Evaluating the Performance of Solvers for Integer-Linear Programming

Arno Luppold
Hamburg University of Technology, Institute of Embedded Systems, Germany
Arno.Luppold@tuhh.de

Dominic Oehlert
Hamburg University of Technology, Institute of Embedded Systems, Germany
Dominic.Oehlert@tuhh.de

Heiko Falk
Hamburg University of Technology, Institute of Embedded Systems, Germany
Heiko.Falk@tuhh.de

Abstract

Optimizing embedded systems often boils down to solving complex combinatorial optimization problems. Integer-Linear Programming (ILP) turned out to be a powerful tool to solve these problems, as beyond traditional constraints, Boolean variables may be used to model complex logical expressions and conditionals. One of the key technical aspects is to be able to efficiently express these relations within the ILP. This paper presents formalized solutions for these issues, as well as an assessment of common ILP solvers. Additionally, the performance impact is illustrated using a compiler based cache aging optimization.

2012 ACM Subject Classification Mathematics of computing → Solvers; Mathematics of computing → Integer programming; Computer systems organization → Embedded and cyber-physical systems

Keywords and phrases Integer-Linear Programming, ILP, Solvers, Evaluation

Category Technical Report

Funding This work received funding from Deutsche Forschungsgemeinschaft (DFG) under grant FA 1017/3-1. This work is part of a project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 779882.

Digital Object Identifier 10.15480/882.1839

1 Introduction

Over the last decades, the complexity of embedded systems has grown drastically. If the initial design of such a system does not meet all of its requirements, it must be optimized specifically towards this criteria.

Traditionally, this task is performed manually by the system designer. E.g., certain functions are manually mapped to a faster but small memory or code snippets are reformulated. However, this is a tedious task as every change may lead to unforeseen side effects which, again, might even degrade the system's adherence to the design requirements.

Compilers have proven to be powerful tools in order to automatically optimize a system with respect to hard design constraints. E.g., Falk et al. provide a compiler framework specifically tailored towards optimizing programs' Worst-Case timing behavior [5]. Recent

research also provides ways in order to automatically optimize multi-tasking systems with regard to their schedulability [11].

These optimizations can be boiled down to a combinatorial optimization problem. One powerful tool to solve such problems is Integer-Linear Programming (ILP). By definition, an ILP contains a set of *linear* equations or inequations, where all variables must be of integer-values only. Constant coefficients may be real numbers. Despite these limitations, ILP constraints may be used to model Boolean expressions, conditionals and give the mathematical foundations to solve complex combinatorial problems in the domain of embedded system design.

One of the key benefits of ILPs is that their solvers are able to return provably optimal solutions. Although solving ILPs is NP-hard, current solvers are able to solve ILPs with tens of thousands of variables and constraints in a couple of minutes or even seconds. Modern ILP solvers like Gurobi natively support Boolean logic like AND or OR operations and even conditional constraints directly using so-called "general" constraints. However, this makes the model incompatible with other solvers which might not support the same so-called general constraints. Therefore, it is also important to be able to express these constraints as a set of regular integer-linear (in)equations.

Apart from solver-dependencies, the question arises if and to what extent these general constraints impact the time needed to solve the ILP. In the following, we will therefore not only give a guide on how to model commonly used logical and arithmetical expressions as sets of ILP constraints. We also aim at evaluating these models for their impact on solving times for the three commonly used solvers lpSolve, IBM CPLEX and Gurobi.

Finally, we show-case that these concepts may not only be used for worst-case timing oriented optimizations, as shown by previous research [9, 15] but also come handy for other optimization problems in the domain of embedded system design. To illustrate this, we evaluate ILP solvers' performance on an ILP based cache aging mitigation optimization.

This is not the first work which describes Boolean, arithmetic and other logic expressions in ILP. However, between manuals, white papers and manufacturer websites, it is hard to find reliable and concise descriptions. Often, precise descriptions of, e.g., bounds on necessary auxiliary variables, are not given or their safety is not clear. Evaluations on the impact of these formulations on the solving times and comparisons between different ILP solvers are mostly missing as well.

The key contributions of this paper are:

- We provide an overview of ILP expressions commonly needed to model logical and arithmetical expressions as a set of linear constraints.
- We intend to give a guide to the interested reader in order to show current possibilities and limitations when using ILPs for problems commonly found in the domain of embedded system design.
- We give safe upper and lower bounds on any auxiliary variables which must be introduced for some kinds of expressions.
- We compare the performance impact on different ILP solvers with and without solver-specific features like Gurobi's "general constraints".

This paper is organized as follows: Section 2 will first give a brief introduction to the history of ILP, previous approaches and use cases in embedded system design. Section 3 gives an overview of the mathematical notations used in this paper, as well as the evaluation setup. Section 4 introduces basic Boolean operations for ILPs. Section 5 covers more complex logical constructs like conditionals or base number decomposition. Section 6

provides an exemplary real-world optimization problem illustrating the combination of most aforementioned operations and constructs. This paper closes with a conclusion.

2 Related Work

Many problems in the design of embedded systems can be reduced to a combinatorial problem with integer or sometimes even binary coefficients. To cope with the often huge complexity of the optimization problem, the optimization may be split into smaller parts which are then optimized independently. However, such local optimizations may come at a cost. E.g., memory consumption might increase, leading to changed (and not necessarily improved) cache behavior in other parts. As a result, the compiler has to tackle the issue of optimizing those parts of the program which lead to the best global improvement - and cannot simply rely on optimizing local parts of the program in isolation.

One way of solving this issue is by expressing the problem as a set of integer-linear (in)equations. The roots of this kind of mathematics go back to ancient Chinese mathematicians [16]. However, major improvements in order to solve large-scale (in)equation systems were made by Dantzig et al. in the mid 1950s [3]. They proposed the so-called "simplex method", allowing to not only solve large-scale linear (in)equation systems efficiently, but also giving a provably optimal solution with regard to a given objective function. Later on, Dantzig also published an exhaustive description of the simplex method as well as possible applications as a book [2]. Modeling an optimization problem as a set of linear (in)equations was respectively called "Linear Programming" (LP).

Over the years, numerous tools emerged which build up on the simplex algorithm in order to provide a user-friendly and computationally efficient front-end for solving these ILPs. For the following work, we picked out 3 of them:

- lpSolve [10]
- IBM CPLEX [7]
- Gurobi [6]

We picked lpSolve as it is commonly used due to its free availability under the GNU Lesser General Public License 2.1. CPLEX and Gurobi are both commercial ILP solvers. Both solvers aim at solving large-scale ILPs in minimal time.

In the domain of embedded system design, especially in use cases such as the resource mapping example provided in Section 6, so-called *Integer*-Linear Programming (ILP) is used. This is a specialized version of LPs where all variables are forced to integral values only. Especially due to fast commercial solvers like CPLEX and Gurobi, ILPs are frequently used for optimally solving common problems like the efficient analysis of the worst-case execution path through a program [9] or Memory-Allocation in Worst-Case Execution Time Aware Compilation [15, 4, 13].

For these kinds of optimizations, integer variables with a value of only 0 or 1 are introduced into the ILP as so-called *binary* variables. Additionally, *logical* relations between these variables like, e.g., AND or XOR often have to be modeled. Previous works do, of course, implement these features but often only describe them marginally. Some works exist [1] which give an overview of some "Integer Linear Programming Tricks". However, they do not cover binary arithmetic and they do not evaluate the impact of their structures on different solvers. Additionally, to the best of our knowledge, some commonly used formulations like if-then-else constraints with variable result values have not been formally described before at all.

Performance analyses of different solvers have also been done before, e.g., [12]. They

mainly focus on complete real-world problems, disregarding the possible effects of specific logical or conditional operations on the ILP's complexity.

In the following, we are aiming at closing this gap. We will provide a brief but complete formal description of the most common Boolean and conditional operations. We then evaluate each problem class stand-alone showing its performance impact with growing number of logical or conditional operations within the ILP. Because modern solvers like Gurobi feature solver-specific implementations of some Boolean operations, e.g., logical AND or OR, we will also investigate the impact of these solver-specifics on the time needed to find an optimal solution.

3 Prerequisites

This section gives an overview of notational conventions used throughout this paper, as well as an overview of the evaluation setup.

3.1 Nomenclature

We formulate an ILP constraint as follows:

$$B_0 \cdot a_0 + B_1 \cdot a_1 + \cdots + B_n \cdot a_n \geq C \tag{1}$$

Relational operators may be $\leq$, $\geq$ or $=$. Constants in the ILP are always written in capital letters while variables are written in lower-case letters.

As we solely focus on *Integer*-Linear Programs, we assume that all variables are integer variables. Unless stated otherwise, constants are not limited to integers but may be arbitrary real numbers. If the valid range of an ILP variable a must be known at model creation time, we denote the valid interval of its result values by $\left[\check{A}_0, \hat{A}_0\right]$. $\check{A}_0$ is the minimum allowed value for a_0 and $\hat{A}_0$ the maximum allowed value. Unless otherwise stated, $-\infty < \check{A}_0 < \hat{A}_0 < \infty$. We consider these maximum and minimum allowed values to be constants in the ILP model. Note that in practice, ILP solvers may have far more rigorous restrictions on the maximum and minimum values of integer variables.

We call a variable to be "Boolean", if its valid range is in the interval $[0, 1]$.

Although the ILP solvers will not distinguish between logical "input" and "result" variables but treat all variables equally, we will use these terms to ease understanding of the formulations. Especially when considering Boolean operations like $x = a \wedge b$, we call a and b *input* variables and x the *result* variable. This is solely used for the purpose of better readability and has no deeper meaning in terms of the formal ILP formulation.

When we use any special solver-specific features (mainly for performance evaluation reasons), we note so explicitly. Some of the upcoming constraints may be simplified by using solvers' special features. Unless we explicitly aim at evaluating the performance of these special features, we will intentionally not use any solver specific features. As a result, we give solver independent formulations for each formulated problem.

Note that logical operations which are modeled within the ILP are set in sans serif font. Therefore, max and min denote logical operations. On the contrary, min and max are used to signify the ILP's objective function which can be set to either minimize or maximize. Despite similar notation, the corresponding use is always made clear in the respective context.

3.2 Evaluation Setup

In all evaluations, we compared the three solvers lpSolve 5.5.0.13, IBM CPLEX 12.5 and Gurobi 7.5.0. Gurobi was evaluated in 2 different configurations: First, logical operations like AND were modeled using regular ILP (in)equations. Then, those operations were modeled using Gurobi's specialized general constraints as far as supported by Gurobi.

The resulting ILPs were solved on a dual CPU Intel XEON server with 96 GB RAM on Ubuntu 16.04.3 LTS. Each CPU consists of 10 cores with a nominal speed 2.30 GHz.

In real-world setups, the user is usually interested in the actual wall time a solver needs to return the optimal solution to a given problem. Therefore, we did not artificially restrict the maximum number of cores to be used by the solver. Therefore, including Intel's Hyper Threading Technology, both CPLEX and Gurobi spawned up to 40 threads. lpSolve only makes use of one thread, therefore, the multi-core setup could not enhance lpSolve's performance. Each solver was called with a time limit of 2 h wall time.

The exact evaluation setup for each operation is described in the respective subsection. All experiments were conducted based on uniform randomly generated numbers. To reduce the risk of statistical spikes in the solving times, we repeated each experiment 10 times with different numbers. To ensure fairness for all solvers, all solvers always had to solve the identical 10 repeats with identical numbers.

In the evaluation, we then plot the arithmetic mean over each of the 10 runs. When a solver was not able to solve all of the 10 repeats within the time limit but was canceled due to timeout in some cases, we mark this explicitly in the respective section.

4 Boolean Operations

This section covers commonly needed Boolean operations. Proofs of correctness can easily be deducted by trying out all possible combinations for the Boolean input variables. The formulations are mostly quite straight-forward.

4.1 NOT

The most basic Boolean operation is the logical NOT operation, used to negate a Boolean ILP variable:

$$x := \bar{a} \tag{2}$$

$$x, a \in [0, 1] \tag{3}$$

This results in the simple ILP formulation:

$$x = 1 - a \tag{4}$$

4.2 AND

A logical AND operation is defined as:

$$x := a \wedge b \tag{5}$$

$$x, a, b \in [0, 1] \tag{6}$$

This can be transcribed into ILP inequations as:

$$x \geq a + b - 1 \tag{7}$$
$$x \leq a \tag{8}$$
$$x \leq b \tag{9}$$

Eq. (7) ensures that x is forced to 1 in case that both a and b equal 1. Eqs. (8) and (9) ensure that x is forced to 0 if either a or b are 0, respectively. In this case, the first equation will resolve to $x \geq -1$ and will thus be fulfilled as well.

Gurobi has a so-called general constraint for this logical expression. Therefore, in Gurobi, an AND may be expressed directly.

If a logical AND is to be created over multiple variables, the logic operations can easily be chained. I.e.,

$$x \coloneqq a \wedge b \wedge c \tag{10}$$

can be transformed into

$$v = a \wedge b \tag{11}$$
$$x = v \wedge c \tag{12}$$
$$a, b, c, v, x \in [0, 1] \tag{13}$$

The additional binary variable v is inserted as an auxiliary variable. In Gurobi's AND expression, multiple variables may directly be connected, thus no additional auxiliary variables are needed.

4.3 OR

The logical OR is written as:

$$x \coloneqq a \vee b \tag{14}$$
$$x, a, b \in [0, 1] \tag{15}$$

It can be described as ILP formulas by:

$$x \geq a \tag{16}$$
$$x \geq b \tag{17}$$
$$x \leq a + b \tag{18}$$

Eqs. (16) and (17) enforce x to be 1 if either a or b are 1. Eq. (18) ensures that x is set to 0 if both a and b are 0. In case that $a \equiv b \equiv 1$, this last equation still holds ($x \leq 2$).

In analogy to the AND operation, multiple ORs may be chained. Also, Gurobi has a general constraint for OR as well, allowing to directly model an OR over 2 or more binary variables, as described in detail in the previous section.

4.4 XOR

We define the logical XOR as follows:

$$x \coloneqq a \oplus b \tag{19}$$
$$x, a, b \in [0, 1] \tag{20}$$

In contrast to AND and OR, there is no general constraint in Gurobi for the XOR. Therefore, all solvers must always use the ILP formulation given below:

$$x \geq a - b \tag{21}$$
$$x \geq b - a \tag{22}$$
$$x \leq a + b \tag{23}$$
$$x \leq 2 - a - b \tag{24}$$

Eqs. (21) and (22) force x to 1 if the difference between a and b is non-null, i.e., a and b differ. Eq. (23) covers the case that $a \equiv b \equiv 0$ and forces $x \equiv 0$. Finally, Eq. (24) forces $x \equiv 0$ in the case of $a \equiv b \equiv 1$.

4.5 Evaluation of Boolean Operations

We evaluated each of the AND, OR and XOR operators in two different ways:

Independent: We created ILPs with a pre-defined number of independent logical operations. For a sample size of S, the ILP for evaluating the AND would be:

$$x_0 \coloneqq a_0 \wedge b_0 \tag{25}$$
$$x_1 \coloneqq a_1 \wedge b_1 \tag{26}$$
$$x_2 \coloneqq a_2 \wedge b_2 \tag{27}$$
$$\dots \tag{28}$$
$$x_{S-1} \coloneqq a_{S-1} \wedge b_{S-1} \tag{29}$$
$$\forall i = 0, \dots, S-1 : x_i, a_i, b_i \in [0, 1] \tag{30}$$

The objective function is set to:

$$\max \left(\sum_{i=0}^{S-1} x_i \right) \tag{31}$$

Chained: In this setup, we chained all constraints with their preceding constraints. For a sample size of S, this basically connects S randomly pre-determined binary variables by the analyzed logical operation. E.g., for the AND operation, this will result in:

$$x_0 \coloneqq a_0 \wedge s \tag{32}$$
$$x_1 \coloneqq a_1 \wedge x_0 \tag{33}$$
$$\dots \tag{34}$$
$$x_{S-1} \coloneqq a_{S-1} \wedge x_{S-2} \tag{35}$$
$$\forall i = 0, \dots, S-1 : x_i, a_i, \in [0, 1] \tag{36}$$
$$s \in [0, 1] \tag{37}$$

Due to the fact that the first condition does not have any predecessor, we introduce an additional binary variable s which we connect x_0 with. The objective is set to maximize the final (logical) decision variable:

$$\max \left(x_{S-1} \right) \tag{38}$$

To enforce one exact solution, uniformly random constants were generated for all logical input variables a_i, b_i as well as s. The values are enforced in the ILP by setting an appropriate

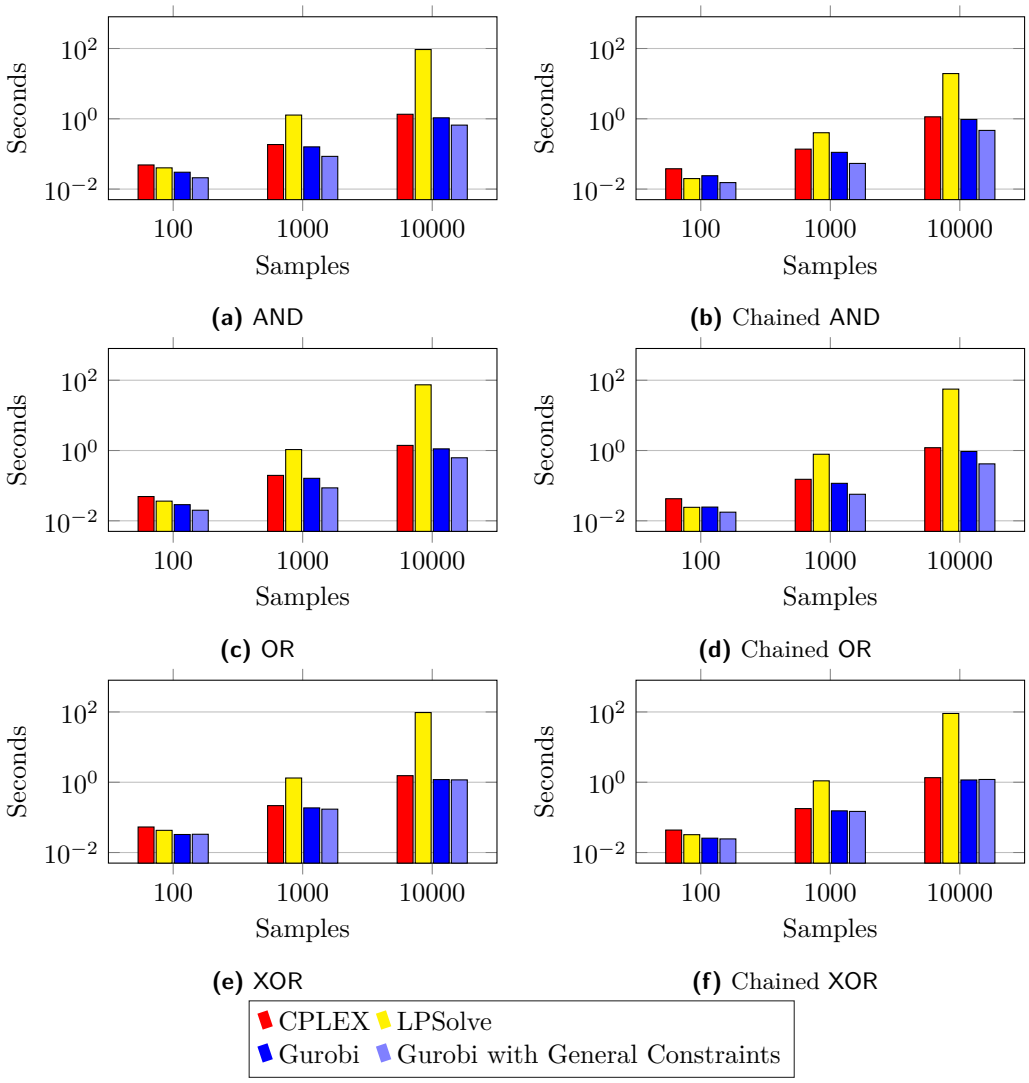

Figure 1 Solving times for Boolean operations for different sample sizes.

constraint for each variable. E.g., if $a_0 \equiv 1$ and $a_1 \equiv 0$, this results in the constraints:

$$a_0 = 0 \tag{39}$$

$$a_1 = 1 \tag{40}$$

To give a good overview of the performance of the different ILP solvers, we chose $S = \{100, 1000, 10000\}$. We randomly generated new values for the fixed variables for each sample size and used the same values for each solver to obtain comparable results.

Fig. 1 shows the evaluation results of the Boolean operations. It can be seen that for small sizes of 100 logical operations, the chosen solver does not really matter. However, starting with 1000 samples, lpSolve's results start getting worse. For 1000 samples, lpSolve needs almost 100 times longer than CPLEX or Gurobi to solve the ILP.

It is noteworthy to mention that in any case, Gurobi outperformed the other solvers and, at the same time, using Gurobi's internal versions of AND and OR brings a noticeable additional performance boost.

Additionally, it is interesting that the results for the individual and the chained logical operations look almost identical. Apart from LPsolve in Fig. 1b, which outperforms its results for the individual AND (Fig. 1a), there is no notable difference in solving times.

5 Complex Operations

After covering basic Boolean operations in the previous section, this section will focus on more complex operations, namely conditional constraints as well as base number decomposition. Conditional constraints are then used to model max and min operations.

5.1 Conditional Constraints

Occasionally, a constraint in an ILP only has to hold if a certain condition is (not) met. This can be modeled by introducing a binary indicator variable b which denotes whether this condition is met or not. Consider, e.g., the computational load of a task in a multi-core setup which only has to be accounted for on a given CPU, if a binary indicator variable has a given value. This operation is still fairly straight-forward, but needed as a basis for upcoming formulations.

To express that a given constraint only has to hold if a binary decision variable b is 1, we denote:

$$b \equiv 1 \Rightarrow a_0 \cdot C_0 + a_1 \cdot C_1 + \dots = D \tag{41}$$

Accordingly, $b \equiv 0 \Rightarrow \dots$ describes that the constraint must hold if b is chosen to be 0. Some solvers like, e.g., Gurobi, offer the possibility to describe these conditional relationships directly, in the case of Gurobi with a syntax very similar to the one in Eq. (41). In this case, the linear equation might use $\geq$, $\leq$ or $=$ as comparison operator. In the general case, however, the equality relation cannot be expressed directly. Thus, in the following, we will limit ourselves to using conditional constraints only for $\geq$ and $\leq$ constraints. Luckily, the equal-condition in Eq. (41) can easily be reformulated:

$$b \equiv 1 \Rightarrow a_0 \cdot C_0 + a_1 \cdot C_1 + \dots \leq D \tag{42}$$

$$b \equiv 1 \Rightarrow a_0 \cdot C_0 + a_1 \cdot C_1 + \dots \geq D \tag{43}$$

If the solver does not support conditionals natively, a sufficiently large constant Z multiplied by the binary indicator variable b is added to the greater-equal side of the constraint. This term equals Z in case that the indicator variable b does *not* hold the desired value. I.e., in case of $b \equiv 0 \Rightarrow$, this results in:

$$a_0 \cdot C_0 + a_1 \cdot C_1 + \dots + b \cdot Z \geq D \tag{44}$$

$$a_i \in [\check{A}_i, \hat{A}_i] \forall i \tag{45}$$

If b is chosen to 0 by the ILP solver, the term $b \cdot Z$ will result in zero, thus the inequation must hold. Otherwise, if $b \equiv 1$, the term $b \cdot Z$ results in Z and if Z was chosen sufficiently large, the equation will always be fulfilled, regardless of the values of the other variables.

Accordingly, in case of an $b \equiv 1 \Rightarrow$ constraint, $\bar{b} \cdot Z = Z - b \cdot Z$ is added instead.

The minimum value for Z is easily determined: If the indicator variable b "disables" the constraint, Eq. (44) will always hold as long as

$$Z \geq D - \check{A}_0 \cdot C_0 - \check{A}_1 \cdot C_1 - \dots \tag{46}$$

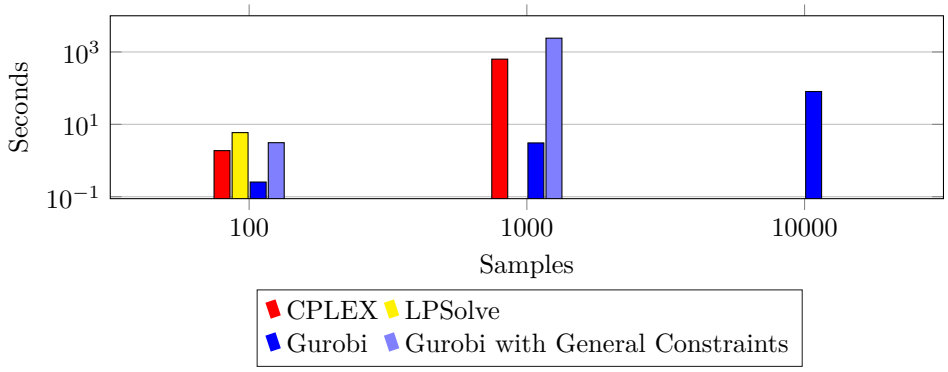

Figure 2 Solving times for the conditional operation for different sample sizes. The bars for lpSolve (1000 and 10000 samples) and IBM CPLEX and Gurobi with general constraints enabled (10000 samples) are missing because these solvers did not manage to solve any of the ILPs of the respective size within the 2 h time limit.

Note, that the *lower* bounds on all variables have to be used here to guarantee a safe result for Z.

As a result, these conditional constraints can only be used if the designer of the optimization framework knows a sound lower bound on the maximum values of each integer variable used in the constraint. Due to the fact that common solvers use signed 32 bit integers for their underlying algorithms, choosing $Z := 2^{31} - 1$ might seem to be a good idea. However, using such large values will most certainly lead to high solving times or broken results due to numeric issues with floating-point arithmetic in the analyzer's solving algorithms. As a last resort, if no safe bound is known, solver-specific solutions can be chosen if the solver supports such. However, the upcoming evaluation will show that, e.g., Gurobi is not that efficient at solving such ILPs. Using the solver's general approach for conditionals should therefore only be used in very small ILPs where performance is not an issue, or as a last resort.

Evaluation

Solving ILPs consisting of conditional constraints turned out to be a challenging problem for all tested ILP solvers. We evaluated the conditionals' performance for sample sizes of $S = \{100, 1000, 10000\}$. For each sample size, we created the same number of constraints with 3 summands and identical integer variables but differing binary indicator variables. E.g., for a sample size of 100, the resulting ILP is:

$$b_0 \equiv 1 \Rightarrow a_0 \cdot C_{0,0} + a_1 \cdot C_{0,1} + a_2 \cdot C_{0,2} \geq D_0 \tag{47}$$
$$b_1 \equiv 1 \Rightarrow a_0 \cdot C_{1,0} + a_1 \cdot C_{1,1} + a_2 \cdot C_{1,2} \geq D_1 \tag{48}$$
$$\dots \tag{49}$$
$$b_{99} \equiv 1 \Rightarrow a_0 \cdot C_{99,0} + a_1 \cdot C_{99,1} + a_2 \cdot C_{99,2} \geq D_{99} \tag{50}$$
$$\forall i = 0, \dots, 99 : a_i \in [-10, 10] \tag{51}$$
$$\forall i = 0, \dots, 99 : b_i \in [0, 1] \tag{52}$$

Similar to the evaluation of the Boolean operators, we randomly generated all constants for each sample size with uniform distribution. The allowed values for the randomly generated constants $C_{i,j}$ and D_i were also in the range of $[-10, 10]$. While this seems to be somewhat arbitrary and quite small, any greater values, especially in the number of summands of

each inequation drastically increased the solving time for all solvers, thus prohibiting any meaningful evaluation.

Fig. 2 shows the results for the evaluation of the conditional constraints. Obviously, despite its user-friendliness, Gurobi's built-in conditional constraints proof to be exceptionally bad for solving performance. The benefit of not having to think about a sufficiently large constant Z comes at the cost of a massive increase in solving time. As the results show, Gurobi has a slow down by more than a factor of 100 on average, when using the general constraints, instead of providing an appropriately chosen large constant Z.

Additionally, it can be seen that lpSolve fails to solve even one of the problems with a sample size of 1000 whereas Gurobi finishes in under 10 s on average (without conditional expressions). For a sample size of 10000, both CPLEX and Gurobi with its native conditional constraints are not able to solve any of the given problems within the 2 h time cap. With the solver-independent formulation, however, Gurobi is able to solve the problems in just over 80 s on average. Due to these results we did not use Gurobi's conditional constraints in any of the following evaluations. Instead, we solely use the general constraints from Section 4 when evaluating Gurobi with general constraints.

5.2 If-Then-Else Structures

It is often required to formulate if-then-else-like structures in the model, such as

$$x \coloneqq \begin{cases} a_0 \cdot C_0 & \text{if } b \equiv 1, \\ a_1 \cdot C_1 & \text{else.} \end{cases} \tag{53}$$

where b is a binary decision variable. Theoretically, such an if-then-else expression can be formulated using two conditional constraints as presented in Section 5.1:

$$b \equiv 1 \Rightarrow x = a_0 \cdot C_0 \tag{54}$$

$$b \equiv 0 \Rightarrow x = a_1 \cdot C_1 \tag{55}$$

However, for an if-then-else structure this can be simplified resulting in a smaller auxiliary constant.

The if-then-else structure is therefore expressed via the following constraints:

$$x \geq a_0 \cdot C_0 - (1 - b) \cdot Z \tag{56}$$

$$x \leq a_0 \cdot C_0 + (1 - b) \cdot Z \tag{57}$$

$$x \geq a_1 \cdot C_1 - b \cdot Z \tag{58}$$

$$x \leq a_1 \cdot C_1 + b \cdot Z \tag{59}$$

$$a_0 \in [\check{A}_0, \hat{A}_0], a_1 \in [\check{A}_1, \hat{A}_1] \tag{60}$$

In case b is set to 0, Eqs. (56) and (57) only restrict x to be in the range $a_0 \cdot C_0 - Z \leq x \leq a_0 \cdot C_0 + Z$. In the contrary case, x is forced to be equal to $a_0 \cdot C_0$. Eqs. (58) and (59) enforce the corresponding relationships, yet for a_1.

For Eqs. (56) and (57) the constant Z can be chosen as follows: In case of $b \equiv 1$, Z is multiplied by 0 and does not matter. In case of $b \equiv 0$ Eq. (56) can be rewritten to:

$$x \geq a_0 \cdot C_0 - Z \Leftrightarrow Z \geq a_0 \cdot C_0 - x \tag{61}$$

As we only have to consider $b \equiv 0$, we know that the "else" part of Eq. (53) must hold. Thus, $x \in \left[\check{A}_1 \cdot C_1, \hat{A}_1 \cdot C_1\right]$. A requirement on the lower bound of Z is obviously given by

subtracting the minimal x from the maximum allowed value of $a_0 \cdot C_0$. Thus:

$$Z \geq \hat{A}_0 \cdot C_0 - \check{A}_1 \cdot C_1 \tag{62}$$

By proceeding accordingly, the lower bound on Z due to Eq. (57) is bounded by:

$$Z \geq \hat{A}_1 \cdot C_1 - \check{A}_0 \cdot C_0 \tag{63}$$

In total, a safe lower bound for Z is therefore given by:

$$Z \geq \max\left(\hat{A}_0 \cdot C_0 - \check{A}_1 \cdot C_1, \hat{A}_1 \cdot C_1 - \check{A}_0 \cdot C_0\right) \tag{64}$$

The identical bound on Z may also be derived from Eqs. (58) and (59) accordingly.

The if-then-else structure is almost directly used to model the min and max operations. Therefore, we postpone the evaluation to the next section, as results of evaluating if-then-else in confinement would not look any different.

5.3 Min and Max

This section describes how to build a logical minimum and maximum over two linear expressions:

$$x \coloneqq \mathsf{min}\left(a_0 \cdot C_0, a_1 \cdot C_1\right) \tag{65}$$

$$y \coloneqq \mathsf{max}\left(a_0 \cdot C_0, a_1 \cdot C_1\right) \tag{66}$$

In order to formulate a min or max function inside an ILP, the following constraints are created first:

$$a_1 \cdot C_1 \leq a_0 \cdot C_0 + b \cdot Z \tag{67}$$

$$a_0 \cdot C_0 \leq a_1 \cdot C_1 + (1 - b) \cdot Z \tag{68}$$

b is a binary variable. Using Eqs. (67) and (68), b is forced to 1 in case $a_1 \cdot C_1 > a_0 \cdot C_0$ holds. In case of $a_1 \cdot C_1 < a_0 \cdot C_0$, b is forced to 0. In case that both terms are equal, b may be set to either 1 or 0 by the solver.

Using this binary variable and a corresponding if-then-else structure as shown in Section 5.2, a min and max function can be formulated:

$$x \coloneqq \mathsf{min}(a_0 \cdot C_0, a_1 \cdot C_1) = \begin{cases} a_0 \cdot C_0 \text{ if } b = 1, \\ a_1 \cdot C_1 \text{ else.} \end{cases} \tag{69}$$

$$y \coloneqq \mathsf{max}(a_0 \cdot C_0, a_1 \cdot C_1) = \begin{cases} a_1 \cdot C_1 \text{ if } b = 1, \\ a_0 \cdot C_0 \text{ else.} \end{cases} \tag{70}$$

A safe value for the constant Z required for the if-then-else structure and Eqs. (67) and (68) is described in Section 5.2. min or max functions with more than two parameters can be represented by cascading them:

$$\mathsf{min}\left(a_0, a_1, a_2\right) = \mathsf{min}\left(\mathsf{min}\left(a_0, a_1\right), a_2\right) \tag{71}$$

Similar to AND and OR, Gurobi features general constraints which can directly model min and max. When using these general constraints, cascading is not necessary for building the maximum or minimum over several variables. Instead, multiple variables can be added as parameter of the general max and min keywords.

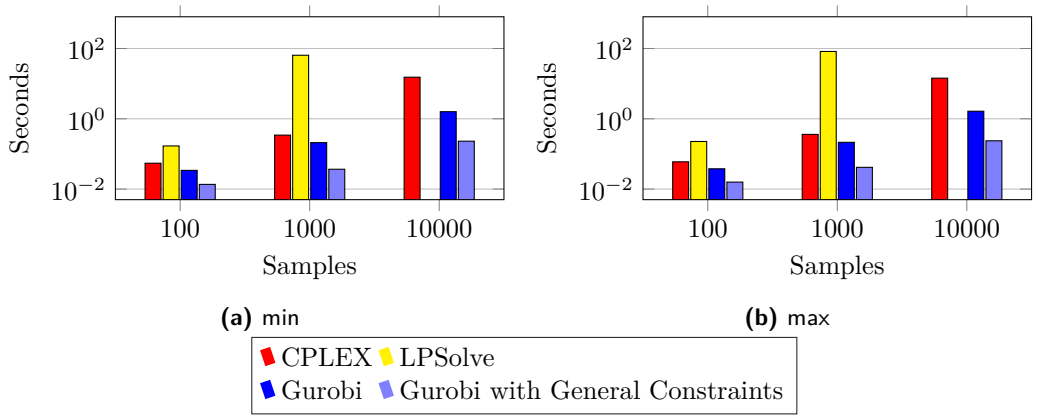

Figure 3 Solving times for min and max operations. The bars for lpSolve at 10000 samples are missing, because lpSolve was not able to solve any ILP of that size within the 2 h time limit.

Evaluation

The evaluations of min and max are very similar. For each sample size $S = \{100, 1000, 10000\}$, we uniform randomly create the same number of integer constants in the range of $[-1000, 1000]$. We then create one general constraint expressing

$$m = \mathsf{max}\,(r_0, r_1, \ldots, r_S) \tag{72}$$
$$m \in [-1000, 1000] \tag{73}$$

The variable bounds of each $r_i, i = 0, \ldots, S$ are fixed to the corresponding random constant previously determined. We repeat this for each S to evaluate each sample size. The ILP for min is built identically. For evaluating max, we set the ILP objective to $\min(m)$. Accordingly, for evaluating min, we use $\max(m)$ as objective.

Fig. 3 shows the results of the evaluation. Both Gurobi and CPLEX manage to solve the ILP for sample sizes 100 and 1000 in well under 1 s. For a sample size of 1000, lpSolve already needs 82 s and for 10000 samples, lpSolve cannot solve the ILP within the 2 h time limit. Notably, especially for the 10000 samples evaluation, Gurobi heavily outperforms CPLEX. For the max operation, Gurobi outperforms CPLEX by a factor of 8.75 (CPLEX: 14 s, Gurobi 1.6 s). With general constraints, Gurobi finishes within 0.2 s on average for 10000 samples. The results of the min operation are very similar to the max with only minimal differences.

5.4 Abs

We want to express:

$$x \coloneqq \mathsf{abs}\,(a) \tag{74}$$
$$x, a \in [\check{A}, \hat{A}] \tag{75}$$

This results in

$$b \equiv 0 \Rightarrow x \geq -1 \cdot a \tag{76}$$
$$b \equiv 0 \Rightarrow x \leq -1 \cdot a \tag{77}$$
$$b \equiv 1 \Rightarrow x \geq a \tag{78}$$
$$b \equiv 1 \Rightarrow x \leq a \tag{79}$$
$$a \leq Z \cdot b - 1 \tag{80}$$
$$a - Z \cdot b + Z \geq 0 \tag{81}$$
$$\tag{82}$$

The idea behind this formulation is the following: First, we introduce a binary variable $b \in [0, 1]$. This variable will be used as an indicator to denote whether a is negative or not. The conditional operator introduced in Section 5.1 is then used to set $x \equiv a$ in case a is positive and $x \equiv -1 \cdot a$ in case it is negative (Equations (76) to (79)).

Eqs. (80) and (81) are used to force the binary helper variable b to 1 if $a \geq 0$ or 0 else: In case that $a < 0$, Eq. (80) is always fulfilled. Then, however, as long as $-1 \cdot a < Z$, Eq. (81) is only fulfilled if $b \equiv 1$.

In the complementary case of $a \geq 0$, Eq. (80) only holds if $b \equiv 1$ and $Z - a \geq 1$. Eq. (81) always holds.

Therefore, we can easily deduce a safe lower bound on Z:

$$Z \geq \max\left(\left|\check{A}\right|, \left|\hat{A}\right|\right) + 1 \tag{83}$$

Evaluation

To evaluate abs, we chose S uniformly distributed random integers for $S = \{100, 1000, 10000\}$. Each random integer $R_i, i = 0, \ldots, S-1$ is in the interval of $[-1000, 1000]$. We then created an ILP as follows:

$$a_0 = \mathsf{abs}\,(r_0) \tag{84}$$
$$a_1 = \mathsf{abs}\,(r_1) \tag{85}$$
$$\cdots \tag{86}$$
$$a_{S-1} = \mathsf{abs}\,(r_{S-1}) \tag{87}$$

The allowed value range for the a_i variables was not limited explicitly. the r_i variables were bound to their pre-determined random value R_i by variable bounds. The optimization goal was then set to

$$\min\left(\sum_{i=0}^{S-1} a_i\right) \tag{88}$$

Fig. 4 shows the results of the evaluation of the ABS operation. For a low number of operations, the choice of the solver does not matter a lot, although even with only 100 ABS operations, significant differences can be observed. While Gurobi needs 38 ms on average to solve the ILP (without its built-in ABS operation) and only 17 ms with using its internal ABS directive, CPLEX needs almost twice the time (62 ms). lpSolve is outperformed by far by both other solvers, with an average solving time of 114 ms. While this performance difference may not matter a lot for small ILPs, as the absolute solving time is always very low, performance differences show up drastically for large-scale ILPs. While Gurobi (without

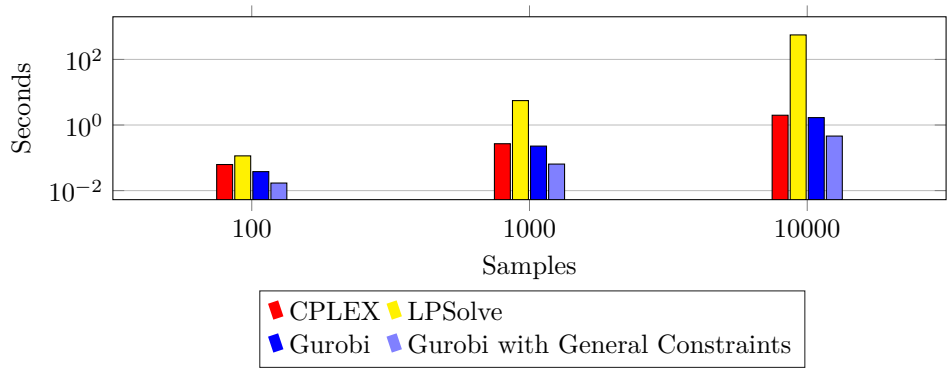

Figure 4 Evaluation results of the ABS operation for different sample sizes.

general constraints) needs 1.7 s on average for 10000 ABS calculations and CPLEX follows with 1.9 s on average, lpSolve needs over 9 minutes to complete the identical calculations. Similar as before, Gurobi with enabled general constraints shows a noticeable additional timing benefit with solving times of only 460 ms on average for the largest sample size.

5.5 Base Number Decomposition

The idea of base number decomposition is to express a positive given number as a sum of its base factors. E.g., the number 102 can be expressed to the base of 10 as:

$$102 = 1 \cdot 10^2 + 0 \cdot 10^1 + 2 \cdot 10^0 \tag{89}$$

Or, to the base of 2:

$$102 = 1 \cdot 2^6 + 1 \cdot 2^5 + 0 \cdot 2^4 + 0 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 0 \cdot 2^0 \tag{90}$$

There are 2 properties of the base number decomposition which can be used to easily perform it as part of an ILP: First, there is exactly one representation of any number for any given base B. Second, the factors prior to each base number are limited in the interval $[0, B-1]$. As a result, $102 = 102 \cdot 10^0$ or similar is not considered to be a valid base number decomposition.

Given this information, any ILP variable may easily be decomposed into its base number representation, as long as an upper bound for the variable is known. To decompose the ILP variable x into its base number representation with base B, the following single constraint may be used:

$$x = b_N \cdot B^N + b_{N-1} \cdot B^{N-1} + \cdots + b_0 \cdot B^0 \tag{91}$$

$$x \in [0, \hat{X}] \tag{92}$$

$$b_i \in [0, B-1], i = 0 \ldots N \tag{93}$$

$$N = \left\lfloor \frac{\log \hat{X}}{\log B} \right\rfloor \tag{94}$$

Eqs. (92) and (93) can be expressed as variable bounds, thus no constraints must be added.

Eq. (94) can be calculated prior to creating the ILP, as long as the maximum value of x, $\hat{X}$, is known a priori.

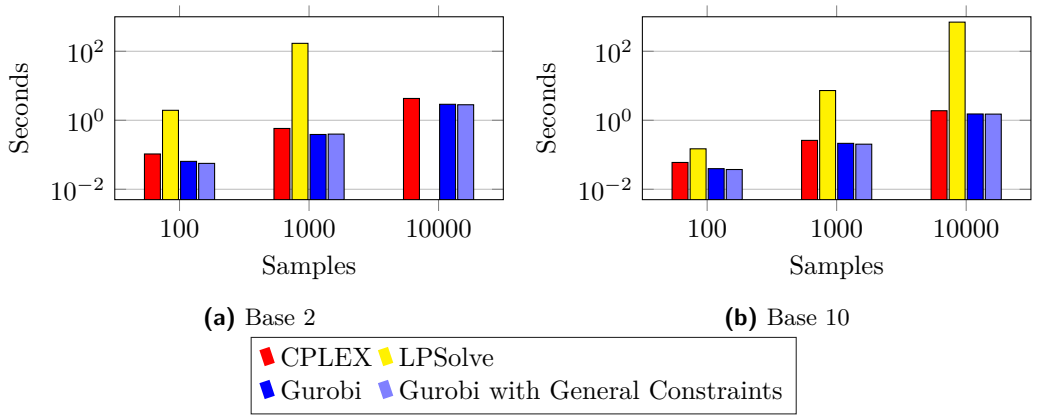

Figure 5 Solving times for base number decomposition for base 2 and base 10. The bar for lpSolve in Fig. 5a is missing, because lpSolve was not able to solve any problem of that size within the 2 h time limit.

A trivial upper bound on x is given by the maximum allowed values of integer variables for a given ILP solver. Common solvers like Gurobi, CPLEX or lpSolve use signed 32 bit numbers for their calculations, so a trivial upper bound is always given by $2^{31} - 1$. However, such large values may easily cause several numerical issues with the solvers' underlying algorithms. Therefore, if no tight upper bound is known for x, the user should consult the solver's manual for recommended maximum values used in the model. Exceeding these values might cause degraded or even wrong results, thus they should be adhered to in any case.

Although this base number decomposition might look useless at first, it has meaningful use cases when optimizing embedded systems. For example, it can be used to model cache behavior for compiler-based code optimizations and will be used in the upcoming case-study in Section 6.

Evaluation

To evaluate the performance of base number decomposition, we created the following setup for sample sizes of $S = \{100, 1000, 10000\}$. For each sample size, we then generated S uniformly distributed random integers in the range of $[0, 1000000]$. We then added the constraint for base number decomposition for each integer to the ILP. E.g., for a sample size of 100, the ILP contains 100 random integers and the constraints to calculate their base number decomposition.

To ensure to only evaluate the performance of the decomposition itself, we additionally defined one single integer variable $r \equiv 0$ and added it to the ILP. We then set the ILP's objective to $\max(r)$. This way, the ILP solver will finish as soon as it finds *any* valid solution which will equal the base number decomposition for all random numbers.

We performed this evaluation once for a base number decomposition with base 2 and once with base 10.

Fig. 5 shows the evaluation results. Obviously, decomposition with a base of 2 proves to be more challenging than with a base of 10. This was quite to expect, as each constraint has $\left\lfloor \frac{\log 1000000}{\log 2} \right\rfloor + 1 = 20$ coefficients, while for the base number decomposition with base 10, only 6 coefficients are needed (cf. Eqs. (91) and (94)).

However, except from lpSolve which shows a significant performance slowdown for base 2 and is not able to finish calculations for 10000 samples, both Gurobi and CPLEX solve all

sample sizes within a couple of seconds.

Due to the fact that base number decomposition does not use any general constraints, differences in the solving time between the two Gurobi instants are purely statistical effects. These stem from the fact that Gurobi uses internal heuristics in order to find a result as soon as possible. Therefore, solving the identical ILP multiple times might have slight performance variations.

6 Case Study

As a case study, we present an ILP-based optimization of a cache-aware data placement. This is a slightly modified adaption of the optimization previously presented by Oehlert et al. [14]. The original approach is tightly integrated into a compiler framework for the Infineon TriCore architecture. We reformulated the approach to make it more abstract and independent from a concrete architecture or compiler framework. This allows us to solely benchmark solvers' behavior without any side-effects from the compiler framework or target architecture specific details.

It includes several of the presented operations, such as Boolean operators, base number decomposition, if-then-else structures and a `max` function.

Typically, data objects are placed in the `.data` section of a program without any special consideration by the compiler. They are simply placed continuously in their order of declaration in the `.data` section. As the address of an object determines in which cache line(s) it may be stored, this placement decision can be crucial to the average, but also to the worst-case timing of a program. A poor data object placement may drastically increase the miss-rate if all frequently accessed data objects are mapped to the same cache line.

Simultaneously, this issue also influences the reliability of the memory. One of the key degradation issues of modern circuits is *negative-bias temperature instability* (NBTI). In order to reduce the effects of NBTI, it is advisable to have a signal probability of 50% per SRAM cell inside the memory [8], hence to achieve a well-balanced usage of each cache line. This in fact may lead to a reduced life-cycle of the embedded system itself in case the cache is not used in a balanced way. The approach presented in the following minimizes the accesses per cache line in order to decrease cache conflicts and degradation.

We assume a direct-mapped data cache with L cache lines and a line size of I bytes. Besides, we assume a data section with a total size of D bytes. The aim is to determine the address for each data object, such that the maximum number of accesses per cache line is minimized.

For each data object O_i of size S_i and start address a_i, a set of ILP variables is introduced which holds the cache lines the object is mapped to.

$$\forall O_i : 0 \leq l_i^0 \leq L - 1 \tag{95}$$

$$\dots$$

$$0 \leq l_i^{F_i} \leq L - 1 \tag{96}$$

$$F_i = \left\lceil \frac{S_i}{I} \right\rceil - 1 \tag{97}$$

For each of these variables holding the cache line to which the object is (partially) mapped

to, a base number decomposition to the base 2 is performed.

$$\forall O_i : \forall j = 0, \ldots, F_i :$$
$$a_i + j \cdot I = b_N^{i,j} \cdot 2^N + ... + b_0^{i,j} \cdot 2^0 \tag{98}$$
$$l_i^j = b_X^{i,j} \cdot 2^0 + b_{X+1}^{i,j} \cdot 2^1 + ... + b_{X+Q}^{i,j} \cdot 2^Q \tag{99}$$

The constants are defined as follows:

$$N = \lfloor \log_2(D) \rfloor \tag{100}$$
$$X = \lceil \log_2(I) \rceil \tag{101}$$
$$Q = \lceil \log_2(L) \rceil - 1 \tag{102}$$

X defines the number of offset bits, whereas Q defines the number of index bits. Using Eqs. (98) and (99), the line variables l_i^j are fixed to the corresponding cache line index to which the data object part is mapped to. It is assumed, that the `.data` section starts at address `0x0`. As long as the real start address of the data section is a multiple of the cache's line size, this will only introduce a constant shift but not change the analysis' behavior.

In case a data object is not mapped to its minimum number of cache lines F_i, it requires an additional cache line l_i^A.

$$a_i + S_i - 1 = b_N^{i,A} \cdot 2^N + ... + b_0^{i,A} \cdot 2^0 \tag{103}$$
$$l_i^A = b_X^{i,A} \cdot 2^0 + b_{X+1}^{i,A} \cdot 2^1 + ... + b_{X+Q}^{i,A} \cdot 2^Q \tag{104}$$

If the data object is only mapped to its minimum number of cache lines, l_i^A equals $l_i^{F_i}$. This is evaluated using the following constraint and inserted for each data object.

$$l_i^{F+1} = \begin{cases} l_i^A \text{ if } l_i^A \neq l_i^{F_i}, \\ -1 \text{ else.} \end{cases} \tag{105}$$

This conditional assignment is formulated using an if-then-else structure as presented in Section 5.2.

We introduce a binary variable $m_{i,j}^p$ which is forced to 1, in case the jth part of data object i is mapped to the cache line p:

$$\forall p = 0, \ldots, L-1 : \forall O_i : \forall j = 0, \ldots, F_i + 1 :$$
$$m_{i,j}^p = \begin{cases} 1 \text{ if } (l_i^j \equiv p), \\ 0 \text{ else.} \end{cases} \tag{106}$$

Subsequently, the number of accesses per cache line h_p are defined.

$$\forall p = 0, \ldots, L-1 : h_p = \sum_{i=0}^{|O|-1} \sum_{j=0}^{F_i+1} E_i \cdot m_{i,j}^p \tag{107}$$

E_i is the total number of accesses during the program runtime to the data object O_i. This could be, e.g., the worst-case or average number of accesses.

Additionally, the data object addresses have to be restricted such that the objects are not overlapping. Two objects O_i and O_j are overlapping if

$$(a_i \leq a_j \leq a_i + S_i - 1) \vee (a_j \leq a_i \leq a_j + S_j - 1) \tag{108}$$

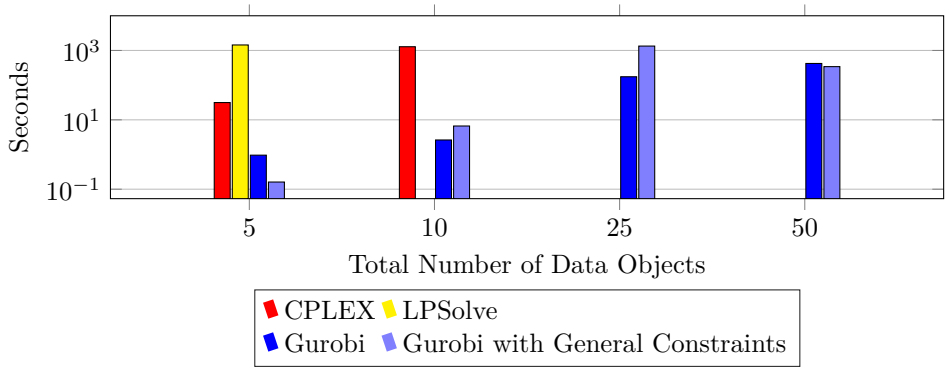

Figure 6 Solving times for the case study for different total number of data objects.

holds. For each side of the $\vee$ operator, a binary variable is created, representing whether the depicted range restriction holds. Both are then combined using the logical OR operator, formulated as shown in Section 4. The binary variable, indicating if O_i overlaps O_j, is named $g_{i,j}$. To enforce no overlapping objects, the following constraint is inserted.

$$\bigvee_{i=0}^{|O|-1} \bigvee_{j=i+1}^{|O|-1} g_{i,j} = 0 \tag{109}$$

Finally, the objective term is inserted.

$$\min : \mathsf{max}(h_0, h_1, ..., h_{L-1}) \tag{110}$$

Evaluation

To evaluate the case study, we created the following setup: We created ILPs with sample sizes of $S = \{5, 10, 25, 50\}$, S representing the total number of data objects. For each S, we generated S uniformly distributed random integers in the range of $[0, 1024]$ which represent the number of accesses per object. Additionally, S uniformly distributed random integers in the range of $[8, 256]$ were generated, denoting the size per object in byte. The number of cache lines was set to 4 with a cache line size of 128 B. The total size of the data section was set to 8192 B.

Fig. 6 shows the results of the evaluation. For a total of 5 data objects, lpSolve is able to find a solution in 2 out of the 10 experiments given the 2 h time limit. Besides, CPLEX and Gurobi are able to find a solution for 5 data objects over all experiments in the given time limit. Considering the experiments where lpSolve finds a solution in time, the average solving time is 46 times larger compared to CPLEX. The average time CPLEX required to find a solution is 32 times higher than compared to Gurobi without general constraints. Yet, Gurobi with general constraints yields the best results in this case with an average running time 6 times lower compared to Gurobi with general constraints deactivated.

For a total of 10 data objects, the average solving times drastically increase. For none of the 10 experiments lpSolve is able to find a solution within the 2 h time limit. CPLEX requires 1280 s in average to find a solution. Gurobi without general constraints reaches the best performance in average for this configuration with an average solving time of 2.62 s. With general constraints activated, Gurobi shows a slight performance degradation with an average solving time of 6.64 s.

For 25 and 50 data objects in total, neither lpSolve, nor CPLEX are able to find a solution within the 2 h time limit. For 25 data objects, Gurobi performs significantly better without general constraints (174.66 s compared to 1338.17 s average solving times). At a total of 50 data objects, Gurobi with general constraints performs slightly better than without.

7 Conclusions

We provided a guide on how to model logical, conditional as well as some arithmetic operations in ILPs. We provide safe bounds on any auxiliary variables needed. For each operation, we compared the performance of the popular solvers lpSolve, IBM CPLEX and Gurobi.

We finally show-cased the usability on a cache optimization which can be used by compilers for embedded systems. Our evaluation results show huge differences between the performance of the different solvers. While lpSolve's poor performance was to be expected due to its lack of multi-threading support, it is noteworthy that we could not produce one single test case in which CPLEX finished faster than Gurobi. Instead, for the more complex formulations and the practical use-case, Gurobi even managed to outperform CPLEX by orders of magnitude.

We therefore showed the applicability of ILPs for complex combinatorial problems in the domain of embedded system design, as long as a highly optimized solver like Gurobi or (for smaller problems) IBM CPLEX is being used.

While definitely not being the first using ILPs to model such problems, we hope that this work can provide both support for future projects, as well as a guide on which solver should be used depending on the actual problem.

References

1 Johannes Bisschop. *AIMMS. Optimization Modeling*. Paragon Decision Technology, Haarlem, Netherlands, 3rd edition, 2009.

2 George B. Dantzig. *Linear Programming and Extensions*. Princeton Landmarks in Mathematics. Princeton University Press, Princeton / USA, 11th edition edition, 1998.

3 George B. Dantzig, Alexander Orden, and Philip Wolfe. The Generalized Simplex Method for Minimizing a Linear Form Under Linear Inequality Restraints. *Pacific Journal of Mathematics*, 5(2):183–195, October 1955.

4 Heiko Falk and Jan C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the 46th Design Automation Conference*, pages 732–737, July 2009.

5 Heiko Falk and Paul Lokuciejewski. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems*, 46(2):251–298, 2010.

6 Gurobi Optimization, Inc. Gurobi Optimizer, 2018. URL: `https://www.gurobi.com`.

7 IBM Corporation. IBM ILOG CPLEX Optimization Studio, 2018. URL: `https://www.ibm.com/products/ilog-cplex-optimization-studio`.

8 Y. Kunitake, T. Sato, and H. Yasuura. Signal probability control for relieving NBTI in SRAM cells. In *Proceedings of the 11th International Symposium on Quality Electronic Design (ISQED)*, pages 660–666, March 2010.

9 Yau-Tsun S. Li, , Sharad Malik, and Andrew Wolfe. Efficient Microarchitecture Modeling and Path Analysis for Real-Time Software. In *Proceedings of Real-Time Systems Symposium (RTSS)*, pages 298 –307, December 1995.

10 lp_solve. lpSolve, 2018. URL: `http://lpsolve.sourceforge.net/5.5/`.

11 Arno Luppold and Heiko Falk. Schedulability-Aware SPM Allocation for Preemptive Hard Real-Time Systems with Arbitrary Activation Patterns. In *Design, Automation and Test in Europe (DATE)*, pages 1074–1079, 2017.

12 Bernhard Meindl and Matthias Templ. Analysis of commercial and free and open source solvers for linear optimization problems. *Eurostat and Statistics Netherlands within the project ESSnet on common tools and harmonised methodology for SDC in the ESS*, page 20, 2012.

13 Dominic Oehlert, Arno Luppold, and Heiko Falk. Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems. In *Proceedings of the 29th Euromicro Conference on Real-Time Systems (ECRTS)*, June 2017.

14 Dominic Oehlert, Arno Luppold, and Heiko Falk. Mitigating Data Cache Aging through Compiler-Driven Memory Allocation. In *Proceedings of the 21st Workshop on Software and Compilers for Embedded Systems (SCOPES)*, May 2018.

15 Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, et al. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of Real-Time Systems Symposium*, pages 223–232, December 2005.

16 Lam Lay Yong. Zhang Qiujian Suanjing (The Mathematical Classic of Zhang Qiujian): An overview. *Archive for History of Exact Sciences*, 50(3):201–240, Sep 1997.