Constraint Analysis and Heuristic Scheduling Methods

P. Poplavko[1], C.A.J. van Eijk[2], and T. Basten[3]

[1] Eindhoven University of Technology, Department of Electrical Engineering, Eindhoven, The Netherlands
peter@ics.ele.tue.nl
[2] Magma Design Automation, Eindhoven, The Netherlands
koen@magma-da.com
[3] Eindhoven University of Technology, Department of Electrical Engineering, Eindhoven, The Netherlands
a.a.basten@tue.nl

Abstract—**Scheduling is one of the main problems that need to be solved by high-level hardware and software compilers. Existing heuristics are often incapable of finding feasible solutions for practical examples, because the tight time and resource constraints make the feasible-solution subspace very small compared to the size of the full search space. For that reason, constraint-analysis techniques that help the scheduler find feasible solutions are nowadays a subject of research. In this paper, the effect of constraint analysis on heuristic schedulers is experimentally quantified to investigate to which extent constraint analysis improves the quality of such schedulers, and also to see which heuristics complement it well. The results show that, for most experiments, constraint analysis helps to improve the obtained schedules in terms of the latency of the schedule. It combines particularly well with the freeing-count heuristic.**

Keywords—**high-level synthesis; scheduling; heuristics; constraint analysis; topological permutation scheduling**

I. Introduction

SCHEDULING is one of the important problems for high-level hardware and software compilers. High-level synthesis [1] is an important stage in the design of digital signal processors for embedded systems. Given the behavioral description of the system to be designed and a processor architecture description, computer-aided design tools have to schedule operations and map them onto the available resources. Unfortunately, the scheduling problems in high-level synthesis are very hard to solve. The difficulty of scheduling stems from the requirement that a schedule must satisfy certain restrictions, or *constraints*, on performance as well as resource usage. It is relatively easy to achieve only good performance or only low resource usage, but when both are required, it is often very difficult to find a solution; furthermore, the complexity grows drastically with the size of the problem instance. In this paper, the latency-minimization problem under some given set of resource constraints is addressed. The problem statement is defined in more detail in Section II.

Because most practical instances of the scheduling problem are too large to be solved efficiently by exact methods, it is necessary to employ heuristic algorithms. Extensive research has been performed in this area during the last two decades; many heuristic schedulers have been developed, e.g., force-directed schedulers [5], topological permutation schedulers [4] and list schedulers [3], to name a few. Lately, research has also been focused on constraint-analysis techniques. These techniques decrease the apparent scheduling freedom, without removing any feasible schedules. This means that the search space of the scheduler is reduced, while the solution space is left unaltered. Constraint analysis constitutes the basis of the scheduling research tool FACTS [2].

The objective of this work is to quantify the impact of constraint analysis on heuristic scheduling methods. In particular, this paper considers the combination of *topological permutation scheduling* introduced in [4] and the constraint-analysis techniques of FACTS. The advantages of topological permutation scheduling are its simplicity and the great degree of freedom it leaves for choosing the particular heuristic for scheduling. Furthermore, it has been shown in [4] that, for the above mentioned latency-minimization problem, topological permutation scheduling does not restrict the feasible solution space and usually gives reasonably good results. Section III introduces topological permutation scheduling in some more detail, whereas Section IV discusses constraint-analysis techniques.

Topological permutation scheduling lifts the scheduling problem to a higher level of abstraction, namely the level of priority functions. The choice of a priority function determines the particular scheduling heuristic. In Section V, we present a number of well-known priority functions that often lead to good schedules. They were tested on a set of relevant benchmarks. The influence of constraint analysis is quantified to investigate which heuristics complement constraint analysis well. The results are reported in Section VI.

The paper is completed with brief conclusions on the scheduling heuristics discussed in this paper and on the impact of constraint analysis on these heuristics.

II. Scheduling

Scheduling can be defined as a combinatorial optimization or decision problem. In this paper, an instance of the scheduling problem is called a *model*. It contains the *behavioral description* of the system to be designed and the description of the *resources* being used. The system behavior is specified by a *data flow graph*; note that although this behavioral description specifies the *algorithm* to be executed by the system, in this paper, the word ‘algorithm’ is only used to refer to *scheduling* algorithms. The resources are described by a set of *module types*. Constraint information is contained in the graph as well as in the set of module types.

A module type models a functional unit such as an adder, a multiplier, an ALU, or some filter section. Each module type is characterized by the number of available modules, the *instances*, and the number of clock cycles before new input data can be consumed after the last consumption of input data, the *data introduction interval*.

A data flow graph (DFG) is a graph where the set of vertices or nodes V models the operations that have to be carried out by the hardware. Each operation has a module type, specifying the functional unit that can perform that operation. Each operation has a *delay* that specifies the number of clock cycles required by the assigned module type to complete the operation. The set of edges E of a DFG specifies the data dependencies in the model. An edge $(u, v) \in E$ indicates that operation v consumes the result of operation u.

We assume that DFGs are polar. This means that a DFG must have two special nodes namely a source v_{source} and a sink v_{sink} such that every node lies on a path from the source to the sink. The source and sink have zero delay by definition. In this paper, we also consider only acyclic DFGs.

A schedule of a DFG is a function $\phi : V \longrightarrow \mathbb{N}$ that assigns to each operation a starting clock cycle. We assume that clock cycle 0 is assigned to the source node: $\phi(v_{\text{source}}) = 0$.

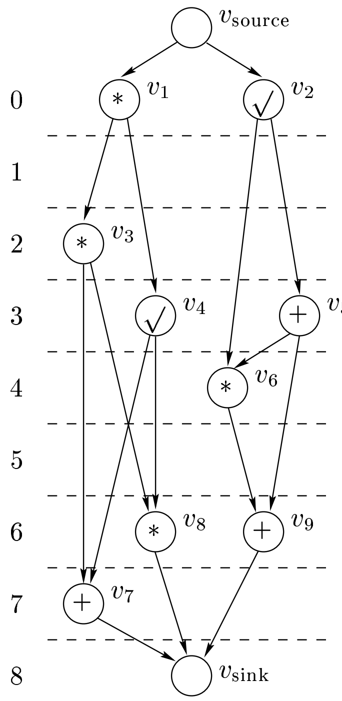

Fig. 1. A scheduled DFG.

Figure 1 shows a scheduled DFG. Every vertex (except for the source and sink) is marked with one of three module types: addition, multiplication, and square root. Clock cycles are shown on the left. The dashed lines separate nodes scheduled in the same clock cycle.

Not every possible schedule is a valid solution to the scheduling problem; it has to satisfy all the *precedence* and *resource* constraints defined by the model. A schedule that satisfies these constraints is called *feasible*. *Precedence constraints* require that no operation start before all its input data is available. If an operation uses a result of some other operation, called a (direct) *predecessor*, it cannot start before the predecessor produces its result. Recall that the edges in a DFG model these data dependencies. Furthermore, recall that for every operation the execution delay is known. The precedence constraints can thus be expressed as follows:

$$\forall (u, v) \in E \;\Rightarrow\; \phi(v) \geq \phi(u) + d(u), \tag{1}$$

where $d(u)$ denotes the delay of operation u.

From the precedence constraints, it follows that no operation can start before the source starts, and that the sink starts when all other operations are completed. So, the total schedule delay in clock cycles is the difference between the start times of sink and source. This delay is called the *latency* of the schedule.

Consider again Figure 1. Assume that the delay of an addition, a multiplication, and a square root is 1, 2, and 3 clock cycles, respectively. The schedule shown in Figure 1 satisfies all precedence constraints, because every data edge crosses at least as many clock-cycle borders as the delay of the operation it originates from. The schedule has latency 8.

Resource constraints model the limitation on the resources available on a chip. A schedule cannot use more modules of a certain type than the number of instances defined in the model.

Consider again Figure 1. Assume that only one instance of every module type is available and that data introduction intervals are such that no module can start a new operation before it completes the previous one. Taking into account only precedence constraints, square-root operation v_4 could have started in clock cycle 2, but square-root operation v_2 has not yet finished in clock cycle 2. That is why operation v_4 has to be postponed until clock cycle 3.

An additional constraint that is often imposed on a feasible schedule is the so-called *latency constraint* which requires that the latency of the schedule is at most some upper bound $T_{\max}$. The *latency-constrained scheduling decision problem* can then be defined as follows: Given a model, find a feasible schedule that satisfies the latency constraint. The latency-constrained decision problem is closely related to the *latency-minimization problem*, where one has to find a feasible schedule with minimal latency.

The latency-constrained scheduling problem is known to be NP-complete ([7]). This fact provides a strong reason to believe that no polynomial-time algorithms exists to solve either the latency-constrained decision problem or the latency-minimization problem. All known exact algorithms for the scheduling problem have exponential running times. For that reason, research has been concentrated on heuristic algorithms, which can find solutions for practical examples in acceptable times.

For simplicity reasons, we do not consider sequence edges and software pipelining in this paper (see [2], for more details).

III. Topological permutation scheduling

In this paper, a class of heuristic scheduling algorithms, called topological permutation schedulers, is used. A topological permutation scheduler schedules operations one by one. When, at some point, the start time of an operation is fixed, the operation is said to be *scheduled*; scheduling decisions are never reconsidered. The order in which the operations are scheduled is determined by a *topologically sorted permutation* of the operations in V.

Let us explain what this means. Recall that we assume that a DFG is acyclic. Thus, it induces a (strict) partial order $\prec$, where $u \prec v$ $(u, v \in V)$ if and only if the DFG has a (non-empty) path from u to v; if $u \prec v$, then u is called a *predecessor* of v and v a *successor* of u. A permutation is topologically sorted if it is consistent with the partial order induced by the graph; that is, if $u \prec v$, then node u must precede v in the permutation. In other words, the permutation respects the data dependencies in the DFG. The requirement that a DFG be acyclic guarantees that there always exist topologically-sorted permutations.

Consider again the DFG of Figure 1. Permutation v_{source}, v_1, v_3, v_4, v_2, v_7, v_5, v_8, v_6, v_9, v_{sink} is topologically sorted. Permutation v_{source}, v_5, v_2, v_6, v_9, v_1, v_4, v_3, v_7, v_8, v_{sink} is not, because v_5 goes before v_2 which is not consistent with edge (v_2, v_5).

As mentioned, a topological permutation scheduler schedules nodes of a DFG one by one. At any point, the nodes whose predecessors have all been scheduled are called *ready* nodes. Consider, for example, Figure 2. The figure shows the DFG of Figure 1 in a state where the four gray nodes have been scheduled. For now, the intervals, the edge annotations, and the dashed arrow can be ignored. The ready nodes are nodes v_3, v_4, and v_6. In each step, a topological permutation scheduler selects one of the ready nodes as the node to be scheduled next. It is clear that such a scheduling order is a topologically-sorted permutation of the nodes of the DFG.

In which particular topological order the nodes are scheduled depends upon the selection scheme used to choose in every step a node from the set of ready nodes. There are two basic selection schemes: *static* and *dynamic* priority functions. A priority function assigns (integer) values to the nodes of a DFG where a smaller value indicates a higher priority. A static priority function is a priority function that can be computed before the actual scheduling starts; a dynamic priority function is a priority function that is computed *during* scheduling. An arbitrary permutation of the nodes in a DFG is a typical static priority function, where the priority of a node is the position in the permutation. In Section V, we give some examples of dynamic priority functions.

In our implementation of topological permutation scheduling, a selected node is scheduled in the earliest clock cycle that satisfies precedence, latency, and resource constraints. To see how the precedence constraints are satisfied, consider all the paths from the source to the selected node. We define the length of such a path as the sum of the delays of all nodes on the path except for the last one, which is always the selected node. The precedence constraints imply that the selected node cannot be scheduled in a clock cycle smaller than the length of any path from the source to the node (see Equation 1). Therefore, to satisfy all precedence constraints, the selected clock cycle cannot be smaller than the length of the longest path from the source to the node. That length is called the *asap* (as soon as possible) value of the node. In Figure 2, the edge weights correspond to operation delays, which makes it easy to compute asap values; the lower bounds of the intervals shown in the figure are the asap values of the corresponding nodes.

To see how a latency constraint $T_{\max}$ is taken into account, consider all the paths from the selected node to the sink of

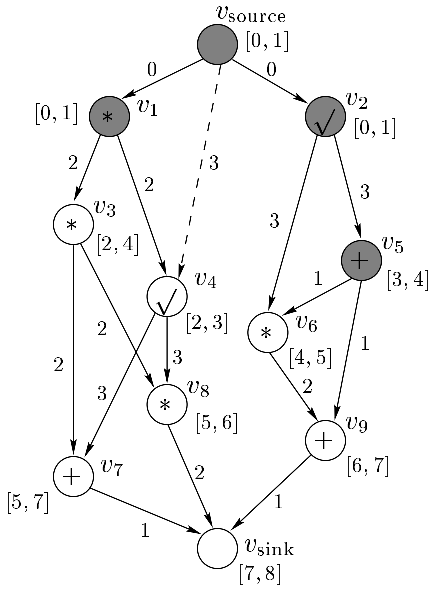

Fig. 2. Topological permutation scheduling in action.

the DFG. The latency constraint means that the sink cannot be scheduled in a clock cycle greater than $T_{\max}$. Thus, Equation 1 implies that the selected node cannot be scheduled in a clock cycle greater than $T_{\max}$ minus the length of any path from the node to the sink. In other words, the node cannot be scheduled later than $T_{\max}$ minus the length of the longest path from the node to the sink. That upper bound is called the *alap* (as late as possible) value of the node. In Figure 2, assuming $T_{\max} = 8$, the upper bounds of the intervals are the alap values of the corresponding nodes.

To satisfy the resource constraints, the scheduling algorithm must know what module type the selected node has. Suppose, for example, that it has type "multiplier" and that the model specifies that there are 3 multipliers available. In order to avoid resource overuse, the scheduler must find the earliest clock cycle within the asap/alap interval at which at most 2 multipliers are busy executing the previously scheduled operations; furthermore, this multiplier must be available for a number of clock cycles at least equal to its data introduction interval. It is straightforward to keep track of resource usage during scheduling.

The above considerations yield the following pseudocode for our topological permutation scheduler.

```
     TPS(DFG, ModTypes)
 1:  {  int n, t;
 2:     ready_node_set = source;
 3:     for (int i=1; i<=|V|; i++)
 4:     {  n = SelectReadyNode();
 5:        t = SelectCycle(n);
 6:        if (t > alap(n)) Exit(fail);
 7:        Schedule(n, t);
 8:        UpdateResourceUsage(n, t);
 9:        UpdateAsap(n, t);
10:        UpdateReadySet(n);
11:     }
12:  }
```

In line 2, the set of ready nodes is initialized to contain a single element, the source. The main loop of the algorithm has as many iterations as there are nodes in the DFG. During every iteration, first, a ready node is selected using some priority function (line 4). Second, the earliest clock cycle at least equal to the asap value of the node and satisfying the resource constraints is determined (line 5). If this clock cycle is larger than the node's alap value, then the selected clock cycle cannot be used and the scheduling algorithm fails to find a feasible solution and exits (line 6). Third, if the algorithm does not exit, the decision is taken to schedule the node in the selected clock cycle (line 7). Fourth, the data structure keeping track of resource usage is updated to remember that an extra module is used (line 8). Fifth, the asap values of successors of the scheduled node are updated (line 9). Consider, for example, Figure 2. Assume that selected node v_4 is scheduled in clock cycle 3; then, the asap value of all successors of v_4 must be updated as if the extra dashed edge with weight 3 is present in the graph. In the example, the asap values of nodes v_7, v_8, and v_{sink} are incremented by one. This update is essential to guarantee that subsequent scheduling decisions satisfy the precedence constraints. Such an update is only necessary if a node is scheduled later than its asap value. Note that alap values need not be adapted. Finally, the set of ready nodes is updated to take into account that the selected node has been scheduled (line 10).

In [4], it has been shown that, for any latency-constrained model that has a feasible schedule, there exists a permutation of the DFG nodes such that the topological permutation scheduler instantiated with the corresponding static priority function yields a schedule with minimal latency. This result means that the transition from the latency-minimization problem to the level of priority functions does not exclude all optimal solutions, which is one of the main reasons for choosing topological permutation scheduling as the basis for our experiments.

IV. CONSTRAINT ANALYSIS

The total number of schedules for a model can be extremely high; it is bounded by ${T_{max}}^{|V|}$. This means that a scheduler can have an enormous search space of schedules in which it has to find a feasible one. Because in most practical cases, only a small fraction of the schedules is feasible, it would be useful for a scheduler to be able to exclude large parts of this search space that certainly do not contain a solution to the scheduling problem. Determining such a *reduced search space* is exactly the problem addressed by *constraint analysis*. Because finding all infeasible schedules is clearly at least as difficult as solving the scheduling problem, practical constraint-analysis techniques are themselves heuristic algorithms. In this section, we briefly discuss how the reduced search space is represented in FACTS and explain one of its analysis techniques to illustrate the basic idea behind constraint analysis. More detailed information can be found in [2] and [6].

In FACTS, the reduced search space is represented by the *distance matrix*. This matrix administrates the minimum and maximum difference between the start times of each pair of operations in a DFG. It forms the common representation on which all constraint-analysis techniques operate. All techniques can use the relative timing information stored in the matrix, and also store their results in it. The effects of these results on the reduced search space are computed by updating the distance matrix, i.e., incrementing the minimum distance or decrementing the maximum distance between two operations.

We have already seen a simple example of a constraint-analysis technique, namely the calculation of the asap and alap values as discussed in the previous section. Because it can easily be proven that scheduling an operation outside the interval guarded by its asap and alap values cannot result in a feasible schedule, many schedules can be excluded. Note that asap and alap values are maintained in the distance matrix: for each operation v, these values correspond to the minimum and maximum distance of the source node to v, respectively.

Standard asap-alap analysis only considers the data dependencies in a DFG. More advanced techniques also take the resource constraints in a model into account. As an example, consider the DFG of Figure 2; assume that only a single multiplier is available. Thus, operations v_3, v_6, and v_8 need to be executed on a single resource. Since a multiplication has a delay of 2 and since the asap value of v_3 equals 2, it can be concluded that the earliest start time of operation v_{sink} is cycle 8, whereas it has a current asap value of 7. This example can be generalized to the following general technique: Given an operation v and the set of all its predecessors, each with an earliest possible start time (asap value), determine a lower bound on the number of cycles it takes to execute these operations on the available module types; this gives a lower bound on the earliest possible start time of v that can be used to update the asap value in the distance matrix. A similar technique can be formulated for the latest possible start times (alap values) of operations.

The integration of constraint analysis into the algorithm of the previous section is straightforward. Before selecting the node to be scheduled, constraint analysis can be applied which means that more accurate asap and alap values can be obtained from the distance matrix.

V. HEURISTICS

The literature contains a number of scheduling heuristics that are interesting in the context of this paper. In our experiments, we used the following four priority functions, each one based on a meaningful heuristic for obtaining good schedules. Note, that the lower values of priority function correspond to the higher priorities.

- **Global freedom**. The global freedom of a node is the difference between its alap value and its asap value. This heuristic gives operations with little global freedom a high priority, because their range of possible start times is small.
- **Alap**. Nodes with a low alap value are given a high priority, resulting in an earliest-deadline-first heuristic.
- **Asap**. Nodes with a low asap value are scheduled first, giving an approach similar to list scheduling.
- **Freeing count**. Priorities are defined by the number of successors that would become ready if the node would be scheduled. The priority of a node increases with its freeing count, to try to maximize the number of ready nodes.

Because the asap values (and also the alap values if constraint analysis is applied) can change during scheduling, the first three priority functions are dynamic ones.

To illustrate the priority functions, consider again Figure 2. As already mentioned, the ready nodes are v_3, v_4, and v_6. Nodes v_4 and v_6 have the lowest global freedom (which means that they have the highest priority according to this heuristic). Node v_4 has the smallest alap value among the ready nodes. Nodes v_3 and v_4 have the smallest asap value. Node v_6 has the highest priority according to freeing count, because its freeng count is 1 whereas nodes v_3 and v_4 have freeing count 0.

VI. Experimental results

We have implemented topological permutation scheduling in FACTS and performed experiments on several benchmarks. Every benchmark contains a single DFG and several versions of the set of module types, which differ only in the number of module instances, e.g., one version of benchmark fdct_block (fast-discrete-cosine-transform block) has 4 adders and 8 multipliers while another one has only 1 adder and 1 multiplier. Several versions of the set of module types combined with the same DFG produce several models. For every model, the minimum latency is known.

For all models contained in every benchmark, experiments are performed using *five* priority functions, namely the four priority functions of the previous section plus the priority function where the (static) priority of a node is determined by a randomly generated permutation of the nodes. For every model and every priority function, 100 experiments have been performed. Every experiment uses a single randomly-generated permutation of the operations. The random priority function simply uses this permutation as a static priority function. The other priority functions use this random permutation to break ties between two or more ready nodes with the highest priority. The experiments with the random priority function are used as a reference to quantify the quality of the other four priority functions.

Every experiment consists of several runs of our topological permutation scheduler. In the first run, the latency constraint is fixed to the known minimum latency of the model. If the algorithm fails, the latency constraint is increased by one and another run is performed. This is repeated until a feasible schedule is found. The difference between the latency of this schedule and the minimum latency is registered as the result of the experiment.

To quantify the impact of constraint analysis, in every experiment actually two sets of runs are performed, one with constraint analysis disabled and one with constraint analysis enabled. Our aim is to see if the use of constraint analysis allows the scheduling algorithm to solve problems with tighter latency constraint. For every set of 100 experiments the minimal, the average, and the maximal result is calculated with and without constraint analysis. For every benchmark, these values are averaged over the models contained in the benchmark. The results are shown in Tables I through IV. In these tables, priority function random is abbreviated as 'rnd', global freedom is abbreviated as 'gfrd', and freeing count is abbreviated as 'frcnt'. Table I contains results for the wave-digital-filter block (wdelf_block), Table II for an inverse-discrete-cosine-transform block (loef_io), Table III for the fast-discrete-cosine-transform block (fdct_block), and Table IV for the threefold-expanded fdct block (fdct3s_expand).

TABLE I
EXPERIMENTAL RESULTS FOR WDELF_BLOCK (4 MODELS).

	ca disabled			ca enabled		
	min	avg	max	min	avg	max
rnd	0.00	1.06	2.25	0.00	0.41	1.50
gfrd	0.50	0.60	0.75	0.00	0.03	0.25
alap	0.25	0.48	0.50	0.25	0.25	0.25
asap	0.25	0.79	1.00	0.00	0.26	1.25
frcnt	0.50	0.85	1.50	0.25	0.51	1.00

TABLE II
EXPERIMENTAL RESULTS FOR LOEF_IO (7 MODELS).

	ca disabled			ca enabled		
	min	avg	max	min	avg	max
rnd	1.71	3.80	6.00	0.14	2.39	4.86
gfrd	1.86	2.75	4.14	0.29	1.28	2.71
alap	0.71	1.61	2.71	0.43	1.22	2.43
asap	1.43	3.68	5.86	0.00	2.04	4.57
frcnt	0.43	1.09	2.29	0.14	0.36	1.43

TABLE III
EXPERIMENTAL RESULTS FOR FDCT_BLOCK (8 MODELS).

	ca disabled			ca enabled		
	min	avg	max	min	avg	max
rnd	0.38	2.45	5.00	0.25	1.43	3.88
gfrd	0.50	1.66	2.75	1.25	1.85	2.50
alap	2.00	2.70	3.75	2.00	2.26	3.25
asap	1.25	1.82	2.62	0.25	1.25	3.50
frcnt	0.12	1.03	2.75	0.12	0.65	2.12

TABLE IV
EXPERIMENTAL RESULTS FOR FDCT3S_EXPAND (2 MODELS).

	ca disabled			ca enabled		
	min	avg	max	min	avg	max
rnd	4.00	7.29	12.00	1.50	6.30	11.50
gfrd	2.50	6.30	8.50	2.50	5.39	9.50
alap	3.50	4.83	6.50	3.00	4.84	7.50
asap	2.50	5.25	9.00	0.50	5.21	10.50
frcnt	1.00	4.58	9.50	0.00	3.06	8.00

From the tables, it can be observed that constraint analysis improves the quality of the scheduler significantly in most cases. However, there are also cases where the results of the asap, alap and global-freedom heuristics get worse, in particular in Tables III and IV. This holds for the best as well as the average and worst schedules found. This somewhat surprising effect can be explained by the fact that the priority functions of these heuristics depend on the asap and alap values and therefore these actually change due to constraint analysis. In the other two cases, constraint analysis has no effect on priorities; it only influences the selection of the cycle in which an operation is scheduled, excluding decisions that would result in an infeasible schedule.

When comparing the heuristics, the freeing-count priority function gives the best overall results. However, the results also show that one has to be careful to decide which heuristic performs the best, because their relative performance is not consistent over the four benchmarks. In fact, although all of them typically perform better than the random permutations, there are also a few cases where this is not true.

VII. CONCLUSIONS

In this work, we have quantified the effect of constraint analysis on the quality of heuristic schedulers for the latency-constrained scheduling problem. The synthesis tool FACTS has been used as the platform for our experiments. Four different priority functions have been used with a topological permutation scheduler. The results show that, in most cases, the results of the scheduler clearly improve with constraint analysis. However, if the priority function depends on the asap or alap values of operations, it can also happen that the results deteriorate due to constraint analysis.

The results show that each heuristic usually performs better than just selecting random permutations. The freeing-count priority function gives often the best results and combines well with constraint analysis. However, there are always a few exceptions, making it difficult to pick the 'best' heuristic. Based on the results, we feel that a good heuristic scheduler should apply some amount of backtracking to traverse a larger part of the search space and thus become robust.

Software pipelining has not been considered here, because in that case it can no longer be guaranteed that there exists a permutation such that topological permutation scheduling gives a schedule with minimum latency [4]. The constraint-analysis techniques in FACTS can handle software pipelining without a problem. In fact, we expect constraint analysis to have an even stronger impact on the results of heuristic schedulers in that case.

REFERENCES

[1] G. De Micheli. *Synthesis and Optimization of Digital Circuits*. McGraw-Hill, Inc., 1994.

[2] C.A.J. van Eijk, B. Mesman, C. Alba Pinto, Q. Zhao, M. Bekooij, J.L. van Meerbergen, and J.A.G Jess. *Constraint Analysis for Code Generation: Basic Techniques and Application in FACTS. ACM Transactions on Design Automation of Electronic Systems*, 5(4), October 2000.

[3] G.F. Girczyc and J.P. Knight. *An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling*. Proc. Int. Conf. on Computer Design, pp. 726–731, 1984.

[4] M.J.M. Heijligers. *The Application of Genetic Algorithms to High-Level Synthesis*. PhD thesis, Eindhoven University of Technology, Dept. of Electrical Engineering, The Netherlands, 1996.

[5] P.G. Paulin, and J.P. Knight. *Force Directed Scheduling in Automatic Data Path Synthesis*. Proc. 24th Design Automation Conf., pp. 195–202, 1987.

[6] A.H. Timmer. *From Design Space Exploration to Code Generation*. PhD thesis, Eindhoven University of Technology, Dept. of Electrical Engineering, The Netherlands, 1996.

[7] W.F.J. Verhaegh, E.H.L. Aarts, J.H.M. Korst, and P.E.R. Lippens. *Improved Force-Directed Scheduling*. Proc. Europenan Conf. on Design Automation, pp. 430–435, 1991.