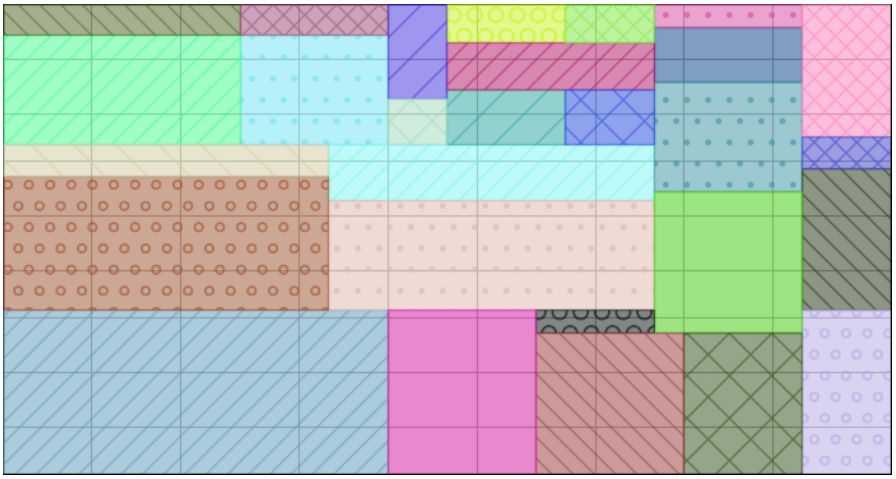
Very Large Scale Integration: solutions with CP, SAT and SMT

Pietro Fanti, Simone Montali

July 2021

Abstract

VLSI (*Very Large Scale Integration*) is the process of integrating **circuits** into silicon **chips**. Technological progress has made it possible to **decrease the transistors' size** and this has **pushed the integration** of more and more transistor into a single chip, allowing modern smartphones to include many advanced features in such small devices. In order to keep acceptable sizes, a proper choice of the **spatial arrangement** of the transistors must be done. This is the problem addressed in this paper. The proposed solution is based on the intuition of **dealing with a single dimension of the problem first**, as if it was a **task scheduling** constraint problem. Then, using the partial solution as an additional constraint, the other dimension will be approached, resulting in a smaller search-tree.



Contents

1 Introduction

The creation of **smaller, yet more powerful** chips has been a highly compelling task in the last years, and with all probability it will also be in the years to come: the **impact on society** it has is tremendous. The possibility of creating powerful calculators having portable sizes is what made electronic devices like smartphones so **widespread** today.

Technological progress played a **central role** in this process, making it possible to decrease transistors size, but the spatial arrangement of the circuits is important too, and technology can truly make a contribution, making the choice of the arrangement efficient and automatic.

In this project we propose a solution to this problem. As fixed input data we have w, the width of a silicon plate on which we have to place n circuits, each one with a certain horizontal size and a certain vertical size. The goal is to find an arrangement that **minimizes the resulting height** h of the plate.

The problem was solved with 3 different approaches: **Constraint Programming** (CP), **Boolean SATisfiability** (SAT) and **Satisfiability Modulo Theories** (SMT). See, respectively sections 3, 4 and 5.

In all the 3 methods, which are available on GitHub [1], our proceeding way revolves around the key concept that it is possible to **decompose the problem** in 2 sub-problems. In the first one the goal is to find the **lowest possible height** and the **vertical coordinates** to reach that. This can be treated as a **cumulative scheduling** problem. In the second and last sub-problem we use the data obtained in the first one to obtain a set of **horizontal coordinates** that **satisfies** the instance.

2 General concepts

2.1 Decomposition and transformation into a cumulative scheduling problem

At first the problem was tackled as a CP (Constraint Programming) problem, so a first approach was to look for a suitable **global constraint** among those available on MiniZinc. Of course packing constraints fit well, and an attempt with the `diffn` predicate has been done, but results were not so satisfactory.

The key concept of this paper is that the VLSI problem can be properly transformed into a **task scheduling** problem, in order to exploit the **cumulative** constraint.

If you take a look to figure 1 you can clearly see that with just a rotation the problem can be treated as it was a task scheduling problem. w is now the **resource capacity**, the objective function h is the **time** that must be minimized and circuits are now tasks: their horizontal size is the time they take to be executed and their horizontal size is the amount of resources they need.

What we look for is the start time of each task, which corresponds to the vertical coordinate of each circuit.

Therefore, from now on, the terms "task" and "circuit" will be used interchangeably depending on which fits better with the current context. The same for "duration" and "vertical size" and so on, according to the analogy just illustrated which is summed up in the table 1.

This partial (because we miss horizontal coordinates) solution allows us to **eliminate many symmetries**, indeed we are interested only in one of the two dimensions of the search space, so all the different solutions which **share vertical coordinates** correspond to only **one solution**. Moreover, we get the **minimum height possible**, so when in the next step we look for vertical coordinates the problem is just a **satisfaction problem**, which is computationally faster than an optimization problem.

2.2 Variables' ranges

The choice of variables' ranges has an **important role** in the optimization of the problems, since a **shorter range** means a search tree with **less branches**.

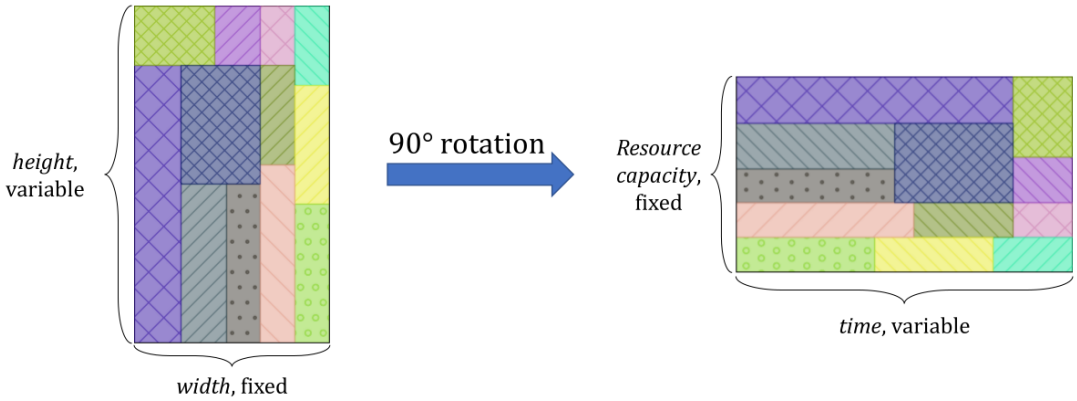


Figure 1: Rotating the plane 90 degrees clockwise we can see the problem as a tasks scheduling problem

VLSI	Cumulative Scheduling
width w	resource r
height h	make-span t
number of circuits n	number of tasks n
vertical coordinate y_i	start time s_i
horizontal coordinate x_i	n/A
circuit's width w_i	task's resource need r_i
circuit's height h_i	task's duration d_i

Table 1: Analogies between original problem and cumulative scheduling problem

For what concerns the start times (i.e. the vertical coordinates) they can span in the interval which goes from 0 to s_{max} where s_{max} is:

$$s_{max} = \sum_{i=1}^{n} d_i - \min_{i=1...n} d_i \tag{1}$$

with d_i being the duration (i.e. the vertical size) of the i-th circuit. Indeed $\sum_{i=1}^{n} d_i$ is the time-span we obtain if we execute just **one task at a time**, consecutively. But a task can have a starting time which, at most, is the above time-span minus its duration. Therefore, to obtain s_{max} the minimum duration possible must be subtracted since which task will be the last one is unknown.

For the same reason the objective function h has an upper limit of

$$h_{max} = \sum_{i=1}^{n} d_i. \tag{2}$$

But a **more substantial optimization** comes from the choice of the **lower bound**.

When circuits can be arranged without leaving blank spaces, which is not always possible but depends on the instance, **it is not possible to find a better configuration**. In this case h will be equal to the **sum of the areas** of all the circuits **divided** (integer division) by w. In a general case the total area occupied by circuits is **not divisible** by w, so in that case at least one blank cell will be present, meaning that the lower bound of h will be at least **increased** by 1.

Considering all this, the lower bound of h can be summarized as:

$$h_{min} = \begin{cases} \sum_{i=1}^{n} x_i y_i \div w & \text{if } \sum_{i=1}^{n} x_i y_i \mod w = 0 \\ (\sum_{i=1}^{n} x_i y_i \div w) + 1 & \text{otherwise} \end{cases} \tag{3}$$

Therefore, h belongs to the range:

$$h \in [h_{min}, h_{max}] \tag{4}$$

2.3 A more general case: circuits rotation allowed

Since it has now been assumed that circuits must keep a **fixed orientation** between each other to work, they can not be rotated. But what happens when this constraint is broken, **allowing circuits to be rotated** to reach a better spatial arrangement, given that it exists?

Since all the circuits are rectangular, they have only two possible rotations: **standard** and **rotated by 90 degrees**. The rotation of a circuit can be achieved by simply **swapping** its vertical dimension h_i with its horizontal dimension w_i, that in the cumulative scheduling analogy corresponds to swap the resource demand r_i of a task with its time demand d_i.

To check whether each circuit is rotated or not we just need a **vector** of **Boolean** values $b_1, b_2, ...b_n$. When b_i is **true** we **swap** w_i with h_i (d_i with r_i).

While the general ideas proposed in section 2.1 also hold in this case, ranges showed in section 2.2 must be reviewed.

h now spans the range $[h_{min}, h_{max_{rot}}]$ where h_{min} is not changed while $h_{max_{rot}}$ has become

$$h_{max_{rot}} = \sum_{i=1}^{n} \max\{d_i, r_i\} \tag{5}$$

which is a value grater or equal than $h_{max} = \sum_{i=1}^{n} d_i$.

To understand this change it is helpful to imagine an instance like the following one:

4
3
5 4
6 3
7 4

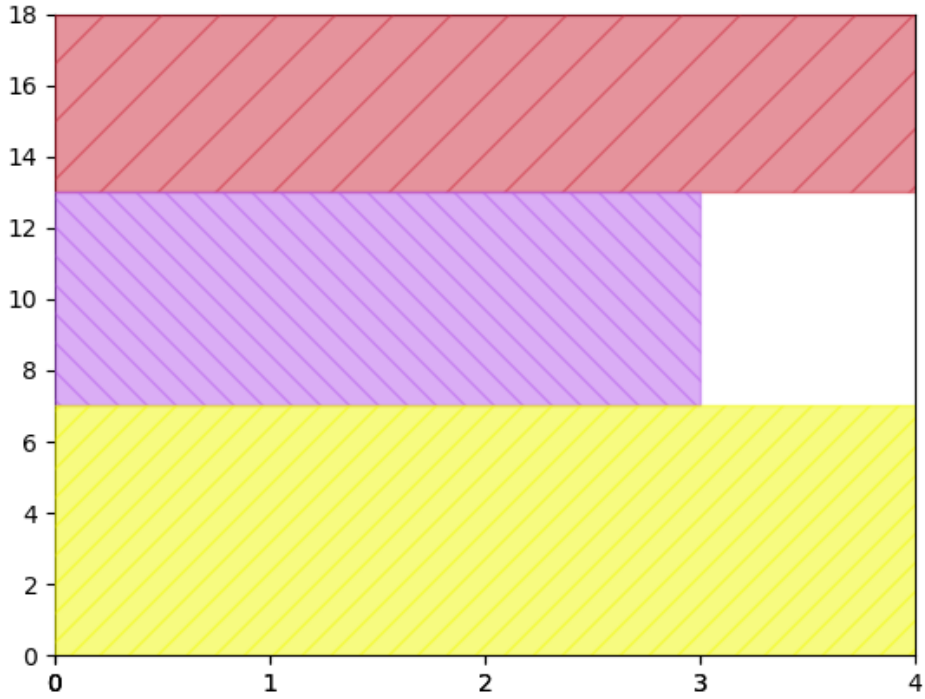


Figure 2: This instance can not be solved without rotating circuits and its makespan reaches a value higher than `sum(duration)`, which is the upper bound when rotations are not allowed. This shows why that bound must be reviewed when rotations are allowed.

This instance cannot be solved without rotation, since none of the 3 circuits can fit in a plate of width equal to 3 without being rotated. While when rotations are allowed the provided solution is the one in figure 2, and the makespan takes the value 18, which is higher than

$$\sum_{i=1}^{3} d_i = 4 + 3 + 4 = 11 \tag{6}$$

The worst case is when, as in the above shown example, each circuit can enter in the plate **only when its largest dimension is parallel** to the plate height, and in that case the makespan takes the value $h_{max_{rot}}$ as defined in formula 5.

Also, possible start times now belong to the range $[0, s_{max_{rot}}]$ where

$$s_{max_{rot}} = h_{max_{rot}} - \min_{i=1\ldots n} \min\{d_i, r_i\} \tag{7}$$

If we compare it with s_{max} as defined in formula 1 we can see that $\sum_{i=1}^{n} d_i$, which corresponds to h_{max}, has been substituted with $h_{max_{rot}}$ since now this is the max value that h can take. Then, instead of subtracting $\min_{i=1\ldots n} d_i$ we subtract $\min_{i=1\ldots n} \min\{d_i, r_i\}$ to reflect the fact that the orientation of the circuit with the biggest vertical coordinate is unknown.

2.4 Particular cases

Originally the project had a different structure. Exploiting the idea of separating the search for vertical and horizontal coordinates, these two tasks were done in two different and completely separated steps, resulting in better performances than the one presented here in most of the

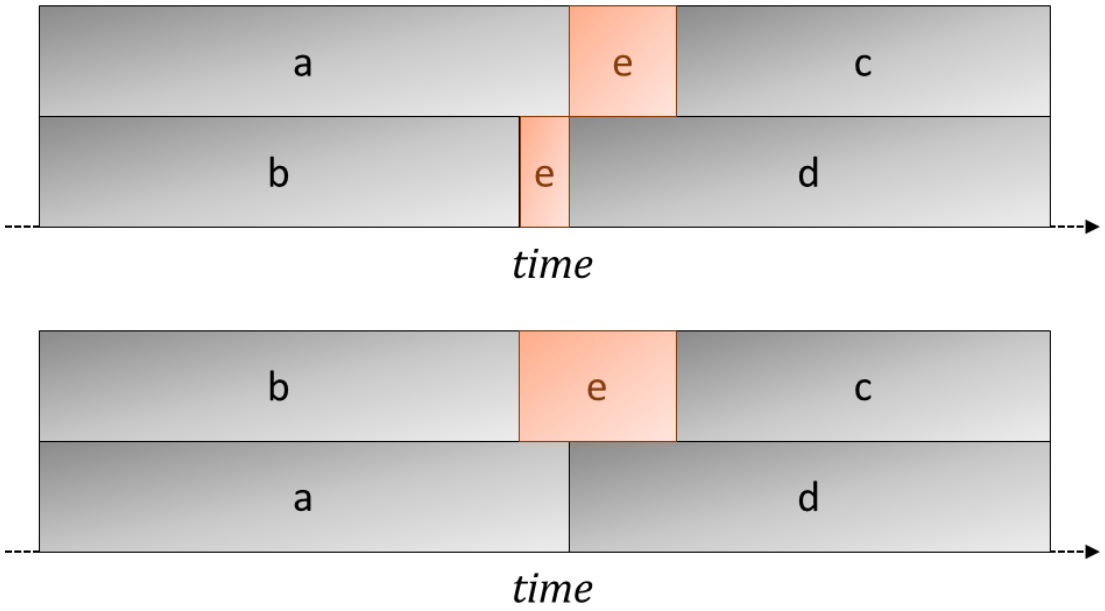


Figure 3: The task e does not need to be *spatially united* to have enough resources to be executed. In the most of cases a permutation of the tasks allows to restore spatial unity preserving all the start times as in this figure, however in some rare cases this could not be possible.

cases (especially in the Constraint Programming implementation). However, while the original model worked great most of the times, during final phases of the testing process **some problems arose** with particular instances: 19 or 29, depending on the implementation. Indeed there are particular solutions of the cumulative constraint which however **do not lead to a possible permutation** of horizontal coordinates. Since tasks are time concepts and not spatial ones, cumulative does not constraint them to be *spatially* united. Even if in most cases spatial unity can be restored as in figure 3, in some rare cases this could not be possible, leading to an unsatisfiable instance.

Several methods are possible to solve this problem. At first, it has been decided to keep the general structure and in case of an unsatisfiable choice of vertical coordinates, to **restart the model** constraining it to find a **different partial solution**. However it was a quite *naive* and inefficient solution, leading to a **huge overhead** in case the model entered in a unsatisfiable partial solution, and not leading to any partial solution when the model reached the time cap. Therefore, more elegant and solid solutions were found. For CP, in particular, the two different models are now **united in a single one**, and thanks to **search annotations** the model searches for vertical coordinates first, and only then for horizontal ones, but the two tasks are not totally separated anymore, leading to a solution in every possible satisfiable case.

3 CP

Once the problem has been transformed as explained in section 2.1 the implementation [1, `CP`] is actually quite easy. After the input file has been suitably **preprocessed** in Python, a MiniZinc model, called `vlsi.mzn`, solves the time scheduling problem with input `n`, `duration`, `w` and `req`. Ranges are the ones illustrated in section 2.2. The model uses **search annotations** to implement a behaviour which is similar to the one described in section 2: at first it searches for vertical coordinates that minimize the total height, and only after that, it looks for a set of horizontal coordinates that satisfies the problem:

```
solve :: seq_search([
```

```
            int_search(start, smallest, indomain_min),
            int_search([makespan], input_order, indomain_min),
            int_search(x, input_order, indomain_min)])
minimize makespan;
```

To implement the cumulative task scheduling constraint we have of course used the **global** `cumulative` constraint from `globals.mzn`, while for vertical coordinates we simply constrained circuits to n**ot overlap** and to **stay inside the plate**:

```
constraint cumulative(start, duration, req, w);

constraint forall(i in CIRCUITS, j in CIRCUITS where i != j)(
    (x[i]+req[i] <= x[j]) \/
    (x[j]+req[j] <= x[i]) \/
    (start[i] >= start[j]+duration[j]) \/
    (start[j] >= start[i]+duration[i]));

constraint forall(i in CIRCUITS) (x[i] + req[i] <= w);
```

Output is then post-processed in order to fit with the **project requirements** and then it is used to also show a graphic representation, which is done with MatPlotLib.

3.1 Circuits rotation

To implement the possibility of rotations in MiniZinc we modified the model `vlsi.mzn` to create a new model called `vlsi-rot.mzn` where we added an **array of Boolean values** deciding whether each circuit is rotated or not and we linked it to the constraints and to the value of the make-span:

```
constraint cumulative(
    start,
    [if (rotation[c]) then req[c] else duration[c] endif |
        c in CIRCUITS],
    [if (rotation[c]) then duration[c] else req[c] endif |
        c in CIRCUITS],
    w);

constraint forall(i in CIRCUITS, j in CIRCUITS where i != j)(
    (x[i] + if (rotation[i]) then duration[i]
        else req[i]
        endif <= x[j]) \/
    (x[j] + if (rotation[j]) then duration[j]
        else req[j]
        endif <= x[i]) \/
    (start[i] >= start[j] + if (rotation[j]) then req[j]
        else duration[j]
        endif) \/
    (start[j] >= start[i] + if (rotation[i]) then req[i]
    else duration[i]
    endif));

constraint forall(i in CIRCUITS)(
    x[i] + if (rotation[i]) then duration[i] else req[i] endif <= w
);

var lower_bound..sum(c in CIRCUITS)(max(duration[c], req[c])):
makespan = max(c in CIRCUITS)(
    start[c] +
```

```
if (rotation[c]) then
    req[c]
else
    duration[c]
endif);
```

From the previous snippet of code you can also see that ranges change according to what is discussed in section 2.3.

3.2 The global `cumulative` constraint

The cumulative constraint is decomposed in the MiniZinc library as depicted in the *Why Cumulative Decomposition Is Not as Bad as It Sounds* [2] paper by Schutt et al., using concepts from Lazy Clause Generation, an approach which mixes up Finite Domain propagation and SAT theory by generating Boolean clauses *lazily* from the finite constraints. In particular, two decompositions are proposed in the paper (which will be further explored in section 5): the **Time-RD** decomposition, containing at most nt_{max} Boolean variables and constraints, and the *Task-RD* decomposition, which relaxes the constraints and reduces the complexity of the problem. Both the decompositions [3] are available in the library, with the first being chosen for small time-horizons, while the latter is chosen for larger time-horizons. Lazy Clause Generation provides a major speed up in the performances of the solver, as seen in section 3.3.

3.3 Performances

MiniZinc comes with the possibility of using different solvers to find a solution of the model, and also other options to customize the solving process.

Between the solvers natively available only two are based on Constraint Programming: Gecode, which is the default solver, and Chuffed [4, sec. 3.4]. Both of them can be executed with 6 different levels of optimization:

- **O0**: No optimization;
- **O1**: Single pass optimization (default);
- **O2**: Double pass optimization;
- **O3**: Double pass optimization with Gecode;
- **O4**: Double pass optimization with Gecode and shaving;
- **O5**: Double pass optimization with Gecode and singleton arc consistency.

Also, Gecode has the possibility of being executed in parallel on multiple processes, from 1 (default) to 99.

According to [4, sec. 3.4.1.2] and to our empirical results, Chuffed is often much faster than Gecode if we are able to take the full-advantage of Chuffed performances. In order to do this we need search annotations and to allow Chuffed to switch between this defined search and activity-based search heuristics. This switching behaviour is activated in MiniZinc with the *Free Search* option.

Since our model already takes advantage of search annotations, it's clear that Chuffed, when started in *Free Search* mode is the best solver for our model. From our attempts also it also emerged that changing the optimization level has not a big impact on performances, but the default O1 level is generally a bit faster.

Results on all the instances for Chuffed with optimization O1, in both standard and rotation cases, are shown in figure 4. Execution times shown are obtained as the mean of the times of 5 different executions and they are capped at 300 seconds (5 minutes).

4 SAT

4.1 SAT solving

While Constraint Programming provided really good results in the test runs, another possibility is definitely interesting to explore: **SAT solving**. *SAT* stands for *Satisfiability* in *Boolean Satisfiability*, namely one of the most discussed logic and computer science problems ever. With satisfiability, we're asking whether a valid interpretation can be found for a given Conjunctive Normal Form boolean formula. This was the first problem to ever be proved NP-complete [5]. Since, as already mentioned, no polynomial-time solution has been found yet (and might never be), heuristics algorithms have been proposed, being able to solve several-thousand variables problems in acceptable times. As by definition, any NP-complete problem can be reduced to SAT, these heuristic algorithms provide an exceptional tool for the solution of **complex problems** in computer science. Very Large Scale Integration is among these.

4.2 Encoding the problem in SAT

The encoding of a problem in Conjunctive Normal Form is definitely the **most complex step** of the process, now that SAT solvers are widely available and straightforward to use. Furthermore, we're dealing with an **optimization** problem, not a satisfaction one, so we will have to enforce a last condition to each solution: it must be a **better solution then the preceding one**. This is also known as *Branch and Bound approach*, and it iterates until the problem is infeasible. We could therefore start from a heuristically-decided height for the chip, and if it is satisfiable reduce it, while if it is not, enhance it. We will still want to keep the decomposition we've used in the Constraint Programming solution, namely the `cumulative` constraint to find the **optimal** Y coordinates of the circuits, then a separate solver that simply **satisfies** the constraints.

4.2.1 Encoding the cumulative constraint

The most intuitive encoding for the problem is the following: we introduce nt_{max} Boolean variables B_t^i that tell whether a given task i is **active** at time t. Then, what we do is imposing the first resource constraint: at each timestep, we want the active tasks **not to exceed the resource constraint**, meaning that the sum of the active tasks' requirements cannot be higher than the resource maximum.

$$\bigwedge_{i=0}^{i \leq t_{max}} \left(\sum_{t=0}^{t \leq n} r_t \leq res_{max} \right)$$

Encoding this in SAT is easy: at first we find the possible subsets of tasks that **respect the resource constraint**, then we make a disjunction (OR) between all the elements of each subset. We need to **negate** the non-mentioned tasks too. Basically, what happens **for each timestep** t is the following:

1. Find the subsets given the resource maximum availability (namely, the width of the circuit), including the empty subset meaning that no task is active;
2. For each subset, create an AND condition:

$$\bigwedge_{i=0}^{i \leq n} \begin{cases} B_t^i \text{ if the task is in the subset} \\ \neg B_t^i \text{ if the task is not in the subset} \end{cases}$$

3. For all the subsets, create an OR condition:

$$\bigvee_{i=0}^{n_{subsets}} \bigwedge_{i=0}^{i \leq n} \begin{cases} B_t^i \text{ if the task is in the subset} \\ \neg B_t^i \text{ if the task is not in the subset} \end{cases}$$

The next step to impose that **durations are respected**: if a task i starts at time t, it will have to stay active until time $t + duration_i$. To do so, we will need to generate all the **sequences of**

timesteps that can contain the duration of task i in the time from 0 to t_{max}. For example, if we had $t_{max} = 5$ and a task with duration 3, the sequences would be the following:

$$[0,1,2] \qquad [1,2,3] \qquad [2,3,4] \qquad [3,4,5]$$

So, for every task i we do the following:

1. We find the timestep sequences given t_{max} and the duration d_i
2. For each time sequence, we impose the AND condition to make the task active during the sequence and false during the other timesteps:

$$\bigwedge_{t=0}^{t \leq t_{max}} \begin{cases} B_t^i \text{ if the timestep is in the sequence} \\ \neg B_t^i \text{ if the timestep is not in the sequence} \end{cases}$$

3. Introduce the OR to decide among the time sequences:

$$\bigvee_{s=0}^{s=n_s} \bigwedge_{t=0}^{t \leq t_{max}} \begin{cases} B_t^i \text{ if the timestep is in the sequence} \\ \neg B_t^i \text{ if the timestep is not in the sequence} \end{cases}$$

We can now get a solution for the `cumulative` and obtain our Y coordinates.

4.2.2 Finding the X coordinates

Now that we have found our Y coordinates, we want to find the X of each circuit. To do so, we will want to proceed as we did for the durations (heights): we find the possible sequences of widths that can stay within the maximum width, then OR the ANDS of the widths. In other words, we are introducing $w \cdot h \cdot c$ Boolean variables (with w being the chip width, h being the chip height, c the number of chips) $B_{x,y}^i$ stating that chip i is found at the position x, y. For each task i, we:

1. Find all the width subsets contained in max_{width}
2. For each Y that was labeled active by the preceding optimization, and for each X of the subset we're taking into account:

$$\bigwedge_{x=0}^{x \leq max_{width}} \begin{cases} B_{x,y}^i \text{ if the } x \text{ is in the subset and the } Y \text{ is active} \\ \neg B_{x,y}^i \text{ if the } x \text{ is not in the subset or the } Y \text{ is not active} \end{cases}$$

3. We OR the preceding conditions to decide a configuration for each width subset

A final condition can be added: if a circuit is found at position x, y, all the other tasks are negated:

$$B_{x,y}^j \longrightarrow \bigwedge_{i=0}^{i \leq n, i \neq j} \neg B_{x,y}^i$$

If we try to satisfy this model, we will obtain the list of active x, y tuples for each task. We finally find the minimum among the X and Y coordinates to find the actual position of the lower left corner of a circuit. Then, the postprocessing we have seen in the MiniZinc part only needs a few tweaks to work with this encoding, finally resulting in a graph of the chip.

4.3 Circuits Rotation

To take rotations into account, we will have to change **the way we find subsets** and **declare Boolean variables**: we introduce a **new dimension** in our variables stating **whether a circuit is rotated**, and impose that if a circuit is rotated for a single x, y, it is **rotated everywhere**. The Boolean variables in the first cumulative optimization become B_{tr}^i for rotated circuits, and B_t^i for non-rotated circuits. Basically, when imposing the first constraint regarding the resources,

we add rotated circuits as if they were new ones, but then, if B^i_{tr} is present in a clause, we add $\neg B^i_t$, the negation of its non-rotated sibling. The same reasoning is inversely applied when we have a non-rotated circuit in the clause and want to add the negation of its rotated sibling. If the circuit is absent from the subset we're working with, we'll want to **negate both** the rotated and non-rotated versions of the circuit. The same procedures are done for the second constraint. Having found a solution for the optimization, we simply **swap the widths and heights** of the rotated circuits and keep the same implementation to find the abscissa.

4.4 Performances

As seen in graph 5, the SAT encoding of the problem **did not perform as well** as Constraint Programming and SMT: all the test instances past `ins-10.txt` reach the 5 minute timeout without being able to find an optimal solution. The reason behind this probably lies in the fact that the models that were chosen for the other two solutions are **highly optimized**, especially the CP one, based on **Lazy Clause Generation**. This does not make SAT solving bad, or outdated. It just shows that **different alternatives and models are always interesting to explor**e for every task, as not to make general assumptions about some type of solver. In fact, SAT solvers are **much more performing** indeed, but the **lack of an easy modeling structure** makes them much more subject to the **optimization of the model**, which would require several papers alone. What is interesting is that, as in most sciences, the best solutions lie in the **unification of different approaches**. This is exactly what is done by SMT and Lazy Clause Generation, and it probably is what we'll **mostly see in the future**.

5 SMT

As it is pretty clear by now, SAT solvers are great tools in terms of performances, explainability and availability on the market. The great successes that have been reached thanks to SAT solvers are **motivation for its frequent use** in diverse domains, but still, as mentioned before, the **encoding** remains the most time-consuming and complex part. It would therefore be nice to be able to express problems with a richer language: **logics** rather than *propositional logic*. Obviously, this takes into account some **loss of efficiency**, but the expressivity of the model encoding might actually make it easier to improve solutions, and consequently, outperforming SAT. **Satisfiability Modulo Theories** solvers are the answer we are looking for: they allow the programmer to define a problem in what is, roughly, first order logic, then autonomously solve the instance. The eager approaches consisted in a **translation to an equivalent SAT instance**, while nowadays SMT solvers are much more complex tools that tightly integrate **DPLL-style boolean reasoning** with **theory-specific** solvers. SMT solvers allow us to define models that are similar to the ones seen in Lazy Clause Generation solving: in SMT, we can mix boolean variables and finite domain constraints as it's done in LCG. As done before, we'll encode the *cumulative* constraint, then simply satisfy the model to find the missing coordinate.

5.1 Encoding `cumulative` in SMT

The literature for `cumulative` is way more dense than the one someone can find for VLSI. A breakthrough paper in the topic is [3], which explained how Lazy Clause Generation can be a great solution for this task. Even though we would need a Lazy Clause Generation solver like **Chuffed** to really enjoy the performance increase in these models, we still can implement them in an SMT instance **obtaining good results**. The paper proposes two different encodings, differing in the number of Boolean variables and constraints:

1. **Time-RD**, which for every time t constrains the sum of all resource requirements to be less or equal to the resource capacity:

$$\sum_{i \in [1..n]} r[i] \cdot B_{it} \leq c$$

where the B_{it} states whether task i is active at time t:

$$B_{it} \leftrightarrow s[i] \leq t \wedge \neg s[i] \leq t - d[i]$$

This model needs at most nt_{max} Boolean variables, nt_{max} conjunction constraints and nt_{max} sum constraints.

2. **Task-RD**, which is a relaxation of the first model: it ensures a non-overload of resources only at the start times. This is sufficient to ensure non-overload at every time, meaning that the number of variables and constraints becomes independent from t_{max}. The decomposition implicitly introduces at most $3n(n-1)$ Boolean variables: B^1_{ij} (meaning task j starts when task i is already started or in the same moment), B^2_{ij} (meaning task j starts before task i ends) and B_{ij} (meaning task j starts when i is running). The constraints for $\forall j \in [1..n], \forall i \in [1..n] \backslash \{j\}$ then becomes the following:

$$\begin{aligned} B_{ij} &\leftrightarrow B^1_{ij} \wedge B^2_{ij} \\ B^1_{ij} &\leftrightarrow s[i] \leq s[j] \\ B^2_{ij} &\leftrightarrow s[j] < s[i] + d[i] \end{aligned}$$

and then,

$$\forall j \in [1..n] : \sum_{i \in [1..n] \backslash \{j\}} r[i] \cdot B_{ij} \leq c - r[j]$$

Finally, we can add some redundant constraints in order to improve the propagation and the learning, which encode the relationship among the B^1_{**} and B^2_{**} for all $i < j$:

$$B^1_{ij} \vee B^2_{ij} \quad B^1_{ji} \vee B^2_{ji} \quad B^1_{ij} \vee B^1_{ji} \quad B^1_{ij} \rightarrow B^2_{ji} \quad B^1_{ji} \rightarrow B^2_{ij}$$

Since we know that t_{max} increases way faster then n, the second decomposition is definitely the most promising. We can therefore proceed to encode it as an SMT model in Z3Py.

5.2 Encoding in SMT

To declare the model in Z3Py, we can start by declaring an `IntVector` to contain the starts of the *tasks*, which will represent our circuits' Y coordinates, and an `Int` makespan to be minimized. We then introduce a support `IntVector` to contain the ends, as $e_i = s_i + d_i$ and just declare the makespan to be the maximum of the ends. We then add some bounds to the makespan to make propagation faster, specifically the ones seen in 2.2. Then, we just need to optimize `makespan`, then create an `x_finder` constraint to find the X coordinates. This constraint will just be the non-overlapping of circuits, defined as:

$$\bigwedge_{i=0,j=0}^{i \leq c, j \leq c, i \neq j} (x_i + width_i \leq x_j) \vee (x_j + width_j \leq x_i) \vee (y_i \geq y_j + height_j) \vee (y_j \geq y_i + height_i)$$

Finally, we impose not to go over the width limit with

$$\bigwedge_{i=0}^{i \leq c} x_i + width_i \leq width_{chip}$$

and not to position circuits under the Y axis with

$$\bigwedge_{i=0}^{i \leq c} x_i \geq 0$$

5.3 Circuits Rotation

To take rotations into account, we can introduce a new `BoolVector` into the model that contains `True` if a circuit is rotated, `False` if not. Then, we can use Z3Py's `If` to check whether a circuit is rotated in the constraints. First, though, it is necessary to change the bounds and how the circuit ends are computed: the upper bound for the makespan is now

$$\sum_{i=0}^{i \leq c} max(width_i, height_i)$$

and the ends are computed as

$$end_i = start_i + \begin{cases} height_i \text{ if the circuit is not rotated} \\ width_i \text{ if the circuit is rotated} \end{cases}$$

The constraints change in the same way: by using an `If` we can alternate between summing $width_i$ and $height_i$ basing on the Boolean value of $rotation_i$.

5.4 Performances

As expected, SMT proved itself **better than SAT**, while still being **inferior to Constraint Programming** with Lazy Clause Generation. The first instances were solved even **faster than in CP**: this was probably caused by the **overhead** that the MiniZinc wrapper for Python introduces. Still, it performed pretty good, but what's more interesting is that **the implementation of SMT took way less time** than the other two. This is definitely a great feature for applications that require **fast development of solutions**, maybe sacrificing a bit of speed in exchange for extreme **easiness of the modeling** task. As seen in figure 6, SMT managed to solve most of the instances, only reaching **timeout** on the ones that had seen the most difficulty in CP too.

6 Conclusions

We have shown how it is possible to **transpose the circuit placement problem from a spatial problem to a temporal one** and solving it using the **cumulative** scheduling constraint, both in CP (section 3), SAT (section 4) and SMT (section 5). This, alongside a **smart choice of the variables ranges** (Section 2.2), allows to solve the problem with a relatively **simple implementation**, particularly in CP, since MiniZinc already provides an implementation for the cumulative constraint: `cumulative` from `globals.mzn`. This transposition works good in the general case too: separately searching a feasible set of horizontal and vertical coordinates **breaks symmetries**. Indeed, when searching for a set of horizontal coordinates we **do not mind about vertical ones**, meaning that all the possible solutions obtained by simply swapping ordinates correspond to a **single solution**.

This leads, despite some unexpected complications (Section 2.4), to **satisfactory performances** in CP, mostly when using the Chuffed solver, and SMT implementation, while in SAT solving it is quite slow. This is probably due to the lack of some typical SAT models optimizations, such as Lazy Clause Generation, in our implementation.

However, our results are probably outperformed by more standard and straightforward approaches, if properly optimized, but hopefully **not too distant**.

In conclusion we believe that even if our idea of facing up to the problem from a different point of view did not lead to the **best performance possible** it could obtain better results if **adequately mixed with constraints** from a more traditional modelling strategy.

As mentioned in 4.4, this project has proved that testing different approaches, solvers and optimization strategies is **always rewarding**: there is **no general best approach**, just lots of trial and error in every optimization task. Furthermore, it showed that NP-complete problems are **not-so-infeasible** nowadays, and that the development of better and faster SAT solvers will play **a central role** in the next century's society progress.

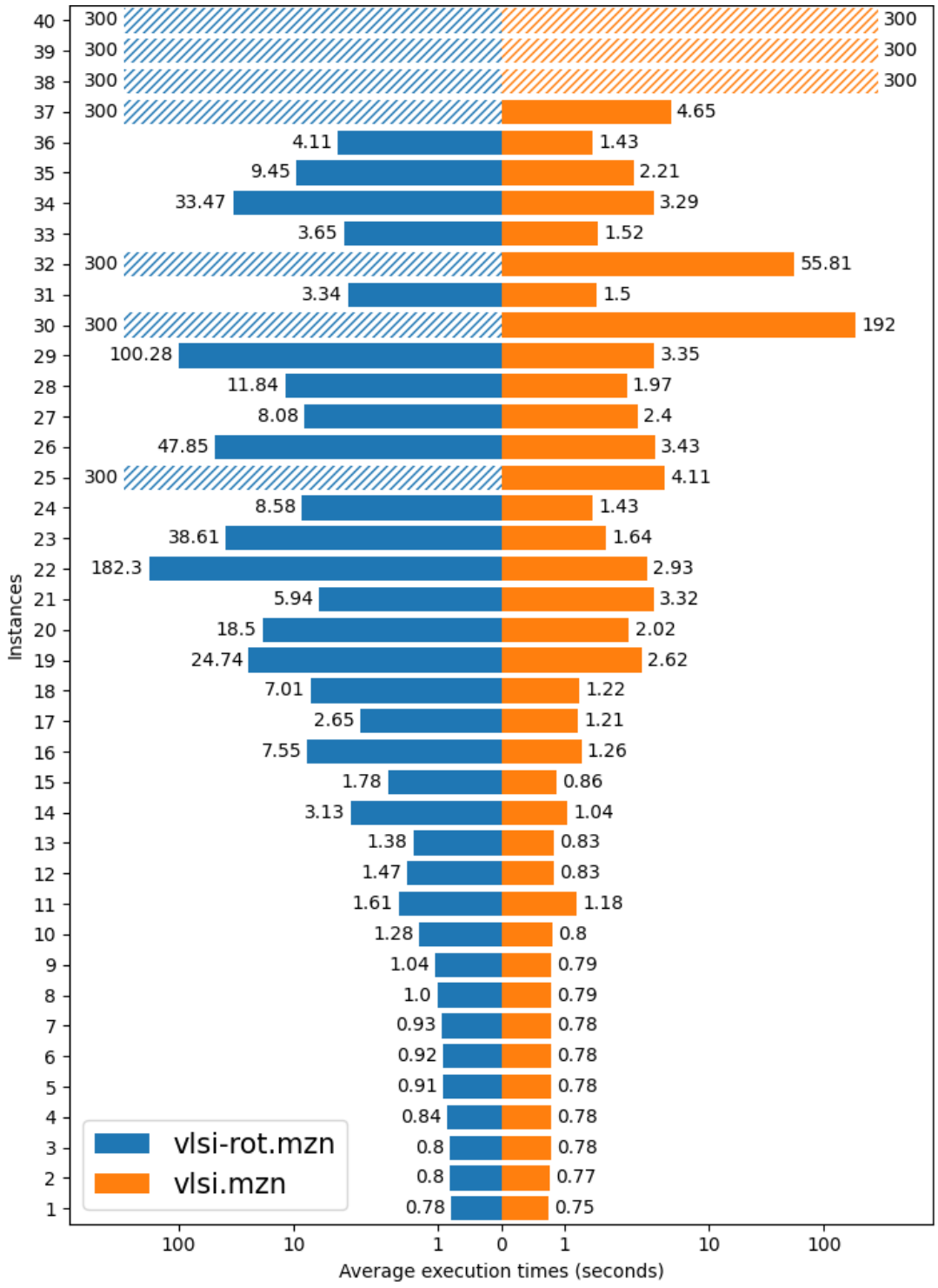


Figure 4: Average running times over 5 executions of Chuffed with optimization O1 in logarithmic scale. After 300 seconds the solver is stopped and a partial solution is returned.

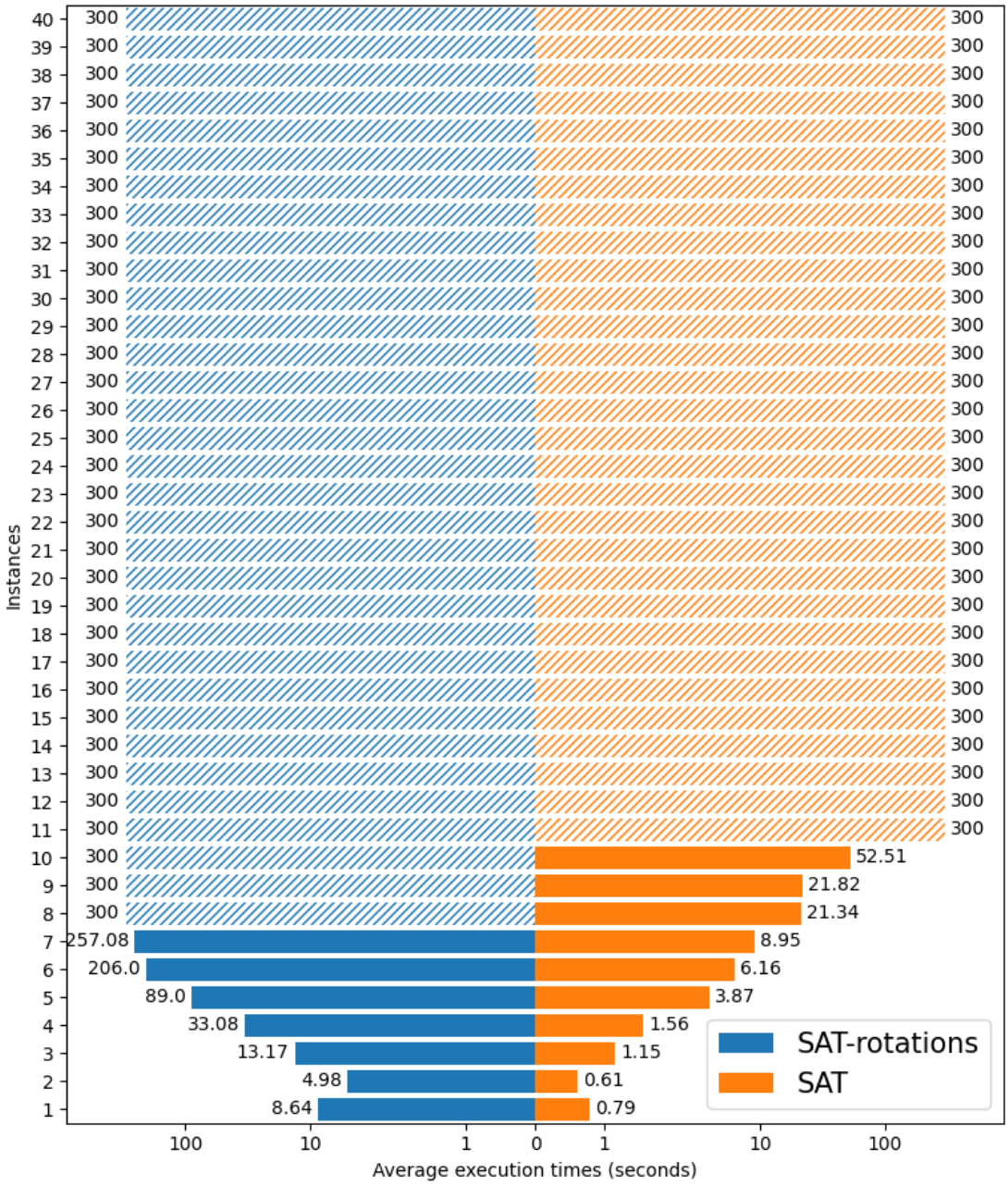


Figure 5: Average running times over 5 executions of the SAT solver in logarithmic scale. After 300 seconds the solver is stopped and a suboptimal solution is returned.

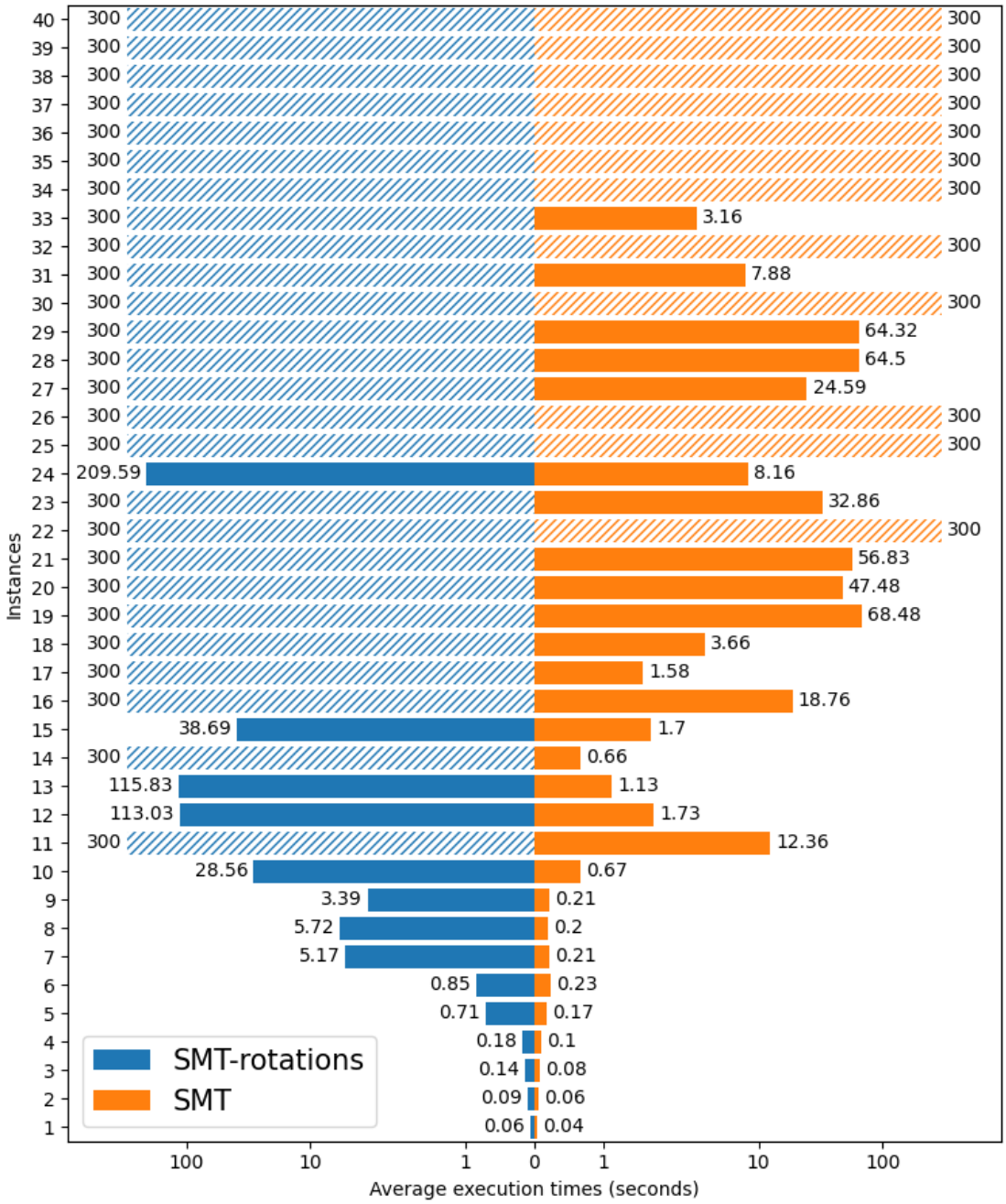
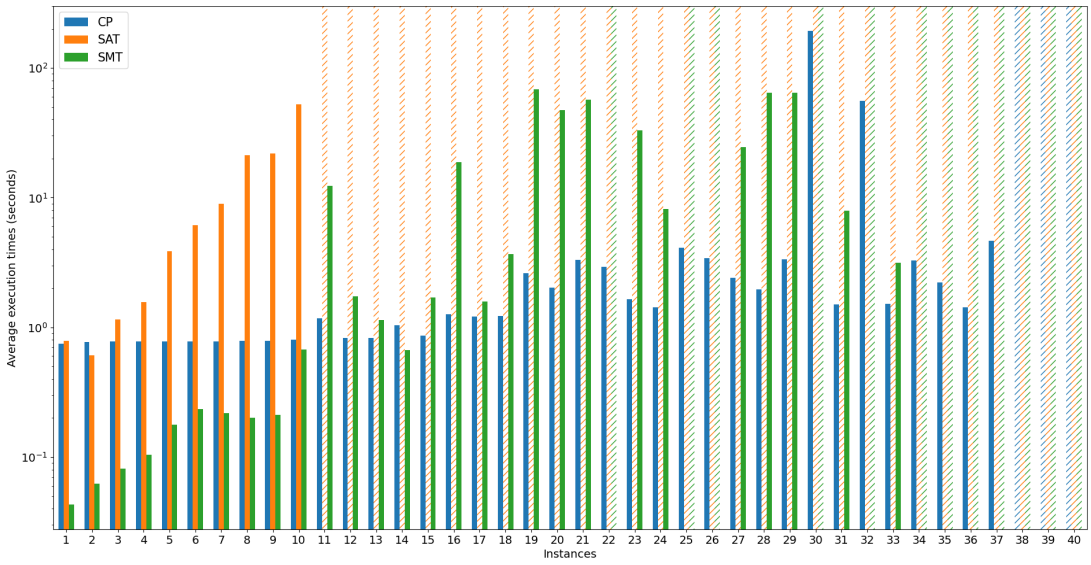
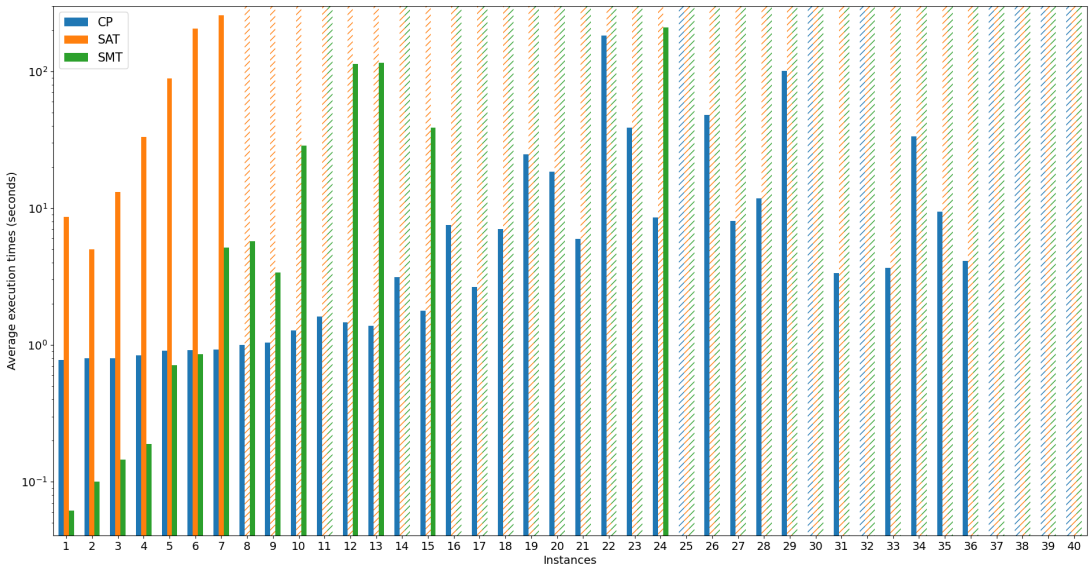


Figure 6: Average running times over 5 executions of the SMT solver in logarithmic scale. After 300 seconds the solver is stopped.



(a) *Standard case*



(b) *Rotation case*

Figure 7: Average running times over 5 executions of every implementation (CP, SAT and SMT) in logarithmic scale on every instance. After 300 seconds the solver is stopped.

References

[1] P. Fanti and S. Montali, "VLSI," 2021. [Online]. Available: https://github.com/TeamFanAli/VLSI

[2] A. Schutt, T. Feydy, P. Stuckey, and M. Wallace, "Why cumulative decomposition is not as bad as it sounds," 09 2009, pp. 746–761.

[3] MiniZinc, "fzn_cumulative_opt.mzn." [Online]. Available: https://github.com/MiniZinc/libminizinc/blob/master/share/minizinc/std/fzn_cumulative_opt.mzn

[4] G. T. Peter J. Stuckey, Kim Marriott, "The MiniZinc handbook," 2018. [Online]. Available: https://www.minizinc.org/doc-2.5.5/en/

[5] S. A. Cook, "The complexity of theorem-proving procedures." New York, NY, USA: Association for Computing Machinery, 1971. [Online]. Available: https://doi.org/10.1145/800157.805047