

Scheduling of Behavioral VHDL by Retiming Techniques*

N. Wehn¹, J. Biesenack¹, T. Langmaier¹, M. Münch^{1,2}, M. Pils¹, S. Rumler¹, P. Duzy¹

¹ Siemens AG, ZFE BT SE 52
Otto-Hahn-Ring 6, D-81730 München

² Institut für Rechnerentwurf und Fehlertoleranz
Universität Karlsruhe, D-76128 Karlsruhe

Abstract

In this paper we present a new approach to the scheduling of behavioral VHDL descriptions for control-flow dominated applications containing a large number of nested conditionals and data dependent loops. The proposed algorithm is able to schedule and re-schedule descriptions for optimization subject to various cost functions. The timing of the I/O signals can be completely fixed, partially fixed or left to the scheduler. In this case the algorithm produces a schedule such that the number of clock cycles required for a complete execution of the behavioral description is minimized. Scheduling is performed as a behavioral VHDL code transformation and allows taking advantage of all the power of commercial RT synthesis systems. The corresponding problem is solved based on an analogy to the re-timing problem on RT-level networks which can be solved in polynomial time. The efficiency of our approach is demonstrated on various examples.

1 Introduction

In the past a lot of effort has been spent in the field of high-level synthesis. Since VHDL is well accepted both in the specification and design of electronic systems, many high-level approaches use VHDL as specification language. In general, the specification of a complex system is hierarchically partitioned by system designers into many smaller units. Each unit is specified in its most appropriate way, i.e., on different levels of abstraction, on logic level (or even lower) as well as on register-transfer or algorithmic level. Units which are reused from previous designs are normally specified on such a low level. Thus, *high-level synthesis is applied only for some parts of the whole design*. This implies that *the algorithmic units must cooperate with all other units especially with respect to the timing*. While combinatorial units and state machine descriptions already imply a fixed timing with respect to a clock (represented by wait-statements in the VHDL description), algorithmic descriptions usually do not reflect timing issues. This causes a severe problem: simulation of the complete system (usually modelled as a set of concurrently executing processes) becomes almost impossible since the signal interfacing with respect to timing between these different types of descriptions is not defined. Unlike many high-level synthesis approaches [1, 2] that do not tackle this problem, ours allows partial or complete specification of the interface timing of the I/O signals on VHDL

source level in the algorithmic description. Although this seems to be an over-specification, there is still a lot of freedom left for scheduling: delays specified in VHDL refer to the scheduling of events on data carriers in future simulation time, with the actual execution of statements taking zero simulation time. As a consequence, the input-to-output delay for a set of assignment statements can be arbitrarily distributed over the corresponding input-to-output path, and thus offers freedom for scheduling. Even in the case where only algorithmic processes have to communicate with each other, a timing problem arises. If two algorithmic processes have to communicate, reading and writing of communication signals often have to satisfy certain timing relations, e.g. a signal written in one process has to be read exactly one clock cycle later in the other process. Hence, scheduling of both processes can not be done independently since the timing relation may not be violated. In this paper we present a new approach for scheduling of control-flow dominated applications which differs from other approaches targeting the same domain of applications [1]:

- The control-flow is globally considered but in contrast to other approaches the potential parallelism among operations due to data-dependencies is also exploited within basic blocks, i.e. the order of the statements is not fixed during scheduling;
- Timing constraints on paths between I/O operations can be fixed;
- Timing relations on communication signals in different processes can be maintained when scheduling the processes;
- The scheduling problem is formulated as an ILP-problem, which is in our case under certain assumptions the dual to the minimum-cost flow problem. As this can be solved in polynomial time, complexity problems with large descriptions do not occur.

The paper is organized as follows. Section 2 introduces the model to specify a design in VHDL and motivates our new scheduling approach. A formal description of the problem and its solution is given in Section 3. Section 4 outlines extensions to the VHDL input model that can be easily integrated into the scheduler. Section 5 compares to existing approaches and in Section 6 experimental results are reported. Section 7 concludes the paper.

*This work was in part supported by the BMFT under JESSI AC-8

2 Basic Idea

In this section the model used to specify a behavioral VHDL description is presented. We consider single VHDL processes in which the behavior is described by `wait`-statements and sequential statements and the schedule of all I/O operations is completely fixed in terms of clock cycles. All `wait`-statements are related to the same edge of a single clock signal: e.g. `wait until clock='1'` (for the remainder of this paper, this statement will be referred to simply as “`wait`-statement”). Such a description is denoted a *sequential FSM model* (abbreviated as SFM). Figure 1 shows a typical example of an SFM. Later we will relax this model in order to process descriptions in which the I/O schedule is only partially or not at all fixed. Due to the VHDL semantics signal accesses in a process represent the I/O operations as signals can not be declared within the process. When a VHDL-simulator has to process a `wait`-statement, the process is suspended, time advances to the next rising clock edge and the process resumes. Hence, the cycle delay between I/O operations depends on the executed path through the behavioral description (which in turn depends on the actual input values) and the *number* of `wait`-statements on this path. It follows that the number of `wait`-statements on each possible I/O path completely determines the schedule of the I/O signals in a process, but not the schedule of the operations between them since the delay can be arbitrarily distributed on such an I/O path. In Figure 1 a further SFM (SFM_2) is shown in which the distribution of the `wait`-statements is modified compared to SFM_1 without changing the I/O schedule.

<pre> 1 architecture SFM_1 of B is 2 process 3 begin 4 statement_IO_1; 5 wait until clock='1'; 6 statement_1; 7 statement_add_2; 8 if (condition='0') then 9 statement_add_3; 10 wait until clock='1'; 11 statement_4; 12 else 13 statement_add_5; 14 wait until clock='1'; 15 statement_IO_2; 16 wait until clock='1'; 17 end if; 18 statement_IO_3; 19 end process; 20 end SFM_1; </pre>	<pre> architecture SFM_2 of B is process begin statement_IO_1; wait until clock='1'; statement_1; statement_add_2; wait until clock='1'; if (condition='0') then statement_add_3; statement_4; else statement_add_5; statement_IO_2; wait until clock='1'; end if; statement_IO_3; end process; end SFM_2; </pre>
--	---

Figure 1: Two equivalent sequential FSM models

Obviously the `wait`-statements partition the behavioral description into smaller description units, each bounded by `wait`-statements. Since there is no `wait`-statement within such a unit, all statements of a unit must be executed in the same clock cycle or state, respectively. The transition from one state to another state is controlled by the conditions on the corresponding program paths.

There are four states q_i in the process description SFM_1 shown in Figure 1 (in the following, statements are labeled by their line numbers in SFM_1): state q_1 contains

statements 6, 7, 8, 9 and 13, q_2 contains statements 11, 17, 18, 19, 3, 4, q_3 contains statement 15 and q_4 statements 17, 18, 19, 3 and 4. In comparison, process description SFM_2 consists of three states: q'_1 contains statements 6 and 7, q'_2 statements 8, 9, 11, 13, 15, 17, 18, 19, 3 and 4, and finally statements 17, 18, 19, 3 and 4 make up q'_3 .

Hence, SFM_2 can yield a better hardware implementation than SFM_1 since the controller needs one state less and the datapath can be implemented with a single adder (note that statements 9 and 13 can share the same hardware, since they are on different program paths).

This example motivates the idea of our scheduling algorithm. The method exploits the fact that the scheduling of non-I/O operations of the process can be changed without changing the I/O behavior and timing. A scheduling problem which does not change the schedule of signal accesses in an SFM is called *WYSIWYS* (What You Simulate Is What You Synthesize) [3, 4] scheduling problem since simulation before and after scheduling matches completely at the process boundary on clock cycle level. Hence, we also use the term “re-scheduling” in the following. Our scheduler assigns operations to clock cycles by generating or deriving new positions for the `wait`-statements in the VHDL description and changing the order of the sequential statements within basic blocks. It takes into account the fixed I/O behavior as well as additional user-defined constraints like resource or frequency constraints. Various cost functions may be selected to be minimized by the algorithm. In the following we discuss the scheduling technique in more detail.

Note that in our model, *the complete I/O schedule is fixed using only VHDL statements and their semantics*. There is no need for a timing specification which is separated from the VHDL description or which extends the language, and thus can not be interpreted by a VHDL simulator. Additionally, all the timing constraints are “well posed”, which means that you can not specify constraints that contradict each other. The only condition that must hold to make such a description synthesizable in principle is, that each loop in the process (which is not unrolled) is cut by at least one `wait`-statement. This condition may be relaxed by false-path analysis.

3 Formal Description

3.1 Internal Representation

An SFM is transformed into a Control/Data Flow Graph (CDFG) to represent the operations and their control and data flow dependencies.

The CDFG is a directed, cyclic graph $CDFG(V, E)$ with nodes $V = V_{io} \cup V_w \cup V_l \cup V_{vir}$ and edges $E = E_c \cup E_v \cup E_{du}$. Each statement in the VHDL description is represented by a node in $V_{io} \cup V_w \cup V_l$. V_{io} is the set of I/O operations, i.e. signal read and write accesses. Nodes in V_w represent `wait`-statements, whereas those in V_l represent all remaining statements. Finally, V_{vir} is the set of *virtual* nodes required to model control flow. They do not directly correspond to VHDL statements. For instance, `loopJoin`-nodes are used as common exit points for all `exit`-statements within a loop.

The edge set $E_c \cup E_v$ models the control flow. An edge $(v_i, v_j) \in E_c$ is generated if the statement represented by v_i is a predecessor of the statement associated with v_j in the VHDL description. Like V_{vir} , E_v contains all *virtual* edges to model control flow, e.g. of loops. Among them are edges linking a `loopEnd`-node with its corresponding `loopBegin`-node or `processEnd`-nodes with `processBegin`-nodes. To represent data flow, edges (u, v) in E_{du} link nodes u which write a variable directly to nodes v which read the variable written at u . These links are called *definition-use chains* and can be established by global data flow analysis techniques as described in [5].

To clarify these definitions, the left part of Figure 2 shows the CDFG generated from the SFM.1 description in Figure 1. Edges in E_c are represented by solid arcs, virtual edges by dashed arcs. I/O operations are marked as dotted nodes, `wait`s as dashed nodes. Due to the lack of `exit`-statements, there are no virtual nodes: $V_{vir} = \emptyset$. Also, $E_{du} = \emptyset$, as Figure 1 exhibits no data dependencies. For all other subsets of V we have: $V_{io} = \{v_4, v_{15}, v_{18}\}$, $V_w = \{v_5, v_{10}, v_{14}, v_{16}\}$ and $V_l = \{v_3, v_6, v_7, v_8, v_9, v_{11}, v_{13}, v_{17}, v_{19}\}$. In the follow-

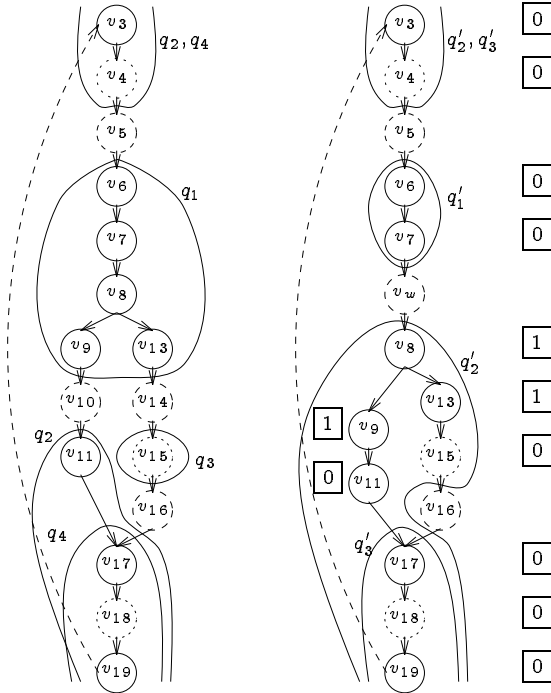


Figure 2: CDFG of example description

ing, we will first concentrate on the control flow part of the CDFG. It can be defined as a subgraph $CFG(V, E \setminus E_{du})$ of the CDFG. A *path* $p(v_1, v_n)$ between two nodes v_1, v_n is defined as a finite sequence of nodes $p(v_1, v_n) = (v_1, v_2, \dots, v_n)$, such that $(v_i, v_{i+1}) \in E_c \cup E_v$ for all $1 \leq i \leq n-1$.

3.2 Mathematical Formulation

Scheduling a CFG can be seen as partitioning the CFG into rooted, acyclic subgraphs. All operations belonging to

such a subgraph are executed within the same state. Due to the nature of the CFG, only one path can be active at any time. Determining a new distribution of `wait`s in a CFG corresponds to *re-scheduling* (or *re-partitioning*) the CFG.

Modeling the re-scheduling is done by exploiting an analogy to the *Retiming*-optimization known from RT-synthesis [6]. Retiming computes a new register distribution between logic blocks without changing the I/O behavior. By establishing a correspondence between wait-statements and registers as well as between the CFG and the network graph, re-scheduling can be considered a special case of retiming.

To mathematically model the problem within our graph framework, we slightly modify the graph structure by removing `wait`-statements from the CFG and replacing them with edges linking the immediate predecessor u and successor v of the `wait`-statement removed. We call this a *weighted CFG*, since the new edge $e = (u, v)$ is assigned a *weight* $w(u, v) = 1$ (removal of n successive `wait`-statements results in an edge $e = (u, v)$ with $w(u, v) = n$). All other edges are assigned zero weight. We can now define the weight $w_p(v_1, v_n)$ of a path $p(v_1, v_n) = (v_1, v_2, \dots, v_n)$ between two nodes v_1 and v_n as the sum of the edge weights on the path:

$$w_p(v_1, v_n) = \sum_{i=1}^{n-1} w(v_i, v_{i+1}). \quad (1)$$

A new distribution of `wait`-statements can be expressed as a recomputation of edge values in the weighted graph. To do so, we assign integer *lag* variables $l(v)$ to each node $v \in V_{io} \cup V_l \cup V_{vir}$. The semantics of a lag variable $l(v)$ can be informally described as the number of `wait`-statements moved over the node v . For instance, an assignment $l(v) = 1$ means that one `wait`-statement is moved from each outgoing edge to each incoming edge of v . Recomputation of a graph's edge weights is done based on the values of the lag variables. The new weight $w'(u, v)$ of an edge (u, v) is computed as follows:

$$w'(u, v) = l(v) - l(u) + w(u, v), \quad (2)$$

as $l(v)$ “`wait`s are moved between u and v ” and $l(u)$ `wait`s are removed. Similarly, the weight $w'_p(v_1, v_n)$ of the path $p(v_1, v_n)$ after a transformation is $w'_p(v_1, v_n) = l(v_n) - l(v_1) + w_p(v_1, v_n)$.

Obviously, calculating a new distribution of the `wait`-statements is equivalent to computing an *integer* assignment to the lag variables. To guarantee the consistency of the resulting VHDL description the transformation is subject to certain constraints to which the assignment must comply. These constraints can be expressed as linear equations in the lag variables; the transformation problem can thus be mapped onto an ILP problem to be solved with respect to an objective function. We will first have a look at the constraints.

To preserve the I/O behavior, the number of `wait`-statements on a path between any pair of I/O operations

must not change:

$$\forall v_i \in V_{io} : l(v_i) = \text{constant}. \quad (3)$$

Also, the consistency of the graph structure requires that no edge weight becomes negative in the transformed weighted CFG and that virtual edges never have a nonzero weight, as these edges do not have any correspondence in the VHDL program:

$$\forall (u, v) \in E_c : w(u, v) + l(v) - l(u) \geq 0, \quad (4)$$

$$\forall (u, v) \in E_v : l(v) - l(u) = 0. \quad (5)$$

The proof that these constraints are sufficient to guarantee the validity of the transformed VHDL program in terms of I/O behavior and program structure is omitted due to space limitations. So far we have presented a technique to model a re-distribution of `wait`-statements on I/O paths. We can use this technique to determine a re-scheduling that meets *resource and performance constraints* and optimizes various *objective functions*.

Suppose the number of functional units available of type τ is n_τ . Then, no more than n_τ operations of type τ must be executed in each state. Let V_τ^2 be the set of all pairs of nodes of operation type τ between which a path exists with exactly $n_\tau + 1$ operations of type τ . At least one `wait` must be placed onto such a path to guarantee that no more than n_τ operations of type τ will be executed within the same clock cycle:

$$\forall (u, v) \in V_\tau^2 : l(v) - l(u) + \min_{p \in p^*(u, v)} \{w_p(u, v)\} \geq 1. \quad (6)$$

$p^*(u, v)$ is the set of all paths between nodes u and v . Computation of the minimum in equation 6 guarantees that the number of `waits` on the shortest path (in terms of clock cycles) between u and v will never be less than one.

To specify a resource constraint of one adder for the example in Figure 1, we would have to place at least one `wait` on each of the paths between nodes v_7 and v_9 , v_7 and v_{13} as well as v_9 and v_7 and v_{13} and v_7 , due to the loop characteristic of a process body. The equations generated for v_7 and v_9 and v_{13} and v_7 , respectively, would be

$$\begin{aligned} l(v_9) - l(v_7) + 0 &\geq 1, \\ l(v_7) - l(v_{13}) + 3 &\geq 1. \end{aligned}$$

Similar equations would be generated for the remaining node pairs.

Specifying timing constraints is done in a similar way: To ensure a minimum frequency $1/\Delta$ of the synthesized circuit, we require that the maximum delay within a clock cycle due to operator chaining must be less than or equal to Δ . For this purpose, the E_{du} edge set of the CDFG is used. As above, we force that at least one `wait` is placed between each pair of nodes linked by (possibly a sequence of) data-dependency edges that violate the constraint. Let V_Δ^2 be the set of all such node pairs, so that the delay on the sequence is greater than Δ unless the last node is removed. The constraint equation is:

$$\forall (u, v) \in V_\Delta^2 : l(v) - l(u) + \min_{p \in p^*(u, v)} \{w_p(u, v)\} \geq 1. \quad (7)$$

This requires library information about the delay of the operations executed in a CDFG node.

The objective function optimizes the ILP solution, and as such the transformed VHDL description, with respect to a cost function. This permits optimizing a design's area consumption. Objective functions are linear functions in L , the set of lag variables. We will focus on functions to minimize controller and register area.

Minimizing controller area can be done by minimizing the number of `wait`-statements, as each `wait` can be identified with a controller state. Minimizing `wait`-statements is achieved by moving `waits` over nodes which have more outgoing edges than incoming edges, such as `fork`-nodes. In our ILP formulation, moving a `wait`-statement over a `fork`-node is equivalent to decrementing the edge weight of each outgoing edge by one and incrementing the incoming edge's weight by one. The objective function can thus be expressed as follows:

$$\min \sum_{v \in V} (l(v) \cdot (\text{indeg}(v) - \text{outdeg}(v))). \quad (8)$$

The functions $\text{indeg}(v)$ and $\text{outdeg}(v)$ return the number of incoming and outgoing edges of v , respectively.

Register minimization can be either done by maximizing chaining of operations, thus requiring less registers to store values over state transitions, or by minimizing the maximum number of live values (i.e., the maximum cut) at a state transition. Chaining is achieved by minimizing the "distance" between two operations linked by an edge in E_{du} :

$$\min \sum_{x \in \Phi} \min \sum_{\substack{e=(u,v) \in E_{du} \\ \nu(e)=x}} (l(v) - l(u)) \cdot b_x, \quad (9)$$

with Φ being the set of all variables in the VHDL-description and $\nu(e)$, $e = (u, v) \in E_{du}$ a function returning the variable written at node u and read at node v . b_x is the bitwidth of variable x .

To compute the maximum cut, we need to know which values of a variable are live at a certain state transition. A value of a variable x is called live at an edge $(u, v) \in E_c$ if there is an edge $e = (v_w, v_r) \in E_{du}$ with $\nu(e) = x$ and a path $p(v_w, v_r) = (v_w, v_1, v_2, \dots, v_n, v_r)$ that contains (u, v) , i.e. $\exists i : 1 \leq i < n : u = v_i \wedge v = v_{i+1}$. Let $l_x(u, v) = 1$ if the value of variable x is live at (u, v) , else $l_x(u, v) = 0$. We now minimize the overall sum of all bits required to store live values as follows:

$$\min \sum_{(u, v) \in E_c} ((l(v) - l(u)) \cdot (\sum_{x \in \Phi} l_x(u, v) \cdot b_x)). \quad (10)$$

The different objective functions may also be merged into a single one in which the components are summed and weighted by constants.

Solving the ILP problem with state minimization and a resource constraint of one adder for our example would yield the following solution: $l(v_8) = l(v_9) = l(v_{14}) = 1$ and $l(v_i) = 0$ for all remaining nodes in the weighted CFG. This is exactly the transformation shown in Figure 1. The

transformed CFG is shown on the right in Figure 2. The boxes next to each node specify the value of the node’s lag variable. The `wait`-statements at nodes v_{10} and v_{14} have been merged into the single new node v_w .

3.3 Solution and Complexity

Constraints (3)–(7) are linearly dependent on the lag variables in L . So are the objective functions (8)–(10). Hence, the problem to be solved is an ILP-problem [7]. The number of ILP variables is $\mathcal{O}(|V|)$ and the number of constraints is in worst case $\mathcal{O}(|V|^2)$. Although solving an ILP problem is in general $\mathcal{NP}$ -complete, this specific problem is the dual to the *minimum cost-flow problem* which can be solved in polynomial time [8]. This is due to the fact that each row of the ILP constraint matrix made up of constraints (3)–(7) does not contain more than two entries which are nonzero, as each equation depends on only two lag variables.

It is important to note that the problem size does not depend on the number of paths in the VHDL description. Constraints (6) and (7) are the only ones which deal with nodes between which more than one path may exist. However, due to the computation of the minimum, only one equation is generated for each such node pair.

4 Extensions

The model can be relaxed by various extensions to allow more degrees of freedom while scheduling. Among them are extensions to *relax the WYSIWYS paradigm*, to *re-sequence operations* in basic blocks and for *multi-process scheduling*.

To prevent over-specification of the design, the WYSIWYS paradigm can be relaxed by allowing I/O operations to float within a time window [4]. Let s the size of the window of I/O node v_{io} , then constraint (3) must be replaced by $-s + \text{constant} \leq l(v_{io}) \leq s + \text{constant}$.

If the interface behavior has been incompletely specified, “sources” of `wait`-statements can be defined. Sources mark points in the VHDL description at which `waits` can be created. The lag variable of a source specifies the number of `wait`-statements created at this node. They must be taken into account while computing a path weight, which is now no more a constant but a term with a constant part and the sum of the lag variables of all source nodes on the path. To introduce a source node v_s which produces a maximum of two `waits` between v_7 and v_8 , the constraint to fulfill a constraint of one adder would then, in the case of nodes v_7 and v_9 , be modified as follows:

$$\begin{aligned} l(v_9) - l(v_7) + l(v_s) &\geq 1, \\ l(v_s) &\leq 2. \end{aligned}$$

Sources can be placed onto any path to allow up to n clock cycles for its execution, thus exploring more design alternatives simultaneously. Using time windows or `wait`-sources requires trading off the additional degrees of freedom against design validation. In this case, design validation should take place *after* scheduling has been performed.

As many other control-flow oriented scheduling algorithms, the above model does not allow to change the sequence of operations, thus making it impossible to satisfy certain resource or performance constraints even if such a solution exists. Especially code segments with a high share of dataflow hold a potential for optimization by parallelizing operations through *resequencing*. To identify these code segments, a *flow graph* [5] is constructed, whose nodes are *basic blocks* consisting of straight-line VHDL operations. The constraints are then modified as follows: one set of equations ensures that the overall control flow is retained, i.e. the topology of the flow graph is not changed. For each basic block another set of equations is generated to ensure that *data-dependencies* within the block are not violated. However, the structure of the equations allows interchanging operations within the block to meet resource or performance constraints. The ILP problem still yields an exact solution in polynomial time, as the sequence of operations on which a constraint is defined remains unchanged.

Scheduling multi-process descriptions can be done by merging the ILP problem of each process into a single ILP problem, in which relations between synchronizing operations ensure that the communication protocol is not violated. Imagine, a node v_{p_1} in process p_1 sends a signal to a node v_{p_2} in process p_2 that reads this signal exactly one clock cycle after v_{p_1} has sent it out. For correct synchronization, re-scheduling must not change this timing relation. This is achieved by equating the two lag variables $l(v_{p_1}) = l(v_{p_2})$ provided that the `wait`-statements have already been specified or else by the relation $l(v_{p_1}) = l(v_{p_2}) - 1$.

5 Comparison with other approaches

The presented scheduling approach emphasizes on control-flow dominated VHDL descriptions. Although a lot of papers on scheduling algorithms [9, 10] exist, few techniques suitable for complex control-flow dominated applications have been published. Most are extensions of dataflow oriented scheduling techniques to consider control-flow [11, 12, 13], but these algorithms only feature the efficient processing of conditional branches caused by `if` and `case` statements. Realistic control-flow dominated descriptions, however, contain a lot of nested data dependent loops with several exits as well as nested `if` and `case` statements. There are only some scheduling techniques known which try to tackle efficiently these descriptions [1, 14, 2]. Path-based scheduling [1] is the best known representative, but it contradicts the WYSIWYS principle since it aims at scheduling each I/O path as fast as possible. Fixing the timing on an I/O path requires a complex constraint specification mechanism. Also, the fact that path-based scheduling considers *all* paths explicitly in a description makes it unsuitable to schedule large specifications.

6 Results

The scheduling algorithm has been implemented in our open synthesis platform **VOTAN** (VHDL Optimization,

Design	VHDL Spec			Re-scheduled VHDL				
	Reg Bits	States	Area [μ^2]	Reg Bits	States	Area [μ^2]	CPU t [s]	SP/LP
qrs	410	20	7842	394	23	7575	0.5	11/30
X.25	137	5	1912	120	6	1630	< 0.1	1/4
ellipf	608	14	7388	560	14	7033	0.1	14/14
diffeq	12	5	16941	8	6	15136	0.1	1/6
mw	91	79	2596	84	54	2357	3.3	1/84
gcd	32	3	2324	32	2	2287	< 0.1	1/2

Table 1: Influence of objective function on area consumption.

Transformation and Analysis) which provides all facilities to efficiently manage VHDL code in a hypertext based environment. VOTAN offers various VHDL transformations and fast analysis tools to estimate the effect of a particular transformation.

Table 2 shows the main characteristics of some benchmark circuits. `ellipf`, `gcd` and `diffeq` are standard benchmarks taken from the High-Level Synthesis Workshop Benchmark Suite [15]. `qrs` is a circuit for heart rate monitoring currently under review for the benchmark suite. `mw` is part of an industrial communication circuit. Especially the latter two are heavily control-flow dominated designs. X.25 is the send process of a X.25 communications protocol [16]. Row 6 of Table 2 lists the functional unit constraints as they were specified for scheduling. In the case of `mw` and `gcd` the I/O behavior was fixed, for all other designs the minimum number of waits required to meet all resource constraints was generated using source nodes. For `mw` and `gcd` controller states are listed instead of register bits.

Property	qrs	X.25	ellipf	diffeq	mw	gcd
LOC	272	74	147	69	845	42
Paths	$5.7 \cdot 10^6$	8	1	3	$\geq 4 \cdot 10^9$	3
Loops	3	3	1	2	35	2
ifs	14	2	0	0	52	1
Nesting	4	3	0	1	6	2
FUs	$1(\geq)$ $3(+)$	$2(\leq)$ $1(\pm)$	$2(+)$ $1(*)$	$1(\pm)$ $2(*)$	$2(+)$	$1(-)$
$ L $	240	43	56	40	538	20
ILP-Eqs	508	81	127	87	1019	42

Table 2: Characteristics of benchmark examples

Table 1 lists the results of (re-)scheduling along with run times required for solving the ILP-problem. Objective functions were set to minimize register area, except for `mw` and `gcd` which were set to minimize the number of controller states. Column SP/LP lists the shortest and longest paths for a complete execution of the description after re-scheduling with state optimization. Benchmarks were run on a SPARCstation 10 clocked at 30 MHz with 64 MByte of memory. Even for the large `mw`-example, run times were negligible.

Also shown in Table 1 are results of mapping the scheduled and non-scheduled VHDL descriptions onto a netlist by a commercial RT-synthesis tool. The constraints were set to “minimize area”, timing constraints were not specified.

7 Conclusion

This paper presented techniques to schedule and optimize behavioral VHDL descriptions. Our model respects

the timing of I/O signals to allow integration in existing VHDL-based design methodologies. All transformations are done on VHDL code to enable the designer to exploit the power of subsequent synthesis tools. The low runtimes permit fast exploration of the design space.

References

- [1] R. Camposano, “Path-Based Scheduling for Synthesis,” *IEEE Trans. on CAD*, vol. CAD-10, pp. 85–93, Jan. 1991.
- [2] S. Huang, Y. Jean, C. Hwang, and J. Wang, “A Tree-Based Scheduling Algorithm For Control-Dominated Circuits,” in *Proc. 30th DAC*, pp. 578–582, June 1993.
- [3] J. Biesenack *et. al.*, “The High-Level Synthesis System CALLAS,” *IEEE Trans. on VLSI-Systems*, vol. 1, pp. 244–254, Sept. 1993.
- [4] A. Stoll and P. Duzy, “High-level synthesis with exact timing constraints,” in *Proc. 29th DAC*, pp. 188–193, June 1992.
- [5] A. Aho, R. Sethi, and J. Ullman, *Compilers — Principles, Techniques and Tools*. Addison-Wesley, 1986.
- [6] C. Leiserson, F. Rose, and J. Saxe, “Optimizing synchronous circuits by retiming,” in *Proc. of the Third Caltech Conference on VLSI*, pp. 87–116, 1983.
- [7] A. Gibbons, *Algorithmic Graph Theory*. Cambridge: Cambridge University Press, 1985.
- [8] L. Lawler, *Combinatorial Optimization: Networks and Matroids*. New York: Reinhart and Winston, 1976.
- [9] M. McFarland, A. Parker, and R. Camposano, “The High-Level Synthesis of Digital Systems,” *Proceedings of the IEEE*, vol. 78, pp. 301–318, Feb. 1990.
- [10] R. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*. Norwell, MA: Kluwer Academic Publishers, 1991.
- [11] A. Parker, J. Pizarro, and M. Mlinarr, “MAHA: A program for data path synthesis,” in *Proc. 23rd DAC*, pp. 461–466, Juli 1986.
- [12] K. Wakabayashi and T. Yoshimura, “A Resource Sharing and Control Synthesis method for Conditional Branches,” in *Proc. ICCAD’89*, pp. 62–65, November 1989.
- [13] T. Kim, J. Liu, and C. Liu, “A Scheduling Algorithm for Conditional Resource Sharing,” in *Proc. ICCAD’91*, pp. 84–87, November 1991.
- [14] W. Wolf, A. Takach, C. Hwang, R. Manno, and E. Wu, “The Princeton University behavioral Synthesis System,” in *Proc. 29th DAC*, pp. 182–187, June 1992.
- [15] “High-Level Synthesis Workshop Benchmarks.” Available via anonymous ftp from [mcnc.mcnc.org](http://mcnc.mcnc.org/pub/benchmark) in pub/benchmark.
- [16] A. Tanenbaum, *Computer Networks*. Prentice Hall, Englewood Cliffs, NJ, 1989.