A Technique for Avoiding Isomorphic Netlists in Architectural Synthesis

Peter Marwedel*, Steven Bashford†,
Rainer Dömer†, Birger Landwehr†, Ingolf Markhof

Technical Report #95-28
August 24, 1995

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717, USA
+1 (714) 824-8059

e-mail: marwedel@ls12.informatik.uni-dortmund.de

Abstract

Register-Transfer (RT-) level netlists are said to be **isomorphic** *if they can be made identical by relabelling RT-components. RT-netlists can be generated by architectural synthesis. In order to consider just the essential design decisions, architectural synthesis should consider only a single representative of sets of isomorphic netlists. Nevertheless, many current synthesis algorithms do not take advantage of this potential reduction in search space. This is especially true for approaches which focus on optimizing the wiring between resource instances. In this paper, we are using netlist isomorphism for the very first time in architectural synthesis. Furthermore, we describe how an integer-programming (IP-) based synthesis technique can be extended to take advantage of netlist isomorphism. As a result, the running time required for synthesis is reduced.*

*On leave from the University of Dortmund. Supported through NATO grant # CRG 950910.

†Current affiliation: Universität Dortmund, Informatik XII. Supported by the Commission of the European Communities under contract ESPRIT 6855 (LINK).

1 Introduction

Early approaches to architectural synthesis used simplified cost functions for guiding the search for efficient RT-architectures. In particular, the effect of interconnections between RT-level components has frequently been neglected. The effect of this can be quite dramatic [McF87].

If interconnections have to be taken into account, RT-components must be uniquely labelled in order to identify the end points of interconnections*. Labels are elements of discrete sets, e.g. integers. Now, even if we restrict ourselves to integers, RT-structures can be labelled in a number of ways. Fig. 1 shows two RT-structures with labelled RT-components.

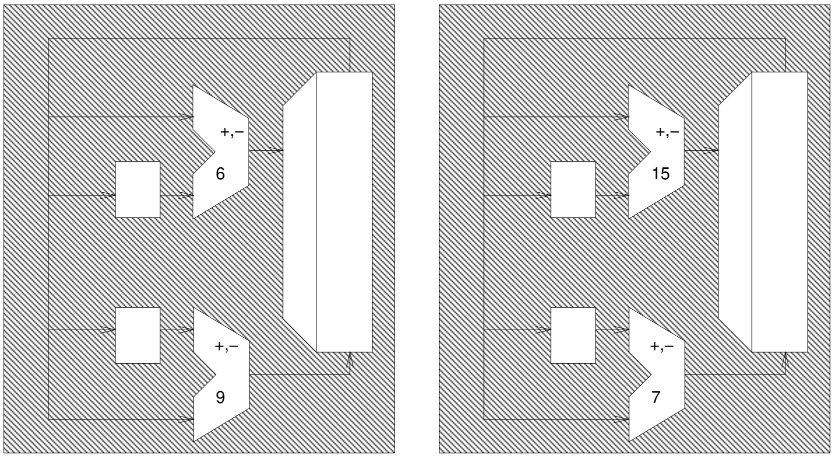

Figure 1: Isomorphic RT-Structures

Assuming that components with equal functionality are actually instances of the same component library element, these structure are obviously very "similar". In fact, we may define a function on the left structure such that its application replaces component labels by the corresponding labels in the right structure.

Structures differing only by their labels are called *isomorphic*. Isomorphism has been used for Graphs, finite state machines [Koh87] etc. It allows us to define the following equivalence relation:

Def. : Let n_1 and n_2 be two netlists. n_1 and n_2 are said to be **renaming-equivalent** (denoted as $n_1 \sim n_2$) if and only if there exists a bijection f on n_1 such that $f(n_1) = n_2$. In this definition, f is supposed to replace only the component labels in the netlist that is passed as an argument.

Relation $\sim$ is an *equivalence relation* and hence defines *equivalence classes*. One would assume that tools synthesizing netlists (such as architectural synthesizers) consider only a single representative of each equivalence class. However, there are several examples (see e.g. [HP83, GE91, Sto90]) which show that this is not true. In these cases, it could be possible to reduce the search space of the algorithms by exploiting netlist isomorphism. In this paper, we will show how netlist isomorphism can be used to reduce the running-time of an architectural synthesis system based on integer programming (IP).

The remainder of this paper is organized as follows: Section 2 describes related work. Section 3 introduces the notation for our mathematical synthesis model. In section 4, we explain how isomorphism can be exploited in our model. The effect of this on algorithm complexity is analysed in section 5. Section 6 lists some practical results. The paper ends with a conclusion.

*For the sake of simplicity, we avoid the discussion about labelling component ports in this paper.

2 Related Work

As explained earlier, netlist isomorphism becomes important if components are labelled.

One of the early approaches to architectural synthesis is the IP-model of Hafer [HP83], which was later adopted for multiprocessor-assignment. The model does not use any kind of *normal form* for labelling components and hence implicitly analyses multiple solutions which are isomorphic. Reduction of the search space would be possible with the technique that will be presented in this paper.

Component labelling is also used in an approach based on simulated annealing [DN89]. In that approach, operations are rebound to various control steps and RT-components. Again, the model does not use any kind of *normal form* for labelling components and isomorphic solutions are analysed.

Just like in the case of Hafer's IP model, Gebotys' approach to interconnect minimization [GE91] using integer programming does not take advantage of isomorphism and reduction of the search space would be possible with the technique that will be presented in this paper.

Note that the speedup by the proposed technique applies essentially to synthesis systems which have placed the very much needed emphasis on interconnect minimization.

3 Synthesis Model

3.1 General Definitions of Terms

Synthesis defines a mapping from behavioural descriptions to structural descriptions. This has frequently been described by arrows in the so-called Y-chart [GK83] describing the different domains in electronic design (see fig. 2).

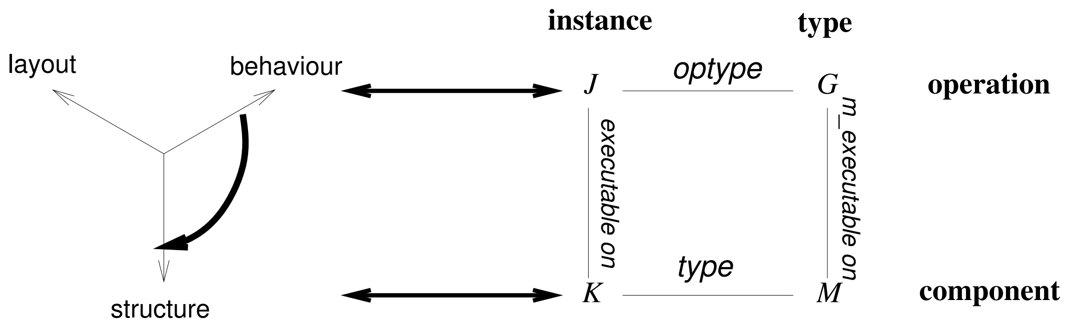

Figure 2: Naming conventions

We assume that the *behaviour* of the system under design is defined by a *dataflow-graph (DFG)*. The nodes of this graph denote operations such as additions and multiplications. More precisely, these nodes contain instances of operation types, such as "+" or "*". Let the nodes of the dataflow graph be uniquely labelled with integers from the corresponding index set $J = \{1..j_{max}\}$. We will use j as a variable to denote such integers. Furthermore, let used operation types be characterized by an index set G. Let function *optype* denote the operation type of DFG-nodes (see fig. 2).

Furthermore, we assume that the *structure* is described by a netlist containing component instances $k \in K = \{1..k_{max}\}$. Each instance is inherited from a corresponding library component type. Let variables $m \in M$ denote library component types. Function *type* denotes the type of a certain instance:

$$type : K \rightarrow M$$

The functionality of component type m is described by relation *m_executable on*:

$\forall m \in M, g \in G : g$ *m_executable on* $m \iff$ component type m is able to perform operation g (this information is available from the library).

From this relation, we derive the corresponding relation *executable on* among instances:

Def.: $j \in J$ *executable on* $k \in K \iff$ *optype*(j) *m_executable on type*(k).

Most synthesis tools do not only generate structure. They also generate a binding between operations and *control steps* in which they are started. This is also the case for all architectural synthesis tools we are aware of. We will use i to denote a certain control step and I to denote the set of all control steps. Note that we consider *multi-cycle operations* (operations which do not terminate in the same control step).

The synthesis task can now be modelled as the problem of binding each operation j to a starting control step i and an executing resource k. Various subtasks of architectural synthesis have been identified; e.g. scheduling, allocation of hardware resources, and the assignment of operations to hardware resources [GDWL92]. Various algorithms and models have been proposed for these subtasks. For example, clique partitioning [ST83], bipartite graph matching [Tim95b], simulated annealing[DN89], integer programming [GE91, Geb92] and many heuristics have been proposed.

3.2 IP-based Synthesis

Among those approaches just mentioned, IP-based models exhibit a number of interesting features, including

- the existence of a formal basis for such models
- the ability of integrating the three main subtasks of behavioural synthesis and a number of extensions
- the fact that the model of architectural synthesis is - to a certain extent - decoupled from the algorithm implementing it.

IP-based models may, however, require long execution times, due to the NP-completeness of integer programming. Nevertheless, it has been shown to be practical if mechanisms for limiting problem sizes exist (see, for example [WGHB95]). In other cases, problem sizes have been inherently small enough to avoid any complexity problems (see e.g. [Mar90]). Finally, faster linear programming (LP)-based approximation algorithms have been used [LMD94]. These cases have shown that IP-based approaches can be used successfully in an area, where the early results indicated excessively long execution times ([HP83]). Careful analysis of some of the recent approaches [GE91] reveals that there is still room for significant speed improvements of IP-based approaches. With such improvements, the application range of such approaches can be extended significantly. Execution interval analysis is one example of such techniques [Tim95a]. In this paper, we elaborate on another technique, one which can be applied to speed-up several of the existing IP-based synthesis algorithms.

For complexity reasons, IP-models (just like other approaches) hardly solve this task in one step. In the following, we will characterize IP-models which include scheduling and integrate other subproblems of HLS to varying extents.

1. The IP-model describes just *control step binding.*

   For example, in [GE91], the essential decision variables are defined as follows:

$$x_{i,j} = \begin{cases} 1, & \text{if operation } j \text{ is started at control step } i \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

   This model is adequate, if *m_executable on* is a one-to-one mapping between operation types and component types (this means, for example, there is just a single adder and a single multiplier in the library). In this case, the solution of the scheduling problem (the problem of binding operations to control steps) implies the number of required components (the "allocation"). Scheduling, allocation and assignment can be solved essentially independently as long as the number of required components is the only variable factor in the cost function (i.e. as long as the interconnect cost is excluded).

2. The IP-model decribes control-step (cs) and *delay binding.*

   This approach is adequate if the library contains mixed speed operations (this means: *m_executable on* is a 1:m relation). cs and delay binding can be modelled with triple-indexed decision variables:

$$x_{i,j,d} = \begin{cases} 1, & \text{if operation } j \text{ is started at control step } i \text{ with delay } d \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

   Achatz, in [Ach93], considers an extension of this approach to the case where the library contains multi-functional units (*m_executable on* is a n:m relation in this case). This case cannot be handled by the model in [GE91].

3. The IP-model describes control step and *type-binding.*

   In type-binding models, triple-indexed decision variables with the following meaning are used:

$$x_{i,j,m} = \begin{cases} 1, & \text{if operation } j \text{ is started on component type } m \text{ at control step } i \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

   This binding provides more detail than delay binding and is recommended, if several component types exist, which execute operations with the same speed.

4. The IP-model describes control step and *instance-binding.*

   In instance-binding models, triple-indexed variables with the following definition are used:

$$x_{i,j,k} = \begin{cases} 1, & \text{if operation } j \text{ is started on component instance } k \text{ at control step } i \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

   Instance-binding is required if any of the following aspects are taken into account:

   (a) Interconnect costs

   Type-binding approaches are not able to take interconnection costs into account. Consideration of just the cost of functional units may, however, result in unreasonable resource folding, e.g. in multiplexing of functional units which are simpler than the multiplexers (for example of AND-units). In order to take interconnection costs into account, the binding of operations to resource *instances* must be considered.

(b) predefined instance binding

Manually predefined bindings have been shown to have a positive effect on the resulting design quality [MS89, AT94]. Predefined bindings can also be generated in a backannotation-like procedure in order to ensure that incremental specification changes result in incremental design changes. Such bindings can also be generated in interactive synthesis environments [JPO93].

Due to the above reasons and due to the recent advances in IP-based synthesis models, we believe that instance binding models will be studied in more detail in the future. We will show how execution times can be shortened for these.

A problem which is common to several integrated scheduling and assignment approaches [Geb92, LMD94, HP83] is the fact that the number instances of a certain component type is usually unknown before scheduling. Hence, for each component type m, a certain number of "potentially required" instance indices $\{k_{m,1}, k_{m,2}, ...k_{m,u_m}\}$ must be used in the IP-model. In this context, u_m denotes an upper bound on the number of instances of m. Upper bounds u_m may be known for a variety of reasons. They may have been defined by the user, computed from the DFG, or computed from the costs of previously generated faster solutions.

4 Exploiting Isomorphism

As a first step, we require that the set of instance indices $\{k_{m,1}, k_{m,2}, ...k_{m,u_m}\}$ forms a contigous range of integers. This means that, without loss of optimality, we restrict ourselves to functions *type* which are step functions. Moreover, we restrict ourselves to *increasing* step functions.

Example: Fig. 3 contains the graphical representation of such a function as well as associated variables ℓ_n, and r_n (these variables will be defined below).

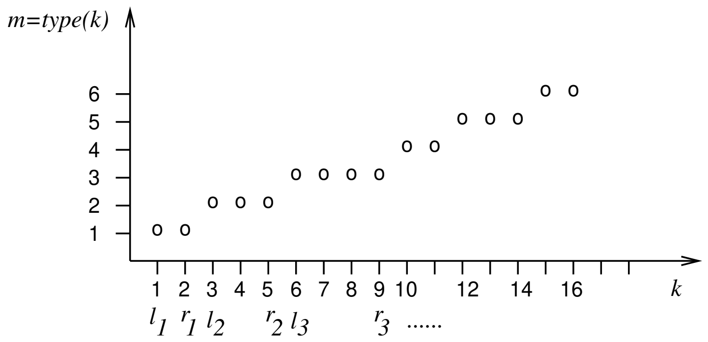

Figure 3: Function $type : K \to M$

□

More precisely, we define ℓ_m and r_m for each $m \in M$ as

$$\ell_1 = 1 \quad (5)$$

$$r_1 = u_1 \quad (6)$$

$$\forall m > 1 : \ \ell_m = (r_{m-1} + 1) \quad (7)$$

$$\forall m > 1 : \ r_m = (\ell_m + u_m) \quad (8)$$

Then, *type* can be defined as

$$type(k) = m \iff \ell_m \leq k \leq r_m \tag{9}$$

The next step for exploiting isomorphism is to restrict ourselves, without loss of optimality, to solutions in which integer label $\ell_m + n$ is used only if there are n or more instances of type m. This means "lower indices are used first".

This can be expressed easily, if the presence or non-presence of a component with a certain index is explicitly modelled. For example, in our synthesis system OSCAR (see [LMD94]), presence of instance k is modelled by a variable b_k:

$$b_k = \begin{cases} 1, & \text{if instance } k \text{ is present in a solution} \\ 0, & \text{otherwise} \end{cases} \tag{10}$$

A straightforward approach for using "lower indices first" could consist in increasing the costs of components by a very small amount. One could define, for example, the cost of the nth component of type m as:

$$cost(\text{instance } \ell_m + n) \quad = \quad cost(\text{type } m) + n * \epsilon \tag{11}$$

Where:

$$\forall m, n \quad : \quad n * \epsilon < cost(\text{type } m) \tag{12}$$

In contrast, we propose another method for "using lower indices first" and we will show that the run-time of our approach is significantly smaller than the straightforward approach. In our approach, we use additional constraints.

With additional constraints, it is quite easy to use "lower indices first". We just have to add the following constraints†:

$$\forall m \; \forall k \in [\ell_m..(r_m - 1)] \;\; : \;\; b_k \geq b_{k+1} \tag{13}$$

Example: If function *type* is defined as in fig. 3, the following constraints will be used:

$$b_1 \geq b_2, b_3 \geq b_4, b_4 \geq b_5, b_6 \geq b_7, b_7 \geq b_8, b_8 \geq b_9, b_{10} \geq b_{11}, b_{12} \geq b_{13}..$$

□

Limitations: The current approach to using a kind of *normal form* for labelling components assures that, for each set of isomorphic netlists, only a single representative is considered. The concept of renaming-equivalent netlists can be extended into a more general concept of equivalence. For example, our approach does not catch effects of "equivalent" wiring.

†The first p b-variables for component m can be set to 1 and the number of additional constraints can be reduced if p is the known lower bound [OKDX95] on the number instances of type m (this was not exploited in the following).

5 Complexity analysis

Most importantly, isomorphic solutions can be avoided without adding *new variables.*

Furthermore, the number of *new constraints* is usually rather small. It can be computed as follows:

$$\text{new constraints} \quad = \quad \sum_{m \in M} (u_m - 1) \tag{14}$$

Let us now analyze the reduction in complexity. Let us assume that the number of component instances which is actually required in an optimal solution is a_m, for each $m \in M$. Obviously it holds that

$$a_m \quad = \quad \sum_{k \in [\ell_m..r_m]} b_k \tag{15}$$

Without the additional constraints (13), there would be $\binom{a_m}{u_m}$ possibilities of labelling a_m components of type m with u_m integers. Hence, for given sets $\{u_m\}$ and $\{a_m\}$, there would be

$$s_{th} \quad = \quad \prod_{m \in M} \binom{a_m}{u_m} \tag{16}$$

isomorphic optimal solutions. Of course, s_{th} is equal to one if the upper bound is either tight or one $(\forall m \in M : (u_m = a_m) \vee (u_m = 1))$. We may consider (16) to be an upper bound on the speed-up *due to excluding multiple isomorphic solutions.*

The actual speed-up, however, may be different from s_{th} due to the following reasons:

- IP-algorithms might, for example, cut-off parts of the search space such that not all isomorphic solutions are actually considered.
- The additional constraints may cause IP-solvers to choose a different sequence for considering the variables. This may have either a positive or a negative effect on the run-time, unless the IP-solver allows maintaining the same sequence by user-controlled sequence guidance.
- Equation (16) ignores the effect of the slightly increased number of constraints.

Due to these reasons, the additional constraints may lead to either smaller or larger computation times. Significantly larger computations times, however, should only occur if the user has no influence on the sequence in which variables are considered.

6 Results

6.1 IP-Solver Independent Results

Using our OSCAR system as an example, we have analysed the actual speed-up. In order to speed up synthesis, we used a cost function considering only the cost of functional units. Constraints included:

precedence constraints, functional unit constaints, assignment constraints and (optional) renaming constraints (see [LMD94] for details).

The library consisted of a simplified library containing an adder, a multiplier and a subtractor. Costs were 20k, 30k and 20k units, respectively. Each component type was single-cycled.

For the elliptical wave filter benchmark, the generated optimal results and the number of variables and constraints are listed in table 1. The results include the upper bound and the actual number of instances of adders ($m = 1$) and multipliers ($m = 2$). Subtractors were not required for this example.

control steps	15	16	17	18	19	20
a_1 (Adder)	3	3	2	2	2	2
u_1	5	5	5	5	5	5
a_2 (Multiplier)	2	1	1	1	1	1
u_2	4	4	4	4	4	4
# Variables	341	518	695	872	1049	1226
equations, excluding (13)	163	233	299	357	415	473
equations, including (13)	170	240	306	364	422	480
speed-up (s_{th})	60	40	40	40	40	40

Table 1: Optimal solutions for elliptical wave filter

Table 2 contains the same type of results for another example: for computing determinantes (see appendix).

control steps	8	9	10	11	12	13	14
a_1 (Multiplier)	3	2	2	2	2	2	1
u_1	6	6	6	6	6	6	6
a_2 (Subtractor)	1	1	1	1	1	1	1
u_2	1	1	1	1	1	1	1
a_3 (Adder)	1	1	1	1	1	1	1
u_3	1	1	1	1	1	1	1
# Variables	224	311	398	485	572	659	746
equations, excluding (13)	96	138	182	225	268	311	354
equations, including (13)	101	143	187	230	273	316	359
speed-up (s_{th})	20	15	15	15	15	15	6

Table 2: Optimal solutions for computing determinantes

6.2 Runtimes for Selected IP-Solvers

For the two examples, actual speed-ups were measured on a Sun SPARCstation-20 running at 60 MHz. We considered four different IP-solvers:

- **lp_solve, version 2.0.1**

  Program lp_solve from the University of Eindhoven [Ber92] is able to solve mixed integer/linear programs. Version 2.0.1 has been derived at our institute from version 2.0 such that variables

b_k are considered first. Results for this solver and the elliptical wave filter are listed in table 3. Runtimes have been included for all combinations of using or not using equation (13) (additional constraints) and equation (11) (modified cost function).

control steps		15	16	17	18	19	20	21
(13)	(11)							
-	-	0.3	4.3	15.5	46.2	98.1	150.8	249.3
+	-	0.7	1.7	5.3	16.1	37.5	48.1	85.9
-	+	0.4	4.9	12.6	75.1	186.2	332.0	577.9
+	+	0.5	1.3	4.5	10.1	21.7	58.0	89.0
speed-up (- -)/(+ -)		0.43	2.53	2.92	2.87	2.62	3.14	2.90

Table 3: Run-times [seconds] of lp_solve 2.0.1 for elliptical wave filter

Fig. 4 shows a graphical representation of the speed-up.

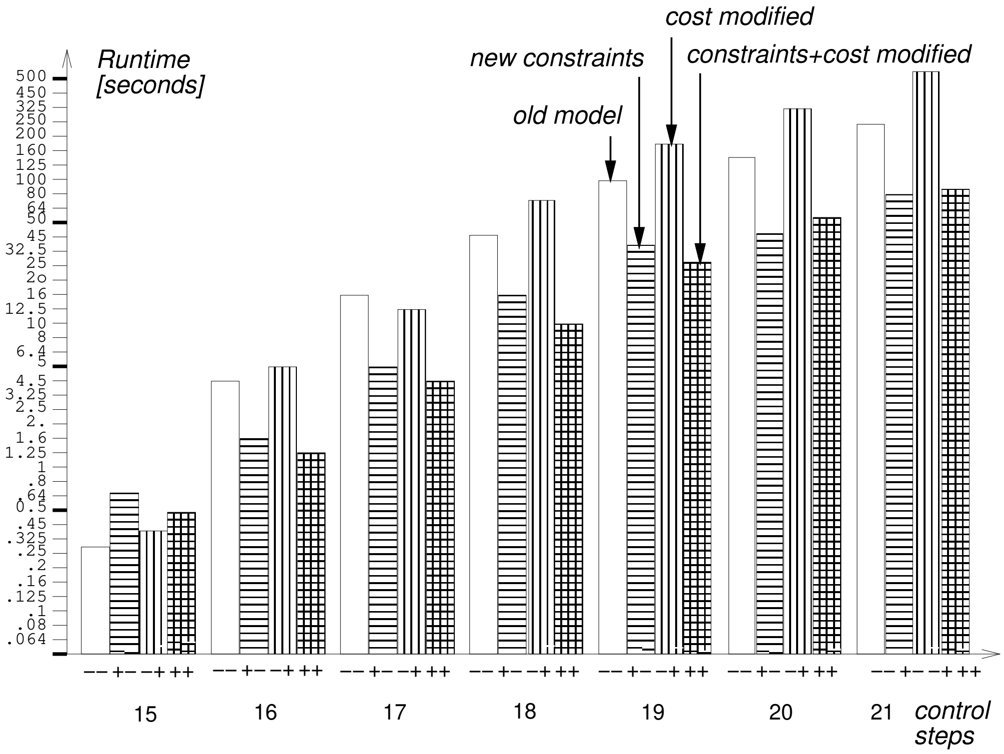

Figure 4: Graphical representation of table 3 (log scale)

From table 3 and fig. 4 it is obvious, that ϵ-terms in the cost function do not speed up IP-solver, except for cases in which the run-time is small and its measurement is imprecise. Additional constraints, however, result in significant time savings, especially if the execution times are high without these constraints. The actual speed-up is significantly smaller than s_{th} (see table 1). Nevertheless, the actual speed-up is important. There is only one case with rather small execution times, in which the speed-up is somewhat smaller than 1.

The large dynamic range of times in combination with the limited resolution of fig. 4 hides the fact that the negative effect of the modified cost function increases with IP-execution times.

Fig. 5 displays normalized runtimes on a linear scale. In this figure, the effect can be seen quite clearly.

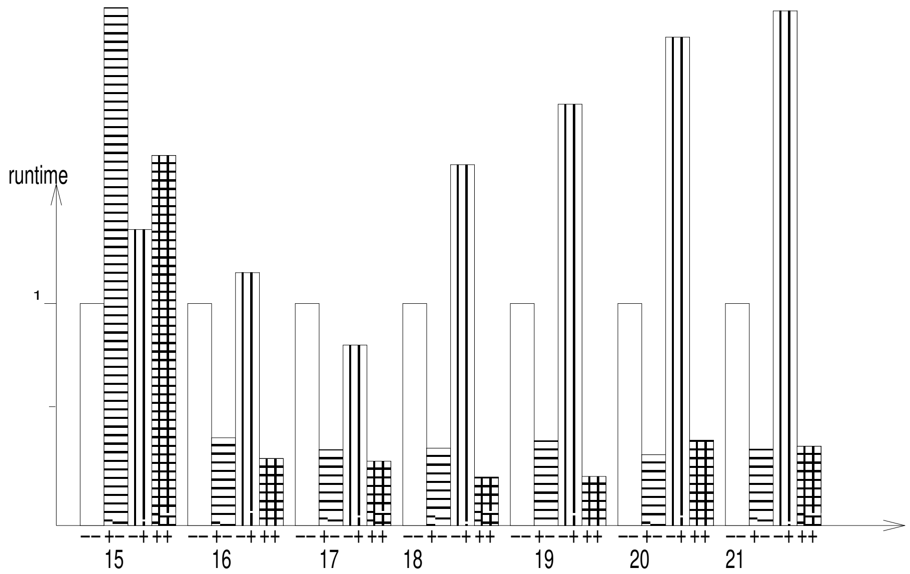

Figure 5: Normalized runtimes (linear scale)

Already from this example, it should be clear that design constraints should –wherever possible– be modelled by IP-constraints and not by modifications of the cost function.

Table 4 shows the same type of results for computing determinates.

control steps		8	9	10	11	12	13	14
(13)	(11)							
-	-	0.8	3.2	6.0	14.7	13.9	31.5	39.3
+	-	0.3	1.1	3.2	3.2	5.5	13.2	9.0
-	+	1.2	5.3	8.6	20.5	31.6	44.1	-
+	+	0.2	1.0	2.5	3.6	5.7	10.1	-
speed-up (- -)/(+ -)		2.67	2.91	1.88	4.59	2.53	2.39	4.37

Table 4: Run-times of lp_solve 2.0.1 [seconds] for computing determinantes

The results from this example confirm the observations made for the elliptical wave filter.

- Beta-release of **osl_solve, release 2**

  The next solver which we considered, is part of the commercial OPTIMIZATION SUBROUTINE LIBRARY (osl)‡. For this solver, we did not specify any sequence for considering variables. Results can be found in tables 5 and 6.

  The actual speed-up exceeds s_{th} (see tables 1 and 2) in some cases by unexpected numbers. This solver seems to be very sensitive to new constraints. For this solver, the new constraints are very important for making the solver usable at all.

‡Copyright: IBM.

control steps		15	16	17	18	19	20	21
(13)	(11)							
-	-	0.7	82.7	6.7	246.3	1155.0	134.5	378.4
+	-	1.3	3.5	8.4	42.3	63.9	58.3	72.7
-	+	0.9	458.7	4.7	114.2	111.8	241.3	509.4
+	+	1.0	1.9	5.3	15.2	76.7	49.1	129.0
speed-up (- -)/(+ -)		0.53	23.62	0.79	5.8	18.1	2.3	5.2

Table 5: Run-times [seconds] of osl_solve for elliptical wave filter

control steps		8	9	10	11	12	13
(13)	(11)						
-	-	3.4	4.0	169.0	11.6	2432.6	>4617
+	-	1.1	2.1	3.2	5.1	16.1	12.4
-	+	7.2	4.3	16.3	6.6	3719.8	>6107
+	+	1.1	2.1	6.2	8.4	10.8	17.4
speed-up (- -)/(+ -)		3.1	1.90	52.81	2.27	151	>372

Table 6: Run-times [seconds] of osl_solve for computing determinantes

- **lp_solve, version 1.0**

  This is an earlier version of lp_solve. This version has not been modified to consider certain variables first. Run-times for this solver are listed in tables 7 and 8.

control steps	15	16	17	18	19	20
runtime, without (13) [s]	1	10	36	244	372	598
runtime, with (13) [s]	1	4	12	88	121	350
speed-up (actual)	1	2.5	3.0	2.8	3.1	1.7

Table 7: Run-times [seconds] of lp_solve 1.0 for elliptical wave filter

control steps	8	9	10	11	12	13	14
runtime, without (13) [s]	8	10	17	37	54	56	40
runtime, with (13) [s]	1	2	3	4	9	15	52
speed-up (actual)	8.0	5.0	5.7	9.3	6.0	3.7	0.8

Table 8: Run-times [seconds] of lp_solve 1.0 for computing determinantes

Since the original version 2.0 of lp_solve does not perform any better than version 1.0 if our examples are used as input, the slower execution speed of version 1.0 is essentially caused by the fact that it is does not necessarily consider variables b_k first. This is consistent with the observation made by Achatz [Ach95]: at least for the IP-problems at hand, it is advisable to consider variables occuring in the cost function first. According to table 3, even the run-times reduced this way can be further reduced by our additional constraints.

- A pre-release of **opbdp V1.0**

  The next IP-solver which we considered is a pre-release of an algorithm for 0/1 integer programs which does not use the simplex algorithm [Bar95]. The corresponding results can be found in tables 9 and 10.

control steps	15	16
runtime without (13) [s]	139.8	>4225
runtime with (13) [s]	2.7	>3313
speed-up (actual)	51.77	-

Table 9: Run-times [seconds] of opbdp V1.0 for elliptical wave filter

control steps	8	9
runtime without (13) [s]	2.4	-
runtime with (13) [s]	0.6	-
speed-up (actual)	4	-

Table 10: Run-times [seconds] of opbdp V1.0 for computing determinantes

  Only few cases of the IP-problems can be solved with this pre-release. For these, however, the speed-up is significant and approaches s_{th}.

A general remark, applying to all four solvers is the following: the number of elements in the considered library is rather small and larger speed-ups can be expected for larger libraries.

7 Conclusion

In this paper, we have presented a technique for exploiting netlist isomorphism in architectural synthesis. With this technique, only one representative for each class of equivalent solutions is generated. The technique presentented can be combined with a variety of synthesis models in order to reduce the run-time of architectural synthesis. Hence, the range of applications of IP-based synthesis is extended despite the fact that synthesis will still be NP-hard. In has been shown that, for the examples we considered, additional constraints result in a larger runtime reduction than cost function modifications. In the OSCAR system, the technique made synthesis of larger examples feasible.

We believe that the concept of netlist equivalence reaches well beyond the current approach. By restricting synthesis systems to consider only the *essential* decisions, (decisions which may actually affect the resulting hardware structure), a lot of computation time might be saved. This might be the right way to go in order to include layout considerations in architectural synthesis.

The authors appreciate the comments of Fadi Kurdahi and Nikil Dutt (UC Irvine).

References

[Ach93] H. Achatz. Extended 0/1 LP formulation for the scheduling problem in high-level synthesis. *EURO-DAC'93*, 1993.

[Ach95] H. Achatz. Personal communication. *UC Irvine*, 1995.

[AT94] L. F. Arnstein and D. Thomas. The attributed behavior abstraction and synthesis tools. *31th Design Automation Conference*, pages 557–561, 1994.

[Bar95] P. Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Technical Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, http://www.mpi-sb.mpg.de/papers/reports/abstracts.html#MPI-I-95-2-003, 1995.

[Ber92] M.R.C.M. Berkelaar. Unix^{tm} manual page of lp_solve. *Eindhoven University of Technology, Design Automation Section*, 1992.

[DN89] S. Devadas and R.A. Newton. Algorithms for allocation in data-path synthesis. *IEEE Trans. on CAD*, 8:768–781, 1989.

[GDWL92] D. Gajski, N. Dutt, A. Wu, and S. Lin. *High-Level Synthesis –Introduction to Chip and System Design–*. Kluwer Academic Publishers, 1992.

[GE91] C. H. Gebotys and M. I. Elmasry. Simultaneous scheduling and allocation for cost constrained optimal architectural synthesis. *28th Design Automation Conference*, pages 2–7, 1991.

[Geb92] C. H. Gebotys. Optimal scheduling and allocation of embedded VLSI chips. *29th Design Automation Conference*, pages 116–123, 1992.

[GK83] D.D. Gajski and R.H. Kuhn. New VLSI tools. *IEEE Computer*, pages 11–14, 1983.

[HP83] L. Hafer and A. C. Parker. A formal method for the specification, analysis and design of register-transfer level digital logic. *IEEE Trans. on Computer-Aided Design, Vol.2*, pages 4–18, 1983.

[JPO93] A. A. Jerraya, I. Park, and K. O'Brien. AMICAL: an interactive high level synthesis environment. *Proceedings EDAC*, pages 58–62, 1993.

[Koh87] Z. Kohavi. Switching and finite automata theory. *Tata McGraw-Hill Publishing Company, New Delhi, 9th reprint*, 1987.

[LMD94] B. Landwehr, P. Marwedel, and R. Dömer. OSCAR: Optimum simultaneous scheduling, allocation and resource binding based on integer programming. *Euro-DAC*, 1994.

[Mar90] P. Marwedel. Matching system and component behaviour in MIMOLA synthesis tools. *Proc. 1st EDAC*, pages 146–156, 1990.

[McF87] M. C. McFarland. Reevaluating the design space for register transfer level synthesis. *IEEE Int. Conf.on Computer-Aided Design(ICCAD)*, pages 262–265, 1987.

[MS89] P. Marwedel and W. Schenk. Improving the performance of high-level synthesis. *Microprogramming and Microprocessing, Vol.27*, pages 381–388, 1989.

[OKDX95] S. Y. Ohm, F.J. Kurdahi, N. Dutt, and M. Xu. A comprehensive estimation technique for high-level synthesis. *Int. Symp. on System Synthesis (ISSS)*, 1995.

[ST83] D.P. Siewiorek and C.J. Tseng. Facet: A procedure for the automated synthesis of digital systems. *20th Design Automation Conf.*, pages 490–496, 1983.

[Sto90] L. Stok. A generalized interconnect model for data path synthesis. *Proc. CompEuro 90, Tel Aviv*, pages 461–465, 1990.

[Tim95a] E. Timmer. Conflict modelling and instruction scheduling in code generation for in-house DSP cores. *32th Design Automation Conference*, 1995.

[Tim95b] E. Timmer. Exact scheduling strategies based on bipartite graph matching. *European Design & Test Conference (ED&TC)*, 1995.

[WGHB95] T. Wilson, G. Grewal, S. Henshall, and D. Banerji. An ILP-based approach to code generation. *in: P. Marwedel, G. Goossens (ed.): Code Generation for Embedded Processors, Kluwer*, 1995.

Appendix: Source code for Example det

```
ENTITY determinante IS
  PORT( a, b, c, d, e, f, g, h, i : IN BIT_VECTOR (31 DOWNTO 0);
        s       : OUT BIT_VECTOR (31 DOWNTO 0));
END determinante;

ARCHITECTURE behaviour OF determinante IS
BEGIN PROCESS
VARIABLE        a_in, b_in, c_in, d_in, e_in, f_in, g_in, h_in,
                i_in, s_out   : BIT_VECTOR (31 DOWNTO 0);
BEGIN
a_in := a;
b_in := b;
c_in := c;
d_in := d;
e_in := e;
f_in := f;
g_in := g;
h_in := h;
i_in := i;
s_out :=          a_in * e_in * i_in
                - a_in * f_in * h_in
                - d_in * b_in * i_in
                + d_in * c_in * h_in
                + g_in * b_in * f_in
                - g_in * c_in * e_in;
s <= s_out;
END PROCESS;
END behaviour;
```