to take interconnection costs into account by merging only registers that were connected to the same functional units, but this technique was not general enough to have much use.

The current version of HAL uses exhaustive clique partitioning but with reduced compatibility graphs, so the problem is no longer NP-complete. This extended version of the earlier HAL register-merging algorithm is called *weight-directed clique partitioning*. In this type of partitioning, we reduce the compatibility graph by considering only the registers that have an interconnection affinity, or structural weight, above a certain threshold. We progressively lower this threshold as we get fewer compatible pairs after each iteration of the merging routine. We determine the structural weights from the preliminary binding of functional units, multiplexers, and interconnections we performed earlier. The register pairs whose merging gives the lowest interconnection cost are give the highest weight, as Figure 11 shows. The weight values of 1 to 4 are for illustration only. The actual values are an estimate of the interconnection area that would

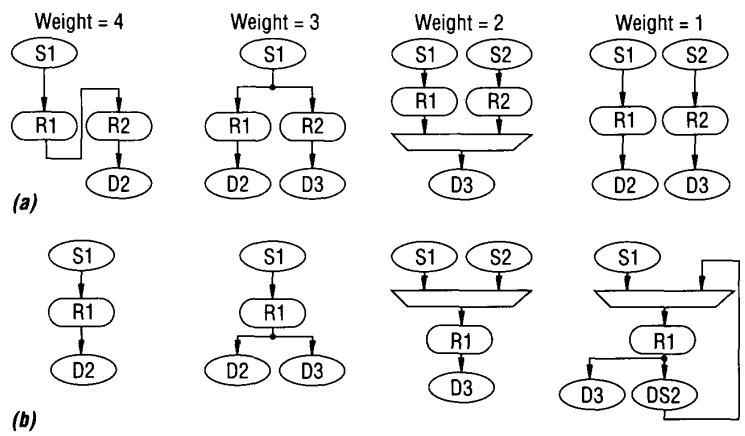


Figure 11. *Interconnection weights for different register merges before* ***(a)*** *and after* ***(b)*** *the merge.*

be saved. We evaluate this area using a function that represents the cost of the interconnection area. Thus, weights can be either positive or negative.

If we set the weight threshold high enough, we can limit the complexity of the clique graph at will. For example, by applying a weight threshold of 4 to the clique graph of Figure 12, we reduce it to the graph represented by the dotted edges. As we refine the selection of registers for merging, the weight value selected varies with the number of remaining edges in the compatibility graph. This number must be small enough to allow exhaustive (or semiexhaustive) clique partitioning in a reasonable time. We then generate all possible merges. For each of these, we evaluate the associated interconnection costs and select the one with the lowest combined register and interconnection cost. For the DiffEq example, the register groups chosen are

{R20, R21}, {R17, R25}, {R16, R23}, R18, R19, R22}.

We naturally get fewer compatible register pairs with each merge, and we repeat the process until no more merges are possible. In our example, by lowering the threshold to 3, we obtain the final solution which is a clique made up of five register groups:

{R20, R21}, {R17, R24, R25}, {R16, R23}, {R18, R19}, R22}

This is the smallest number of registers possible and—perhaps more important—represents a configuration with a very low interconnection cost.

In addition to low register and interconnection costs, we have a good structural partitioning of the design because we group highly connected elements.

MERGING MULTIPLEXERS

A multiplexer is a data-transfer element that has multiple inputs and a single output, while a bus is an element with multiple inputs and multiple outputs. The problem of merging multiplexers into buses is somewhat the same as the problem of merging registers except that a multiplexer is not used continuously. Thus, a multiplexer created in the method described is assigned to a series of c-steps that are not necessarily contiguous. We cannot use algorithms like the left edge for this reason. We can, however, use a clique partitioning method similar to what we used for register merging.

Our threshold for merging multiplexers is not interconnection weights, but the number of common inputs between multiplexer pairs. A merge cannot create more than two levels of buses or multiplexers for each transfer path from register to functional unit to register. Thus, we ensure that the delay through the interconnection paths is minimal. SAW[6] and Splicer[8] allow up to four levels of buses and multiplexers.

DESIGN PARTITIONING

The data paths in Figures 13, 14, and 15 show that in addition to low register and interconnection costs, we have a good structural partitioning of the design. The reason is that we group highly connected elements and so use interconnection information to prune the design space. Although the two merging algorithms we have described are aimed at a general distributed architecture, we can refine them for specific applications. We can introduce different weights to enforce predefined structural or physical partitions that

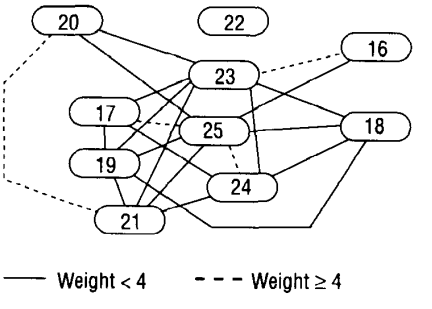


Figure 12. *Compatibility graph for the DiffEq example in Figure 1.*

Our threshold for merging multiplexers is not interconnection weights, but the number of common inputs between multiplexer pairs.

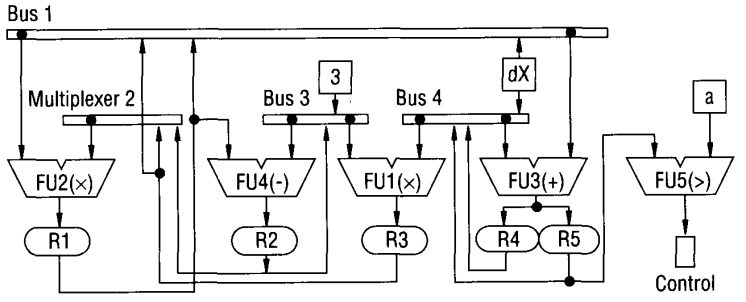


Figure 13. *HAL datapath using nonpipelined functional units.*

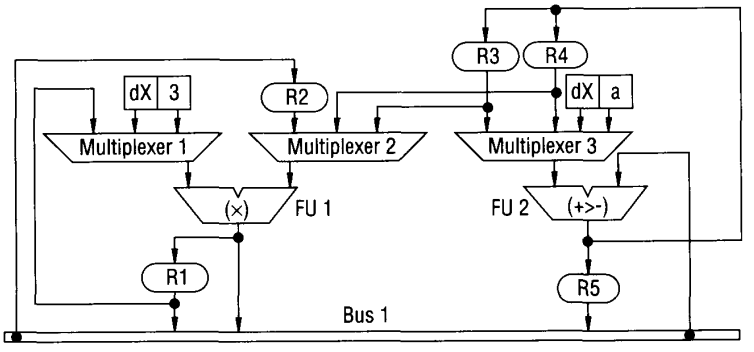


Figure 14. *HAL datapath for DiffEq using a pipelined multiplier.*

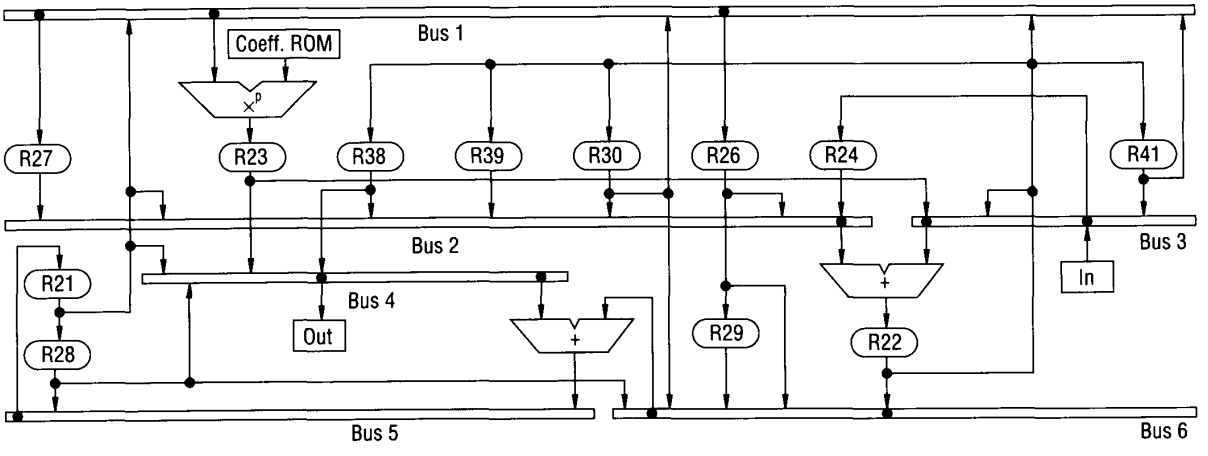


Figure 15. *Data path for wave filter example; $\times^p$= pipelined multiplier.*

correspond to a specific architecture. Registers (multiplexers) within the same partition would have the highest weight to ensure that they are merged first. By varying the value of the weight, we can experiment with compromises in reducing the total number of interconnection lines and preserving the design partitions.

EXPERIMENTAL RESULTS

We present two examples to compare our approach with other systems: the DiffEq differential equation described earlier and a fifth-order waveform elliptic wave filter,[5] which we used earlier to compare scheduling algorithms. We did not fine-tune our algorithm to suit either of these applications. The CPU execution times on a Xerox 1108 are for complete synthesis, including scheduling, allocating functional units, and binding registers and buses.

DIFFERENTIAL EQUATION

The DiffEq example, first presented in a paper from the 1986 Design Automation Conference,[2] is used in Splicer[8] and Catree.[9] The first four columns of Table 2 are a summary of costs for these systems without pipelined functional units and for the current version of HAL using the register and bus merging algorithms. The results of an earlier version of HAL are given as the baseline for the other systems' percentages. Figure 13 shows the HAL data path using nonpipelined functional units.

In column five of the table, we show results for Splicer with a two-stage, pipelined multiplier. The cost of the functional units in this configuration is consequently much lower. In column six of the table, we show results from HAL when we use force-directed scheduling and weight-directed clique partitioning, as shown in Figure 14.

As the table shows, Splicer and Catree improved on the early HAL's results, but the current HAL's register cost is equal to the best result achieved in the other systems, while the interconnection costs are significantly lower.

WAVE FILTER

Table 3 compares the HAL designs with those of SAW,[6] Splicer,[8] and Catree.[9] The table includes the number of multiplexer inputs required—a crude measure of relative interconnection costs. Since HAL also uses buses, this value is actually the combined number of inputs to multiplexers and buses, where we consider a bus the same as a multiplexer with multiple outputs. To help isolate the effects of strategies to allocate registers and interconnections, we compare results with identical time constraints and functional unit allocations.

For all these examples, and with all other costs being equal, the interconnection costs are significantly lower with HAL. Roughly half these savings are the result of using local storage operations, which divide each variable's life into two parts. The total CPU time varies between two and eight minutes.

The bottom row indicates the overall best result from HAL. In this case, HAL uses a two-stage, pipelined multiplier. This design has the lowest interconnection cost of all the examples. The data path for this result is given in Figure 15. The right operand of the pipelined multiplier is a small ROM that contains the filter coefficients.

Our results clearly illustrate the effectiveness of using time frames, distribution graphs, and concurrency balancing.

We still have issues to address such as the need to incorporate preliminary floor-planning information into synthesis.

We can make three observations about the data path in the figure.

1. We used six buses, which is relatively few. The connections to and from these buses are mostly local.
2. Because the path from a single register to a functional unit to a register never crosses more than two levels of multiplexers or buses, we reduce the clock-cycle time. Data paths in SAW and Splicer cross up to four levels.
3. As in the DiffEq example, most interconnections are local to the area defined by a single functional unit. The bipartition in SAW is more clearly defined, however, and probably would be easier to lay out.

Our algorithms complete two important tasks in high-level synthesis: scheduling under time and resource constraints and allocating buses and registers to minimize interconnection costs. The force-directed scheduling and force-directed list scheduling algorithms take into account the cost of functional units as well as the cost of storage and interconnections. Also, by combining these algorithms, we have a flexible stepwise refinement approach to exploring the design space. Our results clearly illustrate the effectiveness of using time frames, distribution graphs, and concurrency balancing. This is confirmed by results from systems

Table 2. *Summary of area costs for DiffEq example.*

System	HAL 86	Splicer	Catree	HAL 89	Splicer	HAL 89
CPU	40 sec	N/A	N/A	50 sec	N/A	120 sec
Intercon-nection (%)	100	86	93	79	107	84
Register (%)	100	100	83	83	100	83
Functional. Unit (%)	100	100	100	100	64	57

Table 3. *Comparison of register and interconnection requirements; + = adder, x = multiplier, x^p = pipelined multiplier.*

System	Number of Control Steps	Number of Multipliers and Adders	Number of Registers	Number of Multiplexer Inputs
HAL	19	2x, 2+	12	28
SAW	19	2x, 2+	12	34 (+21%)
HAL	21	1x, 2+	12	30
Splicer	21	1x, 2+	N/A	35 (+17%)
HAL	17	$2x^p$, 3+	12	31
Catree	17	$2x^p$, 3+	12	38 (+22%)
HAL	19	$1x^p$, 2+	12	26

that exploit the same principles, as reported at this year's *High-Level Synthesis Workshop.*

The approach we have described for allocating registers and buses exploits a simple but powerful weight-directed clique partitioning algorithm based on merging highly interconnected elements. This algorithm prunes the exploration space while favoring merges that reduce interconnection costs.

We still have issues to address such as the need to include floor-planning information in synthesis. We could do this by assigning higher weights to the merging of registers (or buses) in the same floor-plan partition. On the other hand, the whole issue of control costs has gone largely ignored. Although balancing the concurrency of data-path events should reduce the number of control lines, and making the schedule shorter usually reduces the controller size, more accurate metrics for control cost still have to be developed. D&T

Pierre G. Paulin is a member of the VLSI System Design and Synthesis Department at Bell-Northern Research. Prior to joining BNR full time, he was part of a cooperative research project with Carleton University and BNR to do the work reported in this article. He holds a BSc in engineering physics and an MScA in electrical engineering from Laval University and a PhD in electronics from Carleton University. He is a member of the IEEE Computer Society.

ACKNOWLEDGMENTS

We thank Jenny Midwinter for her insight and advice on interconnection allocation and Emil Girczyc who helped lay the groundwork for the original FDS algorithm.

This research was funded in part by grants from the National Society of Electronics Research Council of Canada, from Carleton University, and from BNR, Ottawa, as part of a cooperative PhD project.

REFERENCES

1. M. McFarland, A. Parker, and R. Camposano, "Tutorial on High-Level Synthesis," *Proc. 25th Design Automation Conf.*, July 1988, pp. 330-336.
2. P. Paulin, J. Knight, and E. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis," *Proc. Design Automation Conf.*, July 1986, pp. 263-270.
3. P. Paulin and J. Knight, "Force-Directed Scheduling for the Behavioral Synthesis of ASICs," *IEEE Trans. Computer-Aided Design*, Vol. CAD-8, Vol. 6, June 1989, pp. 661-679.
4. S. Davidson et al., "Some Experiments in Local Microcode Compaction for Horizontal Machines," *IEEE Trans. Computers, Vol. C-30, No. 7, July 1981, pp. 460-477.*
5. S. Kung, H. Whitehouse, and T. Kailath, *VLSI and Modern Signal Processing*, Prentice Hall, Englewood Cliffs, N.J., 1985.
6. D. Thomas et al., "The System Architect's Workbench," *Proc. Design Automation Conf.*, July 1988, pp. 337-343.
7. M. McFarland, "Reevaluating the Design Space for Register-Transfer Hardware Synthesis," Proc. Int'l Conf. Computer-Aided Design, Nov. 1987, pp. 262-265.
8. B. Pangrle, "Splicer: A Heuristic Approach to Connectivity Binding," *Proc. Design Automation Conf.*, July 1988, pp. 536-541.
9. C. Gebotys and M. Elmasry, "VLSI Design Synthesis with Testability," *Proc. Design Automation Conf.*, July 1988, pp. 16-21.

John P. Knight is an associate professor in the Department of Electronics, Carleton University, Ottawa. He holds a BSc from Queen's University and an MScA and a a PhD from the University of Toronto—all in electrical engineering.

Direct comments or questions on this article to P. Paulin, BNR, PO 3511, Station C, Ottawa, K1Y 4H7, Canada, or to J. Knight, Carleton Univ., Dept. of Electronics, Ottawa, K1S 5B6, Canada.