An Efficient ILP-Based Scheduling Algorithm for Control-Dominated VHDL Descriptions

Michael Münch[1] Norbert Wehn[1,2] Manfred Glesner[1]

[1] Darmstadt University of Technology
Institute of Microelectronic Systems
Karlstr. 15
D-64283 Darmstadt, Germany

[2] Siemens AG
Semiconductor Division
Balanstr. 73
D-81549 München, Germany

Abstract

In this paper, we present for the first time a mathematical framework for solving a special instance of the scheduling problem in control-flow dominated behavioral VHDL descriptions given that the timing of I/O signals has been completely or partially specified. It is based on a code-transformational approach which fully preserves the VHDL semantics. The scheduling problem is mapped onto an integer linear program (ILP) which can be constrained to be solvable in polynomial time, but still permits optimizing the statement sequence across basic block boundaries.

1. Introduction

As opposed to logic synthesis, architectural synthesis can considerably optimize a design by exploiting the potential given by the incomplete timing of the input description. Scheduling, which fixes the timing of operations in a design, is thus one of the key problems in architectural synthesis [11].

Scheduling for *data-flow dominated* designs is relatively well understood and covered by a large number of algorithms. These mostly deal with optimizing resource utilization, latency or pipelining and operate on acyclic graphs [11]. They are specialized to deal with a large share of arithmetic operations and moderate control-flow characteristic for digital signal processing (DSP) applications or the like [17].

Control-flow dominated designs, however, are characterized by a large share of nested loops and conditionals with only few arithmetic operations. The number of paths in a modestly sized design of only a few hundred lines of code can be in the order of some millions. Common examples are controllers, parsing engines for packetized data streams (e.g. in ATM and video compression/decompression applications) and protocol processors.

Typically, these systems are also characterized by complex *timing constraints*, i.e. the number of clock cycles to elapse between a pair of operations is either upper or lower bounded or statically fixed and must not be changed during scheduling. This is particularly true for the interface timing which must adhere to a fixed protocol if the component is to be used along with components re-used from earlier projects or highly optimized full-custom building blocks.

Control-flow dominated designs are represented using *cyclic* control/data flow graphs to account for loops. Similarly, timing constraints are not limited to operations within a single basic block (i.e. a code sequence of maximum length in which control flow does not branch), but can span an arbitrary number of conditionals and loops. This type of timing constraint is termed *path-activated* in [21]. Control flow and timing constraints thus interact in a complex manner. YEN and WOLF show in [21] that even without taking allocation costs into account, the problem of finding a feasible schedule under path-activated timing constraints is $\mathcal{NP}$-complete.

This paper presents an approach to scheduling control-flow dominated VHDL descriptions by behavioral code transformation. It allows partial or full specification of the I/O timing and of path-activated timing constraints. The scheduling problem is formulated in terms of an integer linear program [10], the problem size of which is independent of the number of paths. To allow computationally efficient evaluation of design alternatives within an interactive design environment, we impose restrictions on the *partial* order of statements, thereby making the ILP problem solvable in polynomial time. However, the model still permits optimizing the statement sequences across basic blocks to meet performance and/or resource constraints. Scheduling can be done subject to the number of states to minimize the length of each path or to the number of registers.

2. Related Research

Apart from data-flow oriented schedulers which have been extended to efficiently handle conditional branches (such as [18]), only few approaches which explicitly deal with control-flow dominated descriptions exist. Path-based scheduling [3] aims at scheduling each path as fast as possible. Although there is a mechanism for specifying overall timing constraints, specification of *exact* timing relationships between pairs of I/O operations is not supported. Path-based scheduling considers *all* paths explicitly and does not permit optimizing the sequence of operations. Also, the start of each path is defined to be at the beginning of a loop, thus possibly yielding non-optimum results after scheduling. In [12] this is relaxed by allowing a path to start at arbitrary points, however, all paths still need to be considered explicitly.

Tree-based scheduling [7] does optimize the statement sequence, but by considering all paths its complexity is also exponential in the number of paths. It does not support specification of the I/O timing.

Relative scheduling [9] does not consider path-activated constraints and breaks up cyclic control-flow. In [20] the design and its timing constraints are initially represented separately using automaton models which are subsequently merged into a product automaton from which a feasible schedule is derived. For complex constraints, the product automaton grows exponentially in size which makes the approach impractical for large designs.

Other ILP-based approaches include [15, 6, 4]. All of them, though, are targeted towards data-flow dominated designs with only moderate control flow. Moreover, the associated ILP problems are all exponential in complexity and hence computationally inefficient for large designs. The approach presented in [13] permits global optimization of the execution order of operations. It relies on representing all possible schedules in a compressed OBDD representation. The OBDD essentially represents an ILP formulation and hence suffers from some of the drawbacks conventional ILP techniques as mentioned above suffer from.

Of particular interest within the context of this work are [8] and [21]. In [8], a methodology for HDL-based specification is proposed which has been adopted in the Synopsys Behavioral Compiler. There are three different scheduling modes, depending on how tight the timing is to be specified: in *cycle-fixed mode* the exact temporal relationship between I/O-operations is specified in terms of clock cycles in the behavioral description. *Superstate-fixed mode* permits "stretching", i.e. clock cycle insertion at certain points. *Free-floating mode* permits specification of overall timing constraints but leaves the exact position of clock cycle transitions open to the scheduler. Similar approaches have been presented earlier in [16, 19], however, unlike the one presented in this paper, these did not globally optimize the order of statements across basic blocks.

The approach presented in [21] is based on so-called Behavior Finite State Machines (BFSMs) which permit more powerful specification of timing constraints than VHDL does, if it has not been semantically enhanced.

Our approach can be classified in-between the latter two. It relies on a similar methodology as the one suggested in [8], but allows the combination of all three types of timing specifications within a single description. The BFSM model supports more complex timing constraints, however, it cannot directly be incorporated into a VHDL-based design and simulation environment. Moreover, its scheduling algorithm is based on a heuristic which might not always find a solution even though a feasible schedule exists. Due to its exact nature, the ILP formulation presented here will always find an *optimum* solution if it exists, assuming a partial order on certain operations.

3. Modeling and Problem Definition

3.1. Timing Specification

We will consider single multiple-wait VHDL processes in which the timing behavior is specified in relation to the same edge of a single clock signal `clk` using the following wait-statement: `wait until clk = '1'`. The latter will be in this paper referred to simply as "wait-statement". For instance, two consecutive writes of an output signal `ack` between which exactly two clock cycles must elapse (as in a tight communication protocol), can be specified as shown in Program 1 (a).

Program 1 VHDL modeling examples

```
ack <= '1';
wait until clk = '1';
-- some_action
wait until clk = '1';
ack <= '0';
```
(a)

```
if (req = '1') then
--  some_action
  wait_source (2, 4);
  ack <= '1';
end if;
```
(b)

Interface timing can be specified incompletely using so-called *wait-sources*. A wait-source is an abstract statement which is replaced during scheduling by as many wait-statements as required to fulfill all frequency and resource constraints. It can thus be identified with a *superstate* which is during scheduling broken up into "ordinary" states. The range of into how many wait-statements a wait-source may be converted can be specified by passing an upper and/or lower bound to the wait-source. This is useful in specifying more flexible communication protocols. Program 1 (b) shows a code fragment from a module which after receiving a request `req` must carry out some action and issue an acknowledge `ack` no sooner than two clock cycles after the request and not later than four clock cycles.

The waits generated by the wait-source can be arbitrarily distributed on the path between the two I/O operations; in particular, they can be useful in satisfying resource and/or frequency constraints within the block that carries out `some_action`.

3.2. Graph Model

The VHDL specification is mapped onto two graph structures: a *weighted control/data flow graph (CDFG)* and a *flow graph*. The CDFG(V, E, w) is a weighted, directed, cyclic graph (V, E, w) with nodes $V = V_{io} \cup V_l \cup V_{vir}$ and edges $E = E_c \cup E_{vir} \cup E_{du}$ along with a weight function $w : E_c \cup E_{vir} \rightarrow \mathbb{N}$. Except for wait-statements, each statement in the VHDL description including wait-sources is represented by a node in $V_{io} \cup V_l$. Nodes in V_{io} represent I/O statements (i.e. signal read and write operations), all others statements are represented by nodes in V_l. V_{vir} is the set of *virtual nodes*. These nodes do not have a one-to-one correspondence to VHDL statements but are required to model control flow.

The edge set $E_c \cup E_{vir}$ models the control flow. An edge $(v_i, v_j) \in E_c$ is generated if the statement represented by v_i is the direct predecessor of the statement associated with v_j in the VHDL description. Similar to V_{vir}, E_{vir} contains *virtual edges* required to model control flow, e.g. of loops.

Edges in E_{du} represent data flow. There is an edge $(u, v) \in E_{du}$ if and only if the statement associated with u writes a variable so that the new value of the variable can be read by node v (i.e. a write operation at node u affects a read operation at node v). The edges in E_{du} are called *definition-use chains* and can be constructed by global data flow analysis techniques as described in [1].

Wait-statements are represented by edge weights $w(e), e \in E_c \cup E_{vir}$ rather than by nodes. Consider the code sequence in Program 2(a). Suppose the assignment to d is mapped onto node u and the `if`-statement is mapped onto node v. Then there is an edge $e = (u, v)$ in E_c with $w(e) = 1$. In general, a sequence of n wait-statements in the VHDL source is replaced by an edge e in the CDFG with edge weight $w(e) = n$. All other edges are assigned an edge weight of zero.

During data flow analysis, a flow graph $FG(B, E_B)$ is generated. Its nodes B are basic blocks, its edges E_B define a precedence relationship in the global control flow: there is an edge $e_b = (b_i, b_j) \in E_B$ if and only if control flow can transfer directly from basic block b_i to basic block b_j.

3.3. Definition of the Scheduling Problem

Scheduling determines the start time of an operation on clock cycle level or, in other words, assigns each operation a state in which its execution is initiated. In the model proposed above, each wait-statement can be identified with a state in the corresponding state machine. Consequently, the set of statements on a path between two wait-statements can in turn be associated with a state. Scheduling a statement s into a different state requires a code transformation to bind s to a different wait-statement. Consider the code sequence in Program 2 (a). Let us call the states the two wait-statements can be identified with q_i and q_j, respectively. Note that due to the simulation semantics the values of `out1` and `out2` will only be visible to the environment two clock cycles after the multiplications have been initiated. This description has a resource requirement of two multipliers and two adders/subtracters, since both multiplications and additions/subtractions are initiated in state q_i and state q_j, respectively. Assuming that each multiplication requires two time units and each addition requires one time unit, the maximum delay in states q_i and q_j is four time units.

Program 2 A simple VHDL code transformation

```
a := b * c;
d := a * f;
-- q_i
wait until clk = '1';
if (cond) then
  x := x + a;
  y := y + d;
else
  x := x - a;
  y := y - d;
end if;
out1 <= x;
out2 <= y;
-- q_j
wait until clk = '1';
```

(a)

```
a := b * c;
if (cond) then
  x := x + a;
  -- q_i_1
  wait until clk = '1';
  d := a * f; y := y + d;
else
  x := x - a;
  -- q_i_2
  wait until clk = '1';
  d := a * f; y := y - d;
end if;
out1 <= x; out2 <= y;
-- q_j
wait until clk = '1';
```

(b)

Program 2(b) shows the same chunk of code after code transformation. It is easy to see that both code sequences are semantically equivalent. Particularly the *external* behavior of the circuit has not been touched even though an additional state was created. The delay on each of the two paths from the start of the code sequence until the output signals `out1` and `out2` change is still two clock cycles. However, the transformed code has a resource requirement of only one adder/subtracter and multiplier each. Furthermore, the maximum delay has been reduced to three time units.

To mathematically model scheduling by code transformation, each node $v \in V$ is assigned a *schedule-variable* $s(v)$ whose value reflects the *relative change in state* of node v in the new schedule with regard to the initial schedule. Based on the schedule-variables, we recompute the temporal relationship of any pair of nodes (and hence their associated VHDL statements) after scheduling. The "distance" $d_q(v_i, v_j)$ of two operations v_i, v_j in terms of controller states (or equivalently, the delay in clock cycles) is implicitly given in the original specification and can be derived from the CDFG's edge weights. Based on $d_q(v_i, v_j)$, the distance of the two operations $d'_q(v_1, v_n)$ *after* scheduling can be computed based on the schedule-variables as follows:

$$d'_q(v_i, v_j) = s(v_j) - s(v_i) + d_q(v_i, v_j). \tag{1}$$

We can enforce certain properties of the scheduled code by constraining $d'_q(v_i, v_j)$ for selected pairs of nodes. For a data dependence $(v_i, v_j) \in E_{du}$, for example, this distance must obviously be greater than or equal to zero: $d'_q(v_i, v_j) \geq 0$. In other words, v_j must be executed in the same state (through chaining) or in a later state than v_i.

Note that this model does not yet take wait-sources into account which also contribute to d_q. The number of wait-statements generated by a wait-source v will be its schedule value $s(v)$. We will deal with wait-sources separately in Section 4.2 and modify the model accordingly.

Obviously, a code transformation can be expressed in terms of an assignment to schedule-variables. Such an assignment is termed *feasible* if it does not alter the semantic properties of the original description. The scheduling problem dealt with in this paper can hence be defined as follows: *Given a weighted control/data flow graph CDFG*(V, E, w)*, a maximum clock cycle length* $\Delta_{\max}$, n_τ *instances of each functional unit of type* τ *and a set of schedule variables* $\mathcal{S} = \{s(v) \mid v \in V\}$*, a schedule of CDFG is defined by a feasible assignment to* $\mathcal{S}$.

In the following section we will define more rigorously what exactly makes an assignment feasible and how this can be expressed in terms of linear inequalities on the set $\mathcal{S}$.

4. Feasible Assignments

4.1. Basic ILP Model

We require that the overall *structure of the flow graph be retained*, i.e. we will only allow non-control flow statements such as assignments to be moved in and out of basic blocks.

For each basic block $b \in B$ in a flow graph *FG*(B, E_B) we define an entry node $\text{entry}(b)$ and an exit node $\text{exit}(b)$ which uniquely mark the entry and exit points of basic block b. These nodes may be defined so that for two basic blocks b_i and b_j $\text{exit}(b_i) = \text{entry}(b_j)$ if and only if $(b_i, b_j) \in E_B$. Then, it can be shown that the following two (in)equalities guarantee that the assignment to $\mathcal{S}$ will retain the structure of the flow graph:

$$\forall (u,v) \in E_{vir} : s(v) - s(u) = 0 \tag{2}$$

$$\forall b \in B : s(\text{exit}(b)) - s(\text{entry}(b)) + d_q(\text{entry}(b), \text{exit}(b)) \geq 0. \tag{3}$$

If there is a *data dependence* from a node u to a node v, i.e. $(u,v) \in E_{du}$, u may, to preserve the code semantics, never be scheduled after v. This is enforced by the following constraint:

$$\forall (u,v) \in E_{du} \;:\; s(v) - s(u) + \min_{p \in \mathcal{P}(u,v)} \{d_q^p(u,v)\} \geq 0, \tag{4}$$

where $\mathcal{P}(u,v)$ is the set of all paths in $E_c \cup E_{vir}$ from u to v and $d_q^p(u,v)$ the delay on a path p.

It can be shown that Eq. (4) guarantees that data dependencies as defined by the set E_{du} will be preserved after scheduling. However, while moving nodes into other basic blocks, we must also ensure that no other data dependencies are *violated*. In [14] a set of basic code transformations are defined based on which any relocation of a statement may be described by composition. In the following we will, for the sake of brevity, concentrate on *boosting-up* which moves a statement from an `if`- or `case`-branch into the basic block preceding the branching statement (*preamble* for short in the following).

To determine whether boosting a node would violate data dependencies in other branches, we exploit data flow information gathered at the entry and exit points of basic blocks during global data flow analysis.

Let *Live*(b) be the set of variables read after the end of basic block b (such a variable is called *live* on exit of b). Then, a node v_s within a branch block b_s may be moved into its preamble b_f if and only if the following condition holds:

$$D(v_s) \cap \mathit{Live}(b_f) = \emptyset,$$

where $D(v_s)$ is the set of variables defined at node v_s. This constraint prevents that a node v_s defining for instance a variable x which is also read before being re-defined on a path through a neighboring branch b_t (i.e. $x \in \mathit{Live}(b_f)$) is moved into the preamble, since this would invalidate the value of x read on the path through b_t.

Similar conditions can be derived for other code motions. Based on these conditions we can, using a breadth-first search technique, in $\mathcal{O}(|B| + |E_B|)$ time construct for each node v a "backward barrier" $\mathcal{B}(v)$ and a "forward barrier" $\mathcal{F}(v)$, i.e. sets of basic blocks into which v may not be moved. Moving v in any block on the path from its current block to a barrier block, however, will be safe. Hence, for the backward barrier sets the following constraints are required (forward barriers are treated accordingly):

$$\forall b \in \mathcal{B}(v) : s(v) - s(\text{exit}(b)) + \min_{p \in \mathcal{P}(\text{exit}(b), v)} \{d_q^p(\text{exit}(b), v)\} \geq 0. \tag{5}$$

The *interface timing* of a circuit, i.e. the temporal relationship between pairs of signal access operations, is determined by the number of controller states that are traversed between the two signal access operations. To guarantee that the external timing before and after scheduling coincide on clock cycle level, the number of controller states on each path between a pair of I/O operations must not be changed during scheduling, i.e.

$$\forall (v_i, v_j) \in 2^{V_{io}} \forall p \in \mathcal{P}(v_i, v_j) : d_q^p(v_i, v_j) = d_q^{p'}(v_i, v_j).$$

Substituting Eq. (1) into the latter equation returns $s(v_i) = s(v_j)$. The assignment to the schedule-variables of all I/O operations must hence be the same:

$$\forall v_i \in V_{io}, i = 1, 2, \ldots, n-1 : s(v_i) = s(v_{i+1}), \tag{6}$$

if the nodes in V_{io} are labeled v_1 through v_n.

Suppose a *resource constraint* of n_τ instances of a functional unit of type τ has been defined. Then, any path in the *control flow* with more than n_τ operations of type τ and a path weight of zero must be partitioned into at least two states:

$$\forall(v_i, v_j) \in V_\tau^2 : s(v_j) - s(v_i) + \min_{p \in \mathcal{P}(v_i,v_j)} \{d_q^p(v_i, v_j)\} \geq 1, \quad (7)$$

where V_τ^2 is the set of node pairs violating the constraint outlined above. V_τ^2 can be computed in $\mathcal{O}(|V|^2)$ time using a breadth-first technique. Instead of considering paths in the control flow, frequency constraints can be formulated in a similar way by considering chains in the *data flow* for which the delay exceeds some upper bound $\Delta_{\max}$.

4.2. Modifications for Wait-Sources

In the presence of wait-sources the number of states between two nodes can obviously not be determined statically, since it is not known how many wait-statements (or controller states) are generated by a wait-source. We thus have to generalize the definition of d_q.

Given a pair of nodes (v_i, v_j) and the set of wait-sources V_s between v_i and v_j, the distance $d_q^+(v_1, v_n)$ taking into account wait-sources can be computed as follows:

$$d_q^+(v_1, v_n) = d_q(v_1, v_n) + \sum_{v \in V_s} s(v). \quad (8)$$

The constraints have to be modified accordingly. Since it is no more possible to find the minimum distance between a pair of nodes, each of the constraints including a minimum-term has to be replaced by a set of constraints, one for each possible path between the two nodes. This guarantees that the constraints for each path will be met simultaneously. Moreover, the following constraint ensures that for a wait-source v_s at least $b_l(v_s)$ and at most $b_u(v_s)$ wait-statements are generated:

$$b_l(v_s) \leq s(v_s) \leq b_u(v_s). \quad (9)$$

4.3. Objective Functions

The constraint equations defined in the preceding subsections are all linear in the unknown schedule-variables; together with a linear objective function we thus arrive at an *integer linear programming* problem [10]. In the presence of wait-sources the minimum number of wait statements will be generated with the following objective function: $\sum_{v \in V_s} s(v)$. Note that this will implicitly *exactly* minimize the length of each path containing wait-sources, since only as many wait statements will be generated as are required to satisfy all resource and frequency constraints.

Similarly, cost functions can be formulated to minimize the number of controller states or the number of registers, either by minimizing the "maximum cut" or by maximum chaining. Minimization of registers is important from a low-power aspect, since the clock net accounts for a large percentage of the overall power budget. For more information on the objective functions, the reader is referred to [19].

4.4. Complexity

Let us first look at the complexity of a scheduling problem without wait-sources. The number of ILP variables in the model presented in the preceding sections is $\mathcal{O}(|V|)$, the number of constraints is $\mathcal{O}(|V|^2)$. In general, solving an integer linear program is an $\mathcal{NP}$-hard problem [5]. Note, however, that constraints (2)-(7) are all linearly dependent on only *two* ILP variables and that the corresponding coefficients are $+1$ and -1 in each equation. The constraint matrix defined by constraints (2)-(7) is a $(0, +1, -1)$ matrix in which no row has more than one $+1$ and one -1. This makes the constraint matrix *totally unimodular* and it follows that the optimum solution to the ILP problem can be found in *polynomial time* [10] as opposed to the approach in [21].

Moreover, the size of the ILP problem is independent of the number of paths in the description. Constraints (4)–(5) and (7) do consider pairs of nodes between which more than one path may exist, however only one constraint per pair is generated considering only the *minimum* delay on each path.

Obviously, we are not solving the general scheduling problem, which belongs to the class of $\mathcal{NP}$-complete problems. This is achieved by imposing a partial order on pairs of statements which belong to the set V_τ^2: Eq. (7) implicitly defines node v_j to be scheduled after node v_i, even though this order may not be imposed by any data dependence. In practice, we have found that this is not really a restriction, since the target applications are control-flow dominated and usually only consist of a few blocks of data-flow intensive code. Also note that due to the static nature of data-flow analysis, the mobility of a node may change when some other node is moved, so that the mobility as defined in Section 4.1 may actually over-constrain the ILP problem.

In the presence of wait-sources the total unimodularity of the constraint matrix is lost. Since in this case a distance d_q is not a constant term but a sum of schedule variables, there may be more than two non-zero entries in each row of the constraint matrix. Furthermore, as explained in Section 4.2, some constraints are replaced by *sets* of constraints, so that the problem size theoretically becomes dependent on the number of paths. In reality, though, we have observed that due to the relative small number of wait-sources in a typical application (cf. Section 5), the penalty in runtime is tolerable.

5. Results

The scheduling algorithm was implemented in the VOTAN (**V**HDL **O**ptimization, **T**ransformation and **AN**alysis) synthesis platform, an interactive framework for analysis and optimization of behavioral VHDL code designed to be used as a front-end optimizer to logic synthesis.

Comparing our scheduling technique with related work is difficult due to the fact that we require an initial sched-

Design				ILP Size		Register Minimization				State Minimization			
	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{Q}\rvert$	FUs	$\lvert\mathcal{S}\rvert$	Eqns	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{Q}\rvert$	$t[s]$	Gain	$\lvert\mathcal{R}\rvert$	$\lvert\mathcal{Q}\rvert$	$t[s]$	Gain
`qrs`	43	43	$3(+)$	242	573	35	49	0.5	18%	42	40	0.4	7%
`mw`	96	79	$1(-)$	632	1598	76	87	5.2	20%	90	57	4.4	27%
`X.25`	7	4	$2(\pm)$	45	97	6	4	< 0.1	14%	8	3	< 0.1	25%

Table 1. Benchmark Results

ule or at least, in the case of wait-sources, some notion in what part of the code "superstates" are to be specified. Table 1 reviews the benchmark examples on which we ran the scheduler. `qrs` is a circuit for heart rate monitoring, `mw` is part of an automotive control circuit and `X.25` is the send process from an X.25 communications protocol.

Of particular interest in this context are the `qrs`- and `mw`-examples, which are both heavily control-dominated. The `qrs` description is made up of only 272 lines of VHDL code (LOC) but contains approx. $6.7 \cdot 10^6$ paths. The `mw` description is 845 LOC containing more than $4 \cdot 10^9$ paths. The `X.25` example is of modest size with only 74 LOC and 12 paths.

Table 1 lists the number of registers $|\mathcal{R}|$ and states $|\mathcal{Q}|$ in the original description and the optimized descriptions scheduled for register and state minimization subject to the resource constraints given in the first column. The gain in columns three and four are the reduction in number of registers and states obtained with the appropriate objective function. The CPU time t for solving the ILP problems was measured on a SPARCstation 10 with 320 MB of memory clocked at 70 MHz. The ILP solver used was a public domain solver from TU Eindhoven [2]. Note that the gains, particularly for the large `qrs` and the `mw` examples, are significant while the CPU times required were relatively low.

6. Conclusion

We have presented an ILP formulation to solve the scheduling problem in control-flow dominated behavioral descriptions in polynomial time by restricting the *partial* order of statements. The size of the problem is independent of the number of paths in the VHDL description. Furthermore, its structure permits optimization of statement sequences across basic block boundaries. The model supports complete or partial specification of the timing of I/O signals which is retained during scheduling. Subject to resource and performance constraints, this scheduling approach allows optimization of the number of registers and controller states, which traditionally belong to the domain of RT-synthesis, on architectural level by means of code transformation. It is thus well-suited for optimizing VHDL descriptions prior to logic synthesis.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers — Principles, Techniques and Tools.* Addison Wesley, 1986.

[2] M. Berkelaar. lp_solve. Available via anonymous ftp from *ftp://ftp.es.ele.tue.nl/pub/lp_solve*.

[3] R. Camposano. Path-Based Scheduling for Synthesis. *IEEE Tr on CAD*, 10(1), Jan. 1991.

[4] S. Chaudhuri, R. A. Walker, and J. E. Mitchell. Analyzing and Exploiting the Structure of the Constraints in the ILP Approach to the Scheduling Problem. *IEEE Tr on VLSI Systems*, 2(4), Dec. 1994.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, 1979.

[6] C. H. Gebotys and M. I. Elmasry. Global Optimization Approach for Architectural Synthesis. *IEEE Tr on CAD*, 12(9), Sept. 1993.

[7] S. H. Huang et al. A Tree-Based Scheduling Algorithm for Control-Dominated Circuits. In *Proc. of the 30th DAC*, June 1993.

[8] D. Knapp, T. Ly, D. MacMillen, and R. Miller. Behavioral Synthesis Methodology for HDL-Based Specification and Validation. In *Proc. of the 32nd DAC*, June 1995.

[9] D. Ku and G. DeMicheli. Relative Scheduling under Timing Constraints. *IEEE Tr on CAD*, 11(6), June 1992.

[10] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[11] G. D. Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, 1994.

[12] K. O'Brien, M. Rahmouni, and A. Jerraya. A VHDL-Based Scheduling Algorithm for Control-Flow Dominated Circuits. In *Proc. of the Sixth Int'l Workshop on High-Level Synthesis*, Nov. 1992.

[13] I. Radivojević and F. Brewer. A New Symbolic Technique for Control-Dependent Scheduling. *IEEE Tr on CAD of Circuits and Systems*, 15(1), Jan. 1996.

[14] M. Rim, Y. Fann, and R. Jain. Global Scheduling with Code-Motions for High-Level Synthesis Applications. *IEEE Tr on VLSI Systems*, 3(3), Sept. 1995.

[15] H. Shin and N. S. Woo. A Cost Function Based Optimization Technique for Scheduling in Data Path Synthesis. In *Proc. of ICCD '89*, Oct. 1989.

[16] A. Stoll and P. Duzy. High-Level Synthesis from VHDL with Exact Timing Constraints. In *Proc. of the 29th DAC*, June 1992.

[17] J. Vanhoof, K. V. Rompaey, I. Bolsens, G. Goosens, and H. DeMan. *High-Level Synthesis for Real-Time Digital Signal Processing.* Kluwer Academic Publishers, 1993.

[18] K. Wakabayashi and T. Yoshimura. A Resource Sharing and Control Synthesis Method for Conditional Branches. In *Proc. of the 1989 IEEE ICCAD*, Nov. 1989.

[19] N. Wehn et al. Scheduling of Behavioral VHDL by Retiming Techniques. In *Proc. of the Euro-DAC '94*, Sept. 1994.

[20] J. C.-Y. Yang, G. D. Micheli, and M. Damiani. Scheduling with Environmental Constraints based on Automata Representations. In *Proc. of the 3rd EDAC*, Feb. 1994.

[21] T.-Y. Yen and W. Wolf. An Efficient Graph Algorithm for FSM Scheduling. *IEEE Tr on VLSI Systems*, 4(1), Mar. 1996.