

# A Symbolic Approach for the Combined Solution of Scheduling and Allocation

Gianpiero Cabodi<sup>†</sup>  
Sergio Nocco<sup>†</sup>

<sup>†</sup>Politecnico di Torino  
Dip. di Automatica e Informatica  
Turin, ITALY

Mihai Lazarescu<sup>°</sup>  
Claudio Passerone<sup>‡</sup>

<sup>‡</sup> Politecnico di Torino  
Dip. di Elettronica  
Turin, ITALY

Luciano Lavagno<sup>‡</sup>  
Stefano Quer<sup>†</sup>

<sup>°</sup>Cadence Design Systems, Inc.  
Turin, ITALY

## ABSTRACT

Scheduling is widely recognized as a very important step in high-level synthesis. Nevertheless, it is usually done without taking into account the effects on the actual hardware implementation.

This paper presents an efficient symbolic technique to concurrently integrate operation scheduling and resource allocation. The technique inherits all the features of “standard” BDD-based control dominated scheduling, including resource-constraining, speculation and pruning. In addition, it introduces an efficient way of encoding allocation information within a symbolic scheduling automaton with a two-folded target. Firstly, it finds a minimum cost allocation of operation resources satisfying a given schedule. Secondly, it optimizes the amount of registers required to store intermediate results of operations.

Theory and algorithms are developed and presented. Experimental results on a well known set of benchmarks show the potentiality of the approach.

## Categories and Subject Descriptors

B.6.3 [Logic Design]: Design Aids — Automatic synthesis

## General Terms

Algorithms, design

## Keywords

High-level synthesis, scheduling, allocation, BDD, automata

## 1. INTRODUCTION

Embedded systems are often specified at a very abstract level and are successively refined toward an implementation, which is partitioned into hardware and software. To be able to rapidly explore a large design space it is desirable to have an automatic tool to generate a prototype implementation from a high level specification. In this paper we address the problem of synthesizing custom hardware for embedded systems and we thus developed a high level synthesis tool that we use in our hardware/software codesign framework [1].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISSS'02, October 2–4, 2002, Kyoto, Japan.

Copyright 2002 ACM 1-58113-576-9/02/0010 ...\$5.00.

Synthesis of efficient and high performance control units and data paths from high-level behavioral specifications has long been considered a very promising technique for tackling the ever growing complexity of digital design. At the same time, it is a very elusive goal, because after more than twenty years of intensive research, and even the appearance on the market of some industrial CAD tools, high-level synthesis is still far from being widely used as its predecessors, register-transfer level and logic synthesis.

The systems that we are targeting are often a mix of data and control. However, many HLS tools use Control Data Flow Graphs (CDFGs) as their internal model and do not model well constraints coming from input/output operations with the external world (e.g., synchronization, min/max rate, jitter, etc.) and often mostly data dependencies are handled, while control is either ignored or handled by complete case splitting<sup>1</sup>.

Although we use CDFGs as the input specification for our tool, we extend the model introduced by [3], which is at the same time *formal* (based on concurrent automata), *efficient* (it is possible to use symbolic representation techniques with enhancements derived from concurrent specification models), *control-oriented* (condition evaluation and speculative execution are specific features of [3]), and *flexible* (I/O constraints can be represented by restrictions on the automata state space).

Traditionally, the high-level synthesis problem has been split into a sequence of steps in order to make it manageable:

- Allocation chooses the type and number of functional units and registers, and thus determines part of the final cost (the interconnection cost still has to be identified) and performance (the clock cycle is affected by this stage).
- Scheduling assigns time slots (often clock cycles) to I/O, arithmetic and logical operations of the CDFG, and thus determines part of the final performance (the clock cycle still has to be defined).
- Binding assigns a functional unit to each operation, a register to each value that must be preserved across clock cycles, and enough multiplexers or busses to implement all required data transfers (this step further affects the cost and clock cycle).

This separation comes at a cost in terms of optimality. As a consequence, several approaches have tried to combine two or all the three steps in just one phase. However, since any of these problems is NP-complete by itself, the combination generally requires the use of efficient heuristics, that may thus forfeit expected improvements with respect to better and more complex algorithms applied in succession.

<sup>1</sup>Only recently approaches such as [8] that specifically address control-intensive CDFGs have been introduced.

In this paper we also address this issue, by combining the scheduling and allocation steps together, while keeping an implicit representation of the *complete solution space* (as [3] does but dealing with scheduling alone). The designer must still explore the design space by defining the acceptable maximum numbers of functional units and registers. We believe that this data path architecture definition is too critical to be left to a tool, and we provide the designer with a quick feedback on the effect of his decisions.

As [3] we represent implicitly the full solution space by means of the state space of a product of automata, and we represent resource (allocation) constraints by reducing the concurrency of the automata. Our contribution is the introduction of an encoding model for the *allocation* information, that allows us to take into account also *estimated clock cycle length*, in addition to functional unit costs and number of clock cycles.

Once the set of valid schedules is symbolically computed, the extra information we encode allows us to find a schedule with best allocation cost, whereas in [3] a schedule is only checked to fit within the given resource bounds.

## 2. BACKGROUND

### 2.1 High-Level Synthesis Methodologies

Historically two basic approaches have been used for scheduling and binding: Heuristics algorithms and Integer Linear Programming. On the one hand, priority-based heuristic methods (e.g., [10]) can accommodate a variety of data-dominated and control-dominated behaviors, quickly finding good solutions for large problems. They can also take into consideration some loose binding information. On the other hand, they may fail to find an optimal solution in tightly constrained problems, where early pruning decisions may exclude candidates eventually leading to superior solutions. Integer Linear Programming methods (e.g., [7]) can solve scheduling exactly. However, the ILP complexity significantly increases by considering control constraints (if-then-else and loops) and binding information, and thus can lead to unacceptable execution times. Moreover, they consider only one solution at a time, and hence are not particularly suitable for interactive synthesis.

### 2.2 Symbolic Scheduling

More recently [2, 3, 4, 11] symbolic methods have been proved effective in finding exact solutions in highly constrained problem formulations. In these formulations scheduling constraints are represented as Boolean functions, and all solutions are implicitly enumerated. Post-process pruning can be used to apply additional constraints which may not have efficient formulation to be considered within the previous approaches. Moreover, symbolic methods yield a very efficient formulation of control dependencies and environmental timing constraints.

In [11], the authors propose a symbolic formulation that allows speculative operation execution and exact resource-constrained scheduling. In [2, 3], the authors improved the previous method by proposing a new efficient encoding to reduce execution time. This encoding only indicates “whether or not” and not “when” an operation has been scheduled. Finally, [4] handles loops in Data Flow Graph (DFGs).

Their scheduling technique (as well as ours) assumes an input in the form of a CDFG. A CDFG is a directed acyclic graph<sup>2</sup> describing both data-flow and control dependencies between the operations. Operation nodes are atomic actions potentially requiring the use of hardware resources for one or more clock cycles. Directed

<sup>2</sup>We currently model cycles by arbitrarily breaking them and imposing the same binding to data dependencies that have been cut. A better formulation, considering also inter-iteration optimization such as unrolling and pipelining [8], is left to future work.

arcs establish a link between each operation and the predecessors that produce data required by it. A source and a sink are added before every operation without predecessors and after every operation without successors. Conditional behavior is specified by means of fork and join nodes, and directed arcs also establish a link between the operation evaluating the condition and the related fork/join pair. Operations that are neither connected by a directed path, nor mutually exclusive due to a preceding fork node, are concurrent<sup>3</sup>.

EXAMPLE 1. Figure 1 shows an example of CDFG. In particular Figure 1(a) shows the pseudo-code for a conditional statement and Figure 1(b) the relative CDFG.

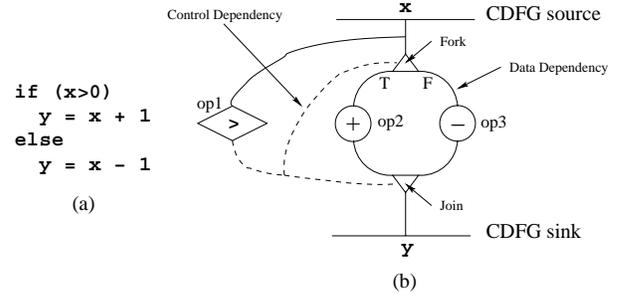


Figure 1: An example of CDFG.

### 2.3 Scheduling Automata

A scheduling problem can be represented by an automaton, defined by the four-tuple  $(V, \delta, S_i, S_f)$ , where  $V$  is the finite, non-empty set of states,  $\delta: V \rightarrow V'$  is the next-state function, and  $S_i$  and  $S_f$  are respectively the sets of initial and final states.

Each operation  $i$  in the CDFG (excluding fork and join operations) is modeled by a two-state automaton, as shown in Figure 2(a). State 0 means that the result of the operation is not available, while state 1 means that it is available.

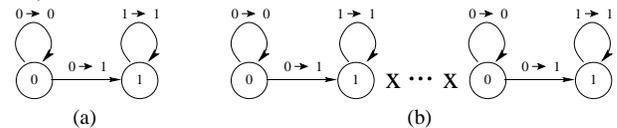


Figure 2: Scheduling automata.

This formulation allows standard symbolic reachability analysis techniques to be employed to determine the exact valid sets of schedules. Present states are described by a vector of  $p$  variables, while a vector of  $n$  variables is used for next states. The characteristic function of a set of states  $S \subseteq V$  is expressed as  $\chi_S(p)$ . With a slight abuse of notation, in the rest of this paper we make no distinction among a set of states, its characteristic function, and its BDD representation. As a consequence, we will thus use  $S(p)$  to represent the corresponding characteristic function  $\chi_S(p)$ .

The transition relation of the  $i$ -th operation is encoded with exactly two Boolean variables ( $p_i$  and  $n_i$ ), as follows:

- $p_i = 0 \Rightarrow n_i = 0$ : operation  $i$  has not been scheduled previously and will not be scheduled in the next cycle.
- $p_i = 0 \Rightarrow n_i = 1$ : operation  $i$  has not been scheduled previously and will be scheduled in the next cycle.
- $p_i = 1 \Rightarrow n_i = 0$ : operation  $i$  has been scheduled previously but the result will no longer be available in the next cycle (because the register holding it has been re-used); this is forbidden in [3],

<sup>3</sup>The same model, if the sink is connected back to the source, can also be viewed as a safe Petri Net. In this paper we use the automata-based notation for consistency with [11].

as well as in our solution, in order to reduce the BDD representation for the sake of efficiency. As a consequence a register cannot be re-used within a scheduling trace. Notice that, however, this possibility is already analyzed in [2], and the methodology is easily applicable to our case.

- $p_i = 1 \Rightarrow n_i = 1$ : operation  $i$  has been scheduled previously and the result remains available.

The complete scheduling is the Cartesian product of the automata (Figure 2(b)), restricted by several constraints. We briefly summarize here dependency and resource constraints, since they will be used in the sequel:

- Data dependencies impose an ordering on operation execution; the automaton modeling an operation is allowed to make the  $0 \Rightarrow 1$  transition only after all those producing values for it have made the same transition, i.e., it is illegal to schedule an operation with a predecessor that has not yet been scheduled:

$$\overline{p_i}n_j \text{ is illegal for all } i \rightarrow j \text{ data dependencies}$$

- Resource constraints limit the number of automata that can make the  $0 \Rightarrow 1$  at a given clock cycle. Given a resource set with  $l$  resources of a given kind (e.g., multipliers) available, and the set  $\rho$  of operations competing for such a resource, it is illegal to schedule more than  $l$  operations from  $\rho$  in a single cycle.

$$(\overline{p_i}n_i \dots \overline{p_k}n_k) \text{ with } \{i..k\} \in \rho \text{ is illegal if } |\{i..k\}| > l$$

Let  $S_0(p)$  be the initial state of the scheduling product automaton, in which no operation has been scheduled. The set of reachable states on the  $i$ -th clock cycle may be computed from the starting point by a standard iterative image computation:

$$S_i(n) = \text{Img}(\delta, S_{i-1}) = \exists_p [S_{i-1}(p) \cdot \delta(p, n)] \quad (1)$$

Valid schedules are represented by state paths that reach a final set of states in which terminal operations have been scheduled, with some additional validity criteria (that will be described more formally in Section 3).

Speculative execution may allow some operations after a fork and before a join to be scheduled before the condition evaluation has been scheduled. However, the condition must be scheduled before the join operation may occur. Moreover, for each possible combination of condition results, all the corresponding operations must be executed in order to complete the schedule.

### 3. COMBINING SCHEDULING AND ALLOCATION

The method of [3] can find all the minimum latency schedules with given resource limits. All allowed schedules are implicitly represented in terms of BDDs as a result of a symbolic traversal process. Nevertheless, the proposed technique is not able to seek for optimal allocations within the bounds.

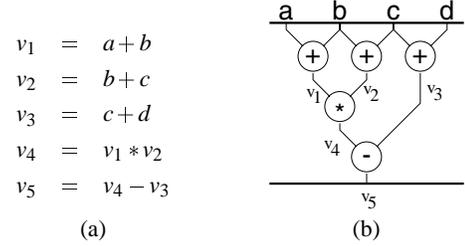
Our method follows [3] to find a symbolic representation of all minimal latency schedules allowed by a given set of resources. Furthermore, each schedule is (symbolically) associated with all valid subsets of allocated resources, so that the combined space can be explored for best allocation purposes. This is achieved by encoding all possible allocations of resources within the given limits. The extra information keeps track of allocations within schedule automaton traversal, and it is finally used to select a schedule with optimal allocation (possibly using less resources than provided by bounds) for a given latency.

Our approach considers the information whether the output of a scheduled operation is used *immediately* or *later*, implying a register in the latter case. A register is required whenever an intermediate result is produced and used in different cycles. A direct

connection, without register, is allowed between predecessor and a successor in the CDFG, provided that the two operations are executed in the same cycle and their combined delays are lower than a specified upper bound. We model this constraint as an additional pruning constraint for the transitions in the product automaton.

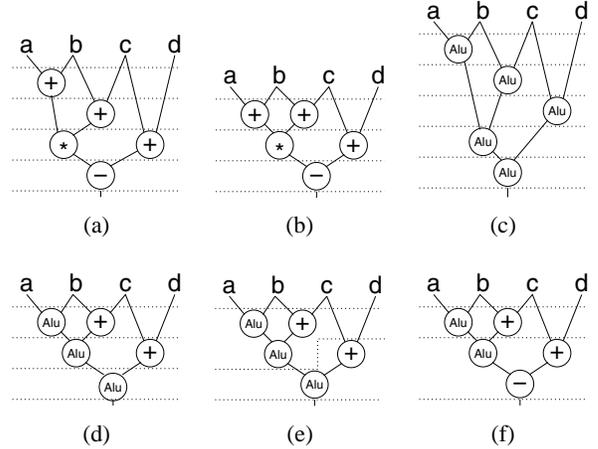
The designer provides the input CDFG, as well as a set of functional units that can implement the CDFG operations, and a bound on the maximum number of registers. Each operation, e.g., an addition, may be implemented by several chosen functional units, e.g., an ADD/SUB or an ALU, with different delay, area and power.

**EXAMPLE 2.** Let us suppose to have the pseudo-code of Figure 3(a), and the corresponding DFG of Figure 3(b). Several scheduling solutions can be found for it, depending on the type and number of operation resources allocated, and on the choice regarding the position of the registers.



**Figure 3: Pseudo-code and corresponding DFG.**

Figure 4 and Table 1 show some of the possible scheduling instances, with different combinational resource and register allocations. In particular, solutions (a) to (d) allocate exactly one register for each combinational operation, whereas solutions (e) and (f) allow combinational propagation of data thus requiring less registers and cycles (traded off by a possibly longer cycle time).



**Figure 4: Scheduling solutions for the proposed DFG.**

As far as resources type is concerned, we consider combinational multipliers and ALUs with the multiply operation. Each row of Figure 4 is executed in a different cycle. Registers are allocated on edges connecting different rows.

Our approach targets both combinational resource and register minimization. In particular, we keep track of every possible allocation of combinational resources while symbolically computing a set of schedules. This allows us to be able to finally select the best allocation, given a table of resource costs (e.g., in terms of area or power). Regarding registers allocation, we accept as constraints the maximum number of register available and the upper bound on combinational propagation delay. We compute, if there exists one,



### 3.1.1 Partial Encoding of Allocated Resources

The technique we propose has an additional cost compared with the original method of [3]. In fact, the resource constraints of [3] can be derived from equation 2 by replacing the  $p_{R_c}$  variables with the (constant) resource bound  $l_{R_c}$ :

$$\text{Illegal}_{R_c}(p, n) = \sum_{\{i..k\} \in \rho_c} (\overline{p_i}n_i \cdot \dots \cdot \overline{p_k}n_k) \cdot (l_{R_c} < |\{i..k\}|)$$

The experimental results section shows a comparison between the two solutions. In particular, it comes out that the full encoding within the allocation space may have a relevant impact on memory and time performance. But this allows an exact search of schedules with minimal allocation. Whenever the additional cost is too high, a sequence of partial explorations of the allocation space may still converge to a nearly optimal solution, at a lower cost. We call this *Partial Encoding* of allocation resources.

An example of such intermediate approach is to encode allocated resources for a given class  $R_c$  only above a lower threshold  $th_{R_c}$ , while associating no allocation encoding for allocations above  $th_{R_c}$ . For instance, one could fully encode all sets on allocated resources with  $5 \leq |R_c| \leq 8$ , while providing no encoding for smaller allocations ( $|R_c| \leq 4$ ). This would obviously allow finding an optimal allocation in the range from 5 to 8, and require a further exploration to look for a solution in lower ranges. The overall process would imply a sequence of scheduling/allocation problems, possibly converging to a final optimal (or sub-optimal) solution.

## 3.2 Register allocation

The target of our register allocation policy is to maximize combinational connections with an allowed propagation delay, so that we possibly avoid the registers to latch the results of some operations. As a motivation for this work, it is worth noticing that the cost of a register (especially in terms of area) is comparable with that of combinational resources like adders and comparators (see, for example, Table 2, Section 4).

We now accept an operation to be scheduled on different cycles without latching its result. We modify the meaning of the operation encoding proposed by [3]. In particular, state 1 for the  $i$ -th operation means that the result of the operation is latched in a register, whereas a combinational operation is possible even in state 0, if a successor requires it. Of course maximum combinational delays must be checked.

In order to support the above encoding, we update the data dependency constraints and the way we account combinational resource usage. We first define the activity of an operation to allow combinational propagation of the result to successors:

$$\text{Active}_i(p, n) = \overline{p_i}n_i + \overline{n_i} \sum_{i \rightarrow j} \text{Active}_j(p, n)$$

A combinational operation  $i$  is active (and allocates a resource) when a  $0 \Rightarrow 1$  transition is scheduled (and the result is latched in a register) or the next state is 0 (no latching) and a successor  $j$  is active (it requires the result of  $i$  through a combinational path).

Data dependencies are replaced by proper checks on combinational propagation delays. Whenever an operation  $j$  is active, it is illegal that one of its operands comes from a combinational path with invalid propagation delay. Let delay $_i$  be the combinational delay of operation  $i$ , let  $D_M$  be the limit for propagation delays on a data path, and let  $\rightarrow j$  be the set of predecessors of  $j$  on a data dependency path. Then

$$\text{Active}_j(p, n) \cdot \left( \sum_{i \in \rightarrow j} (\text{delay}_i) > D_M \right) \cdot \prod_{i \in \rightarrow j} \overline{p_i}$$

is illegal.

Resource allocation encodings are also modified, to take into account the combinational activity of an operation. The  $\overline{p_i}n_i \cdot \dots \cdot \overline{p_k}n_k$  terms in equation 2 are changed to  $\text{Active}_i(p, n) \cdot \dots \cdot \text{Active}_k(p, n)$ . It is worth noticing that an operation is now allowed to be *active* on multiple cycles, but its result may be latched (if required) only once.

As a consequence of the new encoding we chose, a scheduling automaton state now has a  $p_i = 1$  for every operation  $i$  requiring a register. The number of registers required by a schedule path is equal to the number of  $p_i = 1$  bits in the terminal state. The terminal state set thus implicitly contains all required informations to know the register usage of a given set of schedules. More specifically, if we allow register sharing among mutually exclusive schedule traces, an allocation for the number of registers may be checked by simply filtering out states requiring too many registers.

The conclusion is that we are able to find a schedule (if there exist one) with best allocation cost for combinational resources, given a maximum allowed combinational delay, and a limit on the number of registers. And we are able to symbolically select it among *all possible register locations in the CDFG*. The only constraint that we currently assume is to always require latches on operations conditioning a fork/join and on terminal operations (i.e., those that affect the externally visible state of the CDFG).

Circuit	Area Cost	Power Cost
ADD	1.00	1.00
ALU	2.56	2.26
COMPARATOR	0.58	0.56
MUL	8.35	9.70
REGISTER	0.40	0.09
SUB	1.03	1.02
UnaryMINUS	0.42	0.25

**Table 2: Area and Power and costs for the benchmark analyzed.**

Circuit	# Operations	# Conditions
oven_control	12	3
maha	14	6
rotor	35	3
ewf1x1	26	0
ewf1x3	78	0
fdct1x1	42	0

**Table 3: Circuit Complexity in terms of Number of Operations and Conditions Checked.**

## 4. EXPERIMENTAL RESULTS

We show experimental results on the following set of DFG and CDFG benchmarks:

- Elliptic Wave Filter (EWF); **ewf-1** is the standard 34-operation single iteration filter; **ewf-n** is a sequence of  $n$  unrolled iterations of the filter. **ewf-nxm** is the parallel execution of  $m$  copies of the filter, each unrolled  $n$ -times.
- Discrete Cosine Transform (FDCT); **fdct-nxm** follows the same notation as **ewf-nxm**.
- **oven\_control** and **maha** are taken from [9].
- **rotor** is from [11]. It performs a rotation of coordinates.

All the benchmarks are manually translated from the original VHDL sources found in [5].

We present in Table 2 some statistics regarding costs in terms of area, power and time of the cells used to implement our benchmarks. Data are collected starting from a VHDL description of the circuit and using the Synopsys Design Compiler [6] with the AMS

Circuit	Cycle #	Resource # Bound	# Register	Mem. [Mb]	CPU Time
oven_control	8	4+,4C,4-	12	4.6	0.1
maha	7	4+,4C,4-	14	5.4	0.5
rotor	7	4+,4C,4-,4*,4u	35	9.1	5.0
ewf1x1	13	8+	26	4.8	0.3
ewf1x3	28	8+	78	8.5	7.5
fdct1x1	10	4+,4-,4*	42	5.5	0.8

**Table 4: Schedule Results.** Terminology for columns Resource #: ADD=+, ALU=A, COMPARATOR=C, SUB=-, MUL=\*, UnaryMINUS=u.

Circuit	Cycle #	Resource #		Register #	Mem. [Mb]	CPU Time
		Bound	Best Allocation			
oven_control	8	4+,4C,4-	2+,2C,1-	12	4.7	0.2
maha	7	4+,4C,4-	2+,1C,2-	13	6.3	1.0
rotor	7	4+,4C,4-,4*,4u	1+,1C,1-,2*,1u	12	12.4	10.5
ewf1x1	13	8+	4+	26	5.1	0.6
ewf1x3	28	8+	3+	78	14.5	79.0
		8+ (2 partial)	3+	78	13.5	45.0
fdct1x1	10	4+,4-,4*	2+,2-,3*	42	14.0	16.5
		4+,4-,4* (2 partial)	2+,2-,3*	42	6.0	3.0

**Table 5: Schedule Results.** Terminology for columns # Resources: ADD=+, ALU=A, COMPARATOR=C, SUB=-, MUL=\*, UnaryMINUS=u.

library. All variables are 16 bit wide, and we present data on logic only, excluding interconnect costs. All data are normalized with respect to the costs of the adder.

Table 3 shows the complexity of the benchmark set in terms of number of operations, and number of conditions checked. The CD-FGs and DFGs are similar but not identical to the ones presented in [11, 3], and this explains some differences (see Tables 4 and 5) in terms of number of cycles and resources allocated.

We ran our experiments on a 500MHz Pentium III with 256MB of main memory.

Table 4 and 5 compares the results obtained without and with allocation encoding. The experiments in Table 4 are run with an algorithm equivalent to the one presented in [3]. As a consequence, in this case, we do not find the schedule with the best allocation, but we just check the resource bounds. Table 5 shows the same experiments with allocation encoding and search for the best allocation.

For each CDFG we first present the latency of the final schedule (# Cycles), the resource bound and the best allocation (5 only). The number of registers, total memory usage and CPU time for symbolic exploration (including BDD encoding). Notice that in Table 4 the number of register is equal to the number of operations performed, whereas in Table 5 the number of register is usually smaller. To this respect we have to remember that the reduction would be larger if we allowed re-using registers in different cycles, as introduced in [2].

Two of the experiments in table 5 (*ewf1x3* and *fdct1x1*) are run as a sequence of 2 partial allocation sub-problems (see Section 3.1.1) in order to show the lower costs, compared with the previous full ending case.

Overall, all experiments show that the problems are tractable, with an acceptable performance loss, traded off by the ability to find best allocations.

## 5. CONCLUSIONS AND FUTURE WORK

We present a new approach for an integrated symbolic scheduling and resource allocation. The method proposed starts from a state-of-the-art symbolic scheduling technique, and extends it to target both combinational resource and register minimization. As a by-product, it allows trading off latency with cycle time, since reg-

ister optimization is based on allowing combinational connections.

Experimental results on benchmark CDFGs show that our solution is feasible with acceptable performance loss, compared with the improvements proposed.

## 6. REFERENCES

- [1] F. Balarin, E. Sentovich, M. Chiodo, P. Giusto, H. Hsieh, B. Tabbara, A. Jurecska, L. Lavagno, C. Passerone, K. Suzuki, and A. Sangiovanni-Vincentelli. *Hardware-Software Co-design of Embedded Systems – The POLIS Approach*. Kluwer Academic Publishers, 1997.
- [2] S. Haynal. *Automata-Based Symbolic Scheduling*. PhD thesis, University of California Santa Barbara, Dec. 2000.
- [3] S. Haynal and F. Brewer. Efficient Encoding for Exact Symbolic Automata-Based Scheduling. In *Proc. IEEE ICCAD'98*, pages 477–481, San Jose, California, Nov. 1998.
- [4] S. Haynal and F. Brewer. Automata-Based Scheduling for Looping DFGs. *Internal Report EC99\_14*, Oct. 1999.
- [5] <http://ftp.ics.uci.edu/pub/hlsynth/{HLSynth92,HLSynth95}>.
- [6] [http://www.synopsys.com/products/logic/design\\_compiler.html](http://www.synopsys.com/products/logic/design_compiler.html).
- [7] C. T. Hwang, J. H. Lee, and Y. C. Hsu. A Formal Approach to the Scheduling Problem in High-Level Synthesis. *IEEE Trans. on Computer-Aided Design*, 10:464–475, Apr. 1991.
- [8] K. Khouri, G. Lakshminarayana, and N. Jha. High-level synthesis of low-power control-flow intensive circuits. *IEEE Trans. on Computer-Aided Design*, 18(12):1715–1729, Dec. 1999.
- [9] A. C. Parker, J. T. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proc. IEEE/ACM ICCAD'91*, pages 461–466, Las Vegas, June 1986.
- [10] P. G. Paulin and J. P. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASICs. *IEEE Trans. on Computer-Aided Design*, 8:661–679, June 1989.
- [11] I. Radivojevic and F. Brewer. A New Sybolic Techniquer for Control-Dependent Scheduling. *IEEE Trans. on Computer-Aided Design*, C-15(1):45–57, Jan. 1996.