VLSI Design
1995, Vol. 3, No. 1, pp. 21–36
Reprints available directly from the publisher
Photocopying permitted by license only

An ILP Solution for Optimum Scheduling, Module and Register Allocation, and Operation Binding in Datapath Synthesis

T.C. WILSON
VLSI-CAD Group, Department of Computing and Information Science, University of Guelph, Guelph, Ontario, CANADA N1G 2W1

N. MUKHERJEE
Department of Electrical Engineering, McGill University, Montreal, Quebec, CANADA H3A 2A7

M.K. GARG
IBM Canada Laboratories, Don Mills, Ontario, CANADA M3C 1H7

D.K. BANERJI[†]
VLSI-CAD Group, Department of Computing and Information Science, University of Guelph, Guelph, Ontario, CANADA N1G 2W1

(Received June 18, 1993, Revised November 9, 1993)

We present an integrated and optimal solution to the problems of operator scheduling, module and register allocation, and operator binding in datapath synthesis. The solution is based on an integer linear programming (ILP) model that minimizes a weighted sum of module area and total execution time under very general assumptions of module capabilities. In particular, a module may execute an arbitrary combination of operations, possibly using different numbers of control steps for different operations. Furthermore, operations may be implemented by a variety of modules, possibly requiring different numbers of control steps depending on the modules chosen. This generality in the complexity and mixture of modules is unqiue to our system and leads to an optimum selection of modules to meet specified design constraints. Significant extensions include the ability to incorporate pipelined functional units and operator chaining in an integrated manner. Straightforward extension to multi-block synthesis is discussed briefly but the details are omitted due to space considerations.

Key Words: *Datapath; Synthesis; Scheduling; Allocation; Binding; Registers*

1. INTRODUCTION

Scheduling, *module allocation*, *and binding* of operators are three major tasks in behavioral-level datapath synthesis. *Scheduling* involves the assignment of operations to control steps; *module allocation* selects the set of functional units that will participate in the design; and *operation binding* associates these operations with particular instances of the functional units. This association consists of two sub-tasks: (i) *type binding*, which concerns the choice of module type for each operation; and (ii) *instance binding*, which identifies the specific module of the designated type.

In this paper, we present an *optimum and integrated* solution to the problems of scheduling, allocation, and binding. The solution utilizes an integer linear program (ILP) that minimizes a weighted sum of module area and total execution time under very general assumptions of module capability. In particular, a module may execute an arbitrary combination of operations, possibly using different numbers of control steps for different operations. Furthermore, the same operation may be implemented by a variety of modules, possibly involving different num-

[†]Please send all correspondence to the corresponding author.

bers of control steps. Important extensions to this work include the ability to handle multiple-block designs, the use of pipelined functional units, and the possibility of operator chaining. Chaining refers to executing a sequence of operations on the same control step using different functional units without intervening storage. We will show how to incorporate these possibilities within the ILP, not as specially handled cases but as alternative module options and strategic opportunities.

Like functional units, registers can consume significant amounts of space on a chip. An important extension to our basic model incorporates register allocation in an integrated manner. This allows a schedule and module allocation to be found that does not require more than a predetermined number of registers. Alternatively, the number of registers can be determined by the ILP. In this case, register area and module area can be traded off, so that an optimum configuration is found within an overall area limit.

2. BACKGROUND

The problems of scheduling, allocation, and module binding have received considerable attention, and most recent works have stressed the need for an integrated approach. There is also considerable effort being applied to handling a more general mix of module types. Most early efforts, however, assume little or no flexibility in module capability and treat scheduling and module allocation as independent problems, where type binding, module allocation, scheduling, and instance binding are performed in some linear order, often the one just listed. Among many papers in this category, we note the ILP model presented in [1], which considers only the problem of scheduling, but includes pipelined units and operation chaining.

Recently, several papers have presented an "integrated" approach to the problems of scheduling, allocation, and binding. They are integrated in the sense that they consider scheduling, allocation, and binding issues together at each decision point. But the decisions are sequential and are based on knowledge of the past and perhaps on estimates of what may follow. They are constructive heuristics which decide on one operation (or step) at a time and may not yield optimal designs in the end. For example, Cloutier and Thomas [2] extend the force-directed scheduling approach to also include type and instance binding. The binding allows an operation to be associated with any of a variety of module types. Ramachandran and Gajski [3] use a similar approach, but permit a more general module library. They consider different implementations of simple modules but do not allow arbitrary multi-function modules. Balakrishnan and Marwedel [4] develop a design one step at a time and use a linear program to design each step.

Papachristou and Konuk [5] suggest an iterative refinement procedure which alternates between allocation and scheduling. They use force-directed scheduling, followed by an ILP to allocate a minimum area set of modules. The allocation may be used to constrain another round of scheduling, followed by a new allocation, etc. They consider only operations that execute in one control step.

Some approaches are global in the sense of attacking the scheduling and allocation problems for all operations at the same time. Devadas and Newton [6] use simulated-annealing-based algorithms to explore the solution space. Papachristou and Nourani [7] use "moveframe scheduling" and Liapunov's stability function to solve the combined problem. Although their algorithm can handle multi-function units, they cannot accomodate different physical implementations of a functional unit or complex multi-function units taking different times for different operations.

Gebotys and Elmasry [8] is the only other ILP formulation that performs scheduling, allocation, and binding globally and optimally. However, they require the execution time of each operation to be a constant. This restricts their module library, and curtails possibililities for dynamically selecting different implementations of an operation. Their algorithm is "valid for straight line code" [8], and has not been extended to handle multiple block designs. An interesting innovation is to exploit the feature of "facets" which reduces the time required to find an integer solution. They also consider register area, but do not cope directly with values which are used by multiple unrelated operations. They have recently [9] included the notion of control step length in their model. The CATHEDRAL silicon compiler [13] considers many issues in synthesis: operator scheduling, resource allocation, interconnect minimization, memory management, address generation, and instruction word minimization. In contrrast, this work only focuses on a subset of these issues, namely operator scheduling, module and register allocation, and module binding. Interconnect optimization, including module re-binding is handled separately in our synthesis system.

3. BASIC CONCEPTS

Given a library of available functional unit types and the data flow graph (DFG), the problem is to determine a schedule and choose a suitable combination of units, so that:

- the operations are performed by the units having the required capability,
- a module can execute at most one operation at a time,
- the operations are executed in the order specified by the DFG,
- the total area is limited (or minimized),
- the execution occurs within a number of steps that is limited (or minimized).

Two kinds of constraints play a crucial role during scheduling and allocation. First and most familiar are the **data dependency** constraints (or **DD-constraints**). These ensure that no operation begins execution until all of its operand values have been computed. Each DD-constraint corresponds to an edge of the data flow graph (DFG) and bounds the starting time of an operation by the completion times of its predecessors on the DFG. Completion times, in turn, depend on the particular units allocated. Thus, DD-constraints enforce the ordering imposed by the DFG, with consideration given to the type of unit allocated to each operation.

The second important set of constraints focuses on each individual unit – its type and instance. If any unit is used by more than one operation, then those operations must be implemented in some sequential order. No operation can begin until its predecessor (if any) on the same unit has completed its execution. Although this constraint also bounds starting times by the completion times of the predecessors, this ordering is unrelated to data dependency. The ordering here is imposed by the mapping of operations to units and is not predetermined. It must be ascertained *dynamically*, if (multi-step) modules are to be reused. We refer to the ordering among operations that are executed by the same unit as their **unit-use** ordering, enforced by unit-use constraints (**UU-constraints** for short).

In the ILP solution to appear shortly, we introduce 0-1 variables p_{ij} whenever it is feasible for operation i to precede operation j in their use of the same functional unit (whatever unit that may turn out to be). When the ILP sets $p_{ij} = 1$, operation i is presumed to precede operation j. If we restrict attention to those p_{ij} that do become 1 in the final solution, and further restrict attention to those p's referring to immediate predecessors, these p's describe disjoint paths that traverse all the operation nodes of the DFG. Each such "UU" path refers to one allocated funcitonal unit and identifies the sequence of operations that employ that unit.

4. THE GENERAL ILP SOLUTION

The combined scheduling, allocation, and binding problem is formulated and solved as an integer linear programming problem. The following symbols are used to identify object types and to index important objects in the ILP solution:

object	**indexed by**
operation	i, j
unit	m

The unit index, m identifies individual instances of each module (not the module type) being considered in the solution. An upper limit on the number of instances for each type is determined by a preliminary appraisal of the DFG, module library, and global time and resource constraints.

Certain constants pertain to the module library being considered:

A_m is the chip area consumed by including unit m in the design;

M_i is the set of modules that are capable of executing operation i (an index set on m);

D_{im} where $m \in M_i$, is the number of control steps that unit m would require to complete execution of operation i (not defined if $m \notin M_i$).

In addition, the ILP solution makes use of the following predefined values and sets of objects that depend on the particular DFG:

F is the *final* operation set, containing those operations that have no successors in the DFG;

$E[i, j]$ is a 0-1 matrix representing the edge set of the DFG (DD-constraints); if 1 then op_i *immediately* precedes op_j in the DFG;

$Q[i, j]$ is a 0-1 matrix indicating whether operation i could possibly precede operation j in their use of some common module (potential UU-constraints); if 1, then M_i and M_j intersect;

N is some large constant, larger than the maximum number of control steps and the number of operations.

The ILP manipulates the following variables:

y_i: integer; denotes the starting control step for op_i

$x_{im} = 0, 1$	$x_{im} = 1 \Rightarrow op_i$ is executed by unit m (undefined if $m \notin M_i$)
$u_m = 0, 1$	$u_m = 1 \Rightarrow$ unit m is required in the design
$p_{ij} = 0, 1$	$p_{ij} = 1 \Rightarrow op_i$ precedes op_j in their use of the same unit (*undefined if* $Q[i, j] = 0$)

Since both DD and UU-constraints delay the start of operations until their predecessors have finished, the completion step of an operation becomes an important quantity within the solution. The first *free* step, following execution of operation i, when using module m, is given by the expression:

$$y_i + \sum_{m \in M_i} x_{im} D_{im}$$

Note that exactly one of the x_{im} will be non-zero, thereby causing the summation to equal the corresponding D_{im}.

The two major parameters of interest are time and area. The symbol T represents the total number of steps required by the schedule, including the completion of all *final* operations in F. T may be constrained in advance, may be a value to minimize as part of the objective function, or both. Likewise the total module area required by the design may be constrained, may be minimized by the objective function, or both. Its value is given by the expression:

$$\sum_m u_m A_m$$

We can now consider the fundamental ILP formulation. This particular version seeks to minimize a weighted sum of total time, T, and total module area. Their relative weights are denoted W_{time} and W_{area} respectively. Extensions to the ILP are presented later.

OBJECTIVE

$$\min\left[W_{time} * T + w_{area} * \sum_m u_m A_m\right]$$

CONSTRAINTS

(1) All operations must begin on some control step:

$$\forall i \colon y_i > 0$$

(2) Each operation must be assigned to some functional unit:

$$\forall i \colon \sum_{m \in M_i} x_{im} = 1$$

(3) If an operation is mapped to a unit, that unit must be included in the final design:

$$\forall m \colon \sum_i x_{im} \le u_m N$$

(4) The overall completion time, T, is bounded by the completion time of each operation in the final set, F:

$$\forall i \in F \colon y_i + \sum_{m \in M_i} x_{im} D_{im} \le T + 1$$

(5) The starting time of each operation is bounded by the completion times of its predecessors on the DFG (DD-precedence):

$$\forall i, j \text{ where } E[i, j] = 1 \colon y_i + \sum_{m \in M_i} x_{im} D_{im} \le y_j$$

(6) If two operations are assigned to the same unit, then one must precede the other in its unit use (UU-precedence):

$\forall i, j, m$, where $m \in M_i \cap M_j$:

$$x_{im} + x_{jm} - (p_{ij} + p_{ji}) \le 1$$

(7) If one operation precedes another in its use of the same unit, then the starting time of the successor is bounded by the completion time of the predecessor (UU-precedence):

$\forall i, j$ where $i < j$:

(a) If $Q[i, j] = 1$ and $E[i, j] = 0$:

$$y_i + \sum_{m \in M_i} x_{im} D_{im} \le y_j + (1 - p_{ij}) N$$

(b) If $Q[j, i] = 1$ and $E[j, i] = 0$:

$$y_j + \sum_{m \in M_j} x_{jm} D_{jm} \le y_i + (1 - p_{ji}) N$$

Constraint 6 forces either p_{ij} or p_{ji} to be 1, in case operations i and j are ever both assigned to the same unit, m. Constraint 7 will guarantee that at most one of these p's can ever be 1. If either $Q[i, j] = 0$ or $Q[j, i] = 0$, then at most one direction of precedence is possible, at least one of the p's is undefined, and both constraints become simpler. If $Q[i, j] = 0, Q[j, i] = 0$, and $M_i \cap M_j \neq \theta$, then operations i and j must overlap in time, and constraint 6 reduces to

$$x_{im} + x_{jm} \leq 1$$

which enforces mutually exclusive use of unit m.

Constraint 7 is meaningful only if one of the p_{ij} or p_{ji} does, in fact, become 1. In that case, the righthand side of the constraint will contain simply y, the successor's starting time which is being constrainted. However, when $p = 0$, the corresponding righthand side includes a large constant, N, which makes the constraint satisfied automatically, thereby deactivating it.

5. SIMPLE ENHANCEMENTS TO THE GENERAL FORMULATION

The integer linear program presented in Sec. 4 is the basis for many possible variations and extensions. This section discusses some of the simpler functional enhancements, and subsequent sections discuss more substantial enchancements.

5.1 Other Objectives and Constraints

The total time and area available on a chip are often restricted to lie within some prespecified limits. These are easily incorporated by adding the following constraints:

$$(8) \qquad T \leq \text{StepLimit}$$

$$(9) \qquad \sum_m u_m A_m \leq \text{AreaLimit}$$

In practice, the general objective function which minimizes a combination of time and area may either cause the ILP to execute too slowly or simply be inappropriate for the situation. We usually have a specific number of control steps in mind (requiring constraint 8) and seek to minimize the resources that will meet this scheduling deadline. The objective then becomes:

$$\min \sum_m u_m A_m$$

By progressing through a series of plausible time limits using this formulation, the sequence of solutions clearly displays the area/time tradeoffs to the designer.

Likewise, one can find the shortest schedule obtainable with a given set of resources. After specifying either total area (requiring constraint 9), or by specifying which particular functional units are to be used, the objective becomes simply: $\min T$. The fastest option, and often a very useful application, is to verify the existence of some feasible solution that meets both time and total area constraints; this requires no objective function at all.

Of course, additional constraints can be added to limit the instances of certain unit classes or to exclude certain combinations of units from appearing in the design. The designer can always control the module types from which the ILP is allowed to select. For example, to specify that some set of modules, X, is mutually exclusive, we include the constraint: $\sum_{m \in X} u_m \leq 1$. The constraint $x_{im} = 1$ requires that operation i be done using module m; $x_{im} = 0$ (or failing to define x_{im} for this particular value of m) does not allow module m to be employed for operation i. If a module, m, is known to be in the design anyway, u_m should be set to 1, and its cost, A_m, can be set to 0 to promote its reuse within the ILP.

An interesting possibility involves imposing constraints on the interval between the steps on which two operations are scheduled. For example, if operation i must occur exactly k control steps before operation j, then we set $y_j - y_i = k$. Many other simple constraints can specify other requirements of the design.

5.2 Pipelined Functional Units

A simple but powerful extension allows the inclusion of pipelined functional units in the design. These differ from ordinary multi-step units in their ability to begin a new computation before completion an earlier one. The minimum interval between accepting successive inputs is called the *latency* of the unit, denoted L. For non-pipelined units, latency equals execution time, and $L_{im} = D_{im}$. Our model can considers both pipelined and non-pipelined implementation of any operation.

For DD-constrained operations, pipelined units make no difference. Operations cannot begin until their operands have been computed. However, for *data independent* operations that reuse the same (pipelined) unit, the successor need wait only for the latency period of the device. Constraint (7) thus becomes:

$\forall i, j$ where $i < j$

(7*a*) If $Q[i,j] = 1$ and $E[i,j] = 0$:

$$y_i + \sum_{m \in M_i} x_{im} L_{im} \le y_j + (1 - p_{ij})N$$

(7*b*) If $Q[j,i] = 1$ and $E[j,i] = 0$:

$$y_j + \sum_{m \in M_j} x_{jm} L_{jm} \le y_i + (1 - p_{ji})N$$

This simple modification to Constraint (7) is all that is needed to include pipelined units in the design.

5.3 Multiple Block Designs

We use the preceding ILP formulation as the basis for multi-block schedules, allocations, and bindings–not block-by-block or just along *critical* paths, but for all blocks of a design *simultaneously*. In fact, the identity of critical paths cannot always be known until module binding has determined the time for each operation.

The general idea is suggested in Fig. 1. Operations are confined to remain in their original control blocks, and the blocks themselves always execute in some order, never concurrently. These assumptions, respectively, remove the need for either DD or UU constraints between operation pairs appearing in different control blocks. In other words, schedules and allocations in two separate blocks cannot interfere with each other. Thus, the respective data flow graphs for each block remain disconnected as they are submitted to the ILP. While this may seem like independent scheduling of each block, the modules are all being drawn from a common pool, and their total area is counted in the objective function. Thus, each block is being scheduled concurrently with the others, so as to maximize possibilities for reusing modules among all blocks in the design. In addition, *total* execution time is being measured by the objective function. The figure suggests how each control path provides a bound on this time. Therefore, in paths that turn out to be critical, there may be a tradeoff in the times allocated to their individual blocks. Note that step numbers within each block begin at 1, with an appropriate offset added after the ILP. Before this linearizing offset is added, the ILP may appear to map several operations to the same module on the same control step, but operations from the same control block can never overlap in this way.

In practice we usually seek a feasible solution to multiblock designs, in which the total area is constrained, as is the longest execution time through

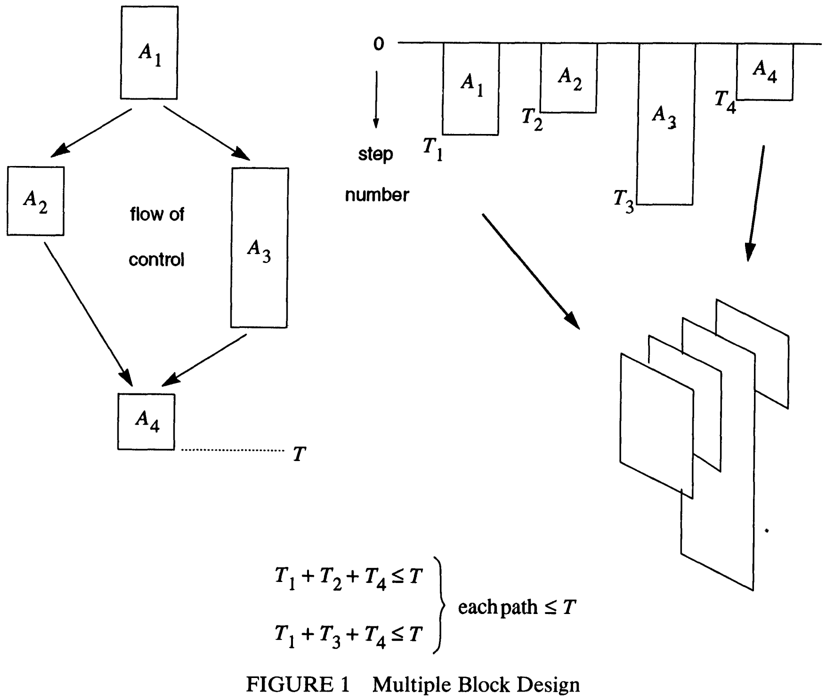


FIGURE 1 Multiple Block Design

any control path. One such design includes both a convolution algorithm and a bandpass filter. This particular design involves 34 control blocks, 58 operations in total, and from 1 to 8 operations per block. Finding a feasible solution requires only 4 seconds of execution time on a SPARCstation 2. The complexity of all the issues involved in multi-block scheduling, allocation, and binding cannot be adequately covered here; these details are provided in a separate manuscript under preparation.

5.4 Additional Enhancements

Some other extensions that have been incorporated in the ILP model will be explained later in separate sections of their own. One of these is to allow *chaining of funcitonal units* within single control steps. Another is to account for *register allocation*, along with module allocation. This can be done in two ways. The schedule can be constrained, so that no more than a predefined number of registers will be required or, alternatively, the registers and modules can both be allocated by the same ILP, so that total area is either minimized or kept below a limiting value. These extensions will be described in Sections 7 and 8, respectively.

6. OPERATIONAL ENHANCEMENTS

Integer linear programs have the potential for long running times on large or inappropriate problems. Thus, it is important to incorporate features which reduce their running times and to use them in appropriate ways. The following two subsections explain specific techniques which greatly accelerate the ILP running times for the problem considered in this paper. The third subsection discusses the appropriate use of these ILPs in the context of a large synthesis system.

6.1 Limited Instance Binding

A dramatic speedup in ILP running time results from exploiting the following observation: For operations using modules which always execute in one control step, instance binding does not affect the schedule or the number of units allocated. As long as some appropriate module is available, the module's exact identity need not be decided until later. All available single-step modules with appropriate functionality are essentially equivalent. *Type binding* alone determines the number of units required from each module class and, hence, the area cost of the design. Type binding by itself is also sufficient to constrain scheduling for single-step operations.

On the other hand, properly scheduling multi-step operations does require *explicit instance binding*, in order to ensure continuity across control step boundaries and non-overlapping use of multi-step units. Even operations on these modules that require only one step have to be identified, in order to avoid conflict with other operations that require multiple steps on the same module. Thus, the need for instance binding depends on the characteristics of the module, as well as the operation.

A functional unit *type* is characterized by its area, operator capabilities, and the time required to perform each operation. As a result, the functional units are categorized into two classes: 1) A *single-step* functional unit, which takes one control step for executing *all* the operations involved in the DFG that it is capable of performing, and 2) A *multi-step* functional unit, which takes multiple steps for executing *at least one* of its operations that also appears in the DFG. For the integrated ILP, the operations are classifed into three categories, according to the types of functional units present in the module library. The categories are:

(*i*) Class A-ops: Operations that always need a *single* control step and, in addition, can be exectued *only* by *single-step* functional units;

(*ii*) Class C-ops: Operations for which all applicable functional units are *multi-step* units (even if the operation in question may take only one step).

(*iii*) Class B-ops: All other operations that do not fall in either of the above two classes.

The categorization of operations depends on the type of functional units available for a design. This, in turn, affects the ILP formulation. Class C is the most general and corresponds to the general ILP solution of Sec. 4. Class A operations will always be associated with single-step modules which may remain anonymous. By themselves, they could be handled by an ILP that is much simpler than the one presented in Sec. 4. Operations in classes A and C will have only those variables and constraints defined that are appropriate to their respective classes. B-ops may wind up being assigned to either module class and will have to participate in both kinds of constraints. If a class B operation is, in fact, mapped to a single-step unit type, the particular unit instance will not matter and will not be decided. If, however, the same class B operation is mapped to a multi-step unit, the specific unit must be determined. It is useful to visualize the accelerated formulation as two tandem linear programs, with class A and C operations belonging to their respective ILPs, and class B operations as straddling both ILPs.

The following symbols are used to identify object types and to index important objects in the ILP solution:

object	**indexed by**
operation	i, j
unit	m, n
module type	k, h
control step	s, t

The unit index, m, identifies individual unit *instances* of each multi-step module type, whereas k identifies only the single-step module *types* being considered in the solution. We introduce constant A_k to denote the chip area consumed by including an instance of a unit from type k in the design. We restrict the earlier definition of $Q[i, j]$ to indicate whether operation i could possibly precede operation j in their use of some common *multi-step* module.

The ILP requires a few new variables and some old ones to be appropraitely reinterpreted:

n_k: *integer*; number of instances from *single-step module* class k

$x_{im} = 0, 1$	$x_{im} = 1 \Rightarrow$	op_i is executed by *multi-step unit m*
$u_m = 0, 1$	$u_m = 1 \Rightarrow$	*multi-step unit m* is required in design
$p_{ij} = 0, 1$	$p_{ij} = 1 \Rightarrow$	op_i precedes op_j in using the same *multi-step unit*.
$w_{iks} = 0, 1$	$w_{iks} = 1 \Rightarrow$	operation i is executed on step s by a unit from *single-step* type k

The total module area is now given by the expression:

$$\sum_m u_m A_m + \sum_{k>0} n_k A_k$$

This expression may appear in the objective function, in an area-limiting constraint, or both.

We now present the complete accelerated ILP that performs instance binding only for operations that require it. For this purpose, a "special" module type is created and labeled as type $k = 0$. Operations in class B can be done by both single-step and multi-step modules. Depending on the type of functional unit allocated, type binding or instance binding is performed. This allocation is not known a priori and, therefore, provision should be made for both the possibilities; the decision is taken *dynamically*. An operation is assigned to class 0 only when it is not assigned to any regular single-step module type and, therefore, must be bound to some *specific instance* of a multi-step module. Module type 0 is thus an "escape" type, implying participation in the other stream of constraints.

OBJECTIVE

$$\min\left[W_{time} * T + W_{area} * \left(\sum_m u_m A_m + \sum_{k>0} n_k A_k\right)\right]$$

CONSTRAINTS

(1) Every operation in class A and operations in class B that are done by *single-step* modules, must be assigned to some time step and module *type*.

$$\forall i \in A \cup B: \sum_{k \geq 0} \sum_s w_{iks} = 1$$

(2) Every operation in classes B and C must be assigned to some control step.

$$\forall i \in C \cup B: y_i > 0$$

(3) The two different step notations for B-ops have to be equated.

$$\forall i \in B: \sum_{k \geq 0} \sum_s s w_{iks} = y_i$$

(4) Every operation in class C must be assigned to an *instance* of a functional unit.

$$\forall i \in C: \sum_{m \in M_i} x_{im} = 1$$

(5) Class B operations which are not assigned to any regular ($k > 0$) module type must be mapped to some module *instance*.

$$\forall i \in B, k = 0: \sum_s w_{iks} = \sum_{m \in M_i} x_{im}$$

(6) If an operation is assigned to a unit, the unit must be included in the design.

$$\forall m: \sum_{i \in B \cup C} x_{im} \le u_m N$$

(7) Enough instances of each module type must be available to satisfy the demand on each control step.

$$\forall s, k > 0: \sum_{i \in A \cup B} w_{iks} \le n_k$$

(8) The overall completion time T is bounded by the completion times of the operations in the final set F.

$\forall i \in F$:

(*a*) *If* $i \in A \cup B$: $\sum_{k>0} \sum_s s w_{iks} \le T$

(*b*) *If* $i \in C \cup B$: $y_i + \sum_{m \in M_i} x_{im} D_{im} \le T + 1$

(9) The starting time for any operation is bounded by the completion times of its predecessors on the DFG.

$\forall i, j$ where $E[i,j] = 1$:

$$\left.\begin{array}{ll} (a)\ \text{If } i \in A \cup B: & \sum_{k>0} \sum_s s w_{iks} + 1 \\ (b)\ \text{If } i \in C \cup B: & y_i + \sum_{m \in M_i} x_{im} D_{im} \end{array}\right\}$$

$$\le \begin{cases} \sum_{k>0} \sum_t t w_{jkt} & \text{if } j \in A \\ y_j & \text{if } j \in B \cup C \end{cases}$$

(10) If two operations are assigned to the same unit, one must precede the other in its unit use.

$\forall i, j, m$ where $i, j \in C \cup B$ and $m \in M_i \cap M_j$:

$$x_{im} + x_{jm} - (p_{ij} + p_{ji}) \le 1$$

(11) If one operation precedes another in its use of the same unit, then the starting time of the successor is bounded by the completion time of the predecessor.

$\forall i, j$ where $i < j$ and $i, j \in C \cup B$:

(*a*) If $Q[i,j] = 1$ and $E[i,j] = 0$:

$$y_i + \sum_{m \in M_i} x_{im} D_{im} \le y_j + (1 - p_{ij}) N$$

(*b*) If $Q[j,i] = 1$ and $E[j,i] = 0$:

$$y_j + \sum_{m \in M_j} x_{jm} D_{jm} \le y_i + (1 - p_{ji}) N$$

Constraint 3 equates the scheduling step notations for B-ops appropriate to each method, *i.e.*, it causes y_i to equal the value of s corresponding to which $w_{iks} = 1$. Constraint 1 is responsible for allocating operations to module *types*, and if any B-op is allocated to type 0, constraint 5 ensures that it gets mapped to a module *instance*. Constraints 10 and 11 are valid only for pairs of operations that can potentially share a multi-step functional unit.

6.2 Utilizing the Span of Scheduling Possibilities

If the maximum number of control steps is specified, it is viable to determine the last (ALAP), as well as the first (ASAP), control steps on which an operation could possibly be scheduled. This information can be used to explicitly constrain the feasible starting step (y_i) for each operation.

In addition, we also compute the *span* for every operation. The *span* of an operation i, introduced in [3], is defined to be the number of steps in which it can possibly be using some functional unit. Mathematically, this is represented as:

$$SPAN_i = (ALAP_i + D_i^{\min}) - ASAP_i$$

where, $D_i^{\min} = \min_{m \in M_i}\{D_{im}\}$. Note that the fastest module is considered for calculating the span of op_i. This is because the *ALAP* partitions have already been determined on the basis of fastest module types available in the library and, therefore, ($ALAP_i$ + $D_i^{\min}$ − 1) gives the step number by which execution of i must complete. In other words, if an operation gets scheduled in its *ALAP* step, it cannot use any module other than the fastest one available for that operation. Note that for multi-step operations the span can extend beyond their *ALAP* step.

We extend this concept to calculate the mutual or joint spans for pairs of operations that can possibly share some functional unit. This information is utilized in removing redundant p_{ij} and p_{ji} variables from the ILP. The span for a pair of operations i and j, in case i precedes j, is represented as:

$$SPAN_{ij} = (ALAP_j + D_j^{\min}) - ASAP_i$$

where, $D_j^{\min}$ is the delay of the fastest module that can perform op_j. $SPAN_{ij}$ is the *maximun time* both these operations may be active, assuming i begins first. $SPAN_{ij}$ and $SPAN_{ji}$ are often unequal. If there is any module that can complete both the operations within the joint span, then these operations can possibly share that module. On the other hand, if a time constraint restricts their joint span, they cannot use the same module. Potential opportunities for sharing resources between operation pairs determine which p_{ij} and p_{ji} variables are selectively introduced in the ILP model. A p_{ij} variable becomes necessary for sequencing operations i and j if and only if:

$\exists m \in (M_i \cap M_j)$, where,

$$SPAN_{ij} \geq D_{im} + D_{jm}$$

i.e., there is some functional unit that can do operations i and j in this order, and can finish their execution within the span. Similarly, a p_{ji} variable is necessary if and only if:

$\exists m \in (M_i \cap M_j)$, where,

$$SPAN_{ji} \geq D_{im} + D_{jm}$$

In a design, there are cases when both p_{ij} and p_{ji} variables appear. Or, it may be the case that there is no need for defining the p_{ij} and p_{ji} variables. This latter situation may arise under two circumstances. One occurs when the spans for a pair of operations do not overlap at all, and the second is when their spans overlap but leave no room for their sequential execution.

6.3 Using the ILP in Context

Although our accelerated ILP solution typically requires only a few seconds, such an ILP runs the risk of slowing down on an ill-suited problem. In most cases, the cause of poor performance is examination of a large number of alternative optimum solutions, in order to determine that an earlier "provisional" solution is, in fact, optimum. Besides employing local ILP "tricks," there are two basic ways of avoiding such problems. One of these is to incorporate features into the ILP that will eliminate certain symmetries and certain hopeless cases. Postponement of instance bindings for single-step operations, for example, eliminates a large number of equivalent solutions, and the use of spans constricts the search space to an area where a realistic solution must be found.

The more one can constrain the formulation (in meaningful ways), the faster is the ILP likely to run. For instance, it is often possible to determine in advance and communicate to the ILP the minimum functionality required to achieve a schedule with acceptable time, or the bounds on control steps for a given module allocation. One good source of additional constraints is a preliminary analysis of the DFG and module library. A synthesis system can preselect modules and determine bounds on their numbers, prior to running the ILP.

Another good source of constraining information is from a previous solution attempt, using a simpler version of the ILP or an altogether different scheduling/allocation heuristic. This brings us to the second major way of speeding up an ILP: not to run it as one monolithic process on a wide-open problem, but rather to progress in steps, saving the full ILP for the final assault on the optimum solution. It is possible to use the ILP formulation itself to find intermediate solutions very quickly. One simply takes the previous best solution, reduces either the time or total area, and submits the newly refined constraints to the ILP solver without any objective function. If any *feasible* solution exists, one will be found rapidly without requiring the ILP solver to pursue even better (or possibly equivalent) solutions in that application. In the end, an ILP may only be asked to find the optimum when it is fine-tuning an already good solution.

7. CHAINING

Chaining refers to performing several data-dependent operations sequentially in a control step. The number of operations that can be serialized in a control step depends on both the propagation delays of the functional units used for executing those operations and the duration of a step.

In our model, the potential operations that can be chained are selected by searching the module library for sequences of single-step functional units capable of performing their respective operations, all within a single control step. During the optimization process, the ILP chains operations if there is any advantage in doing so. Constraint 7 (Sec. 6.1) still holds. It ensures separate instances of units for operations which are chained in a clock cycle. This constraint also restricts the total number of single-step functional units available for any clock cycle and, hence, their area contribution. Therefore, it indirectly controls the use of chaining as well.

If we consider a pair of data-dependent operations i and j to be successfully chained, then we want:

$$w_{iks} + w_{jhs} > 1, \quad \text{only if}, \quad w_{iks}D_{ik} + w_{jhs}D_{jh} \leq 1.$$

(Note that the step number is s for both op_i and

op_j.) D_{ik} and D_{jh} denote the time taken by modules from classes k and h to execute op_i and op_j, respectively. These inequalities state that two operations can be scheduled in a step only if their combined execution time is less than or equal to the duration of one step. The precedence relation (Constraint 9a, Sec. 6.1) for any two operations belonging to either Class A or Class B should be interpreted as follows:

$\forall i, j$ where $E[i, j] = 1$ and $i, j \in A \cup B$:

$$\sum_{h>0} \sum_{t} tw_{jht} - \sum_{k>0} \sum_{s} sw_{iks} \geq \begin{cases} 0 & \text{if } D_{jh} + D_{ik} \leq 1 \\ 1 & \text{if } D_{jh} + D_{ik} > 1 \end{cases}$$

This means that if there are module classes which can do op_i and op_j within a clock cycle, then they can be chained and their step difference $(t - s)$, will be zero. Otherwise, the constraint remains as before, *i.e.*, operations i and j get scheduled in different steps. Constraint 9a can be expressed in a single inequality as:

(9*a*)

$$\sum_{h>0} \sum_{t} tw_{jht} - \sum_{k>0} \sum_{s} sw_{iks} \geq \sum_{h>0} \sum_{t} D_{jh} w_{jht} + \sum_{k>0} \sum_{s} D_{ik} w_{iks} - 1$$

Notice carefully the righthand side. If $D_{jh} + D_{ik} \leq 1$ and op_i and op_j use modules from classes k and h in the same step (when $t = s$), then the righthand side becomes ≤ 0, and chaining can be activated, if necessary. On the other hand, if $D_{jh} + D_{ik} > 1$, the righthand side becomes > 0, which forces a difference of at least one step between the execution of operations i and j.

Our formulation can be extended to cover three or more operations of a data flow graph that can potentially be chained together. This is done (in the case of 3 operations) by including constraints for the two consecutive pairs of operations, as well as constraints for all three together.

8. INCLUDING REGISTER ALLOCATION

Section 5.4 introduced the notion of counting the required number of registers in the design. This section explains how it is actually done. Our model can dynamically assess or control the number of variables that are live after each control step. The maximum of these numbers equals the number of data storage registers required in the design. Note that the formulation presented here does not actually assign variables to specific registers (a process called register binding), but it does accurately consider their number.

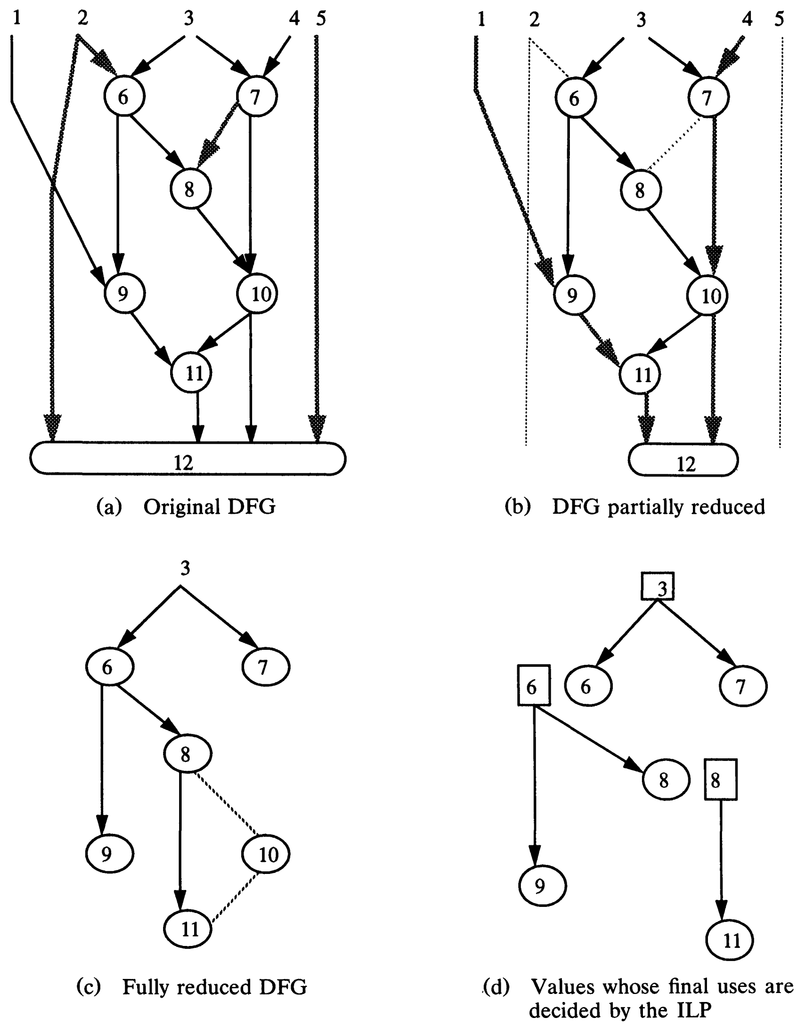
A portion of this problem is solved by Gebotys [8]. We begin in similar fashion but supply the *additional* analysis to permit register allocation to be fully integrated into our model and solved in one application of the extended ILP.

The entire preparatory process can be seen in Fig. 2. Figure 2(a) shows the original DFG. Five separate values, requiring five different registers, are shown from five earlier computations or parameters (1 through 5). Each interior operation (6 through 11) produces a single value which can be stored in a single register, even if it is required at more than one place later on. We use $v(i)$ to represent each distinct value in terms of i, the operation or source that produces it. Values that have more than one ultimate destination may require special attention; we designate such values as *multi-use values* and call the others *single-use values*. Four output values are also shown. Since their destinations need not be distinguished for our purposes, they are shown leading to a generic "end node", numbered 12.

Multi-use values complicate the problem of register allocation. The value in the register must be retained until its *final* use. If the final use is not implied by the DFG, it must be determined by the scheduler. Dynamic determination of the final use is an important contribution of our solution.

The following rules simplify the DFG *for the purposes of register allocation* prior to adding constraints to the ILP. Recall that $E[i, j] = 1$ iff the DFG contains an edge from node i to node j. Following the usual convention, $E^*[i, j] = 1$ iff there is a path from node i to node j. Let I represent the index set of prior operations or parameters that supply input operands, and let e identify the single "end node". (The original DD constraints remain for scheduling purposes, of course):

1. Any input operand, which must be retained unchanged throughout the design, accounts for one register; all of its outgoing edges may be removed, since their destinations can all obtain the value from the dedicated register. This can be stated formally as: if $E[i, e] = 1$ and $i \in I$, then delete every edge corresponding to $E[i, j] = 1$.
2. If a multi-use value has one destination that is necessarily a predecessor of one of its other destinations, the edge to the predecessor node can be removed from the DFG. Thus, if



(a) Original DFG (b) DFG partially reduced

(c) Fully reduced DFG (d) Values whose final uses are decided by the ILP

FIGURE 2 DFG Reduction for Register Allocation

$E[i, j_1] = 1$, $E[i, j_2] = 1$, and $E^*[j_1, j_2] = 1$, then delete edge $E[i, j_1]$. Note that if rule 2 removes all but one edge originating at node i, then $v(i)$ becomes a single-use value.

3. Any directed path containing only single-use values can be replaced by a single edge connecting the first node and the last node of the path. That is, if $E^*[i, j] = 1$, and for every node $i_k \neq j$ in the path, $v(i_k)$ is a single-use value, then every edge in the path can be removed, provided a new edge $E[i, j]$ is added to replace them.

These rules, except the last, essentially follow the analysis of Gebotys [8]. Their purpose is to eliminate edges of the DFG that definitely do not imply an additional live value; all eliminated edges represent values that must be retained for later use anyway. The final rule acknowledges that a register is required for each value entering and each value leaving an operation. Even if different registers are used for the input and output values, their numbers will be the same–which is all that matters here.

Figure 2(b) shows the result of applying the first two rules. The values $v(2)$ and $v(5)$ are global and consume registers throught. The value produced by operation 7 will clearly have its final use by operation 10, allowing edge (7, 8) to be removed and $v(7)$ to become a single-use value.

Figure 2(c) shows the result of applying the final rule. Two paths containing only single-use values, {(1, 9), (9, 11), (11, 12)} and {(4, 7), (7, 10), (10, 12)}, extend throughout the DFG. After replacement by single edges, these edges can be entirely eliminated by rule 1. Of course, two more registers will be consumed. The interior path, {(8, 10), (10, 11)}, can also be replaced by a single edge, which remains in the DFG to depict the equivalent register commitment.

Figure 2(d) shows the three register-related values that must be considered by the ILP. Note that the final schedule could require either one or two registers, in addition to the four already committed. (Consider the sequential schedules: $\{7,6,9,8,10,11\}$ and $\{6,7,8,9,10,11\}$, which require one and two extra registers, respectively.)

At this point, additional variables and constraints must be added to the ILP. Define variables:

r (integer); the number of registers in the design

$g_{is} = 0,1$; $g_{is} = 1$ iff $y_i = s$, *i. e.*, $v(i)$ originates on step s

$f_{is} = 0,1$; $f_{is} = 1$ iff $v(i)$ has its *final* use on step s

The following constraints are added to the version of the ILP given in Sec. 4. Their effects are described below. Minor changes will adapt these constraints for the accelerated version in Sec. 6.1.

$\forall$ remaining $v(i)$:

$$(12) \qquad \sum_s g_{is} = 1$$

$$(13) \qquad \sum_s g_{is} * s = y_i$$

$$(14) \qquad \sum_s f_{is} = 1$$

$\forall$ remaining $v(i)$, and j such that $E[i,j] = 1$:

$$(15) \qquad y_j + \sum_m x_{jm} D_{jm} - 1 \le \sum_s f_{is} * s$$

$\forall$ steps t:

$$(16) \qquad \sum_i \sum_{s \le t} (g_{is} - f_{is}) \le r$$

Constraints 12 and 13 ensure that g_{is} becomes 1 only on the unique step, s, where the value is generated. Constraint 14 ensures that only one final use for this value will be counted (for one value of s). Input source values have $g_{i0} = 1$ to suggest a value generated before step 1, and require neither constraints (12) or (13). Output values have $f_{ie} = 1$ to suggest a final use on step e, the end, past the last step considered by the ILP; constraints (14) and (15) are not required in this case.

Constraint 15 bounds the final use time by the completion times of each of its separate uses. Therefore, the unqiue value of s for which $f_{is} = 1$ will be at least as large as the time of its *final* use, the one that corresponds to the release of its register.

At each control step, the number of previously consumed values is subtracted from the number of previously generated values. This gives the number of live values at that step. These numbers are lower bounds on the number of registers in the design. Constraint 16 provides these bounds for r.

The number of registers can be limited by a constraint at the outset. Alternatively (or additionally), r can be included in the objective function to consider register area and functional unit area simultaneously. The objective function then becomes:

$$\min \left[W_{time} * T + W_{area} * \left(\sum_m u_m A_m + r * A_{reg} \right) \right]$$

where A_{reg} denotes the area of a single register.

9. RESULTS

We have used our system (SYMPHONY), to synthesize some of the standard benchmark examples available in the literature. In this section, we present the results obtained. As expected, our integrated approach often achieves better results than its heuristic counterparts. It also demonstrates the advantage of integrating the scheduling, allocation, and module binding tasks.

The first example is the standard high-level synthesis benchmark: elliptical wave filter [10]. Table I shows the results for 17, 18, and 19 step schedules by using the same module library as used by HAL [10]. It shows the CPU times consumed by our original model (Sec. 4) and the modified model (as presented in Sec. 6) for determining the optimal solutions. Although the (well-known) optimal solu-

TABLE I
Performance of Our Original and Modified Models

C-steps	Model	Adders	Muls	CPUtime*
17	Original	3	3	159.62 mins.
	Modified	3	3	2.2 sec.
18	Original	2	2	183.7 mins.
	Modified	2	2	4.69 mins.
19	Original	2	2	731.4 mins.
	Modified	2	2	5.97 mins.

*On a Sun SPARCstation 2.

TABLE II
Module Library

Component	Types	Area	Delay (C-steps)
Adder	$+_1$	50	1
	$+_2$	30	2
Multiplier	$*_2$	400	2
	$*_3$	250	3

TABLE III
Elliptical Wave Filter

W_{time}	W_{area}	Modules	C-steps	Area
0.9	0.1	$+_1, +_1, +_1, *_2, *_2, *_2$	17	1350
0.8	0.2	$+_1, +_1, *_2, *_2$	18	900
0.6	0.2	$+_1, +_1, + *_2, *_2$	18	900
0.7	0.3	$+_1, +_1, *_2, *_3$	19	750
0.6	0.4	$+_1, +_1, *_2$	21	500
0.4	0.3	$+_1, +_1, *_2$	21	500

TABLE V
Bandpass Filter

Csteps	Model	Modules	Area	CPU time
8	SYMPHONY	$+*, +-, +-, *$	**675**	4.78secs
	ADPS	$+-, *, *, +, -$	685	9secs
9	SYMPHONY	$+-, *, *, +$	**625**	16.87secs
	ADPS	$+-, +-, *, *$	650	64secs
10	SYMPHONY	$+-, *, *, +$	**625**	32.34secs
	ADPS	$*+, +-, *, +$	650	132secs
11	SYMPHONY	$+*, +-, *$	**600**	41.9secs
	ADPS	$+-*, +-, *$	630	197secs

tions are obtained for either case, there is a tremendous improvement in the running times for the modified model.

We also synthesize the elliptical wave filter example by assigning different weights to the area and time costs of the piecewise linear objective function. The module library used is presented in Table II. The results depict the sensitivity of our model to varying weights. By changing the weights randomly, SYMPHONY obtains optimum results corresponding to 17 through 21 step schedules (Table III). These results also show the ability of our model to use different physical implementations for the same operation type. For the 19-step schedule, it utilizes a 2-step and a 3-step mutliplier, rather than using two 2-step multipliers.

The second example is the differential equation benchmark that originally appeared in [10]. This example is synthesized under various assumptions: without chaining and pipeline units in the library; only using pipelined units (pipelined multipliers having a latency of 1); premitting only chaining; and permitting both chaining and the usage of pipelined units. The results are shown in Table IV. Although HAL [10] and ALPS [11] produce the same results for the first three cases depicted in Table IV, *neither of them has shown results when both chaining and usage of pipelined units are allowed.*

Next, we use the bandpass filter example from [5]. Table V compares our results with ADPS [5]. ADPS uses force-directed scheduling [10] together with an ILP to allocate functional units to the predetermined schedule and to bind the operations to individual instances of those allocated units. For every case, SYMPHONY yields better results than ADPS. The times shown are for determining the feasible solutions. Instead of finding the optimal solutions, the time and area are suitably constrained to desired values, and the ILP is asked to determine feasible solutions within those specified limits. This approach eliminates many equivalent results and speeds up the solution process.

Finally, we use the fifth order elliptical wave filter benchmark once again to compare the performance of SYMPHONY with other systems. Table VI compares our results with both HAL [10] and CHASSIS [3]. The module library used is the same as that used by CHASSIS, in order to make a fair comparison. In the table, $+_i$ represents an adder that takes i control steps for executing an addition; $*_i$ is to be interpreted similarly for multiplication. Although HAL assumes a single physical implementation for any given functional unit, we include their figures just to illustrate the improvement possible by using different physical implementations of a functional unit (17, 19, and 20 step cases). CHASSIS is an integrated heuristic approach that can incorporate

TABLE IV
Differential Equation Example

	Neither		Pipelined Units		Chaining		Both	
C-steps	ALUs	Muls	ALUs	Muls	ALUs	Muls	ALUs	Muls
5	not applicable		not applicable		3	3	2	3
6	2	3	1	2	2	3	1	2
7	2	2	1	2	2	2	1	2
8	1	2	1	1	1	2	1	1

TABLE VI
Fifth Order Elliptical Wave Filter Comparison

Csteps	Model	Modules used	Area
17	SYMPHONY	$+_1, +_1, +_1, *_1$	**21500**
	CHASSIS	n/a	n/a
	HAL	$+_1, +_1, +_1, *_2, *_2, *_2$	28500
18	SYMPHONY	$+_1, +_1, *_2, *_2$	**19000**
	CHASSIS	$+_1, +_1, +_3, +_3, *_1$	20600
	HAL	$+_1, +_1, *_2, *_2$	19000
19	SYMPHONY	$+_1, +_1, *_2, *_3$	**14000**
	CHASSIS	$+_1, +_1, *_2, *_3, *_3$	17000
	HAL	$+_1, +_1, *_2, *_2$	19000
20	SYMPHONY	$+_1, +_1, *_2, *_3$	**14000**
	CHASSIS	$+_1, +_1, +_2, *_2, *_3$	14500
	HAL	$+_1, +_1, *_2, *_2$	19000
21	SYMPHONY	$+_1, +_1, *_2$	**11000**
	CHASSIS	$+_1, +_1, +_2, +_2, *_3, *_3, *_3, *_3$	16000
	HAL	$+_1, +_1, *_2$	11000

modules of fairly general type in a design. In all the cases, SYMPHONY produces results that are better than CHASSIS-specially in the 19 and 21 step schedules.

SYMPHONY is implemented in the UNIX/C environment on a Sun SPARCstation 2. The ILP model is solved using the commercial linear programming package LINDO [12].

10. CONCLUSION

In this paper, we have presented an integrated and global solution to the problems of scheduling, module and register allocation, and module binding. This method handles all the interactions among these subproblems. The solution permits the use of complex functional units and allows operations to be implemented by a variety of functional units, possibly requiring different execution times. We have also shown how the model can be extended to effectively use pipelined units or chain operations whenever there is an opportunity for doing so. In a straightforward way, our model can also find optimal schedules and module allocations for *multiple-block* designs, not block-by-block or just along critical paths, but for all the blocks of a design *simultaneously*; the details of multi-block synthesis will be available in a separate manuscript. It has also been extended to totally incorporate register allocation at the same time. We believe SYMPHONY is the most comprehensive and integrated model developed so far for the tasks of scheduling, module and register allocation, and operation binding. Moreover, the capability of using a complex and generalized module library is unique to our system.

References

[1] C-T. Hwang, J-H. Lee, and Y-C. Hsu. "A Formal Approach to the Scheduling Problem in High Level Synthesis". *IEEE Transactions on Computer-Aided Design*, 10(4): 464–475, April 1991.

[2] R.J. Cloutier and D.E. Thomas. "The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm". In *Proc. of the 27th Design Automation Conference*, pages 71–76. ACM/IEEE, 1990.

[3] L. Ramachandran and D.D. Gajski. "An Algorithm for Component Selection in Performance Optimizied Scheduling". In *Proc. of International Conference on Computer-Aided Design*, pages 92–95. IEEE, 1991.

[4] M. Balakrishnan and P. Marwedel. "Integrated Scheduling and Binding: A Synthesis Approach for Design Space Exploration". In *Proc. of the 26th Design Automation Conference*, pages 68–74. ACM/IEEE, 1989.

[5] C.A. Papachristou and H. Konuk. "A Linear Program Driven Scheduling and Allocation Method Followed by an Interconnect Optimization Algorithm". In *Proc. of the 27th Design Automation Conference*, pages 77–83. ACM/IEEE, 1990.

[6] S. Devadas and A. R. Newton. "Algorithms for Hardware Allocation in Data Path Synthesis". *IEEE Transactions on Computer-Aided Design*, 8(7): 768–781, July 1989.

[7] M. Nourani and C.A. Papachristou. "Move Frame Scheduling and Mixed Scheduling-Allocation for the Automated Synthesis of Digital Systems". In *Proc. of the 29th Design Automation Conference*, pages 99–105. ACM/IEEE, 1992.

[8] C.H. Gebotys and M.I. Elmasry. "Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis". In *Proc. of the 28th Design Automation Conference*, pages 2–7. ACM/IEEE, 1991.

[9] C. H. Gebotys. "Optimal Scheduling and Allocation of Embedded VLSI Chips". In *Proc. of the 29th Design Automation Conference*, pages 116–119. ACM/IEEE, 1992.

[10] Pierre G. Paulin. *High-level Synthesis of Digital Circuits Using Global Scheduling and Binding Algorithms*. Ph.D thesis, Carleton University, Ottawa, January 1988.

[11] J-H. Lee, Y-C. Hsu, and Y-L. Lin. "A New Integer Linear Programming Formulation for the Scheduling Problem in Data Path Synthesis". In *Proc. of International Conference on Computer Aided Design*, pages 20–23. IEEE, 1989.

[12] Linus Schrage. *LINDO: User's Manual. The Scientific Press, San Francisco, CA*, 1991.

[13] J. Vanhoof et al. *High-Level Synthesis for Real-Time Digital Signal Processing. Kluwer Academic Publishers, Boston, Ma.* 1993.

Biographies

THOMAS C. WILSON is an Associate Professor in the Department of Computing & Information Science at the University of Guelph, Guelph, Ontario. He holds a BA degree from the University of Iowa, M.Sc from the University of Chicago, and Ph.D. from the University of Waterloo. His research interests lie in the areas of distributed systems and VLSI-CAD, particularly high-level synthesis and retargetable code generation.

NILANJAN MUKHERJEE received his B.Tech degee with honours in Electronics and Electrical Communication Engineering from Indian Institute of Technology, Kharagpur, India, in 1989, and the M.S. degree in Computer Science from University of Guelph, Canada, in 1992. Presently, he is pursuing a Ph.D. in Electrical Engineering at McGill University, Montreal, Canada. His research interests include high-level synthesis, hardware-software codesign, testing, and synthesis of testable designs.

MAHESH K. GARG holds a B.Sc degree in Chemical Engineering from the University of Roorkee, India, M.A.Sc in Chemical Engineering from the University of Waterloo, and M.Sc in Computer Science from the University of Guelph, Guelph, Ontario. Presently, he is working as a System Administrator in the CASE group at the IBM Laboratory in Don Mills, Ontario.

DILIP K. BANERJI is a Professor in the Department of Computing & Information Science at the University of Guelph, Guelph, Ontario. He holds a B.Tech degree in Electrical Engineering from the Indian Institute of Technology, Kanpur, M.Sc. in Electrical Engineering from the University of Ottawa, Canada, and Ph.D. in Computer Science from the University of Waterloo. His primary research interest is VLSI design automation, particularly high-level synthesis.