Parametric scheduler characterization

DOI:
10.1145/3358226

Document status and date:
Published: 01/10/2019

Document Version:
Accepted manuscript including changes made at the peer-review stage

Parametric Scheduler Characterization

JOOST VAN PINXTEN, Océ Technologies B.V. and Eindhoven University of Technology
MARC GEILEN, Eindhoven University of Technology
TWAN BASTEN, Eindhoven University of Technology and TNO ESI

Schedulers assign starting times to events in a system such that a set of constraints is met and system productivity is maximized. We characterize the scheduler behaviour for the case where decisions are made by comparing affine expressions of design parameters such as task workload, processing speed, robot travelling speed, or a controller's rise and settling time. Deterministic schedulers can be extended with symbolic execution, to keep track of the affine conditions on the parameters for which the scheduling decisions are made. We introduce a divide-and-conquer algorithm that uses this information to determine parameter regions for which the same sequence of decisions is taken given a particular scenario. The results provide designers insight in the impact of parameter changes on the performance of their system. The exploration can also be executed with the KLEE symbolic execution engine of the LLVM tool chain to extract the same results. We show that the divide-and-conquer approach provides the results much faster than the generic symbolic execution engine of KLEE. The results allow visualization of the sensitivity to all parameter combinations. The results of our approach therefore provide more insight in the sensitivity to parameters.

CCS Concepts: •**Theory of computation** →**Discrete optimization;** •**Computer systems organization** →**Embedded and cyber-physical systems; Embedded software;** •**Applied computing** →*Industry and manufacturing; Decision analysis;*

Additional Key Words and Phrases: Real Time Scheduling, System Design, Re-entrant Flexible Manufacturing System

ACM Reference format:
Joost van Pinxten, Marc Geilen, and Twan Basten. 2019. Parametric Scheduler Characterization. *ACM Trans. Embedd. Comput. Syst.* 1, 1, Article 1 (October 2019), 25 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

As systems are becoming more complex, they require more and more complex scheduling policies and hence are more complex to design and analyse. Modern systems require schedulers to determine which resource(s) execute the work, as well as the order in which the work is executed. The scheduler decisions become a function of the workload parameters, system configurations, or even the physical layout of the underlying cyber-physical system. Analysing such schedulers becomes very challenging since the space of possible schedules is very large.

Such schedulers can be found in smart and flexible manufacturing systems. An industrial printer is an example. The productivity of a printer is optimized by a scheduler that can be adjusted based

This article appears as part of the ESWEEK-TECS special issue and was presented at the International Conference on Embedded Software (EMSOFT) 2019.

on the specification of the physical printing device and the type of work the printer is supposed to handle. Due to the wide range of parameters for such systems it is useful to understand how the physical system can be co-designed with the scheduler. This understanding can be shared between disciplines to determine the impact of parameters on system productivity, for example, for a given scheduling policy and specific workloads.

We investigate how to characterize the impact of design parameters of a system on the scheduling decisions that are taken in case that the conditions that lead to particular decisions are affine. Such design parameters may include the clock speed and the relative workload of tasks for multiprocessor systems, or the physical dimensions of modules of a manufacturing system. Symbolic execution of the scheduler can determine which *execution paths* in the scheduling algorithm are taken for different parameter values. In this paper we characterise the sequence of decisions and record the (parametric) scheduling outcomes for different parameter values. Our approach achieves a fast characterisation of scheduler decision mechanisms that use affine combinations of parameters to make decisions.

We consider deterministic schedulers, in particular we are interested in those that minimize makespan of a given set of non-pre-emptive jobs. Schedulers need to enforce orderings of events, while meeting certain requirements on the timing of those events associated with the jobs in the system. Our technique requires a scheduler to (re-)construct symbolic affine conditions for its decisions. Our approach uses these symbolic conditions in a divide-and-conquer algorithm to find regions for which the same parametric schedule is produced. Expressions for the makespan of the result can then be found. The problem and approach in this paper are motivated by characterising schedulers for re-entrant manufacturing systems, but the approach is generally applicable to other types of schedulers as well. Our approach requires that the scheduler for the given scheduling instance (1) terminates in finite time for any parameter combination, and (2) leads to a finite number of distinct parametric scheduling results. A key challenge is to cover all parameter combinations such that the parametric decision regions are characterized including their borders, where a scheduler changes its decision because of some critical constraint or requirement. Each parametric decision region relates to an execution branch for concrete parameter values.

Related work is discussed in Section 2. We then define symbolic scheduling (Section 3) and illustrate how to integrate it into a classical scheduling heuristic (Section 4). We show how such a symbolic scheduler is used to evaluate a concrete parameter combination (Section 4) and how to interpret the information it returns (Section 5). In Section 6, we introduce an exact scheduler characterization algorithm that covers all parameter combinations in a given range. In Section 7, we evaluate the algorithm for the classical Shortest Processing Time First (SPTF) scheduler and schedulers for a manufacturing system. We compare our approach to the generic symbolic execution engine LLVM KLEE. Section 8 concludes.

2 RELATED WORK

Determining whether a particular set of tasks and an activation pattern is schedulable is essential for embedded systems. Such schedulability problems have received much attention for varying scheduling policies [2, 19]. The research on schedulability problems has initially focussed on cases where relevant parameters, such as task durations, offsets, relative deadlines, and periods, are given and constant. For a given system, such solutions determine whether the system is schedulable. Such schedulability problems have been extended to parametrized systems, where the problem is to determine for each parameter combination whether the system is schedulable [3, 5]. Schedulability regions have been determined for periodic task sets in [3]. The task execution times in this work are parametric, and it provides solutions for rate-monotonic and fixed-priority scheduling. The results show that it is possible to use hyperplane representations for these scheduling models. The hyperplanes determine limits on the schedulability, i.e., for which task execution times the processors are proven to be capable of executing a workload. The results map each parameter combination in a parameter space to a binary outcome (schedulable or not schedulable). The approach is specific to rate-monotonic and fixed-priority scheduling. The parametric timed automata approach in [5] allows to determine the schedulability for a wider range of systems than [3]. This approach supports parametrized task sets (with parameters for completion time, relative deadline, release times), and activation patterns, also taking into account non-deterministic behaviour for *fixed-priority* scheduling. The approach iteratively bounds the schedulability region. In general, the approach of [5] may not terminate in a finite number of steps. However, the authors show that their approach converges for periodic task sets with bounded activation offsets. Our approach addresses performance characterization, which is more generic than the binary schedulability analysis. The result of our approach is a set of performance regions, and for each region a parametrized expression characterizing performance aspects such as makespan. Our approach targets deterministic systems with schedulers that can take into account the context when scheduling decisions need to be made. Our approach can therefore handle parameters for *deterministic* constraints in completion time, release time, and relative deadlines, as long as the scheduling conditions are affine in nature.

The work of Nasri et al. [13, 14] extends schedulability tests for task sets when the system is scheduled by a job-level fixed-priority scheduling algorithm. Given a set of input jobs with non-deterministic parameters (i.e., with bounded uncertainty), their approach efficiently determines whether the system is schedulable under all execution scenario's, as well as the best-case and worst-case response times of each job. Although the goal of the work of Nasri et al. is similar to our work, the problem that is tackled is different from the one tackled in this paper. In contrast with their work, the problem we present assumes that task execution times and timing constraints are deterministic, and that parameters can be shared among tasks and timing constraints. Moreover, as mentioned before, our work does not only perform schedulability checks, but it also yields performance regions with symbolic makespan expressions characterizing the full scheduler behaviour.

The parametric scheduling work of Subramani [15] also focuses on schedulability queries for real-time systems, using the execution-time-constraints (ETC) framework. This dispatching mechanism only considers constraints from and to *past* events. It does not take into account that constraints from and to known *future* events impact the scheduling decision, and computes a single range in which a task/job can be dispatched.

Parametric timed automata approaches for schedulability such as [5], [9], and [12] also return a set of symbolic conditions, called zones. The typical query, however, is whether a *given* state is reachable. The problem in this paper requires finding whether any accepting state is reachable under certain parameter evaluations, which has been shown to be PSPACE-Complete [1]. Current approaches [9], however, do not give an algorithm to *discover* any accepting states automatically and also capture the symbolic conditions for the decisions leading to these states (in the form of a decision region). Our approach simultaneously discovers the reachable scheduling results and the symbolic conditions under which they are reachable.

The convexity of parametric problems is often based on the resulting parametric affine expressions, as in [7], [10], and [16]. In these works, the throughput expressions and affine schedules are determined for data-flow graphs and iteration vectors respectively. Such results are valid on the border of regions, whereas our problem requires the regions not to overlap. Our approach finds symbolic conditions that are related only to the decision making, but are not necessarily derived from the scheduling result.

Scheduling problems can often be expressed as integer programming problems. Such integer programming problems are often not amenable to solving to optimality due to the complexity of the scheduling problem. The constraints of such integer programming problems can be expressed by parameters [6] [7]. This work solves a different kind of parametric scheduling problem; the decision making is performed by a general-purpose solver. The scheduling result relates directly to the symbolic conditions under which the result is optimal, which is not a limitation of our approach. Our approach characterizes all results achievable by a particular scheduler, thereby characterizing the full scheduler behaviour.

There have been significant advances in the automated generation of test cases in recent years. Such generic techniques can be used to cover all possible execution paths of a program. Symbolic execution engines such as LLVM KLEE [4] keep track of symbolic variables automatically. Using the symbolic information, LLVM KLEE tries to find for each execution branch concrete values for the parameter combination that lead to that particular branch, to be used as a test case, or to show that no concrete values can ever lead to that execution branch. LLVM KLEE is the only related work that can be used to directly address the problem we consider in this paper. We provide an alternative, more specialized approach that outperforms LLVM KLEE for the given problem, and returns more insight into the productivity of schedulers under varying parameters.

3 SYMBOLIC SCHEDULING

Schedulers decide in what order and at what time events should take place. A scheduling problem often contains a set of static timing constraints that are enforced regardless of the scheduler's decisions. For example, there may be a minimum and maximum time constraint between the start and the end of a task. Conditional timing constraints, such as communication, processing, reconfiguration times or sequence-dependent set-up times, are active only when certain conditions are met. The scheduler needs to take into account that the set of active constraints can change when the order of events or the resource allocation changes. Many schedulers first decide on which resource and/or in what order the required events should take place, before start times are determined. The scheduler must ensure that all timing constraints of the scheduling problem are satisfied. A sequence of decisions is taken, based on the evaluation of certain alternatives. The scheduler then computes event realization times to instruct the system.

Schedulers (and programs in general) typically use concrete values to choose between options. In symbolic scheduling, these concrete values are replaced by symbolic expressions, that relate the values to system parameters. The scheduler can then evaluate the symbolic expressions by substituting the parameters with their concrete values. We show that such a symbolic scheduler can determine to what extent the parameters may change before any decision in the decision sequence changes. If the conditions used in the scheduler are restricted to conjunctions of affine inequalities of parameters, then we can relatively efficiently determine for all parameter combinations which decision sequence will be taken. We assume that a symbolic scheduler returns: (i) the schedule produced at a particular parameter point, (ii) a set of necessary symbolic conditions, i.e., affine inequalities on the parameters, defining a region that includes all points that lead to the same decision, and (iii) the result of a concrete decision sequence, e.g., the parametrized timing relations between events.

4 RUNNING EXAMPLE

We first introduce a basic scheduling model: an m-machine n-task allocation problem. We then explain the SPTF heuristic. We use this particular scheduler to show that our approach is applicable to commonly used types of schedulers and as running example.

4.1 m-machine n-task allocation problem

A queue $T = \langle (t_1, d_1), \ldots, (t_n, d_n) \rangle$ of tasks t_i with duration $T(t_i) = d_i$ needs to be allocated on a set of m identical machines, referred to in general as resources. The tasks have no dependencies among each other. A task $t_i \in \operatorname{dom} T$ is executed without interruptions for the specified duration d_i. The scheduling problem is to assign each task to a machine and determine a feasible schedule σ for the start of each task that minimizes the maximum completion time of all machines $C_{\max}$.

We use a simple list scheduler heuristic as an example for our approach. List schedulers iteratively schedule the *highest ranked task* onto the *highest ranked resource*. It suffices to define task and resource ranks to complete the algorithm.

4.2 Shortest Processing Time First Scheduler

An SPTF scheduler is a list scheduler that iteratively schedules a remaining task with the shortest processing time on the earliest available resource (Algorithms 1 and 2). The result is a partitioning of the task queue over resource queues. The resource queues represent dependencies between tasks which share resources, for example $\langle (t_1, a), (t_2, 2a) \rangle, \langle (t_3, b) \rangle$ expresses two resource queues of tasks with parametric execution times with parameters a and b; t_2 depends on completion of t_1, both allocated to the first resource.

At each step, the scheduler must select exactly one option; a tie-breaking condition is required if several options are equally preferable. If several tasks or resources have the same duration or availability time, then Algorithm 1 always selects the first occurrence, which makes it deterministic. In literature, such a tie is typically broken arbitrarily.

Algorithm 1 Shortest processing time first scheduler

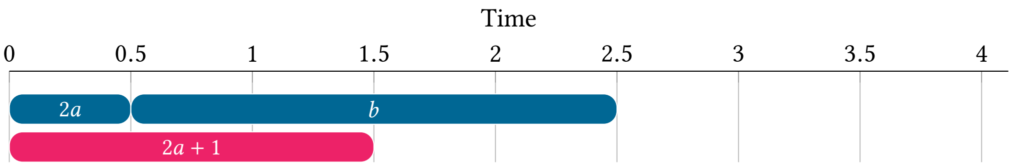
```
1:  function SPTF_SCHEDULE(task queue $T = \langle (t_1, d_1), \ldots, (t_n, d_n) \rangle$, resources $R = \langle r_1, \ldots, r_m \rangle$)
2:    $\sigma = \emptyset$ // start with an empty schedule
3:    for each $r \in R$ do
4:      $Q(r) = \langle \rangle$ // empty queues $Q$ for each resource
5:    repeat
6:      remove task $x = (t_x, d) \in T$ s.t. $d = \min_{(t_i, d_i) \in T} d_i$
7:      select resource $r \in R$ with min. COMPL($Q(r)$)
8:      $s =$ COMPL($Q(r)$)
9:      append $x$ to $Q(r)$
10:     $\sigma(t_x) = s$
11:   until $T$ is empty
12:   return $Q, \sigma$
```

Algorithm 2 Calculate completion time of a queue

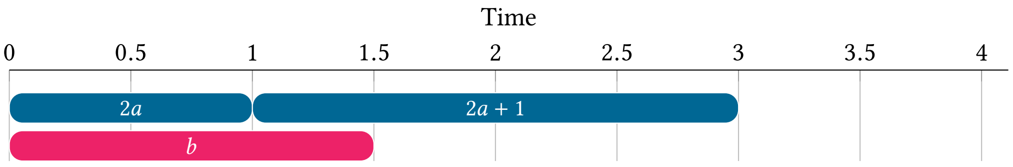
```
1:  procedure COMPL(queue $q$)
2:    return $\sum_{(t_i, d_i) \in q} d_i$
```

5 PARAMETRIZED SCHEDULERS

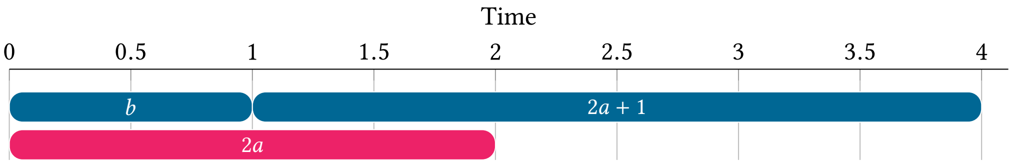
We introduce the concept of decision regions, taking the shape of polyhedra in the design parameter space, in Section 5.1, before extending the SPTF scheduler with symbolic scheduling in Section 5.2. In Section 5.3, we show how to extend the evaluation of conditions in the symbolic scheduler such



(a) Schedule for parameter combination $(a, b) = (1/4, 2)$.



(b) Schedule for parameter combination $(a, b) = (1/2, 3/2)$.



(c) Schedule for parameter combination $(a, b) = (1, 1)$.

Fig. 1. SPTF example schedules for task queue $\langle(t_1, 2a), (t_2, b), (t_3, 2a + 1)\rangle$.

that it can evaluate points that are arbitrarily close to, but not on, the edge of a polyhedron. We use this concept to deal with tie-breaking conditions, characterizing the parameter combinations where a scheduler changes its decision. In Section 5.4, we visualize the branching conditions used in the symbolic scheduler as a decision tree.

5.1 Decision regions

We consider deterministic schedulers for which the conditions of scheduler decisions can be denoted as the conjunction $\bigwedge S$ of a set S of symbolic affine inequalities. The universe of symbolic affine inequalities with d parameters is represented by $U = \mathbb{R}^{d+1} \times \{<, \leq\}$. We denote a symbolic affine inequality by a tuple $(\boldsymbol{e}, \triangleleft)$ with $\boldsymbol{e} \in \mathbb{R}^{d+1}$ and $\triangleleft \in \{<, \leq\}$. A symbolic scheduler evaluates parametric affine expressions $\boldsymbol{e}(\boldsymbol{p}) = \boldsymbol{b} \cdot \boldsymbol{p} + c$, where $\boldsymbol{p}$ is a vector of parameters, $\boldsymbol{b}$ is a vector of weights, c is a constant and $\cdot$ denotes a vector inner product. The affine function $\boldsymbol{e}$ is represented by a vector b consisting of $\boldsymbol{d}$ coefficients and the constant c, and the function evaluation of $\boldsymbol{e}$ parameter point $\boldsymbol{p} \in \mathbb{R}^d$ becomes $\boldsymbol{e}(\boldsymbol{p}) = [\boldsymbol{b}\ c] \cdot [\boldsymbol{p}\ 1]$. A symbolic affine inequality can be evaluated to a boolean at a parameter point p as $\boldsymbol{e}(\boldsymbol{p}) \triangleleft 0$, where $\triangleleft$ is either $<$ or $\leq$. The negation of an affine inequality $\neg(\boldsymbol{e} \triangleleft 0)$ is defined as follows: $\neg(\boldsymbol{e}, <)$ is equivalent to $(-\boldsymbol{e}, \leq)$, and $\neg(\boldsymbol{e}, \leq)$ is equivalent to $(-\boldsymbol{e}, <)$.

A conjunction of symbolic affine inequalities can be interpreted as the intersection of the corresponding half-spaces in the parameter space [8][11][12]. The resulting polyhedron is therefore convex for schedulers that use affine inequalities to perform their decision making. For strict inequalities, the bounding hyperplane is excluded. The convexity of the decision regions cannot be guaranteed when non-linear conditions are used by the scheduler. We want to find a polyhedron for every point in which, the scheduling algorithm makes the *same sequence of decisions*. We use the term *decision region* to refer to such a convex polyhedron.

Algorithm 1 returns the task ordering $Q(r)$ for each resource r, and for each task $t \in \operatorname{dom} T$ the earliest possible starting time $\sigma(t)$. In addition, the scheduler can keep track of the reasons why it made these decisions for the given parameter values. For example, let the queue of symbolic tasks and their duration expressions be $\langle (t_1, 2a), (t_2, b), (t_3, 2a+1) \rangle$. The selection condition $d = \min_{(t_i,d_i) \in T} d_i$ (used in Algorithm 1 Line 6) can be written as the conjunction of inequalities such that the duration d_x of task t_x is at most the duration of any task in T: $\bigwedge_{(t_i,d_i)\in T} d_x \leq d_i$. When $a = 1/4$, $b = 2$, the scheduler picks task t_1 with duration a first, and will do so for other parameter combinations, as long as $2a \leq b$.

Similarly, the minimum completion time condition can also be expressed as a symbolic inequality, as the completion time of a resource is the sum of a set of symbolic duration expressions. In the example, after selecting task t_1, the first (least busy) resource is selected to execute for an additional $2a$ time units. Both resources are available at time 0, and therefore task t_1 is scheduled on r_1. The completion times of r_1 and r_2 are then $2a$ and 0 respectively. Task t_3 with time $2a + 1 = 3/2$ then remains the smallest task as long as $2a + 1 \leq b$. Resource r_2 will be selected for task t_3 under the condition that there is no other resource than r_2 that is idle earlier, i.e., the scheduler requires that $0 \leq 2a$. Moreover, as resource r_1 occurs earlier in the resource list than r_2, the scheduler also requires that r_2 is the *first* resource *in the list* that is idle at the given time, i.e., the scheduler requires that $\neg(2a \leq 0)$. The two conditions combined simplify to $0 < 2a$. After scheduling t_3, the resource completion times are $2a$ and $2a + 1$ for r_1 and r_2 respectively. Finally, the only remaining task t_2 with duration b is added to the resource with minimum completion time. The resource selection condition $2a \leq 2a + 1$ is true regardless of the value of a, and t_2 is therefore scheduled on resource r_1. This schedule is shown in Figure 1a. For this example, $C_{\max} = \max(2a + b, 2a + 1)$, for the parameter values$(a, b) = (1/4, 2)$: $C_{\max} = 5/2$ time units.

5.2 Symbolic SPTF scheduler

Given a particular parametrized set of tasks and resources, a deterministic scheduler makes its decisions sequentially. Algorithm 3 extends Algorithm 1 with symbolic scheduling. It captures the conditions that distinguish between the selection of scheduling options. In Algorithm 3, the condition for selecting task t on Line 6 is captured on Line 7. Similarly, the resource selection condition for r is evaluated on Line 8 and is captured on Line 9.

Note that the symbolic scheduler captures only closed intervals, even though on the interface between two decisions, only one decision is taken. We show in Section 6 that it is not necessary to explicitly return the tie-breaking condition for our approach.

Let the queue of symbolic tasks and their duration expressions again be $\langle (t_1, 2a), (t_2, b), (t_3, 2a+1) \rangle$. Figure 1 shows three symbolic schedules that are produced for this task set for three different parameter combinations ($(a = 1/4, b = 2)$, $(a = 1/2, b = 3/2)$ and $(a = 1, b = 1)$). At these parameter combinations the scheduler returns several necessary conditions for the same decisions to be taken as shown in Figure 2. The relevant task and resource selection conditions are shown in Table 1[1]. These selection conditions result in polyhedra that overlap, yet the scheduler only returns one specific result and schedule for each parameter combination. In Figure 2, the points on the line $b = 2a$ belong to the blue region, as the task with duration $2a$ (as the first task in the task queue) would be picked first. Similarly, the points on the line $b = 2a + 1$ belong to the green region, as task t_2 with duration b is picked before task t_3.

[1] In figures and tables, the task indices have been omitted.

Algorithm 3 Symbolic shortest processing time first scheduler

```
1: function SYMBOLIC_SPTF_SCHEDULE(task queue $T = \langle(t_1, \boldsymbol{d}_1), \dots, (t_n, \boldsymbol{d}_n)\rangle$, resources $R = \langle r_1, \dots, r_m\rangle$, parameter values $\boldsymbol{p}$)
2:   $c = \emptyset$ // initialize an empty set of conditions
3:   for each $r \in R$ do
4:     $Q(r) = \langle\rangle$ // empty queues $Q$ for each resource
5:   repeat
6:     remove $x = (t, \boldsymbol{d}) \in T$ s.t. $\boldsymbol{d}(\boldsymbol{p}) = \min_{(t_i, d_i) \in T} \boldsymbol{d}_i(\boldsymbol{p})$
7:     $c = c \cup \{ (\boldsymbol{d} - \boldsymbol{d}_i, \leq) \mid (t_i, \boldsymbol{d}_i) \in T \}$
8:     select $r \in R$ with min. $\textsc{compl}(Q(r))(\boldsymbol{p})$
9:     $c = c \cup \{ (\textsc{compl}(Q(r)) - \textsc{compl}(Q(r_i)), \leq) \mid r_i \in R \}$
10:    $\boldsymbol{s} = \textsc{compl}(Q(r))$
11:    append $x$ to $Q(r)$
12:    $\sigma(t) = \boldsymbol{s}$
13:  until $T$ is empty
14:  return $c, Q, \sigma$
```

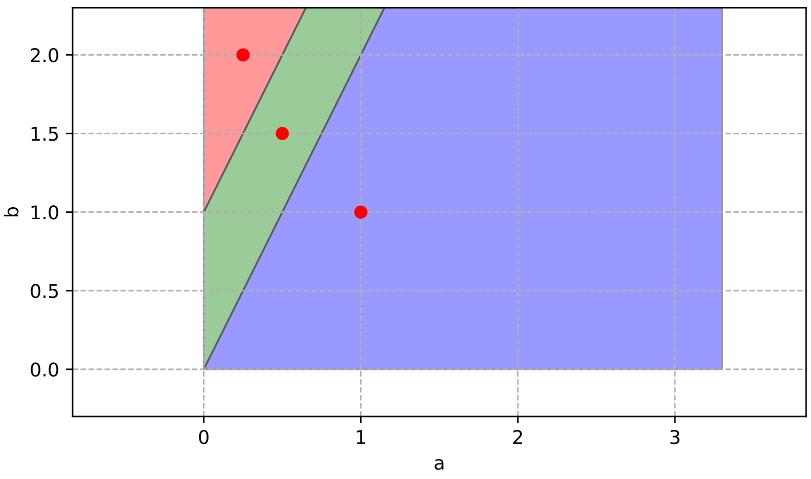


Fig. 2. Example polyhedra returned by the symbolic scheduler for task queue $T = \langle(t_1, 2a), (t_2, b), (t_3, 2a + 1)\rangle$, at parameter points $(a, b) = (1/4, 2), (1/2, 3/2)$ and $(1, 1)$, indicated by the red dots. The regions denote for which parameter combinations the same sequence of decisions are taken.

Table 1. Decision region details for task queue $T = \langle(t_1, 2a), (t_2, b), (t_3, 2a + 1)\rangle$.

$\boldsymbol{p} = (a, b)$	Task	Resource	Queues
$(1/4, 2)$	$2a + 1 < b$	$0 < 2a$	$\langle 2a, b\rangle, \langle 2a + 1\rangle$
$(1/2, 3/2)$	$2a \leq b \wedge b \leq 2a + 1$	$0 < 2a$	$\langle 2a, 2a + 1\rangle, \langle b\rangle$
$(1, 1)$	$b < 2a$	$0 < b$	$\langle b, 2a + 1\rangle, \langle 2a\rangle$

5.3 Evaluating points strictly inside a decision region

For our exploration algorithm, we need to distinguish between parameter combinations that lie on the border of a polyhedron, and those arbitrarily close to, *but not on*, the border to detect in which decision region they fall. If at parameter combination $\boldsymbol{p}$ multiple tasks tie for the smallest duration, Algorithm 3 chooses the task that occurs first in the queue. I.e., on the 'border' between two decision regions, exactly one of several mutually exclusive branching decisions is taken.

The decision regions can be open or closed such as visualized for task queue $\langle(t_1, a), (t_2, 2a), (t_3, 2), (t_4, 5)\rangle$ in 1-dimensional intervals in Figure 3a. This example is synthetic as it allows negative task execution times. However, it illustrates aspects that may occur in realistic, more complex examples, for more complex schedulers. The figure shows that any side of a scheduling region can be open or closed, depending on the conditions used in the scheduler. Moreover, it shows that schedules may occur for only a single parameter value. Point $a = 0$ is the only point in the example that leads to schedule $\langle(t_1, a), (t_2, 2a), (t_3, 2)\rangle, \langle(t_4, 5)\rangle$. The maximum completion times associated with the schedules have different slopes for different decision regions, as shown in Figure 3b. In more complex examples, the makespan may even make discrete, discontinuous steps at borders of regions. The makespan of a schedule results from its critical paths. A parametrized expression for the makespan of a schedule for a given specific parameter combination can be determined by the algorithm presented in [16]. Such critical path expressions of the parametric scheduling result determine the performance slopes inside the regions.

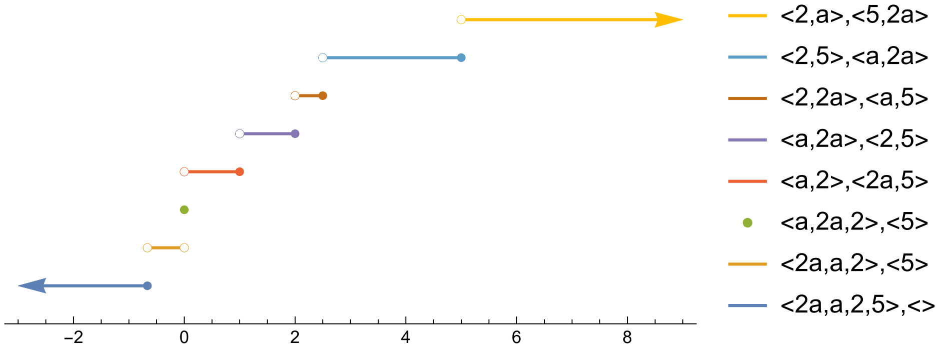
We need to evaluate parameter combinations that are *just off* a region border, to find to which decision region a border belongs. We add an infinitesimally small ϵ to some of the concrete points such that the scheduler evaluates conditions as if the parameter point is strictly inside a polyhedron to influence the decisions that the scheduler makes. This procedure is explained in Section 6.2.

5.4 Interpreting a symbolic scheduler as a decision tree

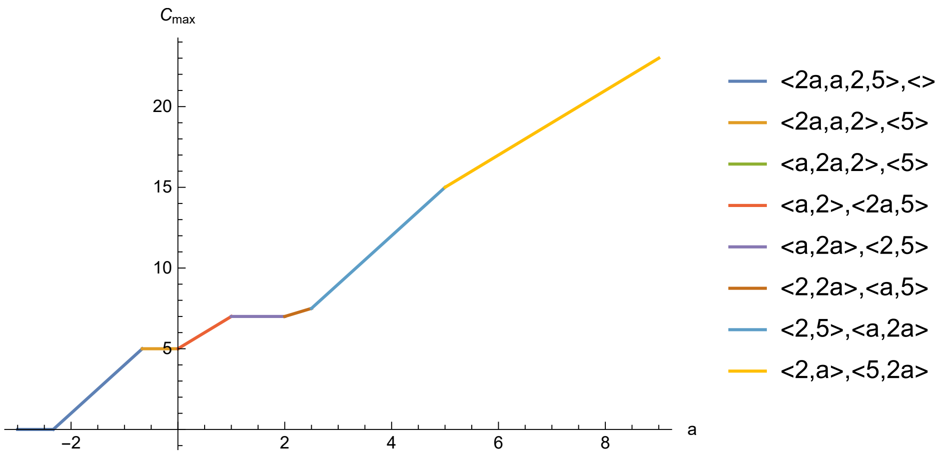
The decision sequences that can be taken by the SPTF scheduler for the task queue $\langle(t_1, 2a), (t_2, b), (t_3, 2a + 1)\rangle$ are visualized in Figure 4 as a decision tree. For each parameter combination $\boldsymbol{p} = (a, b)$ a particular sequence of decisions (i.e., a path in the decision tree) is followed. Starting from the left, the branches of the tree contain the minimum task and resource selection, and the leaves show the resulting resource queues. In Section 6 we present a structured way to find for all parameter combinations which branch of the decision tree is taken. This allows the behaviour of the scheduler to be characterized with respect to the parameters.

There are branches that cannot be taken for any parameter combination $\boldsymbol{p}$. The task t_1, with duration $2a$, for example, is always smaller than t_3 with duration $2a+1$, so task t_3 cannot be scheduled before scheduling t_1. Figure 2 shows the decision regions for strictly positive parameters. Figure 5 shows the geometric decision regions for each combination $\{ (a, b) | -1 \leq a \leq 8/5 \wedge -1 \leq b \leq 8/5 \}$. Only three decision regions fall in the positive quadrant. For the remaining decision regions, at least one parameter is negative.

As a or b becomes negative, at least one task duration becomes negative. As before, we allow negative tasks durations for illustration purposes. Behaviour that we see in this example may also occur in more complex schedulers. For some negative task durations the SPTF scheduler delivers the same schedule as for certain positive parameters, although following a different decision sequence. An example occurs in Figure 8, the two decision regions that are labelled with C correspond to different paths in the decision tree visualized in Figure 4. Different sequences of decisions can lead to the same internal state of the scheduler, which may lead to the same scheduling result.



(a) Open and closed intervals for different scheduling regions for varying values of a.



(b) Maximum completion time of resources for varying values of a.

Fig. 3. Scheduling regions and maximum completion times for scheduling the task queue $\langle(t_1, a), (t_2, 2a), (t_3, 2), (t_4, 5)\rangle$ on two resources.

6 EXPLORING PARAMETER COMBINATIONS

In Section 6.1, we present a divide-and-conquer approach that identifies to which decision regions each parameter combination belongs. The divide-and-conquer approach does so by evaluating the corner points of convex polyhedra, until it is determined that all combinations inside a polyhedron result in the same decision region. To characterize the scheduler for all parameter combinations for polyhedra with strict inequalities, we describe a procedure to generate ϵ corners, i.e., corner points that are an infinitesimal distance away from an open corner of a polyhedron, in Section 6.2.

Compared to LLVM KLEE [4], our divide-and-conquer approach is a depth-first approach. In contrast to LLVM KLEE, we explore an execution path to termination, by evaluating specific concrete points. LLVM KLEE uses a mix of several approaches to identify new (arbitrary) concrete points that continue into different (non-terminated) execution paths.

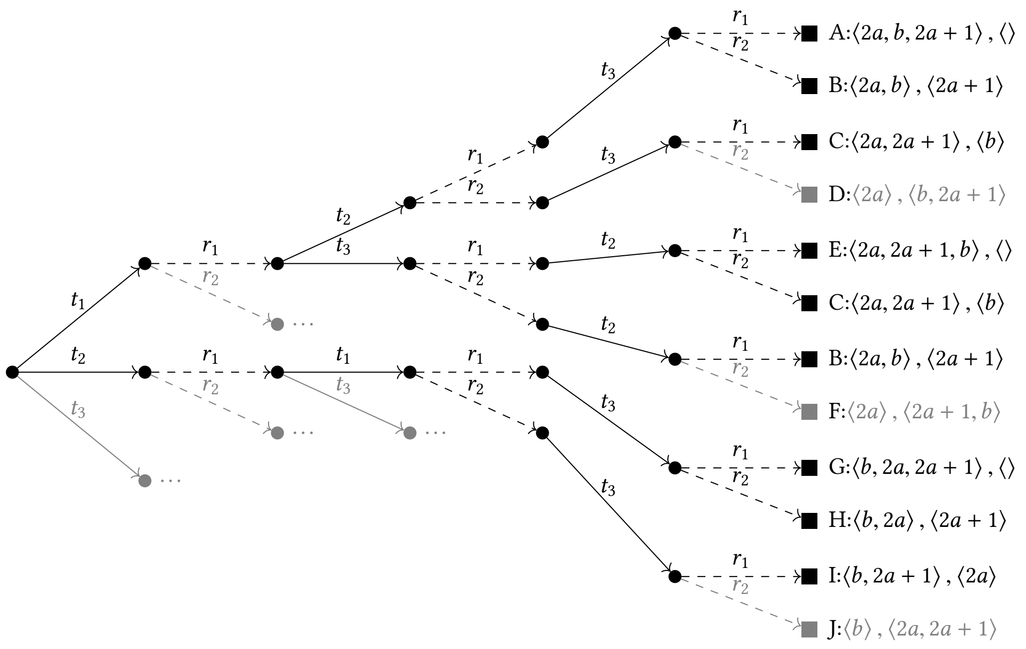

t_1
t_2
t_3
r_1
r_2
A:$\langle 2a, b, 2a+1\rangle, \langle\rangle$
B:$\langle 2a, b\rangle, \langle 2a+1\rangle$
C:$\langle 2a, 2a+1\rangle, \langle b\rangle$
D:$\langle 2a\rangle, \langle b, 2a+1\rangle$
E:$\langle 2a, 2a+1, b\rangle, \langle\rangle$
C:$\langle 2a, 2a+1\rangle, \langle b\rangle$
B:$\langle 2a, b\rangle, \langle 2a+1\rangle$
F:$\langle 2a\rangle, \langle 2a+1, b\rangle$
G:$\langle b, 2a, 2a+1\rangle, \langle\rangle$
H:$\langle b, 2a\rangle, \langle 2a+1\rangle$
I:$\langle b, 2a+1\rangle, \langle 2a\rangle$
J:$\langle b\rangle, \langle 2a, 2a+1\rangle$

Fig. 4. Example decision tree for SPTF scheduling tasks $\langle(t_1, 2a), (t_2, b), (t_3, 2a+1)\rangle$. A solid edge is a decision on a task, and a dashed edge is a decision on a resource. Greyed out decisions cannot be taken due to conflicting requirements on the parameter values.

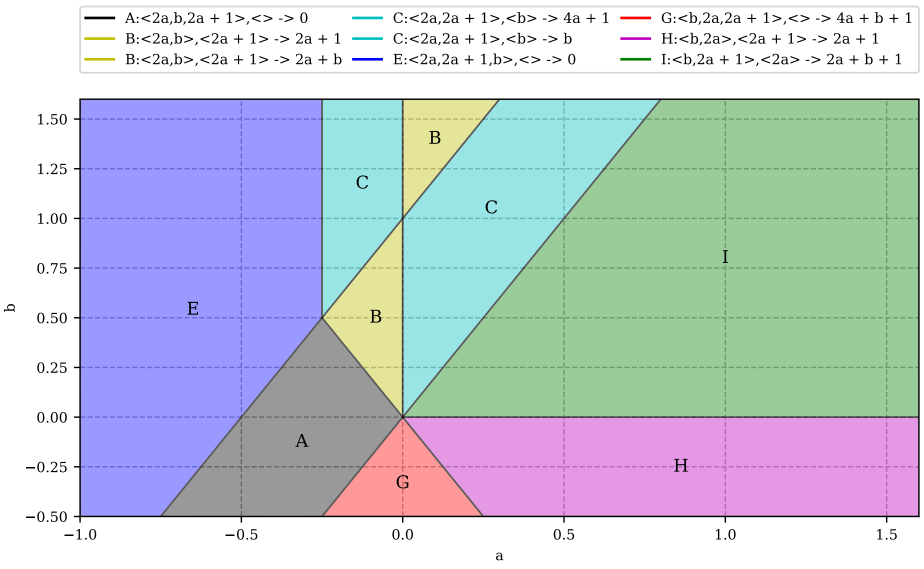


Fig. 5. Geometric visualization of which parameter combinations lead to which resource queues for the example in Figure 4. The legend shows resource queues and symbolic maximum completion times for each explored decision region. The letters relate to the leaves in Figure 4.

6.1 Divide-and-conquer approach

Algorithm 4 shows a divide-and-conquer approach that covers all parameter combinations in a given bounded convex parameter region CP. It returns a set of pairs consisting of a set of constraints defining a convex polyhedral parameter region and the parametric result that the scheduler produces in that region.

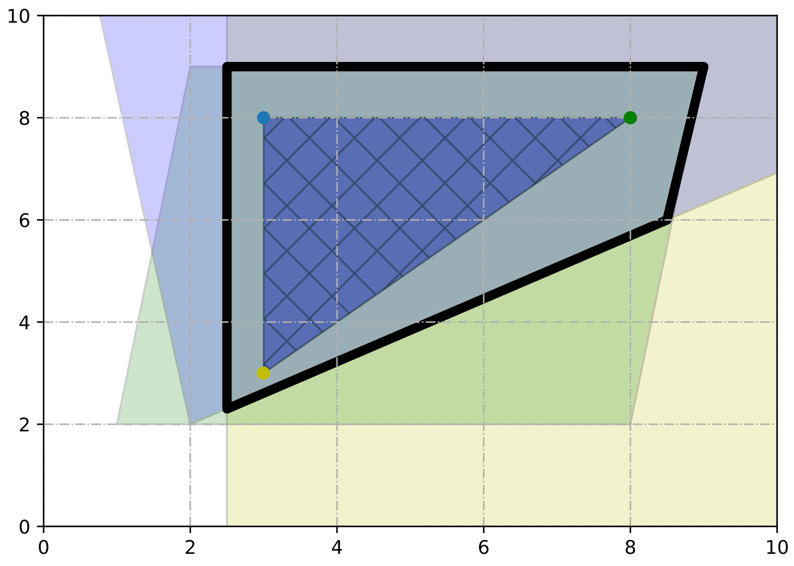
Lines 5 to 7 collect all the constraints and all the scheduler results for all corner points of the parameter region CP. A convex polyhedron can be transformed into a set of corner points using the double description library [8]. Each point in the polyhedron (including the corner points) is related to a particular execution branch in the decision tree. Lines 10 to 12 check if all constraints found (Line 11) hold in all corner points of CP (Line 10). If so, then the recursion ends and we arrive at Line 19. This situation is visualized in Figure 6a. All corner points returned the same scheduling result (g_{tot} contains only one element); therefore we can return a pair with CP and the schedule result, in Line 19. In the case that one of the constraints was found not to hold for one of the corner points (Line 12, Figure 6b), the region CP contains points from more than one decision region. In that case, we use that constraint to split the region CP in two new (smaller) regions in Lines 13, 14; Algorithm 15 then recursively computes the results for the two new, smaller, parameter regions (Lines 15, 16) and returns the union of their separate results (Line 17).

Algorithm 4 Divide-and-conquer for symbolic scheduling

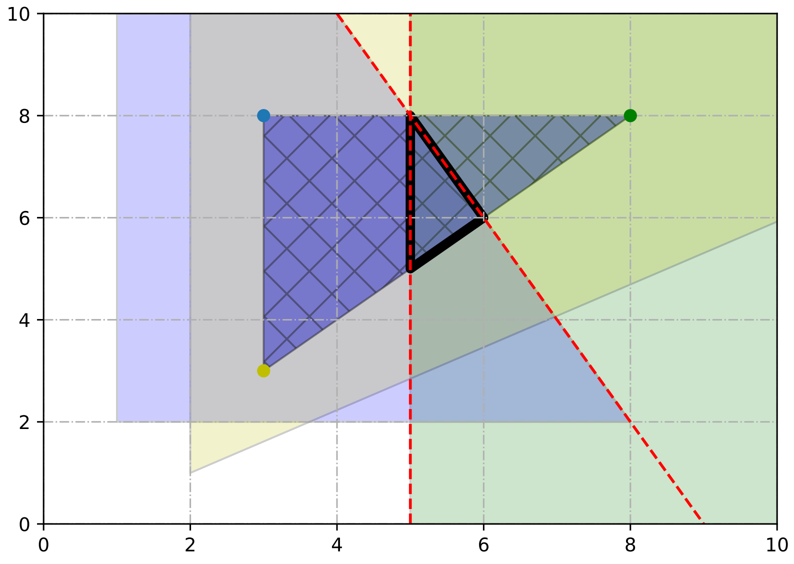
```
1:  function DivideConquer(scheduling problem $P$, bounded convex parameter polyhedron $CP$)
2:    // Empty sets to record conditions and scheduler results
3:    $z_{tot} = \emptyset, g_{tot} = \emptyset$
4:    // Combine the conditions of all corners
5:    for each corner $\boldsymbol{p}_c^\epsilon$ of Epsilon_corners($CP$) do
6:      $z_i, g_i$ = evaluate_problem($P, \boldsymbol{p}_c^\epsilon$)
7:      $z_{tot} = z_{tot} \cup z_i$
8:      $g_{tot} = g_{tot} \cup g_i$
9:    // Check whether all corners admit to all conditions
10:   for each corner $\boldsymbol{p}_c^\epsilon$ of Epsilon_corners($CP$) do
11:     for each $z \in z_{tot}$ do
12:       if $\neg z(\boldsymbol{p}_c^\epsilon)$ then // Divide into two polyhedra
13:         $CP_1 = CP \cup \{ z \}$
14:         $CP_2 = CP \cup \{ \neg z \}$
15:         $R_1$ = DivideConquer($P, CP_1$)
16:         $R_2$ = DivideConquer($P, CP_2$)
17:         return $R_1 \cup R_2$ // Return all regions
18:   Let $g_m$ be the only element of $g_{tot}$.
19:   return $\{ \langle CP, g_m \rangle \}$
```

6.2 Evaluating polyhedra with strict inequalities

The conditions leading to decision regions may include strict inequalities and non-strict inequalities. We represent a polyhedron with strict inequalities that exclude bounding faces using corner points that are at an infinitesimally small distance from the concrete corner (ϵ corners), inside the polyhedron. To determine such ϵ corners for strict inequalities (i.e., an open interface of a polyhedron), we use the technique visualized in Figure 7. This technique creates a smaller



(a) Intersection covers all points in the hatched convex polyhedron.



(b) Intersection does not cover all points of the hatched convex polyhedron. The algorithm can split the polyhedron along one of the two red dashed lines.

Fig. 6. Two situations for the intersection (outlined in thick black) of the green, yellow and blue conditions returned by their respectively coloured corners.

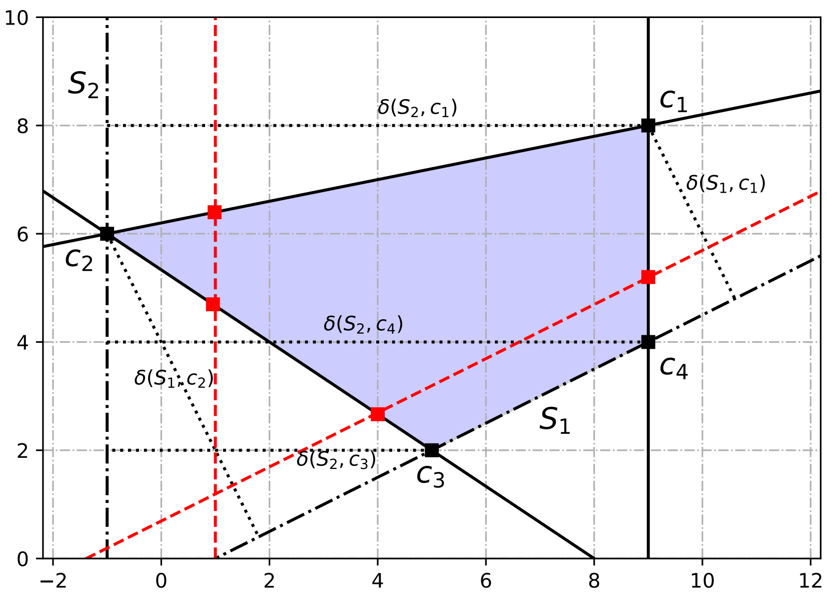


Fig. 7. Creating ϵ corners. c_1, c_2, c_3, c_4 are corners of a polyhedron, and S_1 and S_2 are strict inequalities (dash-dot lines). Calculate new concrete corners by translating s_i a concrete distance, using 1/3rd of the minimum (perpendicular) distance $\delta(S_i, c_k)$ of S_i to each other corner c_k. This generates new corner points (the red dots) without removing corners (such as c_1) on non-strict inequalities (solid lines).

polyhedron, by tightening the conditions (i.e., translation in the figure) such that they cut off the points that lie on the interface of the strict inequality $S_i = (\boldsymbol{e}, <)$. The points $\{\, \boldsymbol{x} \in \mathbb{R}^d \mid \boldsymbol{e}(\boldsymbol{x} = 0) \,\}$ are those that lie on the hyper-surface defined by $\boldsymbol{e}$; we call such points *adjacent* to S_i. We find a distance δ, to be added in the normal direction of the inequality S_i, such that concrete new corner points for a new polyhedron CP' (spanned by the red corners and c_1) are created. δ is chosen small enough to retain all corners that are not adjacent to this strict inequality. The distance at which a corner c_k not adjacent to S_i would be cut off is the perpendicular distance $\delta(S_i, c_k)$. Therefore, any concrete translation distance δ of S_i less than the minimum perpendicular distance of S_i to any non-adjacent corner point c_k does not cut off corner points that are not adjacent to S_i.

Translating the inequality S_i with a concrete value leads to a smaller concrete polyhedron CP'. The corner points of CP' that do not coincide with any corner of CP are used to determine the ϵ corners. The ϵ corner is determined by moving the original corner in the direction of the corresponding concrete corner(s) of CP' by an infinitesimal distance. This creates a corner point for which no concrete point lies between that corner and the original corner. As multiple corners may be cut and generated (as for S_1 and corners c_3 and c_4) we need a way to distinguish which new corners relate to the old corners. To do so, we take the concrete δ small enough so that any new corner is generated closer to its original corner than to any other corner. This distance should be less than half the minimum of the distance to adjacent corners and the perpendicular distances to non-adjacent corners.

In Figure 7, a simpler method seems possible; namely, for each corner point adjacent to a strict inequality, generate new corners that are in the direction of the adjacent corner points. When more than one strict inequality is supplied for a corner point, this method would generate multiple corner points, each of which is still adjacent to at least one other strict inequality. Therefore, this simpler procedure will not always lead to the correct corner points to describe the polyhedron with strict inequalities.

For the procedure sketched here to work, the symbolic scheduler that is used by the exploration algorithm should be capable to evaluate conditions for parameter values with infinitesimal parts.

6.3 Soundness and termination

In this section we provide arguments that the results of our divide-and-conquer approach are sound and that the approach terminates. Soundness of the approach requires that the parameter combinations correspond to the correct execution path and parametric scheduling result. Recall the requirement on the symbolic scheduler to return a (sufficiently) complete set of constraints for the path corresponding to a concrete point.

6.3.1 Soundness. Each concrete parameter combination has a particular execution path associated with it. Each execution path maps to some parametric scheduler result, from which the (parametric) productivity can be deduced. Let $\boldsymbol{p} \in CP$ be a parameter combination in some convex parameter polyhedron CP. Assume the execution path of $\boldsymbol{p}$ is not associated with the execution path of any other point $\boldsymbol{p'} \in CP \setminus \{\boldsymbol{p}\}$, and that our divide-and-conquer approach incorrectly reports that $\boldsymbol{p}$ has the same execution path as all other points in CP.

Our approach only reports all parameter combinations in the same decision region when each corner point of CP admits to the union of the conditions for all corner points (Line 12). The above assumptions imply that the execution path for $\boldsymbol{p}$ occurs only under some condition that does not hold for at least one of the corners of CP. However, as CP is a convex polyhedron, no *affine* condition (i.e., half-space) exists that includes $\boldsymbol{p}$ from CP, while not also including at least one of the corners of CP. This contradiction shows that $\boldsymbol{p} \in CP$ belongs to the same decision region, under the assumption that indeed a sufficiently complete set of conditions is returned at each (ϵ-)corner.

6.3.2 Termination. The termination argument is similar to the argument in [10]. On Lines 13 and 14, the two smaller, non-empty, polyhedra CP_1, CP_2 cover all parameter combinations in CP, without overlap. Recall from the introduction the assumptions that (1) the symbolic scheduler terminates after a finite time for any parameter combination, and (2) that the exploration leads to a finite number of decision regions. As the polyhedra always become *strictly smaller* in the split on Lines 13 and 14, eventually the base case is reached on Line 19. The base case is then recursively propagated back to form the result for the initial call of Algorithm 4.

7 EXPERIMENTAL EVALUATION

We apply our approach to several schedulers to show that the technique is applicable to both classical processor scheduling and scheduling techniques relevant for manufacturing systems. The result of our parametric scheduler characterization allows us to investigate scheduling productivity with respect to design parameters for re-entrant manufacturing systems.

The divide-and-conquer experiments described in this paper have been implemented and executed with Python3, using the Python wrapper for the CDD [8] double description library. To compare our approach to alternative methods, we implemented these simple scheduling algorithms in C, and used LLVM KLEE v2.0 with the recommended LLVM clang 6.0 compiler and Z3 SMT solver to execute the benchmark set.

The parameter combinations inside a given convex polyhedron are forwarded to the actual scheduling algorithm. For the other parameter combinations, the C-program simply exits before

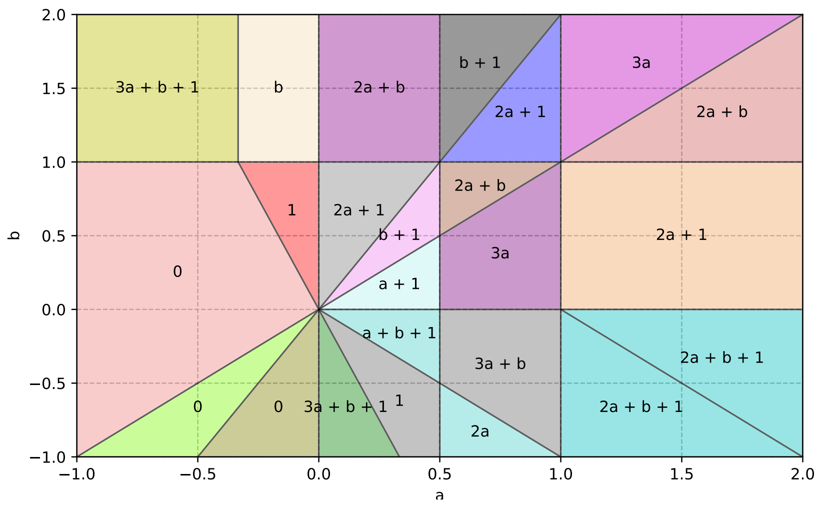


Fig. 8. SPTF performance characterization for task queue $\langle(t_1, a), (t_2, 2a), (t_3, b), (t_4, 1)\rangle$. The maximum completion times are shown per decision region.

performing any scheduling, leading to one additional reported execution path. The compiled C-programs are then symbolically executed using the KLEE execution engine. This execution results in the exploration of all execution paths, and a report of concrete points that lead to the different execution paths. All experiments have been performed on an Intel Core i7 960 with 12 GB RAM, running 64-bit Ubuntu 18.04.

7.1 Shortest Processing Time First scheduler

Figure 4 shows the characterisation of the SPTF scheduler for task queue $\langle(t_1, 2a), (t_2, b), (t_3, 2a + 1)\rangle$. For $\boldsymbol{p}_+ = (a, b)$ with strictly positive parameters ($a > 0, b > 0$) three decision regions are found. As task durations $2a$ and $2a + 1$ are already ordered, these decision regions correspond to the relative ordering of b to $2a$ and $2a + 1$. The decision regions have maximum completion times that depend both on a and b. The static part of the task duration, 1, in $2a + 1$ is not critical as long as $b > 2a + 1$.

Figure 8 shows the performance characterization for the task queue $\langle(t_1, a), (t_2, 2a), (t_3, b), (t_4, 1)\rangle$, for $-1/2 \leq a \leq 1, -1/2 \leq b \leq 2$; 22 different scheduler results are found in 26 decision regions. The maximum completion time in the region $R_1 = \{(a, b) \mid 0 \leq a \leq 1/2, 0 \leq b \leq 1\}$ depends either on a or on b, but not on both. For all parameter combinations in R_1, the task with duration 1 is the last to be scheduled as none of the other tasks are bigger in this region, and therefore always ends up in the maximum completion time expression. The two examples of Figures 4 and 8 illustrate the results of our divide-and-conquer approach. The parametric results per decision region correspond directly to a convex surface describing the makespan of the system, which also holds for the other experiments in this paper.

7.2 Schedulers for re-entrant manufacturing systems

Re-entrant manufacturing systems are interconnected machines that perform operations on a product. The product is processed by one of the machines multiple times; i.e., the product re-enters one of the machines. An example of such a re-entrant manufacturing system is an industrial printer. A duplex printer prints an image on both sides of a sheet using a single printing station. Timing

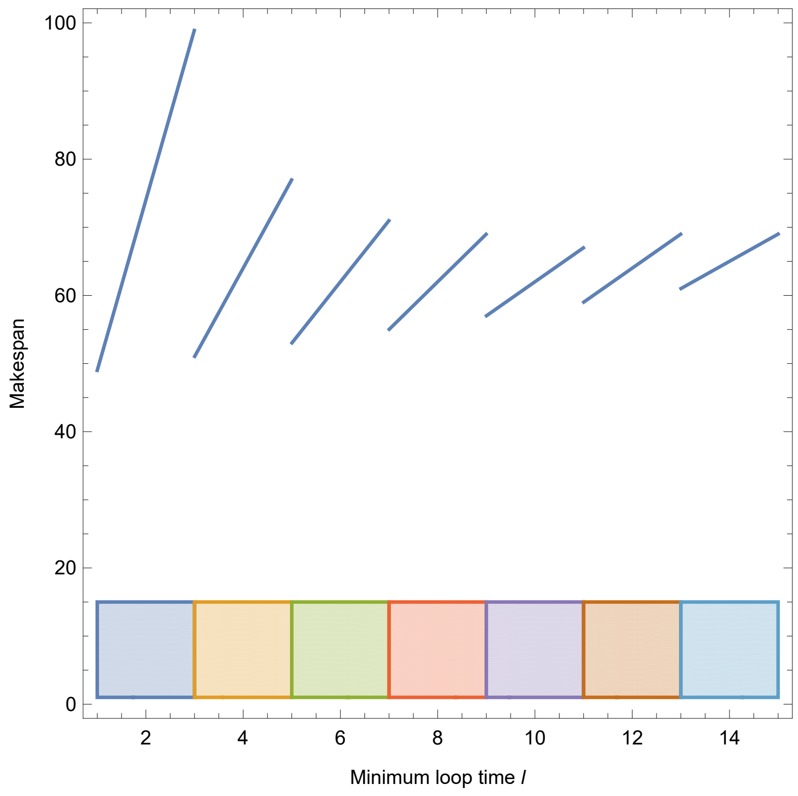


Fig. 9. Eager Scheduler applied to 25 identical sheets, $l_- = l$, $l_+ = l + 3$. The different decision regions are colour coded in the bottom. The top part of the graph shows the makespan as a function of l_-. As the minimum loop time increases, the slope of the makespan becomes less steep, while the starting offset for subsequent line segments is slightly higher.

requirements between products at the printing station are determined by the product's type, as well as the processing order at the printing station. The output order is pre-determined by an operator.

A scheduler for a re-entrant machine then needs to interleave product flows at different stages (passes) to optimize the productivity of the machines. We assume here that (i) the minimum re-entrant loop time l_- and maximum re-entrant loop time l_+ are design-time parameters that are constant during scheduling, (ii) that the set-up times $S : Z \times Z \longrightarrow \mathbb{R}$ between two products $(z_1, z_2) \in Z^2$ at the re-entrant processing station are determined fully by their product types, (iii) that the set-up times between products with the same type are the smallest possible, and (iv) that the products all need to be processed twice by some processing station (i.e., products are processed in two passes). For sake of simplicity of presentation, the set-up times of the machines include the processing time, and are the same between first pass processing and the second pass processing.

7.2.1 Eager Scheduler. The Eager Scheduler [17] selects the 'first fitting slot' in which an operation can be executed. It does so by trying to insert a re-entrant operation into an operation sequence in such a way that it does not influence the chosen timing of previously scheduled operations. The Eager Scheduler iteratively inserts the second pass operation of the next product into a sequence of (initially only first pass) operations. The scheduler fixes the re-entering time for a product after inserting its second pass to the earliest time allowed by the constraints. Evaluating whether an operation can be inserted while satisfying all constraints is possible through a modified longest-path algorithm that detects the presence of positive cycles and does not allow changing the operation times of previously scheduled operations. The presence of a positive cycle indicates that the constraints cannot be met. The symbolic scheduler needs to determine a constraint on parameters for which a given cycle is positive.

Enumerating all cycles in a graph is prohibitively time-consuming, and many of the resulting positive cycle expressions will be redundant. Instead, it is sufficient to return *some* positive cycle expression if an explored choice is infeasible. Our exploration algorithm, Algorithm 4, can then divide the polyhedron on that condition, to separate the decision regions on the outcome of that cycle. Additional positive cycles may appear during recursive evaluation of the smaller polyhedra.

Figure 9 shows how the Eager Scheduler performs for 25 identical products under variation of the minimum loop time, with a maximum buffer time of 3 seconds. The makespan per product increases as the minimum loop time increases, until another product fits in the loop. At that point, the scheduler can avoid some idle time by filling the loop more. Thus, the decision regions returned by the scheduler correspond to the number of products that fit in the re-entrant loop. As the minimum loop time increases, the slope in each region decreases; the minimum loop time parameter occurs fewer times in the critical path. Performing such an evaluation for typical product queues will lead to a better understanding of which minimum loop time is productive.

7.2.2 Pattern Scheduler. If the products in a re-entrant flow-shop are coming in a regular pattern, then instead of optimizing a single product at a time, we can interleave patterns with each other. This is exploited by the Pattern Scheduler [18]. By aligning first and second passes of products from different instances of the pattern, large set-up times can be avoided. A good heuristic is to find the maximum number of patterns that fit in the loop and allow the alignment of the two passes. We analyse a scheduler that determines how many patterns fit in a 2-re-entrant loop, and then calculates which pattern interleaving schedule would be indefinitely repeatable. A pattern $Z = \langle z_1, \ldots, z_k \rangle$ is an ordered list of $|Z|$ product types. The pattern repeats r times in the job.

The number of patterns n that can be concurrently in the re-entrant loop is calculated in a way similar to the steady-state performance estimator [18] for re-entrant flow-shops, which models a steady-state pattern scheduler. It calculates how many interleaved patterns could be executed in the interval between the minimum and maximum loop time. The buffer range is denoted by $l_b = l_+ - l_-$). Figure 10a shows that the first passes (solid rectangles) of pattern k are interleaved with the second passes (dashed rectangles) of pattern $k - n$. Pattern time t_p is derived as a sum of set-up times, given this particular interleaving of first and second passes, and use the knowledge of the pattern that the product type of $i - n \cdot |Z|$ is the same as i:

$$\begin{aligned} t_p &= \sum_{z_i \in Z} (S(z_i, z_{i-n\cdot|Z|}) + S(z_{i-n\cdot|Z|}, z_n i + 1)) \\ &= \sum_{z_i \in Z} (S(z_i, z_i) + S(z_i, z_{i+1})) \end{aligned}$$

In steady state, this means that the first passes of a pattern k and the second passes of a re-entrant pattern $k - n$ are processed every t_p time units. For any product z_i, the deadline $D(z_i)$ (denoted with red dashed arrows in Figure 10b) between the first and second passes must be satisfied for a valid schedule to exist. The relative due date $D(z_i)$ is assumed constant, and parametrized by l_+. Before the *second* pass of a product of pattern k is executed, n patterns ($n \cdot |Z|$ products) must be processed, plus a *first* pass product from pattern $k + n + 1$. That is, for each product $z_i \in Z$, the chosen loop time $t_l^* = n \cdot t_p + S(z_i, z_{i+n\cdot|Z|}) \leq D(z_i)$ should be less than the re-entrant due date $D(i)$. The set-up time $S(z_i, z_{i+n\cdot|Z|}) = S(z_i, z_i)$ in Figure 10a corresponds to the set-up time between a_1^{k+n}, and a_2^k. This constraint is shown as the cycle of bold edges in Figure 10b, and is the same for each instance of the pattern, but may be different for each product type within the pattern. Therefore, we can locally check whether the deadlines for each product type in the pattern can be met for a particular value of n.

The number n of patterns that fit in the loop is determined by the pattern time t_p, the maximum additional set-up time $t_s = \max_{z_i \in Z} S(z_i, z_i)$, l_- and l_+ as follows:

$$n_* = \left\lfloor \frac{l_-}{t_p} \right\rfloor \qquad n_{max} = \left\lfloor \frac{l_+ - t_s}{t_p} \right\rfloor$$

$$n = \begin{cases} 0 & \text{if } n_{max} \leq 0 \\ n_{max} & \text{if } 1 \leq n_{max} < n_* \\ n_* & \text{otherwise} \end{cases}$$

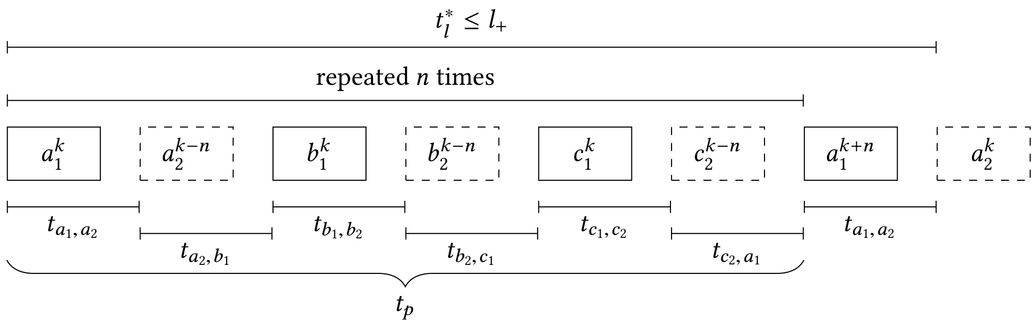
The maximum number of patterns that fit in the loop is then n_{max}. To maintain a high throughput, the scheduler tries to fit n_* patterns in the loop, such that $n_* \cdot t_p \geq l_-$, because then the re-entrant products can return exactly after finishing some previous pattern's first passes. However, if $n_* > n_{max}$ then there exists no n such that the pattern can return within the time window $l_- \leq n_* \cdot t_p \leq l_+$. In that case the scheduler uses the maximum number of patterns that fit in the loop $n = n_{max}$.

The actual timing of operations is determined by first filling the loop with n products, then interleave products of pattern k with $k - n$ in steady-state, until the loop needs to be emptied again. Appropriate sequence-dependent set-up times for this sequence are added, and earliest realization times (i.e., a schedule) are calculated. All conditions in this decision process are captured symbolically as expressions in the minimum loop and buffer times l_- and l_b.

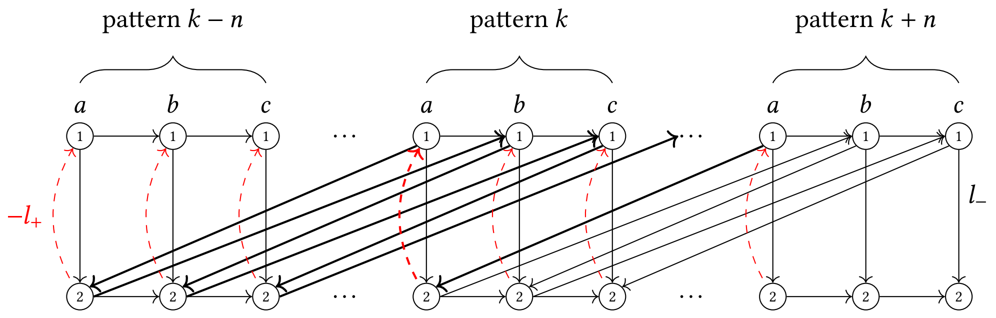
If no complete pattern fits in the loop (i.e., $n_{max} = 0$), then the interleaving procedure schedules both passes of a product to be processed before the next product starts its first pass. This is rather inefficient, so if $n_{max} = 0$ the previously mentioned Eager Scheduler is invoked instead.

Figure 11 shows for a pattern of three products, repeated 5 times, how the minimum loop time and buffer time impact the decisions made by the pattern scheduler. One pattern takes 15/2 time units, while the maximum first to second pass set-up time of the same product is $t_s = 2$. Figure 11 shows that for these combinations of minimum loop times and buffer times at most two patterns are used in the loop. The interpretation of the triangle $\{ (l_-, l_b) \mid 15 \leq l_- \wedge l_- + l_b < 17 \wedge l_b \geq 0 \}$ is that one pattern less is interleaved because the maximum loop time (being the minimum loop time plus buffer time) is not sufficient yet to allow the product's second pass to start. In this region, the adjusted loop time $n \cdot t_p < l_-$ is less than the minimum loop time, and therefore some unproductive idle time has been inserted between patterns. The productivity is therefore decreased if the minimum loop time is increased and idle time is introduced, until an additional pattern fits again. In the bottom left, the more irregular structure of the regions is due to the Eager Scheduler making less regular decisions.

This result enables a designer to study the ability of the buffer to adapt to different patterns with respect to the physical layout of the printer. Improvements can then be explored on how a larger minimum loop time or a larger buffer would impact the ability to adapt to different patterns. For this particular scheduler, adding additional buffering capabilities above some threshold does not yield additional productivity. By performing such an analysis for typical use cases, a designer can make more informed decisions.



(a) A time representation of the interleaving of n patterns, starting at the first pass of pattern k.



(b) An event graph representation of the interleaving of n patterns.

Fig. 10. A steady-state pattern interleaving example.

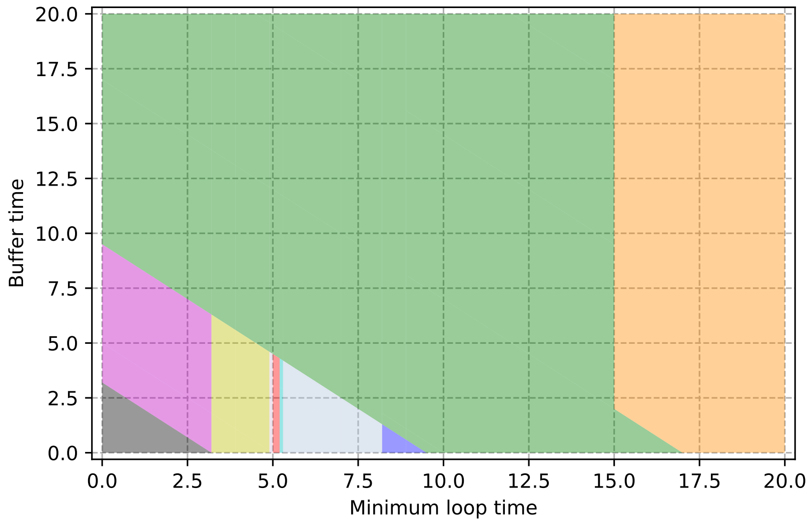


Fig. 11. Pattern scheduling for three products, repeated 5 times. In the green area, one pattern is used in the loop, in the orange area two patterns are used. If no pattern fits in the loop (below left of the green area), then the eager scheduler decides on the operation ordering.

Table 2. Parametric scheduling exploration statistics. The columns show per instance how many evaluations (E) were performed (i.e., symbolic schedule calls), how many regions (R) were generated, and how many different parametric scheduler results (PSR) were found. The last columns show the execution time (t_{dc}) of the divide-and-conquer algorithm including all scheduling calls, and execution time t_{KLEE} for the generic symbolic execution engine.

Scheduler	Instance	E	R	PSR	t_{dc}	t_{KLEE}
SPTF	Figure 2	18	3	3	0.2s	16s
SPTF	Figure 3a	30	8	8	0.8s	>3000s
SPTF	Figure 5	65	10	8	1.3s	20s
SPTF	Figure 8	172	24	21	5.5s	251s
Eager	Figure 9	193	14	14	289s	>3000s
Pattern	Figure 11	1066	45	11	948s	>3000s

7.3 Comparison with LLVM KLEE

Table 2 shows some runtime statistics for the example instances discussed in the previous subsections. The SPTF scheduler is very fast at evaluating single parameter points, and the task queues are rather small. These experiments only contain few different scheduler results and decisions. In these cases, our divide-and-conquer approach does not branch much more than strictly necessary in these cases, as the number of regions is close to or equal to the number of parametric scheduler results found. The Eager Scheduler and Pattern Scheduler require many more evaluations to cover the initial convex parameter region. In all experiments, almost all the time is spent in evaluating corner points, and the time spent gathering the conditions and choosing a branching condition were negligible. In comparison, all but the smallest benchmarks take more than 3000 seconds to execute with LLVM KLEE. Even for such simple algorithms as SPTF, the automatic tracking of symbolic variables and evaluation of many SMT queries seems to lead to an exponential increase in the running time of the benchmarks. Tight loops that depend on symbolic values lead to many calls to the SMT solver, but even with one parameter and four tasks (Figure 3a) the execution time is significant. In addition, the queries generated for the different concrete values do not reveal the particular segmentation directly, nor does it currently allow direct access to the parametric scheduling results.

The experiment of Figure 11 shows that many redundant regions were found, most leading to the same parametric scheduler results, which are in this case part of the same (green) decision region. The green area could have been detected by three cuts, but the cuts that were generated for the eager scheduler also cut the green area in more parts than necessary. These experiments can possibly be sped up by heuristically (instead of arbitrarily) choosing a separation condition that typically leads to fewer corner evaluations.

7.4 Scalability

We present some results to investigate the scalability of our method for the SPTF case. We vary the number of processors, the number of parameters, and the number of tasks. In the previous subsection we have shown that LLVM KLEE has excessive execution times for our parametrized models, even for the smallest cases with one or two parameters. The execution paths of an instance with additional parameters includes at least the execution paths of the instance without those additional parameters. It is therefore unlikely that the exploration mechanism of LLVM KLEE will find execution paths significantly more efficiently when more parameters are used in the instance. We therefore focus only on the scalability of our divide-and-conquer algorithm.

The first scalability experiment consists of SPTF instances with ten tasks, two to five processors, and up to four parameters. We have generated 10 random instances for each combination of processor and parameter count. The coefficients of the parameters and the constants are fractions. The denominator and numerator of the fractions were both drawn uniformly from $\{ 1, 2, \ldots, 10 \}$. The results of this experiment are shown in Figure 12. The time and region axes are shown on a logarithmic scale. The results for LLVM KLEE are not visualized due to the excessive computation times for all but the smallest instances.

The number of parameters has a significant impact on both the number of decision regions that need to be covered, as well as the running time of the divide-and-conquer approach. We observe a growing trend for the worst-case execution time of the algorithm for increasing processor counts. Even though each processor on average receives fewer tasks, there are more decision regions to be determined, and therefore the execution time of our algorithm grows. The running time as well as the number of decision regions grows exponentially for increasing parameter count. As the solutions necessarily grow in size, simply enumerating the answer may require exponential time. This is a common practical limitation for exact parametric methods.

Note that we could also choose to iteratively explore the decision regions for different subsets of the parameters, so as to avoid the exponential running time increase. Once the scheduler behaviour is analysed (and potentially optimized) for one subset of parameters, we can then iteratively choose another subset of parameters. This effectively slices the larger problem into smaller sub-problems.

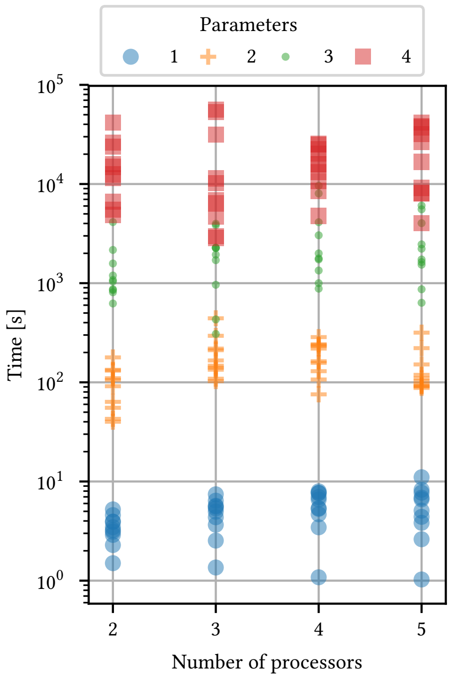
The second scalability experiment varies the task set size from two to 20, and uses two processors and only one parameter. Figure 13 shows the results of our approach on this benchmark. For varying task set size n, the number of unique regions identified grows with about $O(n^2)$. The runtime of our method grows with $O(n^3)$ when only one parameter is used, which can be explained by multiplying the number of regions by the scheduling time of a single SPTF instance, which takes $O(n)$ time.

There is significant variation in running times in both experiments. This variation is caused by the relative values of the randomly generated fractions. This means that, despite the instances having the same general characteristics, the coefficients of each task's parameters have a significant impact on the running time.

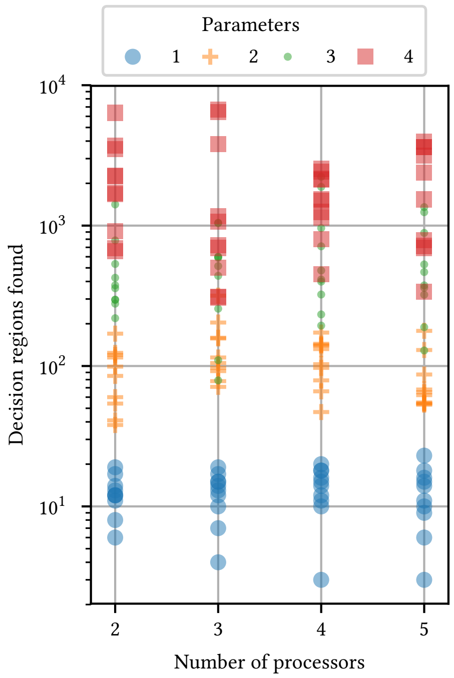
In both experiments, it shows that the number of regions found is highly correlated to the running time. The goal of the approach is to find all regions. In the very best case, the (exponentially many) solutions still need to be enumerated, and the running time therefore scales at least linear to the number of the regions. This trend holds for larger task set sizes in Figure 13c. That is, for this set of experiments our approach scales, on average, with the expected time complexity.

8 CONCLUSION

A scheduler has significant impact on the productivity of a system. We therefore want to find out how a design parameter in a given range influences the sequence of decisions made in a scheduler. These decisions lead to the activation or avoidance of timing constraints, and therefore to a particular system performance. In this paper, we introduced a symbolic scheduling model that captures the conditions under which certain decisions are taken. We introduced a divide-and-conquer algorithm that uses such information from symbolic schedulers to determine which parameter combinations lead to the same scheduling decisions for deterministic scheduling algorithms. It covers all possible parameter combinations in a given range. It can be used to exhaustively test how the system behaves in this range.



(a) Runtime for varying number of processor and parameter counts.



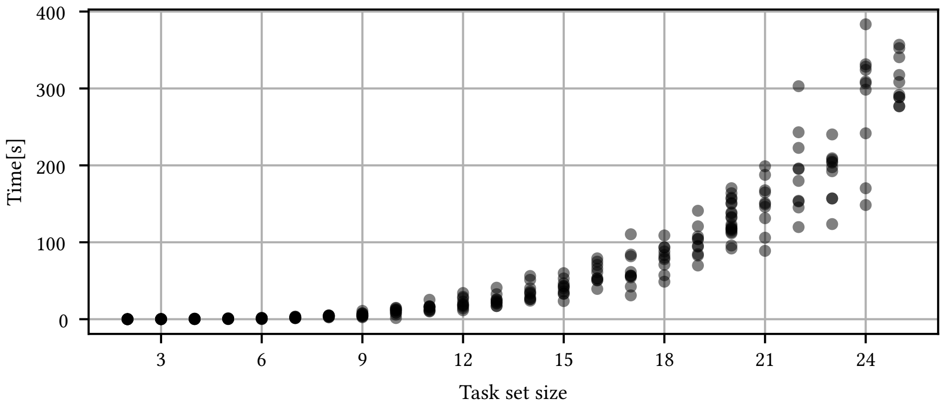
(b) Number of regions for varying processor and parameter counts.

Fig. 12. Parameter and processor count scalability results of the divide-and-conquer method for randomly generated SPTF instances.

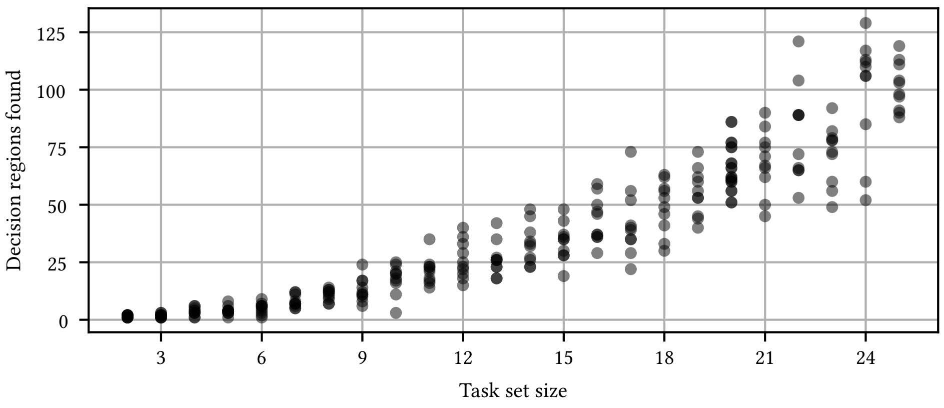
We applied this technique to several schedulers, to show that different kinds of problems and schedulers can be characterised by this technique. The results are gathered much faster than with the generic LLVM KLEE symbolic execution engine. Due to the typically tight loops in a scheduling algorithm, the KLEE engine queries the underlying SMT solver very often. Instead of branching and updating the SMT query for each branch in the tight loop, it could be interesting to append a condition to an SMT query only after each concrete full execution of a tight loop. We have shown in this paper that this gives sufficient information to explore all scheduling results, and we believe that this has the potential to speed up the exploration performed by LLVM KLEE. The construction of condition expressions can be similar to that of our approach. We believe that many deterministic schedulers are amenable to our analysis. The symbolic scheduling can be introduced to an existing scheduler to include symbolic relations with relatively few modifications.

ACKNOWLEDGMENTS

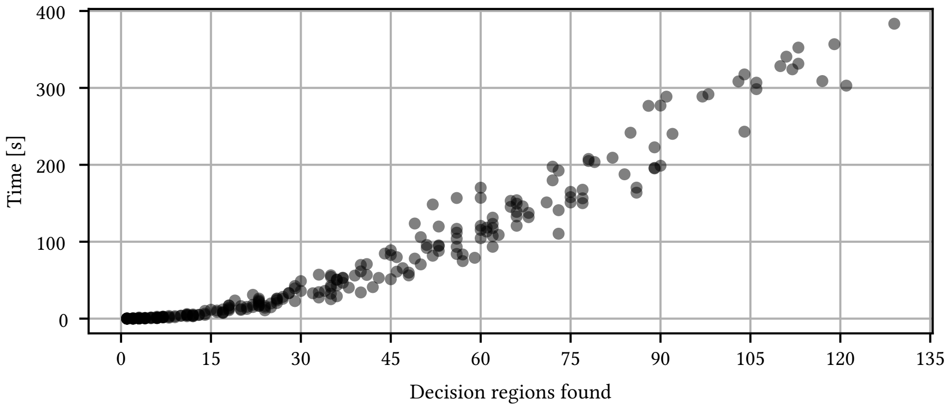
This work is part of the research programme Robust CPS with project number 12693 which is (partly) financed by the Netherlands Organisation for Scientific Research (NWO). We thank the anonymous reviewers for their helpful comments.



(a) Runtime with varying task set sizes.



(b) Number of regions with varying task set sizes.



(c) Number of regions versus running time for varying task set sizes.

Fig. 13. Results of the divide-and-conquer method for SPTF instances with varying task set sizes.

REFERENCES

[1] Rajeev Alur and David L. Dill. 1994. A Theory of Timed Automata. *Theoretical computer science* 126, 2 (1994), 183–235. DOI: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[2] Neil Audsley, Alan Burns, Mike Richardson, Ken Tindell, and Andy J. Wellings. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal* 8, 5 (Sep. 1993), 284–292. DOI: http://dx.doi.org/10.1049/sej.1993.0034

[3] Enrico Bini and Giorgio C. Buttazzo. 2004. Schedulability Analysis of Periodic Fixed Priority Systems. *IEEE Trans. Comput.* 53, 11 (Nov 2004), 1462–1473. DOI: http://dx.doi.org/10.1109/TC.2004.103

[4] Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *OSDI'08*. USENIX Association, Berkeley, CA, USA, 209–224.

[5] Alessandro Cimatti, Luigi Palopoli, and Yusi Ramadian. 2008. Symbolic Computation of Schedulability Regions Using Parametric Timed Automata. In *2008 Real-Time Systems Symposium (RTSS'08)*. IEEE, 80–89. DOI: http://dx.doi.org/10.1109/RTSS.2008.36

[6] Paul Feautrier. 1988. Parametric Integer Programming. *RAIRO-Operations Research* 22, 3 (1988), 243–268.

[7] Paul Feautrier. 1992. Some Efficient Solutions to the Affine Scheduling Problem. I. One-dimensional time. *International Journal of Parallel Programming* 21, 5 (1 Oct 1992), 313–347.

[8] Komei Fukuda and Alain Prodon. 1996. Double Description Method Revisited. *Combinatorics and Computer Science* (1996), 91–111. DOI: http://dx.doi.org/10.1007/3-540-61576-8_77

[9] Paul Gastin, Sayan Mukherjee, and B Srivathsan. 2018. Reachability in Timed Automata with Diagonal Constraints. In *International Conference on Concurrency Theory (CONCUR'18)*.

[10] Amir Ghamarian, Marc Geilen, Twan Basten, and Sander Stuijk. 2008. Parametric Throughput Analysis of Synchronous Data Flow Graphs. In *2008 Design, Automation and Test in Europe (DATE'08)*. IEEE, 116–121. DOI: http://dx.doi.org/10.1109/DATE.2008.4484672

[11] Khaled R. Heloue, Sari Onaissi, and Farid N. Najm. 2012. Efficient Block-Based Parameterized Timing Analysis Covering All Potentially Critical Paths. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 31, 4 (April 2012), 472–484. DOI: http://dx.doi.org/10.1109/TCAD.2011.2175392

[12] Thomas Hune, Judi Romijn, Mariëlle Stoelinga, and Frits Vaandrager. 2001. Linear Parametric Model Checking of Timed Automata. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer, Berlin, Heidelberg, 189–203. DOI: http://dx.doi.org/10.1016/S1567-8326(02)00037-1

[13] Mitra Nasri and Björn B. Brandenburg. 2017. An Exact and Sustainable Analysis of Non-preemptive Scheduling. In *2017 IEEE Real-Time Systems Symposium (RTSS'17)*. IEEE, 12–23. DOI: http://dx.doi.org/10.1109/RTSS.2017.00009

[14] Mitra Nasri, Geoffrey Nelissen, and Björn B. Brandenburg. 2018. A Response-Time Analysis for Non-Preemptive Job Sets under Global Scheduling. In *2018 30th Euromicro Conference on Real-Time Systems (ECRTS)*, Vol. 106. 9–1.

[15] Krishnamurthy Subramani. 2001. Parametric Scheduling - Algorithms and Complexity. In *High Performance Computing (HiPC 2001)*. Springer, Berlin, Heidelberg, 36–46.

[16] Joost van Pinxten, Marc Geilen, Martijn Hendriks, and Twan Basten. 2018. Parametric Critical Path Analysis for Event Networks with Minimal and Maximal Time Lags. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)* 37, 11 (2018), 2697–2708. DOI: http://dx.doi.org/10.1109/TCAD.2018.2857360

[17] Umar Waqas, Marc Geilen, Jack Kandelaars, Lou Somers, Twan Basten, Sander Stuijk, Patrick Vestjens, and Henk Corporaal. 2015. A Re-entrant Flowshop Heuristic for Online Scheduling of the Paper Path in a Large Scale Printer. In *2015 Design, Automation and Test in Europe (DATE'15)*. IEEE, 573–578. DOI: http://dx.doi.org/10.7873/DATE.2015.0519

[18] Umar Waqas, Marc Geilen, Sander Stuijk, Joost van Pinxten, Twan Basten, Lou Somers, and Henk Corporaal. 2016. A Fast Estimator of Performance with Respect to the Design Parameters of Self Re-Entrant Flowshops. In *Euromicro Conference on Digital System Design (DSD'16)*. IEEE, 215–221. DOI: http://dx.doi.org/10.1109/DSD.2016.26

[19] Fengxiang Zhang and Alan Burns. 2009. Schedulability Analysis for Real-Time Systems with EDF Scheduling. *IEEE Trans. Comput.* 58, 9 (2009), 1250–1258. DOI: http://dx.doi.org/10.1109/TC.2009.58

Received April 2019; revised June 2019; accepted July 2019