Improving the Computational Performance of ILP-based Problems

M. Narasimhan J. Ramanujam
Department of Electrical and Computer Engineering, Louisiana State University, Baton Rouge, LA 70803
http://www.ee.lsu.edu/jxr {mukund, jxr}@ee.lsu.edu

Abstract

Many interesting problems in VLSI design are computationally extremely difficult, and as such, there exist no efficient (read polynomial time) algorithms for these problems. Such problems include placement, routing, scheduling and partitioning. A commonly used technique for solving these problems is to model them as an Integer Linear Programming (ILP) problem, and to then solve the resulting model using a generic ILP solver. So far, improving the computational efficiency of these problems was considered equivalent to improving the model. In this paper, we show that large performance improvements can be achieved by incorporating problem specific informations into the ILP solver itself. While the techniques that we present are general in nature, for the sake of concreteness, we will illustrate them by applying them to scheduling problems in high-level synthesis. It will be shown that there is a lot of problem specific information that can be incorporated into the model solver, and that doing so actually improves the performance considerably. We will present experimental results to show that the problem specific ILP solver is considerably faster, often showing improvements by a factor of 1000 in execution time.

1 Introduction

Many optimization problems in High Level Synthesis are modeled as Integer Linear Programming (ILP) models, a solution to which is obtained using a generic ILP solver. ILP solvers almost always rely on a branch and bound methodology, using linear relaxation as the bounding technique. Often, when a particular problem is modeled using ILP, the solver cannot use all the information that may be available from the "original" problem (as opposed to the ILP problem) very effectively. Therefore, using a generic ILP solver to attack the problem is perhaps not the most efficient way to find the solution. A better branch and bound algorithm can be built by exploiting our knowledge of the "original" problem. This is especially true in cases where most of the variables are $0/1$ variables and represent decisions. In such problems, when the solver branches on a decision variable y, it creates two branches, $y = 0$ and $y = 1$. Setting a decision variable to 0 or to 1 has an interpretation in the original problem and it is possible that this decision has other ramifications. So by using a problem-specific ILP solver, we can improve the efficiency of the ILP solution process by exploiting this additional information. Another area in which the knowledge of the problem can be exploited is in the improvement of the lower bound and the selection of the next branch. We will also show that efficient bounding procedures can be applied to reduce the "size" of the ILP problem. We will demonstrate these techniques and provide experimental results for scheduling problems in high-level synthesis [5, 2, 4].

Some problems which can be modeled and solved as ILP models include scheduling, partitioning, module selection and placement. In order to make our discussion concrete, we will specifically discuss the scheduling problem, which Gajski et al. [3] call "perhaps the most important step during structure synthesis." In this paper, two problems are considered

- The *Resource Constrained Scheduling* (RCS) Problem, *i.e.,* minimizing the latency of an implementation, given a fixed set of resources.
- *Time Constrained Scheduling* (TCS) Problem, *i.e.,* minimizing the area of an implementation, given a maximum latency.

These problems are known to be NP-complete in general, and so the existence of a polynomial time algorithm for solving these problem is highly unlikely. Because of the increased computational power available on most modern workstations, it is now feasible to explore the design space using exact algorithms. Blythe and Walker [1] have shown that complete design space exploration can in fact be made considerably more efficient by the use of good bounding strategies. It is shown that the use of bounding reduces the number of iterations of the complete scheduling algorithm that has to be performed. However, even with such techniques, the time taken to solve a few iterations of the scheduling algorithm can be large for medium-sized problems, and so it is still very important to derive fast exact scheduling algorithms. Most exact algorithms transform the scheduling problem into an ILP model. Even though ILP can be computationally very intensive, Chaudhuri and Walker [2] and Gebotys and Elmasry [4] have shown that efficient formulations and the application of polyhedral theory can yield faster solutions for small and medium-sized problems. While it is possible to find a solution to an ILP problem using a generic ILP solver, using a custom ILP solver that takes advantage of the specifics of the problem will result in much faster solutions. In this paper, we show that there is a lot of problem specific information that can be incorporated into the model solver, and that doing so actually improves the performance considerably.

This paper is organized as follows. Section 2 briefly describes our notation and Section 3 describes related work. In Section 4, we present techniques for improving the performance of the ILP solver. Experimental results are described in Section 5 and conclusions are in Section 6.

2 Notation

We assume that the problem to be scheduled is represented in the form of a Data Flow Graph (DFG). We will denote by $V(G) = \{t_1, \ldots, t_N\}$ the set of operations in the DFG and by $E(G)$ the dependence constraints between those operations. A dependence constraint $\langle t_i, t_j \rangle$ implies that operation t_j depends on the result of operation t_i. If there is a path from t_i to t_j in the DFG then t_i is a predecessor of t_j and equivalently t_j is a successor of t_i; we use $\Gamma(t_i, t_j)$ to denote the length of the longest path from t_i to t_j. The set of all predecessors (resp. successors) of t_i is denoted by $\mathsf{Pred(t_i)}$ (resp. $\mathsf{Succ(t_i)}$).

Let $\mathcal{M}$ denote the set of resource classes (multipliers, adders, etc.). The number of available (required) instances of each resource class ℓ is denoted by M_ℓ. Each operation in $t_i \in V$ is associated with a unique member $\mathsf{type(t_i)} \in \mathcal{M}$, and can be executed only on an instance of resource class $\mathsf{type(t_i)}$.

A schedule is a mapping of operations to time steps that satisfies the dependency and resource constraints. The values $\mathsf{ASAP(t_i)}$ (resp. $\mathsf{ALAP(t_i)}$) is the earliest (resp. latest) time step to which operation t_i can be scheduled. We will denote by $\mathsf{TimeFrame(t_i)}$ the set $\{\mathsf{ASAP(t_i)}, \mathsf{ASAP(t_i)}+1, \ldots, \mathsf{ALAP(t_i)}\}$.

We will assume that the ILP formulation is

$$Z_{ILP} = \min \{cx : Ax \leq b, \quad x \in \mathcal{Z}_+^m\},$$

where c, x, A and b are $1 \times m$, $m \times 1$, $n \times m$ and $n \times 1$ respectively. We will denote by (A, b) the polyhedron $Ax \leq b$. Further, we will use $integer(A, b)$ to denote the integer vectors in (A, b). Since we are considering branch and bound techniques, we will be partitioning the polyhedron (A, b), into disjoint polyhedra $(A_1, b_1), (A_2, b_2), \ldots, (A_n, b_n)$ whose union contains all the elements of $integer(A, b)$. Given a constrained optimization problem such as the one above, relaxing some of the constraints will lead to a solution which is a lower bound on the value of the original problem. A commonly used relaxation for the (ILP) problem is its *Linear Relaxation*, in which all integrality constraints are discarded. The Linear Relaxation of the (ILP) problem can be expressed as

$$Z_{LR} = \min\{cx : Ax \leq b, \quad x \in \Re_+^m\},$$

and of course, we have $Z_{LR} \leq Z_{ILP}$.

3 Previous Work

We now describe a simple ILP formulation [5] of the scheduling problem. A set of decision variables, $\left\{y_{ik} : k \in \mathsf{TimeFrame(t_i)}\right\}$ is associated with each operation t_i. The variable y_{ik} is 1 if and only if operation t_i is scheduled to time step k. Since t_i must be scheduled to one and only one step, we have

$$\sum_{k \in \mathsf{TimeFrame(t_i)}} y_{ik} = 1.$$

Resource constraints are modeled as follows

$$\sum_{\mathsf{type(t_i)}=\ell} y_{ik} \leq M_\ell \qquad \forall \text{ time steps } k.$$

A dependence constraint $\langle t_i, t_j \rangle$ is modeled by the inequality

$$\sum_{k \in \mathsf{TimeFrame(t_j)}} k \cdot y_{jk} - \sum_{k \in \mathsf{TimeFrame(t_i)}} k \cdot y_{ik} \;\; \geq \;\; 1.$$

This model is extremely simple and does not take into account the structure of the scheduling polytope. Chaudhuri and Walker [2] and Gebotys and Elmasry [4] present better models that differ from this simple model. Both models differ only in their modeling of precedence constraints. The Resource Constrained Scheduling problem (RCS) and Time Constrained Scheduling problem (TCS) differ only in the objective function (latency or area).

4 Improving the ILP solver

4.1 Better Next-Branch Selection

Let x' be the solution vector of the linear relaxation of the ILP problem. Clearly if x' is an integral vector, then we are done. Suppose that this is not the case. Then there is at least one element in x' which is non-integral. Typically, many of the variables are non-integral, and so the solver picks one of the elements (corresponding to y_{ij}) to branch on. We will assume that the variable picked is a decision variable (i.e. takes on only the values 0 or 1) and so

$$integer\,(A, b) \subseteq (A, b)\,|_{y_{ij}=0} \;\cup\; (A, b)\,|_{y_{ij}=1}.$$

Let $(A_1, b_1) = (A, b)\,|_{y_{ij}=0}$ and $(A_2, b_2) = (A, b)\,|_{y_{ij}=1}$. The solver now has to decide whether it will attempt to find the optimal solution in (A_1, b_1) first or in (A_2, b_2) first. So, the solver has to make two decisions here. First it has to pick the variable to branch on. Then it has to decide which one of the smaller polyhedra resulting from this branch to explore first. A particularly naive solver might just make random choices for both these decisions. More sophisticated solvers might attempt to pick a variable to branch on based on the values of the variables, or branch on a set of variables simultaneously. A better branching scheme can be derived by exploiting the fact that the variables $\{y_{ij} : j \in \mathsf{TimeFrame(t_i)}\}$ are the set of variables deciding the step to which t_i is assigned. Notice that only one of these variables can be set to 1 at a time. Therefore,

$$integer(A, b) \subseteq \bigcup_{j \in \mathsf{TimeFrame(t_i)}} (A, b)|_{y_{ij}=1}.$$

Hence, a good branching scheme would be to pick an operation t_i to branch on, and then branch on this set of variables.

We can use many heuristics to pick the next operation to branch on. If we look at the value of the variable y_{ij} in the vector x' as the probability of scheduling t_i in time step j, then we might decide to pick the operation t_i corresponding to the variable y_{ij} with the maximum value (probability). Then, this scheme would also result in an order in which the branches would be considered. Another scheme which yielded better results involves picking the operation t_i which has not been scheduled or branched on and all of whose predecessors have been scheduled. Once the operation is picked, the branches can be explored in order of the values of y_{ij}. This scheme reduces the number of times that the linear programming solver (bounding routine) is called in most cases.

4.2 Addition of Valid Inequalities

As we already mentioned, branching on a node (set of nodes) has an interpretation in the original problem. We use this information to generate additional valid inequalities, i.e., inequalities which will always be satisfied by any integer solution, but need not be satisfied by non-integer solutions. Suppose that as a result of a branch, a variable $y_{i_1 j_1}$ is set to 1. What this means is that operation t_{i_1} has been scheduled to step j_1. Clearly this means that none of the operations $t_{i_2} \in T_{succ}(t_{i_1})$ can be scheduled to a time step before $\Gamma(t_{i_1}, t_{i_2}) + j_1$. Therefore, we can introduce another constraint

$$\sum_{j_2 \in \mathsf{TimeFrame(t_{i_2})} \text{ and } j_2 < \Gamma(t_{i_1}, t_{i_2}) + j_1} y_{i_2 j_2} = 0$$

for every operation $t_{i_2} \in \mathsf{Succ(t_{i_1})}$. We can derive this inequality based on our knowledge of the "original" problem. In practice, we add up all the inequalities of the form of the inequality shown, and create a single equation. It should be noted that while in general replacing many equations with their sum results in a weaker constraint, in this case there is no weakening. Adding such an equation tightens the formulation, and results in faster solution times. This technique of adding cutting planes (inequalities) in a branch and bound setting is called *branch-and-cut,* and has recently been used to efficiently solve some extremely large ILP formulations.

4.3 Better Bounding

Many researchers have tried to find good bounding procedures for scheduling problems, and some very good techniques have been obtained [9, 4, 2, 7]. However, most people have used these bounding procedures only for static bounding, and not in a branch and bound context, where it can dramatically reduce the time taken for computing the optimal solution. In most of the cases, the ILP solver finds an optimal solution early on, but spends most of the time just confirming that the solution found is indeed the best possible. Therefore, tightening the bounds used by the solver can reduce the problem solving time. Knowledge of the objective function can sometimes be used to improve the actual lower bound Z_{LR} generated by linear relaxation. For the ILP formulation of the RCS problem, we know that the latency will always be integral. Therefore, the objective function must always be integral, and hence, we can use the ceiling of the linear relaxation $\lceil Z_{LR} \rceil$ as a tighter lower bound. The same can be done for the TCS problem. If we are optimizing for area, and if the ratios of the areas of the resources can be expressed (or at least reasonably approximated) as an integer ratio (and this is almost always the case), we can do the same. Doing this is particularly important for the TCS problem because in this case $\lceil Z_{LR} \rceil$ is very often the tightest bound that can be obtained [4, 2]. Therefore, once the optimal solution has been found, just a few iterations of the linear program solver will confirm the optimality of the solution. A big advantage of this scheme is that the increase in computation cost (calculating the ceiling of a number compared to solving a LP formulation) is almost negligible. For the RCS problem, we can improve the lower bound by using an additional lower bounding scheme such as the ones proposed in [9] and [6]. See [7] for more details.

4.4 Better Tests for Optimality

Another improvement would be to run a heuristic each time the solver is forced to branch, to see if the heuristic can come up with a schedule, whose objective function is the same as the solution to the linear relaxation of the problem at hand[1]. The reason why this might prove useful is that there may be many solutions with the same objective function value. It is quite possible, therefore, for the (linear programming/bounding procedure) solver to return a non-integral solution, even though there is actually an integral solution whose objective criteria is the same. On several occasions, running a simple heuristic confirms this, eliminating the need for further search. It is, of course crucial that the heuristic runs fast.

4.5 Reduction of the Search Space

The ILP model for these problems assume that the TimeFrames of all the variables have been calculated as $\mathsf{TimeFrame}(\mathsf{t_i}) = \{\mathsf{ASAP}(\mathsf{t_i}), \ldots, \mathsf{ALAP}(\mathsf{t_i})\}$. Being able to reduce the TimeFrames of the operations in T will improve the ILP model by reducing the size of the search space. One can use the bounding technique presented by Langevin and Cerny [6] to reduce the TimeFrames of the variables. In [7], a technique to further reduce the TimeFrames of the variables is presented. Using these TimeFrames will lead to a ILP model with fewer variables. For the RCS problem, a further reduction may be obtained. This technique relies on the fact that the latency of a schedule is integral, and so, if the best schedule so far has a latency l, then a better schedule has to have latency $l-1$ or less. This is significant as a generic ILP solver cannot know this. Therefore we can use this fact to add additional inequalities which will reduce the search space. This same technique can be applied every time a solution which is better than the previous best solution is found. Therefore, finding a better solution will lead to both better bounding and a reduction in the number of branches that have yet to be explored.

[1] This is not the same as a generic heuristic used to solve the original problem. This heuristic would be required to take the "decisions" already made into account.

5 Experimental Results

The effect of the improvements on various benchmarks is presented in Tables (1), (2) and (3). Since some of these benchmarks are very large, it was not possible to obtain solutions to all instances of the TCS and RCS problems. Cases for which the solution was not obtained within 1 hour on a Sun UltraSPARC is indicated with a †. For all our examples, we assumed that the expression for the area was $5 \cdot \#multipliers + 2 \cdot \#adders$. For all benchmarks, we assume that an ALU takes takes 1 clock cycle to execute, a multiplier takes 2 clock cycles. A non-pipelined multiplier is denoted $*_m$. A pipelined multiplier denoted $*_p$ has a a pipeline initiation interval of 1 clock cycle. The time required by a straightforward *Vanilla* ILP solver are presented in the column labeled *V*, and the times required for the *Refined* ILP solver (i.e., the one that includes our improvements) are listed in the column labeled *R*. For the Refined ILP procedure, we list two set of results for the TCS problem - one obtained by running the procedure on the graph (denoted *R*(F)), and the other obtained by running the procedure on the reversed graph (denoted *R*(R)). For the RCS problem (Table 3), we list only one result from the refined solver. In some cases, our algorithm does take more time than the *Vanilla* ILP solver. This occurs principally because our algorithm has a lot of additional overhead, and sometimes (especially in small problems) this overhead causes our algorithm to run slower. However for a vast majority of the cases, our refined ILP solver which uses problem-specific information is clearly superior, often showing improvements by a factor of 1000 or more in execution time.

6 Conclusions

While ILP is a powerful technique, to actually obtain fast results using ILP, we must use good models [8] and customize the ILP solver to take problem specific information into account. In this paper we showed how the efficiency of the ILP solution process can be tremendously improved by incorporating problem specific information into the ILP solver. We presented a concrete example in which such improvements in performances were achieved.

Acknowledgments The work of J. Ramanujam was supported in part by supported in part by NSF Young Investigator Award CCR-9457768.

References

[1] S. Blythe and R Walker. A methodology for scheduling in 3D design space. *Proc. 8th Intl. Symp. Syst. Synt.,* pp. 78–83, Sep. 1995.

Table 1: Execution times for the RCS problem

Test	$*_m$	ALU	#steps	V	R
EWF	1	1	28	†	3.72
	1	2	21	3.65	1.92
	2	1	28	†	3.40
	3	3	17	0.05	0.01
ARF	1	1	34	†	8.52
	1	2	34	†	10.26
	2	1	18	†	0.32
	2	2	18	†	0.45
	4	2	11	0.02	0.01
FDCT	1	1	34	†	< 0.01
	1	2	34	†	< 0.01
	2	1	26	†	< 0.01
	2	2	18	†	< 0.01
	4	2	13	2.36	< 0.01
	4	3	11	12.80	†
	8	4	08	0.05	< 0.01
Test	$*_p$	ALU	#steps	V	R
EWF	1	1	28	†	3.00
	1	2	19	0.42	2.33
	2	1	28	†	3.70
	2	3	18	< 0.01	0.01
ARF	1	1	19	1403.75	0.39
	1	2	19	0.65	0.37
	2	2	13	0.67	0.04
	2	3	13	0.12	0.04
FDCT	1	1	26	†	< 0.01
	2	1	26	†	< 0.01
	2	3	12	267.95	112.53
	3	3	10	0.22	< 0.01
	4	4	09	0.18	< 0.01

Table 2: Optimal Solutions for the TCS Problem

Test	#steps	$*_m$	ALU	R(F)	R(R)	V
EWF	28	1	1	19.9	29.03	†
	26	1	2	1.97	2.17	†
	25	1	2	1.01	1.22	†
	24	1	2	0.87	0.76	†
	23	1	2	4.12	0.08	†
	22	1	2	2.44	0.26	†
	21	1	2	0.81	0.86	†
	20	2	2	0.17	0.12	†
	19	2	2	0.29	0.23	†
	18	2	2	0.11	0.02	0.05
	17	3	3	0.01	0.01	0.02
ARF	34	1	1	87.36	†	†
	19	2	1	53.54	114.09	3.53
	18	2	1	4.68	3.07	2.46
	17	2	3	†	101.5	†
	16	2	3	130.49	844.43	†
	15	3	2	72.28	92.40	†
	14	4	2	0.32	0.94	†
	13	4	2	0.66	3.52	†
	12	4	2	0.23	0.17	0.14
	11	4	2	0.03	0.03	0.06
FDCT	25	2	2	19.59	†	†
	23	2	2	18.08	11.59	†
	22	2	2	22.10	9.95	†
	21	2	2	489.23	7.8	†
	19	2	2	187.35	107.26	†
	17	3	2	63.04	60.52	†
	16	3	2	27.34	50.03	†
	11	4	3	2.14	18.72	†
Test	#steps	$*_p$	ALU	R(F)	R(R)	V
EWF	28	1	1	†	1014.99	35.11
	26	1	2	2.53	16.64	†
	25	1	2	1.15	1.14	†
	24	1	2	0.92	1.19	†
	23	1	2	0.08	4.53	†
	22	1	2	0.79	0.48	†
	21	1	2	0.50	1.68	†
	20	1	2	0.35	0.83	†
	19	1	2	0.38	0.08	730.77
	18	1	3	0.03	0.12	0.06
	17	2	3	0.01	0.01	0.03
ARF	19	1	1	1872.11	1.77	4.65
	18	2	1	6.53	7.14	†
	17	2	1	1094.49	4.00	†
	16	2	1	73.00	68.25	†
	15	2	1	104.07	377.87	†
	14	2	1	23.24	58.83	†
	13	2	2	74.12	0.33	†
	12	3	2	175.75	323.60	0.14
	11	4	2	0.03	0.02	0.05
FDCT	25	1	2	28.36	†	†
	23	1	2	253.99	214.14	†
	20	1	2	164.95	618.31	†
	18	2	2	38.42	44.29	†
	17	2	2	11.89	40.48	†
	16	2	2	6.67	758.38	†
	15	2	2	18.63	386.01	†
	14	2	2	14.02	1444.42	†
	12	2	3	5.31	5.01	†

[2] S. Chaudhuri and R. Walker. Analyzing and exploiting the structure of the constraints in the ILP approach to the scheduling problem. *IEEE Trans. VLSI*, 2(4):456–471, Dec. 1994.

[3] D. Gajski, N. Dutt, and B. Pangrle. Silicon compilation (tutorial). In *Proc. IEEE Custom Inte. Circ. Conf.*, pp. 102–110, May 1986.

[4] C. Gebotys and M. Elmasry. Global optimization approach for architectural synthesis. *IEEE Trans. CAD*, 12(9):1266–1278, Sep 1993.

[5] C. Hwang and Y. Hsu. PLS: A scheduler for pipeline synthesis. *IEEE Trans. CAD*, 12(9):1279–1286, Sep 1993.

[6] M. Langevin and E. Cerny. A recursive technique for computing lower-bound performance of schedules. *ACM TODAES*, 1(4):443–456, Oct. 1996.

[7] M. Narasimhan. Exact scheduling techniques for high level synthesis, 1998. M.S. thesis, Louisiana State University.

[8] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley–Interscience, 1988.

[9] M. Rim and R. Jain. Lower bound performance estimation for the high-level synthesis scheduling problem. *IEEE Trans. CAD*, 13(4):451–458, April 1994.