Electronic Notes in Theoretical Computer Science 89 No. 4 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume89.html 16 pages

# A BMC–Formulation for the Scheduling Problem in Highly Constrained Hardware Systems

Gianpiero Cabodi Sergio Nocco Stefano Quer
*Politecnico di Torino*
*Dip. di Automatica e Informatica*
*Turin, ITALY*

Alex Kondratiev Luciano Lavagno Yosinori Watanabe
*Cadence Design Systems, Inc.*
*Berkeley, CA*

**Abstract**

This paper describes a novel application for SAT–based Bounded Model Checking (BMC) within hardware scheduling problems.

First of all, it introduces a new model for control-dependent systems. In this model, alternative executions (producing "tree-like" scheduling traces) are managed as concurrent systems, where alternative behaviors are followed in parallel. This enables standard BMC techniques, producing solutions made up of single paths connecting initial and terminal states.

Secondly, it discusses the main problem arising from the above choice, i.e., rewriting resource bounds, so that they take into account the artificial concurrencies introduced for controlled behaviors.

Thirdly, we exploit SAT-based Bounded Model Checking as a verification technique mostly oriented to bug hunting and counter-example extraction. In order to consider resource constraints, the solutions of modifying the SAT solver or adding extra clauses are both taken into consideration.

Preliminary experimental results, comparing our SAT based approach to state-of-the art BDD-based techniques are eventually presented.

## 1 Introduction

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.

Within this framework, BDD-based manipulations have recently attained interesting results, as an alternative to ILP and heuristic techniques. In this approach a non-deterministic finite automata describes design alternatives for highly-constrained control-dominated models. After that, the automata's state space is symbolically visited, adopting model checking's state-of-the-art techniques. These techniques are mix of forward and backward traversals, aimed at finding a scheduling solution as a trace connecting initial and terminal states.

In the simplest case of systems without control choices (*if-then-else* construct), a schedule is a path, and symbolic scheduling works just like invariant checking with counter-example extraction. However, control-dependent behavior produces scheduling instances as DAGs (or trees), where *fork* and *join* nodes are introduced to represent scheduling choices, depending on values of control operands. This has required a specific backward traversal procedure (called *validation* in [3]), which, albeit not far from standard BDD-based traversals, is not directly mapped to standard Model Checking (e.g., CTL) procedures.

In this work we propose to change the original automaton model introduced in [2,3,4,11] for control-dependent systems, so that standard model checking procedures are supported. More specifically we transform alternative subtraces to concurrent behaviors which are followed in parallel. In this way the resulting scheduling is always a path (instead of a DAG) connecting initial and final states. As a byproduct, we can exploit SAT-based Bounded Model Checking. Indeed, as the designer's aim is to *find* a schedule, not to prove its absence, we believe BMC can work at its best, as a verification technique mostly oriented to bug hunting and counter-example extraction, rather than proof of correctness. Nevertheless, in order to enable this method, we also must re-write the resource bounds, so that they take into account the artificial concurrencies introduced for controlled behaviors.

As a final remark, notice that many High Level Synthesis 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[1]. Although we use CDFGs as the input specification for our tool, we adopt 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

[1] Approaches that specifically address control-intensive CDFGs (such as [8]) have been introduced only recently.

specific features of [3]), and *flexible* (I/O constraints can be represented by restrictions on the automata state space). As [3] we represent implicitly the full solution space by means of the state space of a product of automata.

## 2 Background

We assume the reader is familiar with BDDs, SAT and Bounded Model Checking. As a consequence we briefly review only the basic concepts within our application framework.

### *2.1 High-Level Synthesis Methodologies*

Historically two basic approaches have been used for scheduling: 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. 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 thus may 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 [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 Graphs (DFGs).

Their scheduling technique (as well as ours) assumes an input in the form of a CDFG. A CDFG is a directed acyclic graph 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 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 [2] .

**Example 2.1** 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 corresponding CDFG.

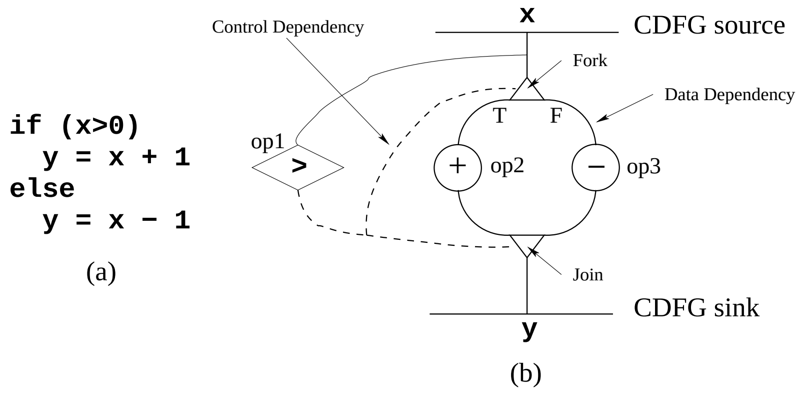

Fig. 1. An example of CDFG.

## *2.3 Scheduling Automata*

A scheduling problem, originally described as a CDFG, can be translated into an automaton, defined by the four-tuple $(V, \mathsf{TR}, S_i, S_f)$, where $V$ is the finite, non-empty set of states, $\mathsf{TR} : V \rightarrow V'$ is the transition relation, and $S_i$ and $S_f$ are respectively the sets of initial and final states.

The generic $i$-th operation in the CDFG (excluding fork and join operations) is modeled by a two-state automaton. Its transition relation is encoded with exactly two Boolean variables ($p_i$ for the present state and $n_i$ for the next state), with the following meaning:

- $p_i = 0, n_i = 0$: operation $i$ has not been scheduled previously and will not be scheduled in the next cycle.
- $p_i = 0, n_i = 1$: operation $i$ has not been scheduled previously and will be scheduled in the next cycle.
- $p_i = 1, n_i = 0$: operation $i$ has been scheduled previously but the result will no longer be available in the next cycle; this is forbidden in [3], as well as in our solution, in order to reduce the amount of equivalent schedules generated.
- $p_i = 1, n_i = 1$: operation $i$ has been scheduled previously and the result remains available.

The complete scheduling is the Cartesian product of the above automata restricted by several constraints, each one representing a particular allowed be-

[2] 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].

havior.

$$\mathsf{TR}(p,n) = \prod_i (\overline{p_i} + n_i) \cdot \mathsf{TR}_{dep}(p,n) \cdot \mathsf{TR}_{res}(p,n)$$

The modeling automaton described by $\mathsf{TR}$ encapsulates all legal execution sequences of a system. Fundamentally, it represents multiple legal execution sequences via nondeterministic choices, yet a real implementation must make deterministic choices. If nondeterministic choices are pruned to leave only one deterministic choice, or if multiple choices are made deterministic by conditions, then a finite state machine controller may be directly synthesized. The criterion used to eliminate nondeterminism is usually minimum execution latency. Variations of this exist for control-dependent behavior, where some control cases might be more favored than others.

Let us briefly summarize here dependency and resource constraints, since they will be used in the sequel:

- $\mathsf{TR}_{dep}$ represents data dependencies, i.e., it is illegal to schedule an operation with a predecessor that has not yet been scheduled:

$$\overline{p_i} n_j \text{ is illegal for all } i \to j \text{ data dependencies (dd)}$$

$$\mathsf{TR}_{dep}(p,n) = \prod_{i \to j \in \mathsf{dd}} (p_i + \overline{n_j})$$

- $\mathsf{TR}_{res}$ represents resource constraints. Let us have a resource set with $b$ resources of a given kind (e.g., multipliers) available, and a set $\rho$ of operations competing for such a resource set. It is illegal to schedule more than $b$ concurrent operations from $\rho$.

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

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

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

Valid schedules are represented by state paths that reach a final set of states in which terminal operations have been scheduled.

The exploration techniques presented here are directed by a minimum latency objective. They determine whether, given all constraints imposed and a target latency $l$, a valid execution sequence of length $\leq l$ exists. With control-dependent models, some additional validity criteria are imposed, and speculative execution may allow some operations after a fork and before a join to be scheduled before the condition evaluation has been scheduled.

## 3 Handling control dependence through concurrency

Unlike the simpler case of data flow graphs, a witness schedule for control-dependent models is not a single path in a path set but rather a set of paths from start to final states. Such a set of paths is called an *ensemble schedule* in [3] and must include a path for each distinct control-dependent execution sequence. For instance, a RISC processor must be able to execute all instructions and therefore an ensemble schedule for a RISC processor contains sequences for every instruction. As a more specific example, consider some control-dependent behavior that branches into two sets of behaviors depending on a true/false control resolution. An ensemble schedule for this example must contain a path from the start state to the final one that represents execution of true control resolution behavior, and another path that covers false control resolution behavior.

In BDD-based formulation, this requires the introduction of control guard variables, representing non-deterministic choices of each controlling operation. A guard is a binary abstraction of the data value controlling a branching condition. A "completeness" check (i.e., all guard values have reached the terminal state) is added to termination conditions.

Furthermore, a validation procedure operates a backward pruning over the state sets computed by forward BFS. Validation is the most expensive symbolic operation and the main cause for BDD blow-up. It consists of a preimage routine with universal quantification of control guards at control resolution points. This is necessary to enforce causality (identical initial sub-path) for outgoing paths at fork points.

Apart from complexity issues, branching schedules and the related validation steps are a major problem for a SAT-based formulation. In order to avoid them, we interpret choice vertices as concurrent forks, and we transform alternative branches into concurrent paths.

So we remove fork and join nodes from the CDFG, and we replace them with unconditioned data dependencies. As a result, a CDFG becomes a DFG, and SAT can explore simultaneously all conditional branches of the original CDFG.

Figure 2 shows the above transformation applied to the example of Figure 1.

Fork and join have been removed, control dependency maintained (as data dependency) for the operations *following the join recombination*. Therefore, in our solution joins work as synchronization points, as no operation following a join is allowed to be executed if both the branches of the control resolution have not been completed yet. This means that our model does not allow control prioritization (as we always have the worst delay), but we have no loss if the objective is minimizing the worst case execution latency. Moreover, since we remove the dependency at forks, speculation is still allowed.

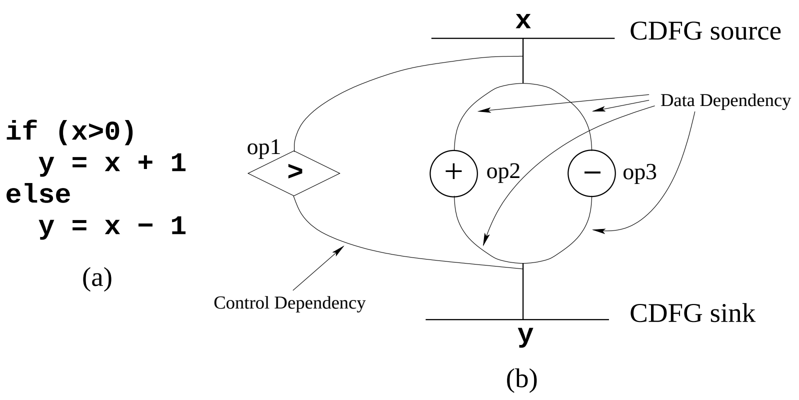

Fig. 2. A CDFG after fork/join removal.

## 4 Concurrent forks and resource constraints

The artificial fork concurrencies we have introduced have no side-effect in the case of scheduling with unbounded resources. In fact, given any set of concurrent operations, if a "large enough" set of resources can be allocated, all operations may always be executed.

The case of bounded resources is less trivial. In this case not all concurrent executions are "real" concurrencies, since some of them are just artificial. As a consequence, not all concurrent operations are competing for resources. As a direct outcome, we need to modify resource constraints, to take into account that some operations could be allocated to the same resource at the same execution time.

More specifically, let us work on a CDFG with a set of $N$ operations $Op = \{op_1, ..., op_N\}$, each one mapped to (i.e., executable by a resource of) a resource class within the set $R = \{r_1, ..., r_M\}$. The generic resource class $r_i$ is characterized by a bound $b_{r_i}$, representing the amount of operation unit available for that class, whereas $n_{r_i}$ is the total number of operations in $Op$ mapped to the $r_i$ resource class. The resource bound problem is obviously trivial for class $r_i$ if $n_{r_i} \leq b_{r_i}$, since there can never be a request of resources greater than the available ones (as for the case of infinite resources).

A much more challenging problem is the case of resource bounds actually reducing the amount of possible concurrencies. Let $op_i$ and $op_j$ be two operations mapped to the same resource class, scheduled for concurrent execution (there is a state transition where $\overline{p_i}n_i\overline{p_j}n_j$ holds). Then, resource allocation may fall in one of the following three cases:

- Unconditioned concurrency. The two operations do not belong to different conditional branches in the original CDFG, so their concurrency is a "real" one, requiring the allocation of two resources.
- Mutual exclusion. The two operations are controlled by mutually exclusive conditions, i.e., they are on different branches of some fork in the original CDFG. Their concurrence is artificial, so just one resource is required.

- Speculative execution. Speculation occurs whenever an operation is executed before its controlling condition is resolved. If $op_i$ and $op_j$ are both executed before their distinguishing condition is known, concurrence is real, and two resources are required.

In other words, we may have couple of operations for which concurrency might be unconditioned (first item in the above list), and other ones characterized by conditioned concurrency (second and third item).

### *4.1 Resource bounds within the SAT solver*

Resource bounds can be accounted for directly by a SAT solver. In this solution the SAT solver has to be properly modified in order to count the allocated resources while recursively building a scheduling solution. This is a special purpose solution to follow only in the case the generated overhead is negligible. It basically relies on identifying active operations through variable decisions and implications, and keeping resource allocation counters. A resource conflict occurs whenever an allocation counter is greater than the allowed bound.

### *4.2 Resource bounds as a Boolean constraint*

Although the above solution is feasible, we prefer exploring an alternative one, that is compatible with a generic SAT solver, since no modification to the SAT algorithm is necessary. We simply follow BDD-based approaches, by generating a resource constraint for the transition relation ($\mathsf{TR}_{res}$), which filters out invalid sets of concurrent executions.

There are various strategies for building such a constraint as a Boolean function returning true on allowed sets of operation executions.

#### *4.2.1 Cliques of concurrency graph*

A straightforward approach works on the graph of possible concurrencies, where operations are nodes and edges connect pairwise concurrent operations. Such a graph can be generated as the transitive closure of a graph where pairs of operations are concurrent if no data dependency connects them and no resolved control makes them mutually exclusive. The graph can be viewed as an upper bound of concurrencies within a schedule. Given the projection of the concurrency graph to resource class $r_i$, cliques of size larger than the allowed bound ($b_{r_i}$) are forbidden.

This is an attractive solution, especially for explicit enumeration, but it is practically limited to small cases, due to its binomial complexity. In particular, it blows up in problems with high degree of concurrency, such as, for example, models of pipelined behaviors.

### *4.2.2 R-combination filtering function*

A more efficient formulation, for the case of unconditioned concurrency, is proposed in [2] for BDD representation. If we omit considering data dependencies, and we simply work on operations of the same resource class, then the resource bound constraint is an r-combination expression, selecting combinations of up to $b_{r_i}$ operations out of $n_{r_i}$. We call this filtering function $\mathsf{Rfilter}(Op, bound)$. Its size complexity, when expressed as a BDD instead of a two level form, is $O(n_{r_i} \cdot b_{r_i})$, i.e., number of operations mapped to the class times the bound for the class. The function is easily translated to CNF format (with intermediate additional variables), with similar complexity.

Unfortunately, as previously shown, we have conditioned (i.e., artificial) concurrencies, that complicate our model compared to [2], and make the above solution exponential in the number of control choices (forks): we should expand one instance of concurrency graph for each case of resolved/unresolved control operation.

### *4.2.3 Hybrid two-level approach*

Since none of the two previous approaches alone is able to efficiently solve our problem, we developed a hybrid technique, which follows the concurrency graph strategy locally, within control components of the CDFG, and the r-combination approach on a global perspective.

More in detail, we express the resource constraint function (for a given resource class $r_i$) as a composition of two sub-functions

$$\mathsf{TR}^{r_i}_{res}(p, n) = \mathsf{Rfilter}(\mathsf{Alloc}(p, n), b_{r_i})$$

The outermost function is (a slight modification of) the previously described r-combination filter, whereas Alloc is a function that remaps operation transitions to a set of allocation variables, with the following rules:

- Each uncontrolled operation $op_i$ is remapped to an allocation variable $a_i = \overline{p_i} n_i$, which evaluates true when the operation is executing.
- Controlled subsets of the CDFG (subgraphs included between fork and join nodes) are globally remapped to a proper set of allocation variables, over whom the Alloc functions returns a number of ones exactly corresponding with the amount of resources required. So all artificial concurrencies and/or speculations are taken into account by this function.

The composition is never computed explicitly, but intermediate allocation variables are kept and transferred to the CNF formulation of $\mathsf{TR}_{res}$, which allows us to face the main size bottlenecks: (1) The complexity of conditional concurrency (function Alloc) is kept within small regions of the CDFG. Especially for the important case of looping and/or pipelined behaviors, modeled by serial and parallel instances of the same reference CDFG, this makes the size of Alloc linear in the number of serial/parallel instances. (2) Rfilter, the function taking care of the overall problem, has size $O(n_{r_i} \cdot b_{r_i})$, i.e., it is linear in the number of operations, for a given resource bound. As an overall result, our result constraint function is scalable, and well suited for looping and pipelined behaviors, which are the most difficult problems in BDD-based approaches.

### *4.2.4 Implementation details*

Figure 3 and 4 show the pseudo-code of the Alloc and Rfilter functions respectively. For sake of simplicity it is assumed that all operations in the CDFG are mapped onto the same resource class, for which maxAlloc units are available.

In our implementation, every operation is labeled with two attributes: (1) the set of all possibly concurrent nodes and (2) a BDD representing the control case for which the operation is enabled. Actually, in order to cover speculation, the meaning of such a BDD is that the operation is *disabled* if the evaluation of the BDD for the already resolved controls returns 0.

As regards the Alloc function, all possible cliques (over the set of operations belonging to the received sub-graph) are recursively built by means of the auxiliary generateCliques function. At each level of recursion, a new node is added to the previously generated clique, checking for speculative execution. In fact, the AND between the node's enable and the clique's enable returns a 0 result only if the current node and at least one node already belonging to the clique are in two different control branches. Therefore, the concurrency of the node w.r.t. the clique is real only if the controlling operations discriminating the branch are not resolved yet (i.e., the nodes in the new clique are executed speculatively). Such controlling operations are therefore added to the *unresolved* set. The transition corresponding to the new clique is then stored as a BDD, and the cliques of bigger sizes are built (the set of possibly concurrent nodes being restricted as the clique has to be completely connected). Eventually, the last loop in the Alloc function defines the allocation variables: variable $a_i$ takes a value of 1 iff there is a transition in the current sub-graph involving the usage of *at least* $i$ resources of the current resource class.

Once all the resource cliques have been generated, the Rfilter function symbolically builds all valid transitions in terms of the allocation variables. To do this, it combines the allocation variables coming from the different calls to the Alloc function to form an expression representing all possible *illegal* allocations (i.e., those requiring at least maxAlloc+1 resources). Then the complementation of such expression, which indeed represents all allocations of at most maxAlloc resources, is returned (and then directly used as a component of TR).

### *4.2.5 A small example*

Let us consider again the CDFG shown in Figure 1 and let us assume that all the operations are mapped on a single ALU. The CDFG is divided by the algorithm into two sub-graphs: the first is composed by the comparison only,

```
ALLOC ($graph$)
   for ($i \leftarrow 1$ TO maxAlloc $+ 1$)
      $tList[i] \leftarrow$ BDD_ZERO
   for ($node \in graph.nodesSet$)
      $cliqueSet \leftarrow \emptyset$
      $unresolved \leftarrow \emptyset$
      $enable \leftarrow$ BDD_ONE
      generateCliques($tList$, $node$, $cliqueSet$, $unresolved$, $enable$, $node.concur$)
   for ($i \leftarrow 1$ TO maxAlloc $+ 1$)
      $graph.a_i \leftarrow$ new_var
      TR $\leftarrow$ BDD_AND(TR, BDD_XNOR($graph.a_i$, $tList[i]$))

GENERATECLIQUES ($tList$, $node$, $cliqueSet$, $unresolved$, $enable$, $concurSet$)
   if $|cliqueSet| >$ maxAlloc
      return
   $newEnable \leftarrow$ BDD_AND_EXIST($enable$, $node.enable$, $unresolved$)
   $newUnresolved \leftarrow unresolved$
   if BDD_IS_ZERO($newEnable$)
      $newUnresolved \leftarrow unresolved \cup$ conflictingControls($enable$, $node.enable$)
      $newEnable \leftarrow$ BDD_AND(BDD_EXIST($enable$, $newUnresolved$),
                                  BDD_EXIST($node.enable$, $newUnresolved$))
   $newClique \leftarrow cliqueSet \cup node$
   $tList[|newClique|] \leftarrow$ BDD_OR($tList[|newClique|]$,
                                  transition($newClique$, $newUnresolved$))
   $newConcur \leftarrow concurSet \cap node.concur$
   for ($op \in newConcur$)
      generateCliques($tList$, $op$, $newClique$, $newUnresolved$, $newEnable$, $newConcur$)
```

Fig. 3. The Alloc function.

```
RFILTER ()
   $allocations[0] \leftarrow$ BDD_ONE
   for ($k \leftarrow 1$ TO maxAlloc $+ 1$)
      $allocations[k] \leftarrow$ BDD_ZERO
   for ($i \leftarrow 1$ TO $N_{graphs}$)
      $newAllocations \leftarrow allocations$
      for ($j \leftarrow 1$ TO maxAlloc $+ 1$)
         for ($k \leftarrow 0$ TO maxAlloc $+ 1$)
            if $j + k >$ maxAlloc $+ 1$
               break
            $alloc \leftarrow$ BDD_AND($allocations[k]$, $graph_i.a_j$)
            $newAllocations[j + k] \leftarrow$ BDD_OR($newAllocations[j + k]$, $alloc$)
      $allocations \leftarrow newAllocations$
   return BDD_NOT($allocations$[maxAlloc $+ 1$])
```

Fig. 4. The Rfilter function.

whereas the second includes both the ADD and SUBTRACT operations. Then the relations defined by the two calls of the Alloc function are respectively:

$$a_1^C = \overline{p_C} n_C, \quad a_2^C = 0$$

and

$$a_1^{AS} = \overline{p_A} n_A + \overline{p_S} n_S, \quad a_2^{AS} = \overline{p_A} n_A \overline{p_S} n_S \overline{p_C}$$

Indeed the first sub-graph may present only a transition requiring 1 ALU unit, whereas the second sub-graph might require 2 ALU units, but only in the case that the ADD and SUBTRACT operations are both executed before the control resolution has been solved. Eventually, the final constraint built by the Rfilter function is:

$$constraint = \overline{a_2^C + a_1^C a_1^{AS} + a_2^{AS}}$$

## 5 BMC Formulation

Once we have generated the transition relation of the CDFG, as previously described, we have to produce the verification problem which will give us the scheduling solution. This is done by unrolling the transition relation a certain number of times and then trying to prove the mutual reachability between initial and final states.

The BDD representing the transition relation (in monolithic or conjunctive form) is stored as a CNF formula as described in [13].

The verification strategy usually starts with a path of length equal to 1 and increases it till the problem is solved or computation resources are exceeded. For the above reasons the technique works well in falsification and partial verification, whereas full verification is usually achieved by BMC with longer and longer bounds.

Our problem is somehow simpler as, with a proper number of registers, there is always a solution to the scheduling problem. Moreover, our experience shows that unsatisfiable problems are much harder to solve than satisfiable instances. To this respect SAT solvers often present an exponential behavior as Figure 5 shows.

For these reasons the standard previously described technique proved to be quite inefficient. On the contrary we do have an estimate of the maximum latency, which is equal to the number of operations in the CDFG. This suggests a second strategy, namely starting from the highest bound and decreasing it in order to find the first unsatisfiable instance. The drawback of this method is that the estimate of the maximum latency can be extremely inaccurate. As a direct consequence, we propose a solution adopting a binary search. Starting with an estimate of the optimal latency, we create the corresponding CNF problem and call the SAT solver giving it a (small) time limit. Accordingly to the result produced by the solver, the estimate of the latency is corrected, and a new bound is tried. Notice that if the SAT solver is unable to solve the CNF problem within the time limit, we consider the instance as unsatisfiable. In general, this might lead to incorrect (i.e., sub-optimal) results, in the sense that a satisfiable instance may be considered as unsatisfiable, but the problem can be solved simply increasing the “unsat” threshold, with an at most linear

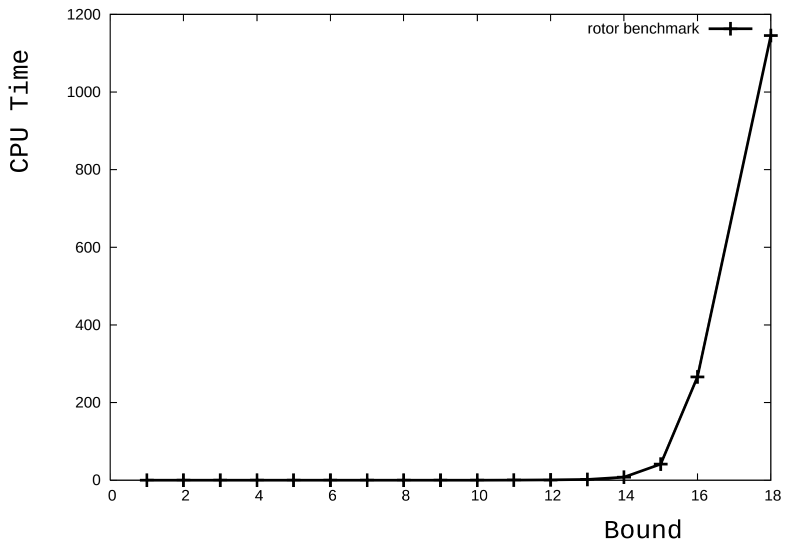

Fig. 5. SAT CPU Time Versus SAT Bound. The data are obtained with a run of the scheduler on the rotor benchmark with two iterates and a resource availability corresponding to the first row of table 2.

loss in performances.

## 6 Experimental Results

We show experimental results on well known benchmarks [2]. Table 1 shows the complexity of the benchmark set in terms of number of operations, and number of conditions checked.

| Circuit | # Operations | # Conditions |
|---|---|---|
| rotor | 28 | 3 |
| s2r | 48 | 6 |
| fdct | 42 | 0 |

Table 1
Circuit Complexity in terms of Number of Operations and Conditions Checked. The data are referred to the acyclic version of the model, i.e., with just one iterate.

We ran our experiments on a 1700 MHz Pentium IV with 1 GByte of main memory. For all the experiments we used BerkMin [12] as SAT engine.

Table 2 summarizes our results. We compare the results obtained with the strategy presented in this paper with the software presented in [2] and locally re-run. More in detail, our data are obtained adopting the binary search (as described in the previous section) with a threshold of two minutes. Notice that all the satisfiable instances were well recognized by the SAT solver; indeed the numbers of scheduled cycles represent the true optimal latencies.

The meaning of columns is the following: # Iterates indicates the number of parallel instances considered (when 1, we refer to the acyclic problem,

| Circuit | # Iterates | # Resources | # Cycles | BDD [2] | | SAT – This Paper | | |
|---|---|---|---|---|---|---|---|---|
| | | | | # BDD [nodes] | Time [s] | # Vars | # Clauses | Time [s] |
| rotor | 1 | 1T,1C,1A | 12 | 74606 | 0.7 | 994 | 4116 | 0.3 |
| | 1 | 1T,1C,2A | 8 | 74606 | 0.7 | 892 | 4232 | 0.3 |
| | 1 | 1T,1C,2A,1∗ | 10 | 84826 | 0.8 | 1142 | 4174 | 0.4 |
| | 1 | 1T,1C,2A,2∗ | 8 | 84826 | 0.7 | 920 | 4152 | 0.3 |
| | 2 | 1T,1C,2A,1∗ | 10 | 871766 | 4.6 | 4749 | 36047 | 10.0 |
| | 2 | 1T,1C,2A,2∗ | 9 | 1447152 | 6.8 | 4392 | 32873 | 10.8 |
| | 2 | 1T,1C,2A,3∗ | 9 | 1864128 | 8.2 | 4428 | 33026 | 8.6 |
| | 2 | 1T,1C,3A,2∗ | 8 | 415954 | 2.7 | 4299 | 31503 | 6.7 |
| | 3 | 1T,1C,2A,1∗ | 12 | 18635148 | 1524.0 | 9573 | 72436 | 118.9 |
| | 3 | 1T,1C,2A,2∗ | 12 | OVF | – | 9861 | 73612 | 308.9 |
| | 3 | 1T,1C,2A,3∗ | 12 | OVF | – | 9957 | 74008 | 288.3 |
| | 3 | 1T,1C,3A,2∗ | 9 | OVF | – | 8229 | 59803 | 37.6 |
| s2r | 1 | 1T,1C,2A,1∗ | 10 | 1006670 | 5.9 | 2532 | 12484 | 1.8 |
| | 1 | 1T,1C,3A,2∗ | 9 | 532462 | 4.1 | 2788 | 14774 | 2.2 |
| | 1 | 1T,–C,2A,2∗ | 8 | 411866 | 3.0 | 4749 | 12832 | 1.7 |
| | 2 | 1T,1C,2A,1∗ | 13 | OVF | – | 12158 | 109623 | 328.6 |
| | 2 | 1T,1C,3A,2∗ | 10 | OVF | – | 10839 | 87817 | 66.8 |
| | 2 | 1T,–C,2A,2∗ | 10 | OVF | – | 10539 | 87137 | 62.9 |
| fdct | 1 | 1+,1−,1∗ | 19 | 306600 | 1.5 | 2775 | 8362 | 133.7 |
| | 1 | 1+,1−,2∗ | 13 | 200312 | 1.2 | 2138 | 6772 | 1.1 |
| | 2 | 1+,1−,1∗ | 32 | – | OVF | 19121 | 223433 | 522.0 |
| | 2 | 1+,1−,2∗ | 26 | – | OVF | 17103 | 188719 | 454.6 |

Table 2

*Schedule Results. Terminology for columns #* Resources*: ADD=+, ALU=A, COMPARATOR=C, SUB=−, MULT=∗, LookUpTable=T. MULT is a two-time steps pipelined multiplier (when not present, multiplications are performed by the ALU). All other resources are single time step. OVF indicates overflow (in terms of memory or CPU time). We use a time limit equal to* 1 *hour and a memory limit equal to* 500 *MBytes.*

otherwise we are handling a looping behavior); column # Resources indicates the number and type of resources allowed; # Cycles is the final solution in term of scheduled cycles. For each experiment we report the data obtained with [2], i.e., the number of BDD nodes and the CPU time required, and with our method (number of variables and clauses generated for the CNF problem corresponding to the solution, and the total CPU time).

Overall, we can make the following observations. For acyclic problems, the times required by the two compared methods are quite similar (with only one exception, the first experiment for fdct). However, when we move to looping behaviors, while the method used in [2] becomes unfeasible, our strategy still produces the optimal result in a limited amount of time. These experiments demonstrate that our solution can be very effective.

## 7 Conclusions and Future Work

We present a new approach for symbolic scheduling based on a new problem formulation and the use of SAT solvers and BMC verification methodology.

Experimental results on DFGs and CDFGs show that our solution can be very effective and competitive with symbolic BDD-based techniques.

Future work will include investigation of better strategies for the CNF problem generation and solution searching.

## 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," 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.

[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. 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.

[9] A. C. Parker, J. T. Pizzarro, and M. Mlinar, "MAHA: A Program for Datapath Synthesis," 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 Symbolic Technique for Control-Dependent Scheduling," IEEE Trans. on Computer-Aided Design, C–15(1):45–57, Jan. 1996.

[12] E. Goldberg and Y. Novikov, "BerkMin: a Fast and Robust SAT-Solver," Proc. IEEE/ACM DATE'02, pages 142–149 Paris, Feb. 2002.

[13] G. Cabodi and S. Nocco and S. Quer, "Improving SAT-based Bounded Model Checking by Means of BDD-based Approximate Traversals," Proc. IEEE/ACM DATE 2003, pages 898–903, Munich, Germany, March 2003.