POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Symbolic Approach for the Combined Solution of Scheduling and Allocation

Original
A Symbolic Approach for the Combined Solution of Scheduling and Allocation / Cabodi, G., Lavagno, L., Lazarescu, M.T., Nocco, S., Passerone, C., Quer, S.. - ELETTRONICO. - (2002), pp. 237-242. (ISSS'02: ACM/IEEE International Symposium of System Synthesis Kyoto, Japan 2-4 ottobre 2002) [10.1145/581199.581252].

Availability:
This version is available at: 11583/1500761 since: 2018-10-30T15:36:57Z

Publisher:
ACM

Published
DOI:10.1145/581199.581252

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
ACM postprint/Author's Accepted Manuscript

(Article begins on next page)

16 October 2026

A Symbolic Approach for the Combined Solution of Scheduling and Allocation

Gianpiero Cabodi[†]
Sergio Nocco[†]

[†]Politecnico di Torino
Dip. di Automatica e Informatica
Turin, ITALY

Mihai Lazarescu[◦]
Claudio Passerone[‡]

[‡] Politecnico di Torino
Dip. di Elettronica
Turin, ITALY

Luciano Lavagno[‡]
Stefano Quer[†]

[◦]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 integrate resource allocation and operation scheduling concurrently. The technique inherits all of the features of BDD-based control dominated scheduling, including resource-constraining, speculation, pruning, etc. It proposes an efficient way of encoding allocation information within a symbolic scheduling automaton with a two-folded target: find a minimum cost allocation of operation resources satisfying a given schedule, and optimize the amount of registers required to store intermediate results of operations.

1. INTRODUCTION

Recent technology advances push the performance of current electronic systems to new heights. However, the design practice does not allow an increase of the productivity of designers to take full advantage of the available performance; this is particularly true in the embedded system world, where product must be designed in very short time.

These 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 [2].

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 20 years of intensive research [12, 4], and even the appearance on the market of some industrial CAD tools [11], 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 CDFG 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[1].

Although we use CDFGs as the input specification for our tool, we therefore decided to extend the model introduced by [6], that is at the same time *formal* (based on concurrent automata), *efficient* (we can use symbolic representation techniques [3] with enhancements derived from concurrent specification models), *control-oriented* (condition evaluation and speculative execution were specific features of [6]), and *flexible* (we can represent I/O constraints 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, hence several approaches have tried to combine two or more steps. However, since any of these problems is NP-complete by itself, the combination generally requires the use of heuristics, that may thus forfeit expected improvements with respect to better and more complex algorithms applied in succession.

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 [6] did for 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 [6] 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.

[1]Only recently approaches such as [1, 10] that specifically address control-intensive CDFGs have been introduced.

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 computed symbolically, the extra information we encode allows us to find a schedule with best allocation cost, whereas in [6] a schedule is only checked to fit within the given resource bounds.

In the sequel, we will briefly overview (Section 2) some preliminary concepts, with a particular focus on the related works on symbolic scheduling. We will then introduce our combined scheduling/allocation method (Section 3), and we will finally present some experimental results attained with a prototype implementation (Section 4).

2. BACKGROUND

2.1 High-level synthesis

Historically two basic approaches have been used for scheduling and binding: heuristics and Integer Linear Programming. Priority-based heuristic methods (e.g., [15]) can accommodate a variety of data-dominated and control-dominated behaviors, quickly finding good solutions for large problems, and can consider also some binding information. On the other hand they may fail to find an optimal solution in tightly constrained problems, where early pruning decisions exclude candidates leading to superior solutions. Integer Linear Programming methods (e.g., [9]) 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 [16, 13, 6, 7, 5] 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 for the previous approaches. Moreover, symbolic methods yield a very efficient formulation of control dependencies and environmental timing constraints.

[16] proposed a symbolic formulation that allows speculative operation execution and exact resource-constrained scheduling. [6, 5] improved the previous method by proposing a new efficient encoding (which only indicates "whether or not" and not "when" an operation has been scheduled) to improve execution time and [7] handles loops in DFGs.

Their scheduling technique (as well as ours) assumes an input in the form of a Control Data Flow Graph (CDFG). A CDFG is a directed acyclic graph[2] describing both data-flow and control dependencies between the operations. Operation nodes are atomic actions potentially requiring use of hardware resources for one or more clock cycles. Directed 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[3]. Figure 1 shows an example of CDFG.

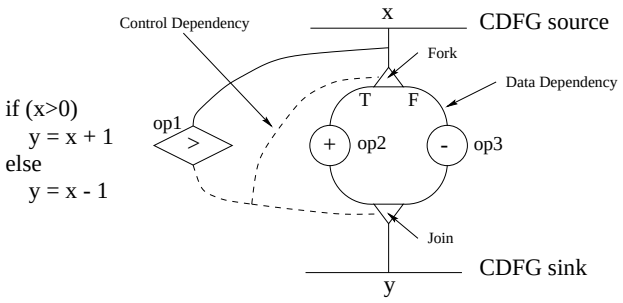


Figure 1: An example CDFG.

2.3 Scheduling Automata

A scheduling problem can be represented by an automaton, defined by the four-tuple (V, δ, 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 sets of initial and final states respectively.

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.

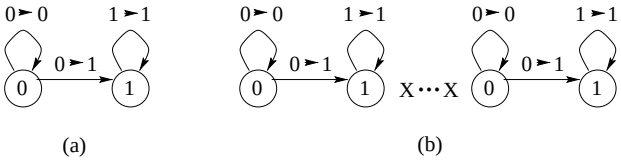


Figure 2: Scheduling automata.

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 liberally identify a set of states, its characteristic function, and its BDD representation. We will thus use $S(p)$ for $\chi_S(p)$.

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 [6], 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,
- $p_i = 1 \Rightarrow n_i = 1$: 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

[2]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 [10], is left to future work.

[3]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 [16].

here dependency and resource constraints, since they will be used in the se**q**uel:

- 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 ρ of operations competing for such a resource, it is illegal to schedule more than l operations from ρ in a single cycle.

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

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 the iterative image computation:

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

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. OUR COMBINED SCHEDULING/ALLOCATION APPROACH

The method of [6] 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. But the proposed technique is not able to seek for optimal allocations within the bounds.

Our method finds a symbolic representation of all minimal latency schedules allowed by a given set of resources (as in [6]). Furthermore, each schedule is (symbolically) associated with all valid subsets of allocated resources, so that the combined allocation-scheduling 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.

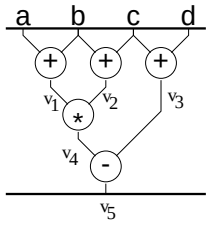
Our approach, on the other hand, considers the information whether the output of a scheduled operation is used *immediately* or *later*, implying a register in the latter case. A register is required whenever an intermediate result is produced and used in different cycles. A direct connection, without 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 1. *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 resources used for each operation (e.g., an ADDER or an ALU for an addition), on the number of resources allocated, and on the choice of where to put registers.*

$$\begin{aligned} v_1 &= a+b \\ v_2 &= b+c \\ v_3 &= c+d \\ v_4 &= v_1 * v_2 \\ v_5 &= v_4 - v_3 \end{aligned}$$

(a)



(b)

Figure 3: Pseudo-code and 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).

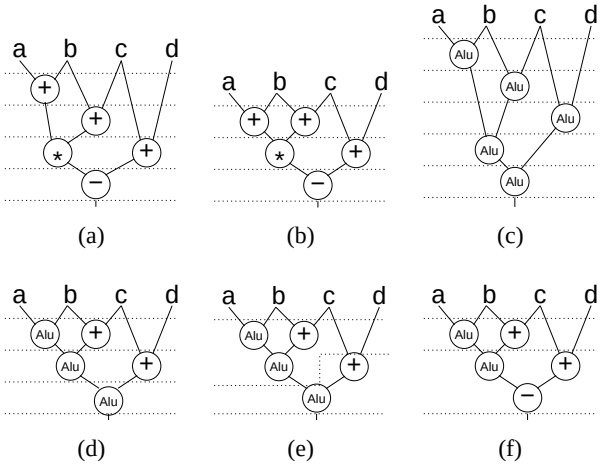


Figure 4: Scheduling solutions for the proposed DFG. Combinational multipliers and ALUs with multiply are considered. Each row is executed in a different cycle. Registers are allocated on edges connecting different rows.

Our approach targets both combinational resource and register minimization. In particular, we keep trace of every possible allocation of combinational resources while symbolically computing a set of schedules, in order to be able to finally select the best one, given a table of costs (e.g., area or power). Regarding registers we accept as constraints a maximum number and an upper bound on combinational propagation delay. We compute, if there exists one, a schedule compatible with the above bounds, which allows register sharing among operations on mutually exclusive schedule traces.

3.1 Accounting for allocation of combinational resources

The first part of our contribution concerns optimal allocation of resources[4].

[4]For sake of simplicity we only consider here combinational resources or sequential resources operating in one cycle. But our method supports also multiple cycle operations, as described in [6].

Figure 4	# Resources				# Cycles	# Registers
	ALU	+	−	×		
(a)	0	1	1	1	4	5
(b)	0	2	1	1	3	5
(c)	1	0	0	0	5	5
(d)	1	1	0	0	3	5
(e)	2	1	0	0	2	1
(f)	1	1	1	0	2	2

Table 1: Resource allocation and latency for the DFG analyzed.

We extend the model of [6] by symbolically encoding the extra information required by the allocation process.

In particular, let $S \in V$ be a state of the schedule automaton, described by its characteristic function $S(p)$. We can associate to it the number of resources allocated for any given resource class R_c, where $c \in 1..N_R$ is the index of the class and N_R is the number of resource classes.

Let us call *allocation instance* a set of allocated resources, divided in classes. For example, $a = (|R_1| = 1, |R_2| = 2, |R_3| = 2)$ is an allocation instance (1 resource is allocated for class R_1, 2 resources each are allocated for classes R_2 and R_3). A schedule solution is compatible with an allocation instance if (due to resource sharing across different cycles) each cycle of the schedule requires at most all the resources in the instance. Let A be the space of all allocation instances. We introduce a set of additional integer variables[5] $p_R = \left\{p_{R_1}, p_{R_2}, \ldots, p_{R_{N_R}}\right\}$, describing the A space. A point in the A space is an allocation instance. A subspace is a set of allocation instances. We are able to express a state in the $V \times A$ space with the set $S_R(p, p_R)$, such that we have a state for any possible instance of allocated resources (see Figure 5).

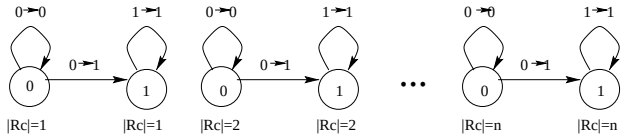


Figure 5: Scheduling automaton with allocation instances.

EXAMPLE 2. *Let us consider 3 resource classes R_1, R_2, and R_3. A scheduler state S(p) with an allocation instance $a = (|R_1| = 1, |R_2| = 2, |R_3| = 2)$, is represented by:*

$$S_{R,a}(p, p_{R_1}, p_{R_2}, p_{R_3}) = S(p) \cdot (p_{R_1} = 1) \cdot (p_{R_2} = 2) \cdot (p_{R_3} = 2)$$

whereas the same scheduler state combined with a set of allocation instances $b = (|R_1| = 1, |R_2| \leq 2, |R_3| \leq 2)$, is expressed as:

$$S_{R,b}(p, p_{R_1}, p_{R_2}, p_{R_3}) = S(p) \cdot (p_{R_1} = 1) \cdot (p_{R_2} \leq 2) \cdot (p_{R_3} \leq 2)$$

Given the p_R variables and an upper bound $|R_c|_{MAX}$ for the resources of the R_c class, the initial state set of the schedule automaton is:

$$s_{i,R}(p, p_R) = s_i(p) \cdot \prod_{c \in 1..N_R} (p_{R_c} \leq |R_c|_{MAX})$$

where each original state in V is augmented with all legal allocation instances.

Each element of the set expresses a possible allocation within the bounds. The target of our scheduler is to find a schedule with lowest resource cost within a given bound on the number of clock cycles.

By also introducing a set of n_R variables describing the next state space, the transition relation δ is extended to δ_R:

$$\delta_R(p, p_R) = \delta(p) \cdot \prod_{c \in 1..N_R} (p_{R_c} = n_R)$$

[5]Our BDD-based implementation uses a Boolean encoding of integer variables.

where the additional product terms captures the fact that the set of resources allocated is kept constant (albeit underused in some cycles or clock cycles).

A schedule may or may not be compatible with an allocation instance, i.e., with a given amount of available resources. For instance, given the set ρ_c of operations that can be executed by the resources of class R_c, scheduling in one cycle a subset $\{i..k\} \subseteq \rho_c$ of operations is not allowed if the set exceeds the allocated resources ($|\{i..k\}| > |R_c|$). This can be handled through additional constraints on the transition relation, one for each resource class:

$$\delta_R(p, p_R) = \delta(p) \cdot \prod_{c \in 1..N_R} ((p_{R_c} = n_R) \cdot C_{R_c}(p, n, p_{R_c}))$$

A C_{R_c} constraint is true for the transitions allowed by a given allocation for the R_c class. We express it as the complement of illegal transitions:

$$C_{R_c}(p, n, p_{R_c}) = \overline{\text{Illegal}_{R_c}(p, n, p_{R_c})}$$

$$\text{Illegal}_{R_c}(p, n, p_{R_c}) = \sum_{\{i..k\} \in \rho_c} (\overline{p_i} n_i \cdot \ldots \cdot \overline{p_k} n_k) \cdot (p_{R_c} < |\{i..k\}|) \quad (2)$$

We use the δ_R transition relation within a symbolic scheduler based on [6]. The set of schedules obtained after the traversal and validation phases implicitly contains all possible schedules and allocations within given resource bounds.

More specifically validation guarantees that all states in the final set of schedules are characterized by a valid allocation instance, i.e. the state is on a valid scheduling trace from the initial state to termination. In particular, the validated initial state set $s_{i,R,\text{validated}}$ includes all possible allocations for the computed set of schedules.

The selection of a minimum cost allocation is done in two steps. We first extract the maximal set of allocation instances common to all initial states in s_i:

$$\text{Alloc}(p_R) = \forall_p(s_{i,R,\text{validated}}(p, p_R))$$

Then we operate a minterm selection using a weighted sum of the allocation instances. Each resource class c is assigned a weight w_c (e.g., an area or power estimated cost, see, for example, Table 2). The allocation cost of a minterm in the A space is defined:

$$\text{AllocCost}(p_R) = \text{Alloc}(p_R) \cdot \sum_{c \in 1..N_R} (w_c \cdot p_{R_c})$$

We finally choose the minterm that minimizes such a cost function:

$$p_{R,\min} = \text{ArgMin}(\text{AllocCost}(p_R))$$

Scheduling selection then resumes, and a scheduling trace originating from $s_{i,R,\text{validated}}(p, p_{R,\min})$ is selected following the strategy of [6].

The above technique can be used in order to find a minimum area or power allocation and the corresponding schedule within a given latency.

3.1.1 Partial Encoding of Allocated Resources

The technique we propose has an additional cost compared with the original method of [6]. In fact, the resource constraints of [6] can be derived from equation 2 by replacing the p_{R_c} variables with the (constant) resource bound l_{R_c}:

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

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 5..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 [6]. 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:

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

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 $\to j$ be the set of predecessors of j on a data dependency path. Then

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

is illegal.

Resource allocation encodings are also modified, to take into account the combinational activity of an operation. The $\overline{p_i} n_i \cdot \ldots \cdot \overline{p_k} n_k$ terms in equation 2 are changed to $\text{Active}_i(p,n) \cdot \ldots \cdot \text{Active}_k(p,n)$. 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); fdtc-nxm follows the same notation as ewf-nxm.
- oven_control and maha are taken from [14].
- rotor is from [16]. It performs a rotation of coordinates.

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 [8] with the AMS 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 CDFGs and DFGs are similar but not identical to the ones presented in [16, 6], 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. In particular the experiments in table 4 were run with the an algorithm equivalent to the one presented in [6], so it could not find the schedule with best allocation, but it just checked

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

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

the resource bounds. Table 5 shows the same experiments with allocation encoding and search for 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). Two of the experiments in table 5 ($ewf1x3$ and $fdct1x1$) were 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 register 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] V. Allan, R. Jones, R. Lee, and S. Allan. Software pipelining. *ACM Computing Surveys*, 27(3):367–432, Sept. 1995.

[2] 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.

[3] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[4] R. Camposano and W. Wolf, editors. *High-level VLSI synthesis*. Kluwer Academic Publishers, 1991.

[5] S. Haynal. *Automata-Based Symbolic Scheduling*. PhD thesis, University of California Santa Barbara, Dec. 2000.

[6] 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.

[7] S. Haynal and F. Brewer. Automata-Based Scheduling for Looping DFGs. *Internal Report EC99_14*, Oct. 1999.

[8] http://www.synopsys.com/products/logic/design_compiler.html.

[9] C. T. Hwang, J. H. Lee, and Y. C. Hsu. A Formal Approach to the Scheduling Problem in High-Level Synthesis. *IEEE Transactions on CAD*, 10:464–475, Apr. 1991.

[10] K. Khouri, G. Lakkshminarayana, 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.

[11] D. Knapp. *The anatomy of a silicon compiler*.

[12] G. D. Micheli. *Synthesis and optimization of digital circuits*. McGraw-Hill, 1994.

[13] C. Monahan and F. Brewer. Scheduling and Binding Bounds for RT–Level Symbolic Execution. In *Proc. IEEE ICCAD'97*, pages 230–235, San Jose, California, Nov. 1997.

[14] A. C. Parker, J. T. Pizzarro, and M. Mlinar. MAHA: A Program for Datapath Sunthesis. In *Proc. IEEE/ACM ICCAD'91*, pages 461–466, Las Vegas, June 1986.

[15] P. G. Paulin and J. P. Knight. Force–Directed Scheduling for the Behavioral Synthesis od ASICs. *IEEE Transactions on CAD*, 8:661–679, June 1989.

[16] I. Radivojevic and F. Brewer. A New Sybolic Techniquer for Control-Dependent Scheduling. *IEEE Transactions on CAD*, C–15(1):45–57, Jan. 1996.

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
ewf1x3	28	8+ (2 partial)	3+	78	13.5	45.0
fdct1x1	10	4+,4−,4∗	2+,2−,3∗	42	14.0	16.5
fdct1x1	10	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.**