Math. Prog. Comp. (2009) 1:1–41
DOI 10.1007/s12532-008-0001-1

FULL LENGTH PAPER

SCIP: solving constraint integer programs

Tobias Achterberg

Received: 6 October 2008 / Accepted: 8 December 2008 / Published online: 20 January 2009

Abstract Constraint integer programming (CIP) is a novel paradigm which integrates *constraint programming* (CP), *mixed integer programming* (MIP), and *satisfiability* (SAT) modeling and solving techniques. In this paper we discuss the software framework and solver SCIP (Solving Constraint Integer Programs), which is free for academic and non-commercial use and can be downloaded in source code. This paper gives an overview of the main design concepts of SCIP and how it can be used to solve constraint integer programs. To illustrate the performance and flexibility of SCIP, we apply it to two different problem classes. First, we consider mixed integer programming and show by computational experiments that SCIP is almost competitive to specialized commercial MIP solvers, even though SCIP supports the more general constraint integer programming paradigm. We develop new ingredients that improve current MIP solving technology. As a second application, we employ SCIP to solve chip design verification problems as they arise in the logic design of integrated circuits. This application goes far beyond traditional MIP solving, as it includes several highly non-linear constraints, which can be handled nicely within the constraint integer programming framework. We show anecdotally how the different solving techniques from MIP, CP, and SAT work together inside SCIP to deal with such constraint classes. Finally, experimental results show that our approach outperforms current state-of-the-art techniques for proving the validity of properties on circuits containing arithmetic.

Keywords Constraint programming · Integer programming · SAT

Mathematics Subject Classification (2000) Primary: 90C11; Secondary: 90-04 · 90-08

T. Achterberg (✉)
Zuse Institute Berlin, Berlin, Germany
e-mail: achterberg@zib.de

1 Introduction

SCIP is a software framework for *constraint integer programming* (CIP), a novel paradigm that integrates *constraint programming* (CP), *mixed integer programming* (MIP), and *satisfiability* (SAT) modeling and solving techniques. SCIP is freely available in source code for academic and non-commercial use and can be downloaded from [102].

Constraint integer programming is a generalization of MIP that allows for inclusion of arbitrary constraints that get reduced to linear constraints on continuous variables after all integer variables have been fixed. CIPs can be treated by a combination of techniques used to solve CPs, MIPs, and SAT problems: propagating the variables' domains by constraint specific algorithms, solving a linear programming (LP) relaxation of the problem, strengthening the LP by cutting plane separation, and analyzing infeasible subproblems to infer useful global knowledge about the problem instance.

The idea of combining modeling and solving techniques from CP and MIP is not new. Several authors showed that an integrated approach can help to solve optimization problems that were intractable with either of the two methods alone (see for instance, [33,60,95]). Other approaches to integrate general constraint and mixed integer programming into a single framework have been proposed in the literature as well, for example [10,17,32,58].

Our approach differs from the existing work in the level of integration. SCIP combines the CP, SAT, and MIP techniques on a very low level. In particular, all involved algorithms operate on a single search tree, which enables close interaction amongst these techniques. For example, MIP components can base their heuristic decisions on statistics that have been gathered by CP algorithms or vice versa, and both can use the dual information provided by the LP relaxation of the current subproblem. Furthermore, the SAT-like conflict analysis evaluates both the deductions discovered by CP techniques and the information obtained through the LP relaxation.

This paper is based on the dissertation of the author [2] and is organized as follows. Section 2 introduces the concept of constraint integer programming and shows that SAT, CP, and MIP are special cases of CIP. Section 3 provides an overview of the main design concepts of SCIP. Then, the paper illustrates the use of SCIP on two different problem classes.

We describe mixed integer programming as the first application of CIP in Sect. 4. On our test set, SCIP 1.1 is only 1.87 times slower than CPLEX 10.2 [59], even though CPLEX is a specialized commercial MIP solver and SCIP supports the more general constraint integer programming model. This performance is achieved by carefully implementing methods used for solving MIPs as described in the literature and by introducing new algorithms that improve the current state-of-the-art. The most important improvements are reviewed in this section and their effects are assessed by computational experiments. Then we compare the performance of SCIP with that of CPLEX and CBC [49].

As a second application, Sect. 5 shows how to employ SCIP to solve chip design verification problems as they arise in the logic design of integrated circuits. Although this problem class features a sizeable kernel of linear constraints that can be efficiently treated by MIP techniques, it also contains few non-linear constraints that are difficult to handle by MIP solvers. In this setting, the CIP approach is very effective: it can apply the sophisticated MIP machinery to the linear part of the problem, while it is still able to deal with the non-linear constraints outside the MIP kernel by employing constraint programming techniques. By giving examples of some of the chip verification specific plugins that are used on top of SCIP, we illustrate how such non-linear problems can be solved by using SCIP as a CIP framework. Experimental results show that our approach outperforms current state-of-the-art techniques for proving the validity of properties on circuits containing arithmetics.

The constraint integer programming paradigm is a step to extend the horizon of classical mathematical programming. The modular design of SCIP and the availability of the source code make it an ideal platform for further research in this direction. In addition, its good performance enables computational research and algorithm development in the area of mathematical programming to also consider interactions between new methods and existing state-of-the-art MIP components, which is essential for the further algorithmic advancement in MIP solving technology. It has to be noted that the latter can also be achieved by employing CBC [49] of the Coin- OR [38] project, which achieves a similar MIP solving performance as SCIP.

Apart from mixed integer programming and the chip design verification problem covered in this paper, SCIP has already been used in various other projects like branch-and-cut [63,87,90] and branch-and-price [37,86] applications, general MIP research [64,66], and other problem specific algorithms [11,29,31,44]. A very interesting topic was studied by Armbruster et al. [14–16], who used SCIP to combine LP and semidefinite relaxations to solve the minimum graph bisection problem within the branch-and-cut framework. Additionally, SCIP is also being used for teaching computational optimization, see Achterberg, Grötschel, and Koch [6].

2 Constraint integer programming

Most solvers for constraint programs, satisfiability problems, and mixed integer programs share the idea of dividing the problem into smaller subproblems and implicitly enumerating all potential solutions. They differ, however, in the way of processing the subproblems.

Because MIP is a very specific case of CP, MIP solvers can apply sophisticated problem specific algorithms that operate on the subproblem as a whole. In particular, they usually employ the simplex algorithm invented by Dantzig [41] to solve the LP relaxations and cutting plane separators like the Gomory cut separator [56].

In contrast, due to the unrestricted definition of CPs, CP solvers cannot take such a global perspective. They rely on constraint propagators, each of them exploiting the structure of a single constraint class. Usually, the only interaction between the individual constraints takes place via the variables' domains. An advantage of CP is, however, the possibility to model the given problem more directly, using very expressive constraints which contain a great deal of structure. Transforming such constraints into linear inequalities can conceal their structure from an MIP solver. Therefore, this restricts the solver's ability to draw valuable conclusions about the instance or to make the right decisions during the search.

SAT is also a very specific case of CP with only one type of constraints, namely Boolean clauses. Since the canonical LP relaxation of SAT is rather useless (if each clause has at least two literals, one can always satisfy the relaxation by setting all variables to 0.5), SAT solvers mainly exploit the special problem structure to speed up the domain propagation algorithm and to employ highly effective data structures.

The motivation behind combining CP, SAT, and MIP techniques is to combine their advantages and to compensate for their individual weaknesses. We propose the following slight restriction of a CP to specify our integrated approach [2,4]:

Definition 2.1 (Constraint integer program) A *constraint integer program* $(\mathfrak{C}, I, c)$ is defined as

$$\text{(CIP)} \quad c^\star = \min\{c^T x \mid \mathfrak{C}(x),^1 \; x \in \mathbb{R}^n, \; x_j \in \mathbb{Z} \text{ for all } j \in I\}$$

with a finite set $\mathfrak{C} = \{\mathcal{C}_1, \dots, \mathcal{C}_m\}$ of constraints $\mathcal{C}_i : \mathbb{R}^n \to \{0, 1\}$, $i = 1, \dots, m$, a subset $I \subseteq N = \{1, \dots, n\}$ of the variable index set, and an objective function vector $c \in \mathbb{R}^n$. A CIP has to fulfill the following condition:

$$\forall \hat{x}_I \in \mathbb{Z}^I \; \exists (A', b') : \{x_C \in \mathbb{R}^C \mid \mathfrak{C}(\hat{x}_I, x_C)\} = \{x_C \in \mathbb{R}^C \mid A' x_C \leq b'\} \tag{1}$$

with $C := N \setminus I$, $A' \in \mathbb{R}^{k \times C}$, and $b' \in \mathbb{R}^k$ for some $k \in \mathbb{Z}_{\geq 0}$.

Restriction (1) ensures that the subproblem obtained after fixing the integer variables is always a linear program. This means that for the case of finite domain integer variables, the problem can—in principle—be solved by enumerating all values of the integer variables and solving the corresponding LPs. Note that this does not forbid non-linear expressions. Only the remaining part after fixing (and thus eliminating) the integer variables must be linear in the continuous variables.

The linearity restriction of the objective function can be avoided easily by introducing an auxiliary objective variable z that is linked to the actual non-linear objective function with a non-linear constraint $z = f(x)$. We just demand a linear objective function in order to simplify the derivation of the LP relaxation:

Definition 2.2 (LP relaxation of a CIP) Given a constraint integer program $(\mathfrak{C}, I, c)$, a linear program

$$\text{(LP)} \quad \check{c} = \min\left\{c^T x \mid Ax \leq b, \; x \in \mathbb{R}^n\right\}$$

is called an *LP relaxation of* (*CIP*) if

$$\{x \in \mathbb{R}^n \mid Ax \leq b\} \supseteq \{x \in \mathbb{R}^n \mid \mathfrak{C}(x), x_j \in \mathbb{Z} \text{ for all } j \in I\}.$$

Using the above objective function transformation, every CP that meets Condition (1) can be represented as constraint integer program. In particular, we can observe the following [2]:

1 $\mathfrak{C}(x) :\Leftrightarrow \mathcal{C}_1(x) = \dots = \mathcal{C}_m(x) = 1$.

Proposition 2.3 *The notion of constraint integer programming generalizes finite domain constraint programming* (*CP*(*FD*))*, mixed integer programming* (*MIP*)*, and SAT solving*:

1. *Every CP*(*FD*) *can be stated as a CIP.*
2. *Every MIP can be stated as a CIP.*
3. *Every SAT problem can be stated as a CIP.*

3 SCIP design concepts

Clearly, in order to solve CIPs a new solver design is needed. In the following we explain the approach that we used for SCIP.

SCIP employs a branch-and-bound approach, which is used in all three areas, MIP, CP, and SAT. This is complemented by LP relaxations and cutting plane separators as they can be found in MIP, by constraint specific domain propagation algorithms as in CP and SAT, and by conflict analysis as in modern SAT solvers.

Branch-and-bound. The problem instance is successively divided into subproblems, usually by splitting the domain of a variable into two disjoint parts (branching), see Fig. 1. The dissection of a subproblem ends if it is infeasible, an optimal solution for the subproblem can be identified, or if it can be proven that no better solution than the currently best known solution can be contained in the subproblem (bounding). Most branching schemes found in the literature split a problem into two subproblems, usually by a disjunction on the domain of a single variable. However, SCIP also supports branching with a larger number of subproblems and branching on constraints.

LP relaxation. The LP relaxation of an MIP is obtained by dropping the integrality restrictions of the variables. It can be solved efficiently and provides a dual bound on the objective value which can be used for the bounding step in branch-and-bound. For

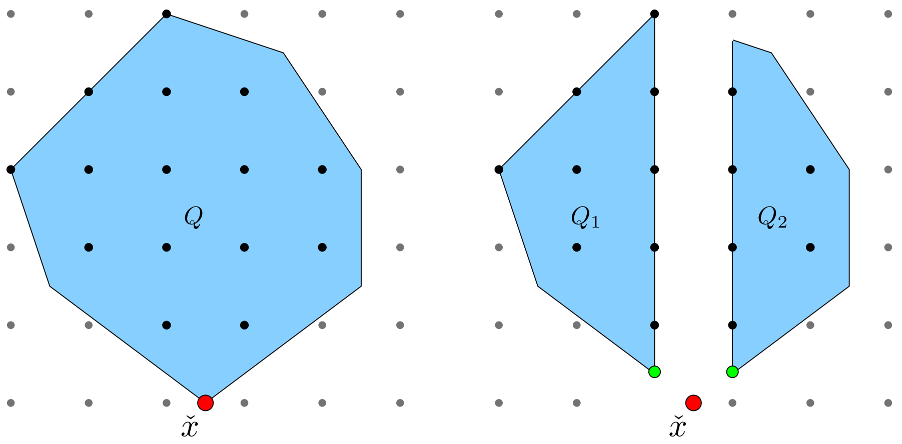

Fig. 1 LP based branching on a single fractional variable

a CIP, we use an LP relaxation that arises as the intersection of linear relaxations of the individual constraints in the problem instance. Note, however, that in SCIP the LP relaxation does not need to be solved at every node of the search tree. It can even be turned off completely, which turns SCIP into a pure constraint programming solver.

Cutting plane separation. After having solved the LP relaxation of a subproblem, it is possible to exploit the integrality restrictions in order to tighten the relaxation and thereby improve the bound obtained. This is achieved by adding linear inequalities that are valid for all integer feasible solutions of the problem but violated by the current LP solution, see Fig. 2. This approach of tightening a *relaxation* of a general problem, ideally until the convex hull of all feasible solutions has been obtained, can be seen as an extension to the approach of Crowder, Johnson, and Padberg [39].

In addition to linear cutting planes, in SCIP, it is also possible to tighten the subproblem formulation with general constraints. For example, Armbruster [14] extracts additional valid non-linear constraints during the solving of a semidefinite programming (SDP) relaxation to tighten the SDP relaxation of the current subproblem and its descendants.

Domain propagation. After having tightened a variable's domain in the branching step of branch-and-bound, domain propagation infers additional domain reductions on the variables by inspecting the individual constraints and the current domains of the involved variables. Figure 3 illustrates domain propagation on the example of the well-known ALLDIFF constraint, which demands that all variables of the constraint take pairwise different values.

Conflict analysis. Infeasible subproblems are analyzed to identify the reason of the infeasibility and to extract additional valid constraints. These can be applied later during the search to prune the search tree and to improve the subproblems' relaxations.

The core of SCIP is a framework that provides the infrastructure to implement very adjustable branch-and-bound based search algorithms with special support for

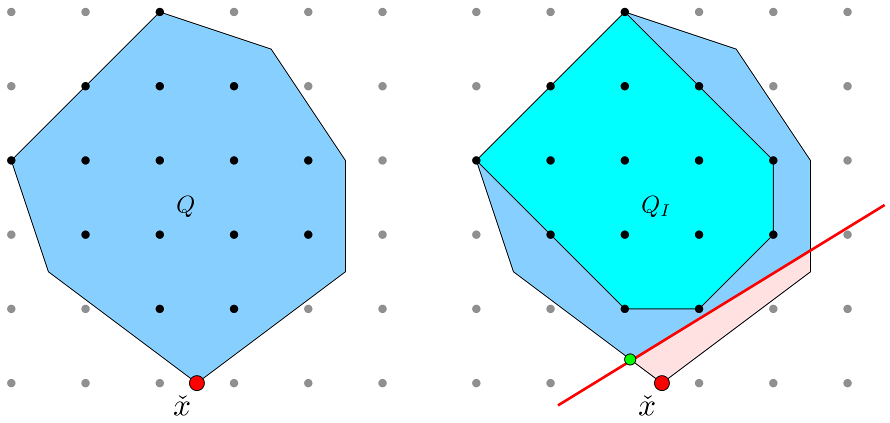

Fig. 2 A cutting plane separates the LP solution $\check{x}$ from the convex hull Q_I of integer points of Q

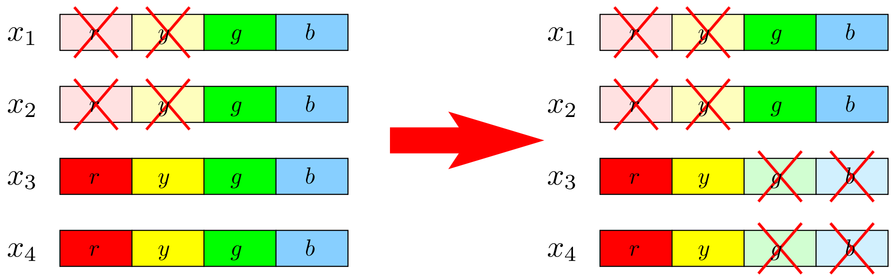

Fig. 3 Domain propagation on an ALLDIFF constraint. In the current subproblem on the *left hand side*, the values *red* and *yellow* are not available for variables x_1 and x_2 (for example, due to branching). The propagation algorithm detects that the values *green* and *blue* can be ruled out for the variables x_3 and x_4

LP relaxations. In addition to the infrastructure, SCIP includes a large library of default algorithms to control the search. These main algorithms are part of external *plugins*, which are user defined callback objects that interact with the framework through a very detailed interface. The different plugin types are described in Sect. 3.1. The infrastructure provided by the framework for communication and data sharing between the plugins is explained in Sect. 3.3. Finally, Sect. 3.4 discusses the advantages and limitations of this plugin approach.

3.1 User plugins

In the following, we describe the various types of user plugins that can enrich the basic CIP framework and explain their role in the solution process. As the *constraint handlers* are the most important plugin types, they are discussed in detail, while the other plugin types are only covered very briefly.

3.1.1 Constraint handlers

The main structure of CIPs is determined by constraints. Hence, the central objects of SCIP are *constraint handlers*. Each constraint handler represents the semantics of a single class of constraints and provides algorithms to handle constraints of the corresponding type.

The primary task of a constraint handler is to check the feasibility of a given solution with respect to all constraints of its type existing in the problem instance. This feasibility test suffices to turn SCIP into an algorithm which correctly solves CIPs with constraints of the supported type, at least if no continuous variables are involved. However, because the constraint handler would then behave like a feasibility oracle without providing additional information about the problem structure, the resulting procedure would be a complete enumeration of all potential solutions.

To improve the performance of the solution process, constraint handlers may provide additional algorithms and information about their constraints to the framework, namely

- presolving methods to simplify the constraint representation,
- propagation methods to tighten the variables' domains,
- a linear relaxation, which can be generated in advance or on the fly, that strengthens the LP relaxation of the problem, and
- branching decisions to split the problem into smaller subproblems, using structural knowledge of the constraints in order to generate a small search tree.

Example 3.1 (Knapsack constraint handler) A binary knapsack constraint is a specialization of a linear constraint

$$a^T x \leq \beta \tag{2}$$

with non-negative integral right hand side $\beta \in \mathbb{Z}_{\geq 0}$, non-negative integral coefficients $a_j \in \mathbb{Z}_{\geq 0}$, and binary variables $x_j \in \{0, 1\}$, $j \in N$. The knapsack constraint handler is available in the SCIP distribution at `src/scip/cons_knapsack.c`.

The feasibility test of the knapsack constraint handler is very simple: for each knapsack constraint it only adds up the coefficients a_j of variables x_j set to 1 in the given solution and compares the result with the right hand side β. Presolving algorithms for knapsack constraints include modifying the coefficients and right hand side in order to tighten the LP relaxation, and fixing variables with $a_j > \beta$ to 0, see Savelsbergh [93] and Achterberg [2] for details.

The propagation method fixes additional variables to 0, that would not fit into the knapsack together with the variables that are already fixed to 1 in the current subproblem.

The linear relaxation of the knapsack constraint initially consists of the knapsack inequality (2) itself. Additional cutting planes like lifted cover cuts (see, for example, [21,23] or [80]) or GUB cover cuts (see Wolsey [98]) are dynamically generated to enrich the knapsack's relaxation and to cut off the current LP solution.

Example 3.2 (NOSUBTOUR constraint handler) The symmetric traveling salesman problem (TSP) on a graph $G = (V, E)$ with edge lengths $c_{uv} \in \mathbb{R}_{\geq 0}$, $uv \in E$, can be stated as a constraint integer program in the following way:

$$\begin{aligned}
\min \quad & \sum_{uv \in E} c_{uv}\, x_{uv} \\
\text{s.t.} \quad & \sum_{u \in \delta(v)} x_{uv} = 2 && \text{for all } v \in V && (3)\\
& \textsc{nosubtour}(G, x) && && (4)\\
& x_{uv} \in \{0, 1\} && \text{for all } uv \in E. && (5)
\end{aligned}$$

Formally, this model consists of $|V|$ degree constraints (3), one NOSUBTOUR constraint (4), and $|E|$ integrality constraints (5). The NOSUBTOUR constraint is a non-linear constraint which is defined as

$$\textsc{nosubtour}(G, x) \;\Leftrightarrow\; \nexists C \subseteq \{uv \in E \mid x_{uv} = 1\} : C \text{ is a cycle of length } |C| < |V|.$$

This constraint must be supported by a constraint handler, which for a given integral solution $x \in \{0,1\}^E$ checks whether the corresponding set of edges contains a subtour C. The linear relaxation of the NOSUBTOUR constraint consists of exponentially many *subtour elimination inequalities*

$$\sum_{uv \in E(S)} x_{uv} \leq |S| - 1 \text{ for all } S \subset V \text{ with } 2 \leq |S| \leq |V| - 2,$$

with $E(S) = \{uv \in E \mid u, v \in S\}$, which can be separated and added on demand to the LP relaxation. Additionally, the constraint handler could separate various other classes of valid inequalities for the traveling salesman problem that can be found in the literature, see Applegate et al. [13] and the references therein. The SCIP distribution contains a NOSUBTOUR constraint handler as C++ example at `examples/TSP/src/Cons-hdlrNosubtour.cpp`.

3.1.2 Variable pricers

Several optimization problems are modeled with a huge number of variables, e.g., with each path in a graph or each subset of a given set corresponding to a single variable. In this case, the full set of variables cannot be generated in advance. Instead, the variables are added dynamically to the problem whenever they may improve the current solution. In mixed integer programming, this technique is called *column generation*.

SCIP supports dynamic variable creation by *variable pricers*. They are called during the subproblem processing and have to generate additional variables that reduce the lower bound of the subproblem. If they operate on the LP relaxation, they would usually calculate the reduced costs of the not yet existing variables with a problem specific algorithm and add some or all of the variables with negative reduced costs. Note that since variable pricers are part of the model, they are always problem class specific. Therefore, SCIP does not contain any "default" variable pricers.

3.1.3 Presolvers

In addition to the constraint based (primal) presolving mechanisms provided by the individual constraint handlers, additional presolving algorithms can be applied with the help of *presolvers*, which interact with the whole set of constraints. They may, for example, perform dual presolving reductions which take the objective function into account.

A particular example of such a presolver is the *probing* [93] plugin, which is located at `src/scip/presol_probing.c` in the SCIP distribution. Probing is a very time-consuming preprocessing technique which consists of successively fixing each binary variable to zero and one and evaluating the corresponding subproblems by domain propagation techniques. The analysis of the two subproblems can identify variable fixings, aggregations, tightened bounds, and valid implications between variables. The latter are stored in an implication graph and exploited during the solution process, e.g., in other preprocessing algorithms or in the branching variable selection.

Note that the domain propagation itself is conducted by calling the domain propagation methods of the constraint handlers. In this way, the probing plugin can exploit the structure of the constraints without having explicit knowledge about their semantics.

3.1.4 Cut separators

In SCIP, we distinguish between two different types of cutting planes. The first type are the constraint based cutting planes, that are valid inequalities or even facets of the polyhedron described by a single constraint or a subset of the constraints of a single constraint class. They are generated by the constraint handlers of the corresponding constraint types. Prominent examples are the different types of knapsack cuts that are generated in the knapsack constraint handler, see Example 3.1, or the cuts for the traveling salesman problem like subtour elimination and comb inequalities which can be separated by the NOSUBTOUR constraint handler, see Example 3.2.

The second type of cutting planes are general purpose cuts, which use the current LP relaxation and the integrality restrictions to generate valid inequalities. Generating those cuts is the task of *cut separators*. Examples are Gomory fractional and Gomory mixed integer cuts (Gomory [56]), complemented mixed integer rounding cuts (Marchand and Wolsey [76]), and strong Chvátal-Gomory cuts (Letchford and Lodi [68]). See Wolter [99] for a detailed discussion of cutting plane separators in SCIP.

3.1.5 Primal heuristics

Feasible solutions can be found in two different ways during the traversal of the branching tree. On one hand, the solution of a node's relaxation may turn out to be feasible with respect to all of the constraints including integrality restrictions. On the other hand, feasible solutions can be discovered by *primal heuristics*, which are called periodically during the search.

SCIP provides specific infrastructure for diving and probing heuristics. *Diving heuristics* iteratively resolve the LP after making a few changes to the current subproblem, usually aiming at driving the fractional values of integer variables to integrality. *Probing heuristics* are even more sophisticated. Besides solving LP relaxations, they may call the domain propagation algorithms of the constraint handlers after applying changes to the variables' domains, and they can undo these changes using backtracking.

Other heuristics without special support in SCIP include local search heuristics like *tabu search* [54], *rounding heuristics* which try to round the current fractional LP solution to a feasible integral solution, and *improvement heuristics* like *local branching* [48] or RINS [40], which try to generate improved solutions by inspecting one or more of the feasible solutions that have already been found. See Berthold [26] for a detailed discussion of primal heuristics in SCIP.

3.1.6 Other plugin types

Other types of plugins that can be added to SCIP are as follows.

Domain propagators complement the constraint based domain propagation of the constraint handlers by dual propagations, i.e., propagations that can be applied due to the objective function and the currently best known primal solution.

If none of the constraint handlers performed a "structure exploiting" dissection of the search space (like, for example, the Ryan–Foster branching rule for set partitioning constraints [91]), the *branching rules* are called to split the problem into two or more subproblems, usually by applying a dichotomy on an integer variable with fractional value in the solution of the current LP relaxation. The well-known *most infeasible*, *pseudocost*, *reliability*, and *strong branching* rules are examples of this type (see Achterberg, Martin, and Koch [7]).

Node selectors decide which of the leaves in the current branching tree is selected as the next subproblem to be processed. This choice can have a large impact on the solver's performance, because it influences the finding of feasible solutions and the development of the global dual bound. Common examples for node selection are *depth first*, *best first*, and *best estimate* search.

In addition to the native support of the LP relaxation, it is also possible to include other relaxations, e.g., Lagrange relaxations or semidefinite relaxations. This is possible through *relaxation handler* plugins. The relaxation handler manages the necessary data structures and calls the relaxation solver to generate dual bounds and primal solution candidates.

Event handlers process events that are generated by SCIP, for example, if a bound of a variable has been changed. Usually, a constraint handler closely interacts with an event handler in order to improve its own runtime performance.

Conflict handlers take care of conflict sets that are produced by the conflict analysis and turn them into constraints. *File readers* parse input data (like `.mps`, `.lp`, or ZIMPL [65] files) and produce a CIP model. *Dialog handlers* can be used to introduce additional dialog options in SCIP's interactive shell. *Display columns* allow for user generated output in the columns of the node log. The output of SCIP is passed through *message handlers*, which easily allows for redirection or suppression of output.

3.2 Solution process

Figure 4 sketches the solution process as it is performed in SCIP. After the user has specified his problem instance in terms of variables and constraints, the presolver plugins and the presolving methods of the constraint handlers are called to simplify the problem instance. Then, the actual search commences by creating the root node of the search tree and by selecting the root as the first node to be processed. Primal heuristics are called at various places in the solving loop, and each heuristic plugin can specify when it should be called. For example, heuristics like farthest insert for the TSP that do not depend on an LP solution can be called before the LP relaxation has been solved. Very fast heuristics like rounding are best to be called inside the cutting plane loop, whereas more time consuming heuristics like diving should only be called once at the end of the node processing cycle.

After a node has been selected and applicable heuristics have been called, domain propagation is applied by calling the domain propagation plugins and the domain propagation methods of the constraint handlers. If specified in the parameter settings,

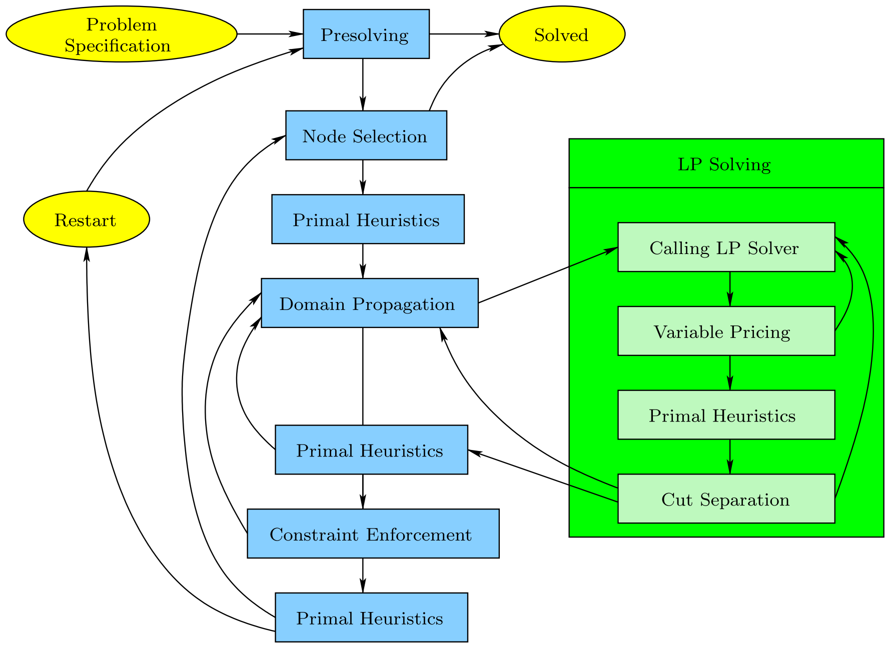

Fig. 4 Flow chart of the solution process in SCIP

the next step is to solve one or more relaxations of the problem, with the LP relaxation being directly supported by the framework.

The LP solving loop consists of an inner pricing loop in which the pricer plugins produce additional variables, and an outer cutting plane loop in which the cut separators and the cut callbacks of the constraint handlers add cutting planes to the LP relaxation. Cutting plane separators, in particular reduced cost fixing, can tighten the bounds of the variables, which triggers another call to the domain propagators in order to infer further domain reductions.

Eventually, no more improvements of the relaxation can be found and the constraint enforcement is executed. If the relaxation of the current node became infeasible during the process, the node can be pruned and another node is selected from the search tree for processing. Otherwise, the constraint handlers have to check the final solution of the relaxation for feasibility. If the solution is feasible for all constraints, a new incumbent has been found and the node can be pruned. Otherwise, the constraint handlers have the options to add further cutting planes or domain reductions, or to conduct a branching. In particular, if there are integer variables with fractional LP value, the integrality constraint handler calls the branching rule plugins to split the problem into subproblems. Finally, a new unprocessed node is selected from the search tree by the current node selector plugin and the process is iterated. If no unprocessed node is left, the algorithm terminates.

After processing a node, there is also the option to trigger a restart, which is an idea originating from the SAT community. Restarting means to exploit collected knowledge like incumbents, cutting planes, and variable fixings in a subsequent repeated

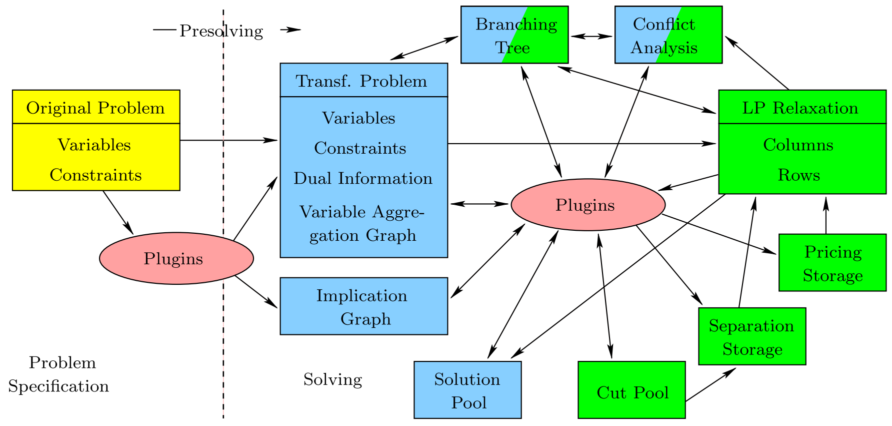

Fig. 5 Infrastructure provided by SCIP. The *arrows* denote the data flow between the components

presolving step and to restart the tree search from scratch. Our experiments with SCIP indicate that in the MIP context, restarts should only be applied at the root node, and only if a certain fraction of the variables have been fixed while processing the root node. This is in contrast to SAT solvers, which usually perform periodic restarts throughout the whole solution process.

3.3 Infrastructure

SCIP provides all necessary infrastructure to implement branch-and-bound based algorithms to solve CIPs. It manages the branching tree along with all subproblem data, automatically updates the LP relaxations, and handles all necessary transformations due to the preprocessing problem modifications. Additionally, a cut pool, pricing and separation storage management, and a SAT-like conflict analysis mechanism are available.

Figure 5 gives a rough sketch of the different components of SCIP and how they interact with each other and with the external plugins. The problem information is represented in three different parts of the diagram. Initially, the user states the CIP instance as *original problem*. The constraint handler and presolver plugins generate the *transformed problem*, which is an equivalent but usually more compact and smaller formulation of the problem instance. Feasible solutions for the instance—i.e., value assignments for the variables such that all constraints are satisfied—are stored in the *solution pool*. Optionally, an *implication graph* and a *clique table* can be associated to the transformed problem.

The third representation of the problem instance is only a partial representation, namely the LP relaxation. It is populated via intermediate storage components, the *pricing storage* and the *separation storage*. Additionally, the *cut pool* can store valid inequalities that can be added on demand to the LP through the separation storage. The *branching tree* and *conflict analysis* components operate on both representations, the CIP model of the transformed problem and the LP relaxation. The user *plugins* can access all of the components, although the LP relaxation can only be modified through the pricing and separation storages.

The infrastructure depicted above provides a communication interface between the plugins. Most notably, the plugins can communicate through the variables' bounds and the LP relaxation, which, together with the integrality information of the variables, form an MIP relaxation of the actual constraint integer program. Because this MIP relaxation is managed by the framework and accessible via interface methods, it can be used by problem-independent plugins even though they do not know the structure of the underlying CIP. Consequently, if the MIP relaxation is a good approximation of the CIP model at hand, standard MIP plugins like primal heuristics or cutting plane separators that are already available in SCIP can be effectively used to improve the CIP solution process.

3.4 Advantages and limitations of plugin approach

The plugin concept of SCIP facilitates the implementation of self-contained solver components that can be employed by a user to solve his particular constraint integer programming model. For example, if some user implemented a constraint handler for a certain class of constraints, he could make this plugin publicly available in order to enable all other SCIP users to use such constraints in their CIP models. Since all MIP specific components that come with the SCIP distribution are implemented as plugins, more general CIP applications can immediately benefit from these components.

Such a modular code design has the consequence that all communication between the plugins has to pass through a unified interface defined by the framework. In SCIP, this communication interface is mainly based on the MIP relaxation of the CIP, see Sect. 3.3. The actual semantics of the various (potentially non-linear) constraints in the model is hidden from all plugins except the responsible constraint handler.

An important aspect of this information hiding is the loss of the dual view, i.e., the column-based representation of the problem. By looking at the columns of the constraint system $Ax \leq b$, a typical MIP solver knows exactly how the feasibility of the constraints is affected if the value of a particular variable changes. Such data is, for instance, used for dual presolving reductions like the identification of parallel and dominated columns and for symmetry detection.

Therefore, SCIP suffers from some handicaps when compared to specialized MIP solvers. To attenuate this drawback, SCIP demands from the constraint handlers to provide a limited amount of dual information to the framework, namely the number of constraints that may block the increase or decrease of each variable. This information suffices to enable the most important dual presolving operations like dual fixing and dual bound reduction and helps to guide primal heuristics and branching, see Achterberg [2].

Additionally, the constraint handler approach itself yields some remedy for the issues that arise from decomposing the problem formulation into individual constraints. Other branch-and-cut frameworks like ABACUS [94] treat each individual constraint as an isolated object. In contrast, a constraint handler in SCIP, which manages *all* constraints of a certain type, can still perform operations on multiple

Table 1 Source code statistics of SCIP 1.1.0

Component	Lines of code	Number	Component	Lines of code	Number
Total	275640		LP solver interfaces	25067	8
Framework	130705		C++ wrapper classes	7600	18
Plugins	112268	160			
Branching rules	3145	8	File readers	11767	12
Constraint handlers	45687	18	Heuristics	17449	24
Cut separators	24096	10	Node selectors	1790	5
Dialog handlers	3319	46	Presolvers	2599	6
Display columns	1078	29	Propagators	1338	2

constraints. In particular, if the limited dual information reveal that some variables are contained only in a single class of constraints, then the corresponding constraint handler can apply dual presolving methods on these variables. Some dual methods are incorporated into the linear constraint handler. Others, like symmetry detection and parallel column aggregation are still under development.

Another drawback of the plugin approach of SCIP is its memory footprint and poor locality. An MIP solver can store the whole problem matrix A in a single memory block. In contrast, the CIP problem data in SCIP is stored locally in the constraint handlers, which means that the data are distributed across the memory address range. This usually leads to a degradation in the cache usage of the CPU and consequently to a performance loss. Additionally, since SCIP manages an MIP relaxation of the problem and additionally employs a black-box LP solver as sub-algorithm to manage the LP relaxation, most of the problem data is copied several times. This yields a significant increase in the memory consumption. Moreover, treating the LP solver as a black box and passing information through an interface layer abandons the opportunity for run-time improvements, namely to take certain short cuts and to exploit integrality during the LP solves.

3.5 Technical remarks

SCIP is implemented in C and is being developed at the Zuse Institute Berlin (ZIB) since 2001. It is the successor of the mixed integer programming solver SIP of Alexander Martin [79] and adopts several ideas and algorithms of its predecessor. It was implemented from scratch in order to obtain a much more flexible design and to support the constraint integer programming paradigm.

Table 1 shows a breakdown of the SCIP 1.1 source code. It has 275 640 lines of code with roughly 50% of the source code lines in the framework. The rest of the code is distributed over 160 plugins, 8 LP solver interfaces, and 18 C++ wrapper classes for the different plugin types. The LP interfaces support CLP [50], CPLEX [59], MOSEK [83], SOPLEX 1.2, SOPLEX 1.3, SOPLEX 1.4 [100], and XPRESS [43]. Additionally, there is an empty LP solver interface which is used when the user does not want to employ LP relaxations.

Roughly 18% of the lines in the source code are comments, and the code contains 16 446 assertions, which amounts to 20% of the total number of code statements. As to the author's knowledge, this constitutes a fairly large amount of comments and assertions compared to other software projects. The hope is that this helps for using and understanding the code, and to ease further development and maintenance of the software.

SCIP compiles and runs on various architectures and compilers, including Linux, Windows, SunOS, MacOS, and AIX. It supports both 32 and 64 bit platforms. The default plugins allow to read in various data formats, such as the MIP `.mps` and `.lp` formats, the SAT `.cnf` format and the pseudo-Boolean `.opb` format. Most notably, it also has a direct interface to ZIMPL [65] in order to parse `.zpl` files without having to generate intermediate `.lp` or `.mps` files. SCIP includes a DOXYGEN documentation, which can be generated from the source code or accessed at [102].

4 Advances in mixed integer programming

Integer programming and mixed integer programming emerged in the late 1950's and early 1960's when researchers realized that the ability to solve mixed integer programming models would have great impact for practical applications (see [42,77]).

A mixed integer program is defined as

$$\text{(MIP)} \qquad c^{\star} = \min \left\{ c^T x \mid Ax \leq b,\ x \in \mathbb{R}^n,\ x_j \in \mathbb{Z} \text{ for all } j \in I \right\},$$

with $A \in \mathbb{R}^{m \times n}$, $b \in \mathbb{R}^m$, $c \in \mathbb{R}^n$, and a subset $I \subseteq N = \{1, \ldots, n\}$. Today's most successful solvers apply the branch-and-cut algorithm, which is a combination of LP based branch-and-bound and cutting plane separation. SCIP takes the same approach and contains many of the ingredients found in the literature. In this section, we focus on the new methods that have been developed in [2] and which improve the MIP performance of SCIP. Improving the MIP solving technology within SCIP is particularly interesting, because it immediately enables more involved applications like branch-and-cut or general CIP solving to benefit from the advancements.

4.1 Branching

Since branching is in the core of any branch-and-bound algorithm, finding good strategies was important to practical MIP solving right from the beginning, see [25,81]. For a comprehensive study of branch-and-bound strategies we refer to [51,67,72] and the references therein.

SCIP implements multiple branching rules from the literature, namely *most infeasible branching*, *pseudocost branching*, *strong branching*, *hybrid strong/pseudocost branching*, and *pseudocost branching with strong branching initialization*, see [2,7] for details. SIP and its successor SCIP employ a new technique called *reliability branching* [7], which is a generalization of *pseudocost branching with strong*

Table 2 Mixed integer programming test sets

Test set	Type	Size	Problem class	Ref.	Origin
MIPLIB	Mixed	30	Mixed	[8]	http://miplib.zib.de
CORAL	Mixed	38	Mixed	[71]	http://coral.ie.lehigh.edu/mip-instances/
MILP	Mixed	37	Mixed		http://plato.asu.edu/ftp/milp/
ENLIGHT	IP	7	Combinatorial game		http://miplib.zib.de/contrib/AdrianZymolka/
ALU	IP	25	Infeasible chip verification	[1]	http://miplib.zib.de/contrib/ALU/
FCTP	MBP	16	Fixed charge transportation	[57]	http://plato.asu.edu/ftp/fctp/
ACC	BP	7	Sports scheduling	[85]	http://www.ps.uni-sb.de/~walser/acc/acc.html
FC	MBP	20	Fixed charge network flow	[18]	http://www.ieor.berkeley.edu/~atamturk/data/
ARCSET	IP/MIP	23	Capacitated network design	[20]	http://www.ieor.berkeley.edu/~atamturk/data/
MIK	MIP	41	Mixed integer knapsack	[19]	http://www.ieor.berkeley.edu/~atamturk/data/

branching initialization. Additionally, SCIP features an *inference branching* rule, which is described in the following.

Classical MIP branching rules like *pseudocost branching*, *strong branching*, and their hybrid variants mentioned above base the branching decision on the change in the objective value of the relaxation. In a constraint satisfaction problem or in a SAT, however, there is no objective function, and these rules do not make sense. One idea, which is employed in the SAT solver SATZ [69,70], is to evaluate the branching decisions in a strong branching fashion, but instead of looking at the objective value change one considers the number of additional domain reductions that are triggered by the branching. In contrast, the *inference branching* rule of SCIP keeps track of the domain reductions in a history similar to the pseudo costs. Then, it selects a variable with largest estimated number of deductions. The *hybrid reliability/inference branching* rule combines the selection criteria of *reliability branching* and *inference branching*, and augments this with the so-called *variable state independent decaying sum* rule as it is used in SAT solvers based on conflict analysis [84], see [2] for details.

Table 2 shows the test sets that we used for evaluating the MIP components of SCIP. The benchmarks have been conducted with SCIP 0.90f on a 3.8 GHz Intel P4 with 1 MB cache, using a time limit of 3,600 s. Further details on the instance selection and the testing methodology can be found in [2].

A summary of computational results for various branching rules is given in Table 3. As one can see, our new *hybrid reliability/inference branching* rule gives an overall speed-up of 8% compared to the previous state-of-the-art *pseudocost branching with strong branching initialization*,[2] ranging from a deterioration of 5% on the FC test set to an improvement of 55% on the ALU instances. The 8% speedup is the result of a 27% reduction in the number of branching nodes. Note that in this regard the *full strong branching* and *strong branching* rules are the most efficient schemes, but this node reduction cannot compensate for the amount of work that they spent on each branching decision.

[2] Using the *hybrid reliability/inference branching* results as 100% base line.

Table 3 Performance effect of different branching rules for solving MIP instances

	Test set	Random	Most inf	Least inf	Pseudocost	Full strong	Strong	Hybr strong	Psc strinit	Reliability	Inference
	MIPLIB	+139	+139	+266	+16	+92	+38	+20	+5	−1	+101
	CORAL	+332	+314	+575	+40	+97	+59	+27	+2	+7	+177
	MILP	+81	+86	+109	+23	+107	+44	+43	+9	+6	+20
	ENLIGHT	+115	−40	+149	−27	+45	+11	+9	+27	+5	−70
Time	ALU	+1271	+1991	+1891	+619	+180	+13	+11	+55	+36	−35
	FCTP	+288	+267	+379	+35	+36	+25	+4	+14	+2	+187
	ACC	+52	+85	+138	−41	+174	+82	+153	+11	+84	−24
	FC	+912	+1152	+837	+98	+14	+18	+14	−5	−2	+188
	ARCSET	+1276	+1114	+1296	+106	+112	+72	+38	+18	−1	+317
	MIK	+10606	+10606	+9009	+102	+59	+8	+11	+35	+2	+5841
	Total	**+226**	**+219**	**+341**	**+33**	**+95**	**+44**	**+30**	**+8**	**+6**	**+95**
	Nodes total	**+543**	**+428**	**+976**	**+98**	**−75**	**−65**	**+3**	**+27**	**+9**	**+217**

The values denote the percental changes in the shifted geometric mean of the runtime compared to the default *hybrid reliability/inference branching* rule. Positive values represent a deterioration, negative values an improvement. The "total" line gives a weighted mean over all test sets. For reasons of comparison, we also include the "nodes total", which shows the percental changes in the number of branching nodes over all test sets

As it turns out, we can achieve an even larger improvement by altering the *score function* to combine the estimated impact q_j^- in the downwards branch of a variable x_j with the upwards estimate q_j^+. For example, if we estimate the downwards objective change for x_1 to be $q_1^- = 2$ and the upwards objective change to be $q_1^+ = 10$, and the estimates for x_2 are $q_2^- = 4$ and $q_2^+ = 6$, respectively, which variable is the more promising candidate? In the literature, one can find a weighted sum score function of the type

$$\text{score}(q^-, q^+) = (1 - \mu) \cdot \min\{q^-, q^+\} + \mu \cdot \max\{q^-, q^+\}$$

with $\mu \in [0, 1]$ being a parameter. Bénichou et al. [25] and Beale [24] propose to use $\mu = 0$, while Gauthier and Ribière [53] employ $\mu = \frac{1}{2}$. However, as can be seen in Table 4, the best value for μ in SCIP is located between 0 and $\frac{1}{2}$, which is also reported by Linderoth and Savelsbergh [72]. They found $\mu = \frac{1}{3}$ to be a good value, while for SCIP a value of $\mu = \frac{1}{6}$ is superior. Nevertheless, none of the weighted sum score functions is able to reach the performance of the product score function

$$\text{score}(q^-, q^+) = \max\{q^-, \epsilon\} \cdot \max\{q^+, \epsilon\}, \quad \epsilon = 10^{-6}. \tag{6}$$

which is, as to the author's knowledge, a new idea that has not been proposed previously. Using the product score function improves the overall MIP performance of SCIP by 14%.

4.2 Cutting planes

SCIP separates knapsack cover cuts [39], complemented mixed integer rounding cuts [75], Gomory mixed integer cuts [22,55], strong Chvátal-Gomory cuts [68],

Table 4 Performance effect of different branching score functions for solving MIP instances

	Test set	Min ($\mu = 0$)	Weighted ($\mu = \frac{1}{6}$)	Weighted ($\mu = \frac{1}{3}$)	Avg ($\mu = \frac{1}{2}$)	Max ($\mu = 1$)
Time	MIPLIB	+26	+12	+28	+30	+53
	CORAL	+25	+20	+27	+49	+113
	MILP	+18	+10	+36	+35	+58
	ENLIGHT	+34	−19	−1	−9	+115
	ALU	+88	+66	+85	+101	+187
	FCTP	+72	−2	+6	+26	+56
	ACC	+43	+70	+29	+41	+50
	FC	+58	−7	−6	−4	−2
	ARCSET	+35	+11	+22	+32	+62
	MIK	+134	+13	+31	+52	+169
	Total	**+29**	**+14**	**+29**	**+37**	**+75**
	Nodes total	**+31**	**+34**	**+54**	**+76**	**+130**

The values denote the percental changes in the shifted geometric mean of the runtime compared to the default product score function (6). Positive values represent a deterioration, negative values an improvement. The "total" line gives a weighted mean over all test sets. For reasons of comparison, we also include the "nodes total", which shows the percental changes in the number of branching nodes over all test sets

flow cover cuts [88,97], implied bound cuts [93], and clique cuts [62,93]. In the default settings, SCIP only separates cuts at the root node. For implementation details, see [2,99].

In addition to implementations of the mentioned cutting plane separators, SCIP features a newly developed ingredient, namely a cut selection algorithm based on *efficacy*, *orthogonality*, and *objective parallelism*. The first two concepts have already been used in [12] for cut selection, but only within a single cutting plane class. In contrast, SCIP performs a cut selection among the generated cutting planes of all cut separators.

Given the current solution $\check{x}$ of the LP relaxation, the idea of the cut selection is that we want to add cuts $r : d_r^T x \leq \gamma_r$ with large efficacy $e_r := (d_r^T \check{x} - \gamma_r)/\|d_r\|$ and large objective parallelism $p_r := |d_r^T c|/(\|d_r\| \|c\|)$. Among each other, the cuts should have a large orthogonality $o_{rq} := 1 - |d_r^T d_q|/(\|d_r\| \|d_q\|)$. In an initial step, the cut selection algorithm calculates the efficacy e_r and objective parallelism p_r for each cutting plane candidate r, and assumes a perfect minimal orthogonality $o_r = 1$ to obtain an initial score

$$s_r := w_e e_r + w_p p_r + w_o o_r$$

with $w_e = 1$, $w_p = 0.1$, and $w_o = 1$ being the default settings for the weights of the addends. Then, it successively adds a cut $r^\star$ of maximal score $s_{r^\star}$ to the LP and updates the orthogonality $o_r := \min\{o_r, o_{rr^\star}\}$ and score s_r for each of the remaining cut candidates r. In addition, cuts with too small efficacy ($e_r < \textsf{mineffi} = 0.01$) or orthogonality ($o_r < \textsf{minortho} = 0.5$) are removed from the cut candidate set.

Table 5 shows that the two simple strategies to add only the deepest cut of each separation round to the LP and to add *all* violated cuts are clearly inferior to the more sophisticated selection rule of SCIP.

Table 5 Performance effect of different cutting plane selection strategies for solving MIP instances

	Test set	One per round	Take all	No obj paral	No ortho
Time	MIPLIB	+97	+105	+4	+28
	CORAL	+98	+38	−8	+19
	MILP	+27	+45	−7	+11
	ENLIGHT	+85	−17	−3	+8
	ALU	−3	+1	+18	−16
	FCTP	+7	+69	+6	+17
	ACC	+1853	+351	−22	+313
	FC	+80	+694	+7	+102
	ARCSET	+35	+49	+6	+1
	MIK	+247	+676	+7	+8
	Total	**+79**	**+71**	**−3**	**+22**
	Nodes total	**+11**	**−5**	**−5**	**−8**

The values denote the percental changes in the shifted geometric mean of the runtime compared to the default cutting plane selection of SCIP, which considers efficacy, orthogonality, and objective parallelism. Positive values represent a deterioration, negative values an improvement. The "total" line gives a weighted mean over all test sets. For reasons of comparison, we also include the "nodes total", which shows the percental changes in the number of branching nodes over all test sets

Additionally, one can see that considering orthogonality improves the overall performance by 22%, even though the number of branching nodes increases by 8%. On the other hand, the idea of including the objective parallelism seems to be ineffective.

4.3 Conflict analysis

One of the key ingredients in modern SAT solvers is *conflict analysis* [78]: infeasible subproblems that are encountered during branch-and-bound are analyzed in order to learn deduced clauses that can later be used to prune other nodes of the search tree. In addition, these conflict clauses enable the solver to perform so-called *non-chronological backtracking* [78].

SCIP features a generalization of conflict analysis to constraint integer programming, which includes mixed integer programming. The details can be found in [1,2].

Table 6 shows the effect of conflict analysis if used with different aggressiveness. Note that in SCIP 0.90f, which was used in this test, conflict analysis was not enabled by default. Thus, negative values in the table reflect an improvement due to conflict analysis.

In the "prop" column, we only apply conflict analysis on propagation conflicts, i.e., we do not analyze infeasible or bound exceeding LPs. The two columns "prop/inflp" and "prop/inflp/age" extend this by the infeasible LP analysis, with the latter adding an aging strategy to get rid of useless conflict constraints. The "prop/lp" column also applies conflict analysis on LPs that exceed the objective cutoff, and the "all" and "full" columns even extract conflicts from infeasible or bound exceeding strong branching sub-LPs, with "full" being more aggressive in reducing the initial conflict set for bound exceeding LPs. As one can see, with a well-balanced aggressiveness

Table 6 Performance effect of different variants of conflict analysis for solving MIP instances

	Test set	Prop	Prop/inflp	Prop/inflp/age	Prop/lp	All	Full
Time	MIPLIB	0	+6	0	+12	+14	+24
	CORAL	−10	−1	−7	+2	+2	+8
	MILP	−13	−26	−28	−31	−24	−18
	ENLIGHT	−11	−21	−49	−21	−26	−26
	ALU	−28	−35	−11	−39	−39	−38
	FCTP	+1	+2	+1	+23	+20	+41
	ACC	+19	+21	+17	−2	−19	+5
	FC	+1	+1	−1	+9	+9	+18
	ARCSET	+1	−6	−6	−4	−1	+1
	MIK	+7	+4	+7	−11	−15	−17
	Total	**−7**	**−8**	**−12**	**−8**	**−6**	**+1**
	Nodes total	**−14**	**−17**	**−28**	**−31**	**−32**	**−36**

The values denote the percental changes in the shifted geometric mean of the runtime compared to disabling conflict analysis. Positive values represent a deterioration, negative values an improvement. The "total" line gives a weighted mean over all test sets. For reasons of comparison, we also include the "nodes total", which shows the percental changes in the number of branching nodes over all test sets

conflict analysis can reduce the overall runtime by 12%. On the ENLIGHT instances, the reduction is almost 50%, which is a speed-up by a factor of 2.

4.4 Further improvements

The previous sections covered some of the newly developed ingredients in SCIP that improve the state-of-the-art in MIP solving. Further improvements have been made in node selection, child selection, in primal heuristics [3], and by applying restarts at the root node (see Sect. 3.2). Within SCIP, these components lead to a speed-up of 4, 9, 7, and 8%, respectively. For details, see [2].

4.5 Comparison to CPLEX and CBC

In order to evaluate the overall performance of SCIP on MIP instances, we compare it to the specialized MIP solvers CBC 2.2, CPLEX 11.0, and CPLEX 10.2. Table 7 shows detailed results on instances obtained from MIPLIB 3, MIPLIB 2003, and Hans Mittelmann's web-site [82] as it was in March 2008. We applied CPLEX 11 [59] and CBC 2.2 [49] with CLP 1.8 [50] on all of the 159 instances in these sets and removed instances that none of the two solvers could solve to optimality within the time limit of 3,600 s on an Intel Pentium D processor with 3.4 GHz and 2 MB cache. Additionally, we also excluded the instances `enigma`, `lrn`, `markshare1_1`, `markshare2_1`, `mitre`, and `timtab1`, because the six tested solvers did not agree on a common optimal objective value. The resulting test set consists of 122 instances.

The table lists the results of the six different solvers that we consider, namely CPLEX 11.0, CPLEX 10.2, CBC 2.2, and SCIP 1.1 linked to the LP solvers CPLEX 11.0, CLP 1.8, and SOPLEX 1.4 [100], respectively. The "Time" columns show the solution

Table 7 Comparison of CPLEX 11.0, CPLEX 10.2, CBC 2.2, and SCIP 1.1 with different LP solvers

Name	CPLEX 11.0		CPLEX 10.2		CBC 2.2		SCIP/Cpx		SCIP/Clp		SCIP/Spx	
	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
10 teams	0.8	1	9.9	415	19.4	602	12.2	154	25.0	161	68.7	502
30_05_100	66.3	9	99.7	81	>3600.0	1354	478.4	146	752.9	204	>3600.0	16
30_95_100	65.6	1	110.5	27	>3600.0	4706	400.6	166	781.1	195	>3600.0	83
30_95_98	59.4	1	100.0	1	>3600.0	6585	326.4	228	698.2	191	>3600.0	96
acc0	0.4	1	0.1	1	1.0	1	34.3	118	90.7	82	6.5	1
acc1	1.6	1	3.4	1	4.2	1	75.1	61	133.1	109	351.6	64
acc2	2.4	1	17.6	14	77.2	265	116.3	84	139.9	77	420.2	164
acc3	59.6	39	51.9	31	257.4	228	466.8	348	311.2	70	1360.3	392
acc4	86.1	70	143.4	130	622.8	794	2569.1	2102	1214.3	626	2075.1	856
acc5	41.0	24	128.4	143	1218.5	1241	607.6	941	800.6	1284	403.2	60
acc6	1152.0	2531	249.8	365	770.7	2583	281.0	593	679.6	1105	299.1	29
aflow30a	15.3	3054	22.8	9533	770.9	97340	19.9	1768	31.6	1553	53.4	3055
air03	0.5	1	0.5	1	2.0	1	49.9	4	47.4	4	66.7	3
air04	15.6	263	16.8	217	159.9	1256	107.4	100	167.7	59	325.5	290
air05	14.0	467	16.1	509	73.6	693	50.4	280	95.4	151	139.2	216
bc1	157.0	5551	110.3	6347	>3600.0	19339	289.0	4392	598.5	6452	969.3	4983
bell3a	2.8	26087	3.8	27345	3.4	4837	36.1	48171	25.2	28144	36.1	25181
bell5	0.2	1129	0.2	1025	3.8	4018	1.2	1620	1.1	1047	0.8	1106
bienst1	45.6	10525	172.9	7272	308.7	15712	51.2	17068	81.5	18776	303.3	8562
bienst2	274.1	92913	3089.2	86006	1532.3	104614	259.9	89778	514.9	111170	1642.6	71745
binkar10_1	17.0	3250	36.8	8397	>3600.0	302208	680.5	157069	1407.7	202038	1648.8	243851
blend2	1.7	1020	1.4	2514	13.3	2063	0.6	195	0.6	160	0.5	44

Table 7 continued

Name	CPLEX 11.0		CPLEX 10.2		CBC 2.2		SCIP/Cpx		SCIP/Clp		SCIP/Spx	
	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
cap6000	1.3	4227	15.0	4527	86.1	22232	7.0	2949	9.0	3315	11.8	2740
dano3_3	90.3	20	113.4	17	117.4	34	120.7	8	506.2	12	591.6	15
dano3_4	112.6	29	128.2	25	190.4	52	290.3	23	1180.2	46	1396.2	32
dano3_5	497.4	202	349.6	222	899.5	363	460.0	226	2291.4	88	>3600.0	80
dcmulti	0.5	76	0.5	55	4.7	277	1.4	53	3.1	101	3.4	84
disctom	11.6	1	327.1	130	16.1	1	3.9	1	14.6	1	16.4	1
dsbmip	0.2	1	0.4	4	12.7	128	0.4	1	0.4	1	0.8	1
egout	0.1	1	0.1	1	0.1	1	0.1	1	0.1	1	0.1	1
eilD76	9.4	263	6.0	85	730.6	13727	26.1	8	88.9	18	178.6	5
fast0507	910.7	2941	887.3	4863	1714.2	6367	1303.6	1998	>3600.0	555	>3600.0	835
fiber	0.3	60	0.3	73	4.8	34	1.5	24	1.4	11	1.8	16
fixnet6	1.0	71	0.9	17	6.8	203	2.6	6	5.2	18	8.8	19
flugpl	0.1	89	0.1	81	0.1	50	0.1	80	0.1	86	0.1	80
gen	0.1	1	0.1	1	0.2	1	0.2	1	0.2	1	0.3	1
gesa2_o	1.6	482	3.1	2298	64.3	3630	2.2	41	2.2	8	2.2	4
gesa2	0.5	147	0.5	82	10.9	420	2.1	7	2.1	5	2.2	6
gesa3	0.9	49	0.4	36	7.3	289	1.9	16	2.9	17	3.0	16
gesa3_o	0.7	58	0.3	32	14.6	226	2.5	12	3.2	12	4.5	20
gt2	0.1	1	0.1	1	0.5	5	0.1	1	0.1	1	0.1	1
harp2	288.8	316170	>3600.0	2551920	1857.4	348829	>3600.0	4814504	>3600.0	2258185	>3600.0	1340772
irp	5.9	1	4.9	6	29.8	275	66.3	650	37.2	13	577.0	9987

Table 7 continued

Name	CPLEX 11.0		CPLEX 10.2		CBC 2.2		SCIP/Cpx		SCIP/Clp		SCIP/Spx	
	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
khb05250	0.1	4	0.1	2	0.8	32	1.1	10	1.6	4	1.3	5
l152lav	0.9	289	0.9	293	7.7	530	3.6	29	9.5	62	10.9	79
lseu	0.1	102	0.1	162	1.6	50	0.3	66	0.5	239	0.9	582
manna81	0.1	1	>3600.0	1505149	1.6	1	1.3	2	1.2	1	2.2	2
markshare4_0	101.9	1600251	76.4	1298307	727.2	691046	824.7	814015	308.7	1234962	862.8	1055223
mas74	625.5	2673089	1239.8	5075684	>3600.0	2790307	1629.2	3070052	3139.7	3719547	>3600.0	2608442
mas76	77.0	398167	112.6	641122	473.3	407866	140.9	329883	203.8	336737	310.3	312443
mas284	5.5	13673	8.4	24079	55.8	19879	34.5	15277	53.8	15753	75.8	14805
mik._-75.1	2.4	8680	6.9	19951	421.1	16004	6.6	7218	9.5	6052	10.0	6262
mik._-75.2	2.5	6735	1.8	3800	154.7	9398	5.6	4746	9.3	4142	12.7	5774
mik._-75.3	5.5	19397	13.7	37735	131.6	8298	6.2	5684	8.7	4432	10.2	5186
mik._-75.4	10.8	46124	7.8	19279	655.4	57172	60.4	95266	71.5	60672	75.7	60564
mik._-75.5	3.5	14429	8.0	21655	97.6	5805	9.3	12576	15.2	11584	14.1	11308
misc03	0.4	188	0.2	206	5.0	72	1.6	176	2.7	178	5.2	151
misc06	0.1	14	0.1	19	1.1	21	0.6	7	0.5	9	1.3	7
misc07	40.1	25645	9.7	11270	103.1	16454	50.1	30984	90.5	35710	49.3	17678
mkc1	23.4	3886	52.4	10622	2321.8	108913	>3600.0	694750	>3600.0	331636	>3600.0	313507
mod008	0.1	343	0.2	718	1.2	44	0.4	90	0.7	175	0.7	211
mod010	0.3	5	0.3	9	1.2	1	2.2	2	2.1	4	3.1	2

Table 7 continued

Name	CPLEX 11.0		CPLEX 10.2		CBC 2.2		SCIP/Cpx		SCIP/Clp		SCIP/Spx	
	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
mod011	40.6	54	51.5	35	81.5	476	154.9	2048	545.4	1409	904.7	1827
modglob	0.2	183	0.1	56	27.7	7815	1.2	66	0.9	11	1.9	83
mzzv11	169.2	498	242.6	2860	2881.1	13001	826.1	3166	3516.7	4204	>3600.0	15
mzzv42z	57.8	298	51.0	149	258.3	370	499.9	251	2069.9	190	>3600.0	9
neos1	2.8	64	2.1	2	30.2	384	8.3	2	12.3	1	9.8	1
neos2	17.9	1477	5.8	866	14.6	460	69.2	17880	296.0	34281	175.3	12028
neos3	34.5	4196	28.6	3644	29.8	1433	748.1	210663	>3600.0	324591	2367.8	135430
neos4	5.3	18	6.0	18	30.5	442	5.5	1	5.4	1	5.5	1
neos5	327.1	1223151	1544.3	6063389	1888.6	885673	2875.1	5375739	>3600.0	4039346	>3600.0	3537144
neos6	46.5	600	481.9	8800	492.6	5839	390.9	7171	1115.5	2868	3076.9	6057
neos7	42.5	39467	31.7	9563	171.2	7518	25.7	6945	392.3	53075	567.2	41246
neos8	2.7	1	4.3	1	30.7	1	80.7	1	80.8	1	80.8	1
neos9	1114.6	2458	>3600.0	19001	>3600.0	7617	136.1	1	>3600.0	763	>3600.0	96
neos10	4.9	19	9.6	36	48.8	63	83.6	7	88.1	6	95.4	10
neos11	254.2	2320	112.0	1566	625.7	4788	427.5	5602	614.0	1597	2417.7	2758
neos12	158.2	226	189.7	151	>3600.0	12096	1106.3	1083	>3600.0	1793	>3600.0	842
neos13	108.9	565	1638.7	1866	1770.1	12355	550.6	2774	1341.9	7767	>3600.0	8323
neos20	26.4	1517	24.5	3170	56.6	2228	12.8	598	29.0	681	66.5	1225
neos21	103.0	4916	36.1	2609	387.6	17081	37.7	1575	95.3	1881	253.3	2426
neos22	4.8	508	23.1	7631	219.1	9394	2.7	1	3.8	1	3.2	1
neos23	48.7	47643	73.4	74525	883.9	210522	40.7	16909	25.4	5570	23.9	4129

Table 7 continued

Name	CPLEX 11.0		CPLEX 10.2		CBC 2.2		SCIP/Cpx		SCIP/Clp		SCIP/Spx	
	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
neos616206	2034.5	381639	>3600.0	408178	>3600.0	206236	3247.4	623286	>3600.0	400485	>3600.0	240650
neos632659	0.1	48	0.1	24	11.1	105	6.8	7676	972.2	778169	>3600.0	3256465
neos648910	0.4	94	0.5	184	46.7	827	2.3	195	2.8	95	3.4	74
neos808444	883.0	725	>3600.0	360	>3600.0	1	>3600.0	25	>3600.0	9	>3600.0	7
neos897005	206.9	1	293.3	1	108.5	1	537.1	21	1859.4	9	>3600.0	13
nug08	13.8	45	16.7	59	13.1	12	77.2	1	486.3	1	224.1	1
nw04	31.6	283	23.7	265	41.5	108	79.3	132	80.6	8	616.1	103
opt1217	0.1	1	>3600.0	6754218	0.4	1	0.8	1	0.6	1	0.9	1
p0033	0.1	5	0.1	4	0.1	1	0.1	2	0.1	1	0.1	4
p0201	0.5	140	0.2	79	23.8	28	1.2	113	2.8	53	3.5	100
p0282	0.1	16	0.1	72	8.5	91	0.8	7	1.1	11	1.2	6
p0548	0.1	1	0.1	1	1.6	48	0.4	8	0.5	7	0.5	8
p2756	0.4	11	0.4	30	9.6	182	3.8	187	4.7	201	4.0	54
pk1	164.7	186390	83.7	338108	326.5	221721	149.3	227351	255.9	252667	357.4	232977
pp08a	1.6	625	1.4	688	23.2	2237	1.9	192	3.2	577	4.8	688
pp08aCUTS	2.2	1102	1.8	999	41.2	5237	2.0	91	4.4	558	5.6	221
prod1	15.9	32134	26.8	46378	345.4	30140	39.5	23482	49.8	27555	59.1	23582
prod2	97.4	87637	420.1	411595	1826.2	105115	186.9	68500	243.8	70897	401.8	96131
qap10	184.7	41	279.0	88	100.5	12	317.1	5	665.8	3	2429.0	7
qiu	58.1	7233	52.0	4233	916.8	47973	157.4	12812	302.5	11445	522.6	13627
qnet1	2.0	86	1.7	37	9.6	30	4.7	71	8.9	50	10.8	29

Table 7 continued

Name	CPLEX 11.0		CPLEX 10.2		CBC 2.2		SCIP/Cpx		SCIP/Clp		SCIP/Spx	
	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes
qnet1_o	1.5	63	1.2	100	6.3	22	2.8	27	4.3	41	14.4	82
ran8x32	6.9	3805	5.0	3486	48.0	9968	21.9	10999	38.9	9613	80.0	16117
ran10x26	19.8	5769	29.9	19740	67.4	19110	62.9	31469	123.9	24508	186.0	30170
ran12x21	90.8	34725	49.3	32815	248.2	62100	115.8	61542	274.8	56590	506.4	90573
ran13x13	15.3	6858	21.0	24343	211.9	37627	69.5	50831	155.9	49970	141.5	36464
rentacar	0.8	7	0.8	11	4.4	16	4.9	15	6.8	14	9.5	13
rgn	0.2	543	0.6	3516	4.8	954	0.5	34	0.4	2	0.4	2
rout	17.5	5260	44.9	16492	375.9	74937	71.5	29025	120.9	20166	120.4	17200
set1ch	0.5	330	0.5	310	1301.5	86157	0.9	9	1.0	8	1.4	22
seymour1	269.5	1786	655.8	6435	2860.6	10516	845.3	5772	1572.1	5112	>3600.0	3909
stein27	0.3	1383	0.2	1111	2.9	626	2.1	4175	2.5	3951	2.3	3757
stein45	13.7	52050	13.5	54137	80.3	5752	51.6	52415	66.9	54557	60.4	51076
swath1	11.9	1783	11.9	4680	463.9	28883	64.6	750	147.9	3981	143.9	3154
swath2	19.4	3598	17.6	6068	532.3	39832	154.5	5157	285.5	9812	827.4	38876
swath3	181.1	43757	64.4	25160	1560.1	113709	2148.7	142792	803.1	37597	546.6	26120
tr12-30	866.8	318526	>3600.0	874501	>3600.0	6695	>3600.0	638788	>3600.0	439921	>3600.0	534581
vpm1	0.1	1	0.1	1	0.6	4	0.1	1	0.2	1	0.3	1
vpm2	0.8	1619	0.6	1880	6.3	327	1.9	177	2.0	114	2.6	194
sh geom mean	11.7	558	17.9	920	77.8	1362	33.5	671	54.6	657	74.9	614

Table 8 Comparison of CPLEX 11.0, CPLEX 10.2, CBC 2.2, and SCIP 1.1 with different LP solvers

	Cplex 11.0	Cplex 10.2	CBC 2.2	SCIP/Cpx	SCIP/Clp	SCIP/Spx
# of instances within 101% of fastest	73	61	6	11	4	6
# of instances within 110% of fastest	82	64	8	16	9	7
# of instances within 150% of fastest	105	83	13	32	19	16
# of instances within 200% of fastest	114	95	25	48	32	23
# of instances within 500% of fastest	121	110	54	88	66	57
# of instances within 1000% of fastest	122	114	77	106	87	79
# of solved instances (of 122)	122	115	111	118	112	102
Shifted geometric time ratio to Cplex 11	–	1.53	6.65	2.86	4.67	6.40
Shifted geometric nodes ratio to Cplex 11	–	1.65	2.44	1.20	1.18	1.10
Shifted geometric iterations ratio to Cplex 11	–	1.62	2.80	1.18	2.22	2.49
Shifted geometric time ratio (solved by all)	–	1.22	6.67	2.63	3.84	5.16
Shifted geometric nodes ratio (solved by all)	–	1.24	2.22	0.93	0.85	0.91
Shifted geometric iterations ratio (solved by all)	–	1.32	2.97	1.04	1.81	2.07

time in seconds for each instance and solver, while the "Nodes" column contains the number of branching nodes needed to solve the instance. A ">" tag in front of the time identifies instances that could not be solved by the respective solver within the time limit of 3,600 s. In the shifted geometric means[3] at the bottom of Table 7 such cases are counted as if the instance was solved after 3,600 s. Note that this gives a small advantage to solvers that often hit the time limit; their geometric time and node statistics would degrade if a larger time limit was employed.

Table 8 gives a summary of the results. The top part of the table provides information which is similar to a performance profile [45]: for each of the solvers it shows how many of the 122 instances can be solved within a certain time relative to the fastest of the six solvers. For example, on 48 instances SCIP 1.1 with CPLEX 11 as LP solver spent at most twice as much time as the respective best solver. It is obvious that CPLEX 11 is the best of the considered solvers on this test set. Among the non-commercial products, the largest number of instances that can be solved in a "reasonable" time is by SCIP/CLP. The performance of CBC and SCIP/SOPLEX seems to be similar.

[3] The shifted geometric mean of values $t_1, \dots, t_n$ is defined as $\gamma_s(t_1, \dots, t_n) = \left(\prod(t_i + s)\right)^{1/n} - s$ with shift $s \geq 0$, see [2]. We use a shift $s = 1$ for time, $s = 10$ for nodes, and $s = 100$ for simplex iterations in order to decrease the strong influence of the very easy instances in the mean values.

This observation is confirmed by the shifted geometric mean ratios in the second part of Table 8. These ratios are calculated by dividing the shifted geometric means of time, nodes, and simplex iterations, respectively, by the results of CPLEX 11. As one can see, CPLEX 10.2 shows a performance degradation of a factor of 1.53 compared to CPLEX 11, which is followed by SCIP/CPLEX with a factor of 2.86. Since there is not much improvement in the LP solver from CPLEX 10.2 to CPLEX 11, this means that the "MIP kernel" of SCIP is only 1.87 times slower than the one of CPLEX 10.2. This supports the results in [2], where a performance difference of a factor of 1.63 was reported between CPLEX 10.0.1 and SCIP 0.90i.

The bottom part of the table concentrates only on those instances that could be solved by all six solvers within the time limit. On these 97 "easy" instances, the difference of SCIP to CPLEX is smaller than on the whole test set. This is also confirmed by our own experience, namely that CPLEX 11 has improved considerably over CPLEX 10 on the harder instances and can now solve many more instances in reasonable time. Additionally, many algorithms in SCIP are designed and tuned for medium sized models.

Comparing time, node counts, and iteration numbers of the different solvers reveals some interesting facts. First, the speedup from CPLEX 10.2 to CPLEX 11 can be attributed to a reduction in the number of nodes while the node throughput (i.e., how many nodes are processed per second) has not changed much. The number of simplex iterations needed to resolve each node in the search tree stayed almost constant.

Compared to CBC, SCIP needs a much smaller number of branching nodes in average to solve the MIP instances. It can even compete to CPLEX in this regard. Together with the time values, this indicates that SCIP spends more work on individual nodes than the other solvers, which can mainly be attributed to the aggressive use of strong branching. Comparing the performance of the LP solvers within an MIP framework, the results of SCIP show that CPLEX is superior to CLP, which in turn outperforms SOPLEX.

5 SCIP as a CIP framework: chip design verification

This section shows the use of SCIP as a CIP framework by the example of the *property checking problem*, which arises in chip design verification (see also [2,4,5]). First, we give a short introduction to the problem.

A recent trend in the semiconductor industry is to produce so-called *Systems-on-Chips* (SoCs). These are circuits which integrate large parts of the functionality of complete electronic systems. They are employed in cell phones, car controls, digital televisions, network processors, video games, and many other devices [61]. Due to the complexity of SoCs, it is a very challenging task to ensure the correctness of the chip design. According to INFINEON [96], 60–80% of the expenses in SoC chip design are spent on verification.

Figure 6 sketches a typical work flow in the chip manufacturing process. The chip design is usually developed in a hardware design language like VERILOG, VHDL, SYSTEM- C, or SYSTEM VERILOG, which are very similar to ordinary imperative computer programming languages like C or FORTRAN. The design is tested by *software*

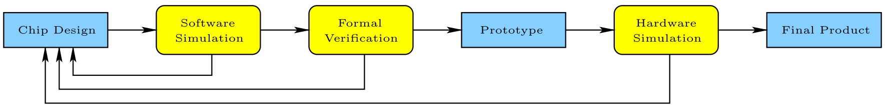

Fig. 6 Chip manufacturing workflow

simulation, which consists of applying many different input patterns to the input connectors of a virtual representation of the chip. If the computed output does not match the expected output, the design is flawed and has to be corrected.

The idea of formal verification is that the verification engineer completely describes the expected behavior of the chip by a set of *properties*, which are formal relations between the inputs, outputs, and internal states of the circuit. Given one of these properties, the task of the *property checking problem* is to prove that the chip design satisfies this property. If this can be shown for all properties, the chip is proven to be correct.

5.1 Constraint integer programming approach

Properties can be defined in a language similar to chip design languages like VHDL or VERILOG that are used to design the circuit. Current state-of-the-art property checking algorithms transform this representation into an instance of the satisfiability problem, which is then solved by a black-box SAT solver.

The reduction of the property checking problem to a SAT instance facilitates formal verification of industrial circuit designs far beyond the scope of classical model checking techniques like BDD [4] based approaches [28,30]. However, it is well known that SAT solvers have problems when dealing with instances derived from the verification of arithmetic circuits (as opposed to logic oriented circuits). Hence, although SAT based property checking can often be applied successfully to the control part of a design, it typically fails on data paths with large arithmetic blocks.

To remedy this situation, word level solvers have been proposed [35,47,52,89,101] that address the property checking problem at a higher representation level, the *register transfer (RT) level*, and try to exploit the structural information therein. Our approach shares this idea but differs from previous work in the techniques that are employed. Namely, we formulate the problem as a constraint integer program. For each type of RT operation, we implemented a constraint handler with specialized domain propagation algorithms that use both bit and word level representations. We also provide *reverse* propagation algorithms to support conflict analysis at the RT level. In addition, we present linearizations for most of the RT operators in order to construct the LP relaxation.

The property checking problem at the register transfer level can be defined as follows:

Definition 5.1 (Property checking problem) The *property checking problem* is a triple PCP $= (\mathfrak{C}, P, \mathfrak{D})$ with $\mathfrak{D} = \mathcal{D}_1 \times \cdots \times \mathcal{D}_n$ representing the domains $\mathcal{D}_j =$

[4] Binary decision diagram, see Akers [9], Bryant [36], or Madre and Billon [73].

$\{0, \dots, 2^{\beta_j - 1}\}$ of register variables $\varrho_j \in \mathcal{D}_j$ with bit width $\beta_j \in \mathbb{N}$, $j = 1, \dots, n$, $\mathfrak{C} = \{\mathcal{C}_1, \dots, \mathcal{C}_m\}$ being a finite set of constraints $\mathcal{C}_i : \mathfrak{D} \to \{0, 1\}$, $i = 1, \dots, m$, describing the behavior of the circuit, and $P : \mathfrak{D} \to \{0, 1\}$ being a constraint describing the property to be verified. The task is to decide whether

$$\forall \varrho \in \mathfrak{D} : \mathfrak{C}(\varrho) \to P(\varrho) \tag{7}$$

holds, i.e., to either find a counter-example ϱ satisfying $\mathfrak{C}(\varrho)$ but violating $P(\varrho)$ or to prove that no such counter-example exists.

In order to verify Condition (7) we search for a counter-example using the equivalence

$$\forall \varrho \in \mathfrak{D} : \mathfrak{C}(\varrho) \to P(\varrho) \quad \Leftrightarrow \quad \neg\,(\exists \varrho \in \mathfrak{D} : \mathfrak{C}(\varrho) \wedge \neg P(\varrho)). \tag{8}$$

The right hand side of (8) is a finite domain constraint satisfaction problem CSP $= (\mathfrak{C} \cup \{\neg P\}, \mathfrak{D})$, which is a special case of CIP, see Proposition 2.3. Every feasible solution $\varrho^\star \in \mathfrak{D}$ of the CSP corresponds to a counter-example of the property. Therefore, the property is valid if and only if the CSP is infeasible.

We model the property checking CSP with variables $\varrho \in \{0, \dots, 2^{\beta_\varrho - 1}\}$ of width β_ϱ and constraints $r^i = C_i(x^i, y^i, z^i)$, which resemble circuit operations with up to three input bit vectors x^i, y^i, z^i, and an output bit vector r^i. For each bit vector variable ϱ, we introduce single bit variables ϱ_b, $b = 0, \dots, \beta_\varrho - 1$, with $\varrho_b \in \{0, 1\}$, for which linking constraints

$$\varrho = \sum_{b=0}^{\beta_\varrho - 1} 2^b \varrho_b \tag{9}$$

define their correlation. In addition, we consider the following circuit operations: ADD, AND, CONCAT, EQ, ITE, LT, MINUS, MULT, NOT, OR, READ, SHL, SHR, SIGNEXT, SLICE, SUB, UAND, UOR, UXOR, WRITE, XOR, ZEROEXT with the semantics as defined in Table 9, see also [2,34]. Each constraint class, including the linking constraints (9), gives rise to a *constraint handler* with a specialized set of algorithms in order to perform presolving, domain propagation, reverse propagation, linearization, cutting plane separation, and feasibility checking.

The whole set of plugins specific to the property checking problem, including constraint handlers, a branching rule, presolvers, and file readers, consists of 58 367 lines of C code. In the following, we provide some very brief anecdotal descriptions of the algorithms that are contained in these plugins in order to convey an impression of how SCIP can be extended to support a complex CIP model.

5.2 Domain propagation

For the bit linking constraints (9) and for each type of circuit operation we implemented a specific domain propagation algorithm that exploits the special structure of

Table 9 Semantics and LP relaxation of circuit operations. l_ϱ and u_ϱ are the lower and upper bounds of a bit vector variable ϱ

Operation	Semantics	Linearization	
r = AND (x,y)	$r_b = x_b \wedge y_b$ for all b	$r_b \le x_b, r_b \le y_b, r_b \ge x_b + y_b - 1$	
r = OR (x,y)	$r_b = x_b \vee y_b$ for all b	$r_b \ge x_b, r_b \ge y_b, r_b \le x_b + y_b$	
r = XOR (x,y)	$r_b = x_b \oplus y_b$ for all b	$x_b - y_b - r_b \le 0, -x_b + y_b - r_b \le 0,$	
		$-x_b - y_b + r_b \le 0, x_b + y_b + r_b \le 2$	
r = UAND (x)	$r = x_0 \wedge \ldots \wedge x_{\beta_x - 1}$	$r \le x_b, r \ge \sum_{b=0}^{\beta_x-1} x_b - \beta_x + 1$	
r = UOR (x)	$r = x_0 \vee \ldots \vee x_{\beta_x - 1}$	$r \ge x_b, r \le \sum_{b=0}^{\beta_x-1} x_b$	
r = UXOR (x)	$r = x_0 \oplus \ldots \oplus x_{\beta_x - 1}$	$r + \sum_{b=0}^{\beta_x-1} x_b = 2s,$	$s \in \mathbb{Z}_{\ge 0}$
r = EQ (x,y)	$r = 1 \Leftrightarrow x = y$	$x - y = s - t,$	$s, t \in \mathbb{Z}_{\ge 0},$
		$p \le s, s \le p(u_x - l_y),$	$p \in \{0, 1\},$
		$q \le t, t \le q(u_y - l_x),$	$q \in \{0, 1\},$
		$p + q + r = 1$	
r = LT (x,y)	$r = 1 \Leftrightarrow x < y$	$x - y = s - t,$	$s, t \in \mathbb{Z}_{\ge 0},$
		$p \le s, s \le p(u_x - l_y),$	$p \in \{0, 1\},$
		$r \le t, t \le r(u_y - l_x),$	
		$p + r \le 1$	
r = ITE (x,y,z)	$r = \begin{cases} y & \text{if } x = 1, \\ z & \text{if } x = 0 \end{cases}$	$r - y \le (u_z - l_y)(1 - x)$ $r - y \ge (l_z - u_y)(1 - x)$	
		$r - z \le (u_y - l_z)\, x$	
		$r - z \ge (l_y - u_z)\, x$	
r = ADD (x,y)	$r = (x + y) \bmod 2^{\beta_r}$	$r + 2^{\beta_r} o = x + y,$	$o \in \{0, 1\}$
r = MULT (x,y)	$r = (x \cdot y) \bmod 2^{\beta_r}$	$v_{bn} \le u_{y_n} x_b, v_{bn} \le y_n,$	$v_{bn} \in \mathbb{Z}_{\ge 0}$
		$v_{bn} \ge y_n - u_{y_n}(1 - x_b)$	
		$o_n + \sum_{i+j=n} \sum_{l=0}^{L-1} 2^l v_{iL+l,j}$	
		$\quad = 2^L o_{n+1} + r_n,$	$o_n \in \mathbb{Z}_{\ge 0}$
r = MINUS (x)	$r = 2^{\beta_r} - x$	replaced by $0 = \text{ADD}(x, r)$	
r = SUB (x,y)	$r = (x - y) \bmod 2^{\beta_r}$	replaced by $x = \text{ADD}(y, r)$	
r = SHR (x,y)	$r_b = \begin{cases} x_{b+y} & \text{if } b + y < \beta_x, \\ 0 & \text{if } b + y \ge \beta_x \end{cases}$	replaced by $r = \text{SLICE}(x, y)$	

the constraint class. In addition to considering the current domains of the bit vectors ϱ and the bit variables ϱ_b, we exploit knowledge about the global equality or inequality of bit vectors or bits, which is obtained in the preprocessing stage of the algorithm. For example, if we know already that for certain bits the input vectors x and y in an equality constraint $r = \text{EQ}(x, y)$[5] are equal, i.e., $x_b = y_b$ for a few bits b, the value

[5] The constraint $r = \text{EQ}(x, y)$ on $r \in \{0, 1\}$ and $x, y \in \{0, \ldots, 2^{\beta-1}\}$ is defined as $r = 1 \Leftrightarrow x = y$.

Table 9 continued

Operation	Semantics	Linearization
r = CONCAT (x,y)	$r_b = \begin{cases} y_b & \text{if } b < \beta_y, \\ x_{b-\beta_y} & \text{if } b \geq \beta_y \end{cases}$	removed in preprocessing
r = NOT (x)	$r_b = 1 - x_b$ for all b	removed in preprocessing
r = SIGNEXT (x)	$r_b = \begin{cases} x_b & \text{if } b < \beta_x, \\ x_{\beta_x - 1} & \text{if } b \geq \beta_x \end{cases}$	removed in preprocessing
r = ZEROEXT (x)	$r_b = \begin{cases} x_b & \text{if } b < \beta_x, \\ 0 & \text{if } b \geq \beta_x \end{cases}$	removed in preprocessing
r = SHL (x,y)	$r_b = \begin{cases} x_{b-y} & \text{if } b \geq y, \\ 0 & \text{if } b < y \end{cases}$	no linearization
r = SLICE (x,y)	$r_b = \begin{cases} x_{b+y} & \text{if } b + y < \beta_x, \\ 0 & \text{if } b + y \geq \beta_x \end{cases}$	no linearization
r = READ (x,y)	$r_b = \begin{cases} x_{b+y\cdot\beta_r} & \text{if } b + y \cdot \beta_r < \beta_x, \\ 0 & \text{if } b + y \cdot \beta_r \geq \beta_x \end{cases}$	no linearization
r = WRITE (x,y,z)	$r_b = \begin{cases} z_{b-y\cdot\beta_z} & \text{if } 0 \leq b - y \cdot \beta_z < \beta_z, \\ x_b & \text{otherwise} \end{cases}$	no linearization

of the resultant r can already be decided after the remaining input bits have been fixed.

Some of the domain propagation algorithms are very complex. For example, the domain propagation of the MULT constraint uses term algebra techniques to recognize certain deductions inside its internal representation of a partial product and overflow addition network. Others, like the algorithms for SHL, SLICE, READ, and WRITE, involve reasoning that mixes bit- and word-level information.

Most of the constraint handlers perform domain propagation on one or even multiple internal representations of the constraints. Thereby, they exploit one of the inherent advantages of SCIP compared to other branch-and-cut frameworks like CPLEX with callbacks, namely that SCIP takes care of all the bookkeeping for the internal variables inside the search tree but does not need to add them to the LP relaxation. Furthermore, the event system of SCIP enables constraint handlers to easily track bound changes on variables and allows for efficient sparse domain propagation implementations.

5.3 LP relaxation

Although the property checking is a pure feasibility problem and there is no natural objective function, its LP relaxation is still useful, because it usually detects the infeasibility of a subproblem much earlier than domain propagation. This is due to the fact that the LP has a "global view" taking all (linearizable) constraints except the integrality restrictions into consideration at the same time, while domain propagation is applied successively on each individual constraint.

Table 9 shows the linearizations of the circuit operation constraints that are used in addition to the bit linking constraints (9) to construct the LP relaxation of the problem instance. Very large coefficients like 2^{β_r} in the ADD linearization can lead to numerical difficulties in the LP relaxation. Therefore, we split the bit vector variables into words of $W = 16$ bits and apply the linearization to the individual words. The linkage between the words is established in a proper fashion. For example, the overflow bit of a word in an addition is added to the right hand side of the next word's linearization. The relaxation of the MULT constraint involves additional variables y_n and r_n which are "nibbles" of y and r with $L = \frac{W}{2}$ bits.

The MINUS and SUB operations can be replaced by an equivalent ADD operation as shown in the table. SHR is replaced by the more general SLICE operator. The operations CONCAT, NOT, SIGNEXT, and ZEROEXT do not need an LP relaxation: their resultant bits can be aggregated with the corresponding operand bits such that the constraints can be deleted in the preprocessing stage of the algorithm.

No linearization is generated for the SHL, SLICE, READ, and WRITE constraints. Their linearizations are very complex and would dramatically increase the size of the LP relaxation, thereby reducing the solvability of the LPs. For example, a straightforward linearization of the SHL constraint on a 64-bit input vector x that uses internal ITE-blocks for the potential values of the shifting operand y already requires 30944 inequalities and 20929 auxiliary variables. The possibility in SCIP to treat such constraints solely by CP techniques without having to specify an LP relaxation is a key ingredient for the success of our CIP based property checker. In contrast, pure IP approaches like [35] are not able to efficiently incorporate highly non-linear constraints like SHL into their models.

The objective function vector c of the CIP model, and consequently the one of the LP relaxation, can be chosen arbitrarily. We experimented with three choices, namely $c = 0$, $c_{jb} = 1$, and $c_{jb} = -1$ for all register bits ϱ_{jb}. It turned out that this choice does not have a large impact on the performance of the solver.

5.4 Preprocessing

Before the actual branch-and-bound based search algorithm is applied we try to simplify the given problem instance by employing the following preprocessing techniques. The individual preprocessing algorithms are applied periodically until no more reductions can be found.

Constraint based presolving is conducted by the constraint handlers. For example, as noted in Sect. 5.3, the constraint handlers for CONCAT, NOT, SIGNEXT, and ZEROEXT just perform the necessary variable aggregations and fixings. Afterwards, they delete all of their constraints from the model.

The presolving of the ADD constraint handler for constraints $r = \text{ADD}(x, y)$ can extract fixings and aggregations of bits r_b, x_b, y_b, and its internal overflow variables o_b by inspecting the bit level addition scheme. Additionally, it can detect the global inequalities $r \neq y$ or $r \neq z$ if $z \neq 0$ or $y \neq 0$, respectively, on the word level. Such global inequality information can be exploited in other presolving and domain propagation algorithms. Furthermore, the ADD constraint handler performs pairwise presolving to detect equivalences between bit variables across multiple ADD constraints.

The presolving of MULT constraints is even more complicated. As can be seen in Sect. 5.3, the LP relaxation of $r = \text{MULT}(x, y)$ is asymmetric w.r.t. x and y. Therefore, such a constraint is replaced by $r = \text{MULT}(y, x)$ if this leads to a smaller number of LP rows and columns in the relaxation. Aggregations and fixings of bit variables are performed by traversing the bit level multiplication table, which contains internal partial product variables $p_{ij} = x_i \cdot y_j$ as well as intermediate sum and overflow variables. Additionally, a symbolic domain propagation is applied which can identify fixings as well as aggregations of variables. Similar to the ADD constraint presolving, a pairwise comparison can detect aggregations and global inequalities across multiple MULT constraints.

In addition to the constraint based presolving conducted by the constraint handlers, our chip verification code features some global presolving methods which are implemented as *presolver plugins*. One of the employed presolvers is *probing*, which is already available in SCIP and also used in MIP solving, see Sect. 3.1.3. In addition, we implemented two special purpose presolving plugins for the property checking problem, which we call *term algebra preprocessing* and *irrelevance detection* and which are described in detail in [2].

These two special purpose presolving methods are based on the high-level constraint description of the problem. The tight cooperation between such problem specific algorithms and general purpose methods like probing can be accomplished easily in the constraint integer programming paradigm through the plugin concept of SCIP.

5.5 Experimental results

In this section we examine the computational effectiveness of the described constraint integer programming techniques on industrial benchmarks obtained from verification projects conducted by OneSpin Solutions. All calculations were performed on a 3.8 GHz Pentium-4 workstation with 2 GB RAM. In all runs we used a time limit of 7,200 s. The specific chip verification algorithms were incorporated into SCIP 0.90i [2]. The LP relaxations were solved using Cplex 10.0.1 [59].

For reasons of comparison, we also solved the instances with classical SAT techniques on the bit level. Before the SAT solver is called, a preprocessing step is executed to simplify the instance at the bit level. We used MiniSat 2.0 [46] to solve the resulting SAT instances. We also tried MiniSat 1.14, Siege v4 [92] and zChaff 2004.11.15 [84], but MiniSat 2.0 turned out to perform best on most of the instances of our test set. Unfortunately, we cannot compare our approach to other word level solvers as mentioned in Sect. 5.1, because they are not available within the framework of our industrial partner.

We conducted experiments on property checking instances of different circuits. Details can be found in [2]. Table 10 compares the results of MiniSat and our CIP approach on some of the non-trivial instances of our test set.

For each circuit and property listed in the “Prop” column the table shows the time in seconds of the two algorithms needed to solve instances of different input bit-widths.

Table 10 Comparison of SAT and CIP approaches on invalid (top) and valid (bottom) properties

Prop	Meth	register width 5	10	15	20	25	30	35	40	∅ Nodes
PipeMult	SAT	0.0	0.0	0.1	0.2	0.3	0.4	0.6	0.7	748
#1	CIP	0.3	0.6	0.7	1.6	2.3	4.7	7.7	10.5	7
PipeMult	SAT	0.0	0.0	0.1	0.2	0.3	0.4	0.6	0.8	932
#2	CIP	0.2	2.2	1.2	2.8	3.5	4.6	7.7	9.4	36
PipeMult	SAT	0.0	0.1	0.1	0.2	0.3	0.6	1.0	1.2	1 770
#3	CIP	0.7	3.0	2.6	9.3	13.5	20.4	21.8	28.8	26
PipeMult	SAT	0.0	0.0	0.1	0.2	0.2	0.3	0.5	0.7	605
#4	CIP	0.5	2.8	6.2	11.1	23.1	33.0	6.3	7.3	40
PipeMult	SAT	0.0	0.0	0.1	0.2	0.3	0.6	0.8	1.0	1 869
#5	CIP	1.1	5.4	15.0	32.5	52.0	92.0	55.1	125.2	53
PipeMult	SAT	0.0	0.1	0.1	0.2	0.3	0.6	0.7	1.1	941
#8	CIP	1.2	8.1	19.8	43.1	45.5	89.9	118.9	91.4	91
ALU	SAT	0.5	—	—	—	—	—	—	—	13 864
muls	CIP	0.0	0.0	0.0	0.1	0.1	0.1	0.2	0.3	0
ALU	SAT	0.1	100.0	—	—	—	—	—	—	53 869
neg_flag	CIP	0.8	3.6	11.6	36.3	81.8	136.6	218.4	383.5	47
ALU	SAT	0.0	0.0	0.1	0.1	0.2	0.4	0.5	0.6	136
zero_flag	CIP	2.3	0.6	1.6	4.0	6.2	10.7	15.6	379.7	56
PipeAdder	SAT	0.0	0.2	0.5	0.8	1.3	1.6	2.1	2.9	73 936
#6	CIP	0.0	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0
PipeAdder	SAT	0.0	0.6	1.3	1.9	2.2	1.7	22.5	16.4	148 840
#7	CIP	0.0	0.1	0.1	0.1	0.1	0.1	0.1	0.2	0
PipeAdder	SAT	0.1	8.9	21.3	764.0	5554.4	81.6	177.1	4087.1	3 535 112
#9	CIP	0.0	0.1	0.1	0.1	0.1	0.1	0.2	0.2	0
PipeAdder	SAT	0.0	0.2	0.5	0.8	1.3	1.6	2.1	2.9	73 936
#10	CIP	0.0	0.0	0.0	0.0	0.1	0.1	0.1	0.1	0
PipeMult	SAT	2.8	—	—	—	—	—	—	—	113 585
#6	CIP	0.0	0.1	0.1	0.2	0.3	0.4	0.6	0.7	0
PipeMult	SAT	8.3	—	—	—	—	—	—	—	241 341
#7	CIP	0.1	0.2	0.5	1.0	1.8	2.9	4.8	6.8	0
PipeMult	SAT	34.5	—	—	—	—	—	—	—	885 261
#9	CIP	0.1	0.5	1.6	3.5	7.1	11.6	19.8	27.9	0
PipeMult	SAT	2.2	—	—	—	—	—	—	—	91 909
#10	CIP	0.1	0.1	0.2	0.4	0.6	0.8	1.1	1.5	0

Solution times are given in seconds. The right most column shows the geometric mean of the number of branching nodes needed to solve the instances of each property. The means include only those instances that both solvers could solve within the time limit

Results marked with "–" could not be solved within the time limit. The right most column shows the geometric mean of the number of branching nodes needed to solve the instances of each property.

The top part of the table contains invalid properties. One can see that SAT clearly outperforms CIP to find counter-examples for those invalid properties. This is due to the fact that SAT processes the nodes of the search tree much faster than CIP and is thereby able to investigate many more nodes in the same amount of time.

For most of the valid properties, however, CIP dominates the SAT approach, sometimes by orders of magnitude. Dramatic improvements can be observed on the *muls* and *neg_flag* properties of the *ALU* circuit and on the *PipeMult* instances. The latter are a result of the structure exploiting presolving methods that are possible within the

Table 11 Comparison of SAT and CIP approaches on multiplier properties (all valid)

Layout	Meth	Register width									∅ Nodes
		6	7	8	9	10	11	12	13	14	
Booth	SAT	0.4	3.3	21.0	135.4	935.1	–	–	–	–	12941
signed	CIP	21.3	70.1	318.7	384.2	904.1	1756.2	2883.7	4995.9	3377.9	3116
Booth	SAT	0.5	2.5	17.9	102.9	879.0	4360.4	–	–	–	27989
unsigned	CIP	15.7	51.7	269.1	911.3	1047.6	2117.7	2295.1	4403.4	7116.8	3499
non-Booth	SAT	0.4	3.4	21.8	134.1	1344.1	–	–	–	–	11248
signed	CIP	12.8	31.2	100.6	265.9	569.8	690.8	1873.0	1976.3	4308.9	1314
non-Booth	SAT	0.3	1.8	16.5	83.1	909.6	5621.5	–	–	–	17527
unsigned	CIP	3.6	22.4	111.2	214.0	335.4	1040.1	1507.5	2347.7	4500.2	1516

Solution times are given in seconds. The right most column shows the geometric mean of the number of branching nodes needed to solve the instances of each circuit layout. The means include only those instances that both solvers could solve within the time limit

CIP paradigm. As can be seen in the geometric means of the nodes, these instances are already solved by preprocessing.

Table 11 shows a comparison on equivalence checking problems for a *Multiplier* circuit. Here, one has to show that a given representation on the bit level indeed implements the multiplication of two input registers. We consider signed and unsigned multipliers and two different bit level implementations, the so-called "Booth" and "non-Booth" encodings, which gives four different layouts in total. Note that this equivalence checking task is not particularly well suited for our approach, since in the bit-level implementation of the multiplication, the arithmetic structure is already lost and cannot be exploited by specialized constraint handlers.

Nevertheless, there seems to be no clear winner on these instances. SAT is faster on the smaller instances, but the running time for the CIP approach increases to a lesser extent with the bit-width of the inputs. The 10×10 multiplication circuits were verified with both techniques in roughly the same time. For larger bit widths, the CIP approach is superior to SAT.

On the whole test set of property checking instances (which is only partly shown in the tables of this paper), SAT failed on 58 out of 258 instances, while CIP was able to solve all instances within the time limit.

6 Outlook

SCIP is still under active development. Future goals are to deal with mixed integer non-linear programming by outer approximation and non-linear relaxations, detecting and handling symmetries, and implementing an exact MIP solver based on SCIP. Additionally, the number of supported constraint types will grow by adding new constraint handlers, for example scheduling constraints or an ALLDIFF constraint handler.

Recently, Berthold et al. [27] extended SCIP to solve pseudo-Boolean problems of optimizing a linear objective subject to polynomial constraints in binary variables.

Computational results on the pseudo-Boolean Evaluation 2007 [74] test set show that SCIP outperforms MiniSat+ 1.14 [46], which is the best of the participating codes in terms of number of solved instances within the time limit of 30 minutes.

Acknowledgments I thank the referees of this paper for their thorough reviews. Their suggestions helped significantly to improve the quality of the paper. Finally, I thank all of my colleagues that worked on SCIP during the recent years. In particular, I thank Timo Berthold, Gerald Gamrath, Stefan Heinz, Bo Jensen, Thorsten Koch, Manuel Kutschka, Alexander Martin, Michael Perregaard, Marc Pfetsch, Christian Raack, Robert Waniek, Michael Winkler, and Kati Wolter. Without their help, my work on SCIP would have been much harder. It is a pleasure to see the project continue even after my departure to Cplex.

References

1. Achterberg, T.: Conflict analysis in mixed integer programming. Discret. Optim. **4**(1), 4–20 (2007) (special issue: Mixed Integer Programming)
2. Achterberg, T.: Constraint Integer Programming. Ph.D. Thesis, Technische Universität Berlin (2007). http://opus.kobv.de/tuberlin/volltexte/2007/1611/
3. Achterberg, T., Berthold, T.: Improving the feasibility pump. Discret. Optim. **4**(1), 77–86 (2007) (special issue: Mixed Integer Programming)
4. Achterberg, T., Berthold, T., Koch, T., Wolter, K.: Constraint integer programming: a new approach to integrate CP and MIP. In: Perron, L., Trick, M.A. (eds.) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, 5th international conference, CPAIOR 2008. Lecture Notes in Computer Science, vol. 5015, pp. 6–20. Springer, Heidelberg (2008)
5. Achterberg, T., Brinkmann, R., Wedler, M.: Property checking with constraint integer programming. Technical Report 07-37, Zuse Institute Berlin (2007). http://opus.kobv.de/zib/volltexte/2007/1065/
6. Achterberg, T., Grötschel, M., Koch, T.: Teaching MIP modeling and solving. ORMS Today **33**(6), 14–15 (2006)
7. Achterberg, T., Koch, T., Martin, A.: Branching rules revisited. Oper. Res. Lett. **33**, 42–54 (2005)
8. Achterberg, T., Koch, T., Martin, A.: MIPLIB 2003. Oper. Res. Lett. **34**(4), 1–12 (2006)
9. Akers, S.B.: Binary decision diagrams. IEEE Trans. Comput. **C-27**(6), 509–516 (1978)
10. Althaus, E., Bockmayr, A., Elf, M., Jünger, M., Kasper, T., Mehlhorn, K.: SCIL—symbolic constraints in integer linear programming. Technical Report ALCOMFT-TR-02-133, MPI Saarbrücken, May (2002)
11. Anders, C.: Das Chordalisierungspolytop und die Berechnung der Baumweite eines Graphen. Master's Thesis, Technische Universität Berlin (2006)
12. Andreello, G., Caprara, A., Fischetti, M.: Embedding cuts in a branch&cut framework: a computational study with $\{0, \frac{1}{2}\}$-cuts. INFORMS J. Comput. **19**(2), 229–238 (2007)
13. Applegate, D.L., Bixby, R.E., Chvátal, V., Cook, W.J.: The Traveling Salesman Problem. Princeton University Press, Princeton (2006)
14. Armbruster, M.: Branch-and-Cut for a Semidefinite Relaxation of the Minimum Bisection Problem. Ph.D. Thesis, Technische Universität Chemnitz (2007)
15. Armbruster, M., Fügenschuh, M., Helmberg, C., Martin, A.: Experiments with linear and semidefinite relaxations for solving the minimum graph bisection problem. Technical Report, Darmstadt University of Technology (2006)
16. Armbruster, M., Fügenschuh, M., Helmberg, C., Martin, A.: On the bisection cut polytope. Darmstadt University of Technology (preprint, 2006)
17. Aron, I.D., Hooker, J.N., Yunes, T.H.: SIMPL: a system for integrating optimization techniques. In: Régin, J.-C., Rueher, M. (eds.) Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, first international conference, CPAIOR. Lecture Notes in Computer Science, vol. 3011, pp. 21–36. Springer, Nice, France (2004)
18. Atamtürk, A.: Flow pack facets of the single node fixed-charge flow polytope. Oper. Res. Lett. **29**, 107–114 (2001)
19. Atamtürk, A.: On the facets of the mixed—integer knapsack polyhedron. Math. Programm. **98**, 145–175 (2003)
20. Atamtürk, A., Rajan, D.: On splittable and unsplittable capacitated network design arc-set polyhedra. Math. Programm. **92**, 315–333 (2002)

21. Balas, E.: Facets of the knapsack polytope. Math. Programm. **8**, 146–164 (1975)
22. Balas, E., Ceria, S., Cornuéjols, G., Natraj, N.: Gomory cuts revisited. Oper. Res. H Lett. **19**, 1–9 (1996)
23. Balas, E., Zemel, E.: Facets of the knapsack polytope from minimal covers. SIAM J. Appl. Math. **34**, 119–148 (1978)
24. Beale, E.M.L.: Branch and bound methods for mathematical programming systems. In: Hammer, P.L., Johnson, E.L., Korte, B.H. (eds.) Discrete Optimization II, pp. 201–219. North Holland Publishing Co., Amsterdam (1979)
25. Bénichou, M., Gauthier, J.M., Girodet, P., Hentges, G., Ribière, G., Vincent, O.: Experiments in mixed-integer linear programming. Math. Programm. **1**, 76–94 (1971)
26. Berthold, T.: Primal heuristics for mixed integer programs. Master's Thesis, Technische Universität Berlin
27. Berthold, T., Heinz, S., Pfetsch, M.E.: Solving pseudo-Boolean problems with SCIP. Report 07–10, Zuse Institute Berlin (2008)
28. Biere, A., Clarke, E.M., Raimi, R., Zhu, Y.: Verifying safety properties of a Power PC microprocessor using symbolic model checking without BDDs. In: Computer-aided verification. Lecture Notes in Computer Science, vol. 1633, pp. 60–71. Springer, Heidelberg (1999)
29. Bilgen, E.: Personalkostenminimierung bei der Einsatzplanung von parallelen identischen Bearbeitungszentren in der Motorradproduktion. Master's Thesis, Technische Universität Chemnitz (2007)
30. Bjesse, P., Leonard, T., Mokkedem, A.: Finding bugs in an Alpha microprocessor using satisfiability solvers. In: Computer-aided verification. Lecture Notes in Computer Science, vol. 2102, pp. 454–464. Springer, Heidelberg (2001)
31. Bley, A., Kupzog, F., Zymolka, A.: Auslegung heterogener Kommunikationsnetze nach performance und Wirtschaftlichkeit. In: Proceedings of 11th Kasseler Symposium Energie-Systemtechnik: Energie und Kommunikation, pp. 84–97, Kassel, November (2006)
32. Bockmayr, A., Kasper, T.: Branch-and-infer: a unifying framework for integer and finite domain constraint programming. INFORMS J. Comput. **10**(3), 287–300 (1998)
33. Bockmayr, A., Pisaruk, N.: Solving assembly line balancing problems by combining IP and CP. In: Sixth Annual Workshop of the ERCIM Working Group on Constraints, June (2001)
34. Brinkmann, R.: Preprocessing for Property Checking of Sequential Circuit on the Register Transfer Level. Ph.D. Thesis, University of Kaiserslautern, Kaiserslautern, Germany (2003)
35. Brinkmann, R., Drechsler, R.: RTL-datapath verification using integer linear programming. In: Proceedings of the IEEE VLSI Design Conference, pp. 741–746 (2002)
36. Bryant, R.E.: Graph-based algorithms for Boolean function manipulation. IEEE Trans. Comput. **C-35**(8), 677–691 (1986)
37. Ceselli, A., Gatto, M., Lübbecke, M., Nunkesser, M., Schilling, H.: Optmizing the cargo express service of swiss federal railways. Transport. Sci. (to appear)
38. COIN-OR. Computational Infrastructure for Operations Research. http://www.coin-or.org
39. Crowder, H., Johnson, E.L., Padberg, M.W.: Solving large scale zero-one linear programming problems. Oper. Res. **31**, 803–834 (1983)
40. Danna, E., Rothberg, E., Le Pape, C.: Exploring relaxation induced neighborhoods to improve MIP solutions. Math. Programm. **102**(1), 71–90 (2005)
41. Dantzig, G.B.: Maximization of a linear function of variables subject to linear inequalities. In: Koopmans, T. (ed.) Activity Analysis of Production and Allocation, pp. 339–347. Wiley, New York (1951)
42. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press, Princeton (1963)
43. Dash Optimization. XPRESS- MP. http://www.dashoptimization.com
44. Dix, A.: Das Statische Linienplanungsproblem. Master's Thesis, Technische Universität Berlin (2007)
45. Dolan, E., Moré, J.: Benchmarking optimization software with performance profiles. Math. Programm. **91**, 201–213 (2002)
46. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) Proceedings of SAT 2003, pp. 502–518. Springer, Heidelberg (2003)
47. Fallah, F., Devadas, S., Keutzer, K.: Functional vector generation for HDL models using linear programming and boolean satisfiability. IEEE Trans. CAD **CAD-20**(8), 994–1002 (2001)
48. Fischetti, M., Lodi, A.: Local branching. Math. Programm. **98**(1–3), 23–47 (2003)
49. Forrest, J.J.H.: COIN branch and cut. COIN-OR, http://www.coin-or.org
50. Forrest, J.J.H., de la Nuez, D., Lougee-Heimer, R.: CLP user guide. COIN-OR, http://www.coin-or.org/Clp/userguide

51. Fügenschuh, A., Martin, A.: Computational integer programming and cutting planes. In: Aardal, K., Nemhauser, G.L., Weismantel, R. (eds.) Discrete Optimization. Handbooks in Operations Research and Management Science, Chap. 2, vol. 12, pp. 69–122. Elsevier, Amsterdam (2005)
52. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL(T): fast decision procedures. In: Proceedings of the International Conference on Computer Aided Verification (CAV-04). pp. 26–37 (2004)
53. Gauthier, J.M., Ribière, G.: Experiments in mixed-integer linear programming using pseudocosts. Math. Programm. **12**(1), 26–47 (1977)
54. Glover, F., Laguna, M.: Tabu Search. Kluwer, Boston (1997)
55. Gomory, R.E.: Solving linear programming problems in integers. In: Bellman, R., Hall, J.M. (eds.) Combinatorial Analysis Symposia in Applied Mathematics X, pp. 211–215. American Mathematical Society, Providence (1960)
56. Gomory, R.E.: An algorithm for integer solutions to linear programming. In: Graves, R.L., Wolfe, P. (eds.) Recent Advances in Mathematical Programming, pp. 269–302. McGraw-Hill, New York (1963)
57. Gottlieb, J., Paulmann, L.: Genetic algorithms for the fixed charge transportation problem. In: Proceedings of the 1998 IEEE International Conference on Evolutionary Computation, pp. 330–335. IEEE Press, New York (1998)
58. Hooker, J.N.: Planning and scheduling by logic-based Benders decomposition. Oper. Res. **55**(3), 588–602 (2007)
59. ILOG. CPLEX. http://www.ilog.com/products/cplex
60. Jain, V., Grossmann, I.E.: Algorithms for hybrid MILP/CP models for a class of optimization problems. INFORMS J. Comput. **13**(4), 258–276 (2001)
61. Jerraya, A.A., Wolf, W.: Multiprocessor Systems-on-Chips. The Morgan Kaufmann Series in Systems on Silicon. Elsevier/Morgan Kaufman, Boston/San Francisco (2004)
62. Johnson, E.L., Padberg, M.W.: Degree-two inequalities, clique facets, and biperfect graphs. Ann. Discret. Math. **16**, 169–187 (1982)
63. Joswig, M., Pfetsch, M.E.: Computing optimal Morse matchings. SIAM J. Discret. Math. **20**(1), 11–25 (2006)
64. Kaibel, V., Peinhardt, M., Pfetsch, M.E.: Orbitopal fixing. In: Fischetti, M., Williamson, D. (eds.) Proceedings of the 12th Integer Programming and Combinatorial Optimization conference (IPCO). LNCS, vol. 4513, pp. 74–88. Springer, Heidelberg (2007)
65. Koch, T.: Rapid mathematical programming or how to solve sudoku puzzles in a few seconds. In: Haasis, H.-D., Kopfer, H., Schönberger, J. (eds.) Operations Research Proceedings 2005, pp. 21–26 (2006)
66. Kutschka, M.: Algorithmen zur Separierung von $\{0, \frac{1}{2}\}$-Schnitten. Master's Thesis, Technische Universität Berlin (2007)
67. Land, A., Powell, S.: Computer codes for problems of integer programming. Ann. Discret. Math. **5**, 221–269 (1979)
68. Letchford, A.N., Lodi, A.: Strengthening Chvátal–Gomory cuts and Gomory fractional cuts. Oper. Res. Lett. **30**(2), 74–82 (2002)
69. Li, C.M., Anbulagan: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of 15th International Joint Conference on Artificial Interlligence (IJCAI 1997), pp. 366–371. Morgan Kaufmann, Japan (1997)
70. Li, C.M., Anbulagan: Look-ahead versus look-back for satisfiability problems. In: Proceedings of third international conference on Principles and Practice of Constraint Programming (CP 1997), pp. 342–356. Springer, Autriche (1997)
71. Linderoth, J.T., Ralphs, T.K.: Noncommercial software for mixed-integer linear programming. In: Karlof, J. (ed.) Integer Programming: Theory and Practice, Operations Research Series, pp. 253–303. CRC Press, Boca Raton (2005)
72. Linderoth, J.T., Savelsbergh, M.W.P.: A computational study of search strategies for mixed integer programming. INFORMS J. Comput. **11**, 173–187 (1999)
73. Madre, J.C., Billon, J.P.: Proving circuit correctness using formal comparison between expected and extracted behavior. In: Proceedings of the 25th Design Automation Conference, pp. 205–210 (1988)
74. Manquinho, V., Roussel, O.: Pseudo Boolean evaluation (2007). http://www.cril.univ-artois.fr/PB07/
75. Marchand, H.: A polyhedral study of the mixed knapsack set and its use to solve mixed integer programs. Ph.D. Thesis, Faculté des Sciences Appliquées, Université catholique de Louvain (1998)
76. Marchand, H., Wolsey, L.A.: Aggregation and mixed integer rounding to solve MIPs. Oper. Res. **49**(3), 363–371 (2001)

77. Markowitz, H.M., Manne, A.S.: On the solution of discrete programming problems. Econometrica **25**, 84–110 (1957)
78. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**, 506–521 (1999)
79. Martin, A.: Integer programs with block structure. Habilitations-Schrift, Technische Universität Berlin (1998). http://www.zib.de/Publications/abstracts/SC-99-03/
80. Martin, A., Weismantel, R.: The intersection of knapsack polyhedra and extensions. In: Bixby, R.E., Boyd, E., Ríos-Mercado, R.Z. (eds.) Integer programming and combinatorial optimization. Proceedings of the 6th IPCO Conference, pp. 243–256 (1998). http://www.zib.de/Publications/abstracts/SC-97-61/
81. Mitra, G.: Investigations of some branch and bound strategies for the solution of mixed integer linear programs. Math. Programm. **4**, 155–170 (1973)
82. Mittelmann, H.: Decision tree for optimization software: Benchmarks for optimization software. http://plato.asu.edu/bench.html
83. Mosek. MOSEK Optimization tools. http://www.mosek.com
84. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the Design Automation Conference, July (2001)
85. Nemhauser, G.L., Trick, M.A.: Scheduling a major college basketball conference. Oper. Res. **46**(1), 1–8 (1998)
86. Nunkesser, M.: Algorithm design and analysis of problems in manufacturing, logistic, and telecommunications: An algorithmic jam session. Ph.D. Thesis, Eidgenössische Technische Hochschule ETH Zürich (2006)
87. Orlowski, S., Koster, A.M.C.A., Raack, C., Wessäly, R.: Two-layer network design by branch-and-cut featuring MIP-based heuristics. In: Proceedings of the Third International Network Optimization Conference (INOC 2007). Spa, Belgium (2007)
88. Padberg, M.W., van Roy, T.J., Wolsey, L.A.: Valid inequalities for fixed charge problems. Oper. Res. **33**(4), 842–861 (1985)
89. Parthasarathy, G., Iyer, M.K., Cheng, K.T., Wang, L.C.: An efficient finite-domain constraint solver for RTL circuits. In: Proceedings of the International Design Automation Conference (DAC-04) June (2004)
90. Pfetsch, M.E.: Branch-and-cut for the maximum feasible subsystem problem. Report 05-46, ZIB (2005)
91. Ryan, D.M., Foster, B.A.: An integer programming approach to scheduling. In: Wren, A. (ed.) Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling, pp. 269–280. North Holland, Amsterdam (1981)
92. Ryan, L.: Efficient algorithms for clause-learning SAT solvers. Master's Thesis, Simon Fraser University
93. Savelsbergh, M.W.P.: Preprocessing and probing techniques for mixed integer programming problems. ORSA J. Comput. **6**, 445–454 (1994)
94. Thienel, S.: ABACUS—A Branch-and-Cut System. Ph.D. Thesis, Institut für Informatik, Universität zu Köln (1995)
95. Timpe, C.: Solving planning and scheduling problems with combined integer and constraint programming. OR Spectr. **24**(4), 431–448 (2002)
96. VALSE-XT: Eine integrierte Lösung für die SoC-Verifikation (2005). http://www.edacentrum.de/ekompass/projektflyer/pf-valse-xt.pdf
97. van Roy, T.J., Wolsey, L.A.: Valid inequalities for mixed 0-1 programs. Discret. Appl. Math. **14**(2), 199–213 (1986)
98. Wolsey, L.A.: Valid inequalities for 0-1 knapsacks and MIPs with generalized upper bound constraints. Discret. Appl. Math. **29**, 251–261 (1990)
99. Wolter, K.: Implementation of cutting plane separators for mixed integer programs. Master's Thesis, Technische Universität Berlin (2006)
100. Wunderling, R.: Paralleler und objektorientierter Simplex-Algorithmus. Ph.D. Thesis, Technische Universität Berlin (1996). http://www.zib.de/Publications/abstracts/TR-96-09/
101. Zeng, Z., Kalla, P., Ciesielski, M.: LPSAT: a unified approach to RTL satisfiability. In: Proceedings of Conference on Design, Automation and Test in Europe (DATE-01) Munich, March (2001)
102. Zuse Institute Berlin. SCIP: solving constraint integer programs. http://scip.zib.de