Synthesizing Schedules Through Heuristics for Hard Real-Time Workflows

Thomas Kothmayr, Alfons Kemper
Chair for Database Systems,
Technische Universität München, Germany
{kothmayr, kemper}@in.tum.de

Andreas Scholz, Jörg Heuer
Corporate Technology, Siemens AG
{andreas.as.scholz, joerg.heuer}@siemens.com

***Abstract*—Task assignment and subsequent schedule synthesis in distributed real time systems is a problem that arises in many fields of industry, such as factory automation or the automotive or avionic sector. Monolithic and bus-based approaches, while easy to schedule in the aforementioned context, are giving way to more flexible hardware environments, either because of increased pressure for flexibility (factory automation) or hardware consolidation (automotive and avionics). This paper presents an approach for synthesizing schedules of a distributed hard real time workflow based on existing heuristics. This approach can find a solution for over 85% of all feasible tested system configurations while being orders of magnitude faster than an approach based on a satisfiability solver. We obtained these results by simulating over 1 million different workflows and system configurations.**

I. Introduction

Embedded systems in application domains like factory automation or the automotive and avionic industry have shifted towards complex hardware architectures of multiple networked processing units. When developing software for such a system, decomposing the application into a set of communicating tasks is a widespread practice [1], [2]. Synthesizing schedules for each processor in such a system is a problem that is hard in multiple respects: First, it is hard in terms of the real-time requirements, meaning that its timing constraints are derived from the physical world and any violations thereof may result in catastrophic outcomes for the system or its users. Second, it is a hard problem regarding its computational complexity. Even the simpler problem of scheduling tasks with release times and deadlines on a single processor is already a NP-hard problem [3]. Third, these systems are hard to develop from an engineering standpoint. Developers often have to integrate legacy systems, deal with shared communication media and lack tools that can be used to explore the implications of task placements or timing behavior early in the development process or that can be used for automated CPU and communication schedule synthesis in large problem instances.

A "distributed network of tasks" model for development of embedded real-time systems can be realized through a service-oriented architecture (SOA) [1] or model driven approaches [2]. The problem of schedule synthesis arises when the application is deployed, at the latest. Scheduling of tasks in real-time embedded systems is a well studied problem in literature [4] [5]. However, combined CPU and message scheduling is a computationally hard problem with ongoing research efforts. Finding a solution with exhaustive search through satisfiability (SAT) solvers [2] or branch-and-bound techniques [6] is an option to tackle NP-hard problems like this.

Our key argument in this paper is that the process of synthesizing both CPU and communication schedules can be achieved through a combination of deadline assignment heuristics [7] [8] and scheduling heuristics [9] in the large majority of cases. Since the heuristics can be completed in mere milliseconds, as opposed to the long computing times required by SAT solvers, they enable engineers to interactively explore the implications of placing tasks on different machines in a distributed real-time environment. Section II gives a brief overview of the related work before we outline our architecture and the chain of heuristics in Section III. At this point we rely on heuristics from the literature, which we explain in Section IV. The feasibility of our approach hinges on whether or not a combination of heuristics is able to generate valid schedules for a given hard real-time workflow on a given distributed hardware environment. This is extensively evaluated in Section V before we conclude the paper in Section VI.

The contributions of this paper are a detailed comparison of popular deadline assignment heuristics against satisfiability solvers over a wide range of task graph layouts. Furthermore, we demonstrate the feasibility of deriving local schedules from a global task-graph based on a combination heuristics.

II. Related Work

Voss and Schätz [2] use SMT-solvers to find both a suitable task placement and schedules for each device. We also formulated our scheduling problem for SMT-solvers to obtain a definitive answer for the solvability of our test cases. The SMT-Formulations are given in Section IV-C. In Section V, we evaluate a heuristics based approach that is several orders of magnitude faster. Voss and Schätz target systems with a bus-based communication channel that has a fixed bound on latency, whereas our model adopts a fine-grained communication model on the level of the individual TDMA-slots.

Our approach applies heuristics for local deadline assignment before using scheduling heuristics on the output of the former. Kao and Garcia-Molina [7] propose several heuristics for deadline assignment in soft real-time systems that are in widespread use. Our approach includes these heuristics as well. Their heuristics are presented for linear parallel task graphs, meaning tasks that follow a multi-step pipeline where each step can consist of several tasks than are executed in parallel. This differs from the general directed acyclic graphs (DAGs) used in our work and we present the modifications in Section IV-A.

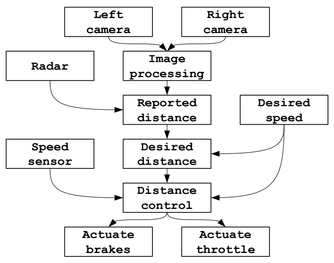
Di Natale and Stankovic [10] introduced a technique for assignment of intermediate deadlines in hard real-time systems based on analysis of the critical path the task graph with several metrics. Jonson and Shin [8] extended the slicing technique [10] with additional adaptive metrics. We implemented all of their heuristics in our prototype and evaluate them against each other.

Marinca et al. [11] proposes two new algorithms for deadline assignment and online admission control of real-time flows. In contrast to the works previously mentioned, it focuses on unicast flows from a sender to a destination instead of general DAGs, making it not directly applicable to our application model. Theoretically, a partitioning approach [12] could be used to separate the task graphs into parallel flows, incurring additional processing overhead and misspent slack during deadline assignment.

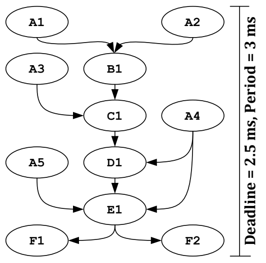
A different approach would be to synthesize a suitable network topology for a given set of schedules [13]. We view the network topology and slot-assignment as constant. This allows for integration with legacy applications running on the same network by marking the TDMA slots used by the legacy application as unavailable during planning. Existing processor schedules can be integrated by introducing dummy tasks with fixed deadlines and release times that block the CPU when it is assigned to the legacy schedule.

III. From Real-Time Workflows to CPU and Communication Schedules

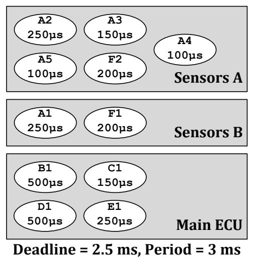
The use cases for our approach comprise distributed, embedded, hard real-time systems such as factory automation systems or applications in the automotive domain. We chose an adaptive cruise control system as a running example: In our example a 3D-vision system is used together with a radar system to measure the distance to leading vehicles and regulate acceleration and deceleration accordingly. The resulting workflow is shown in Figure 1a. This only covers the functional dependencies and modular decomposition of the system so far. The overall period and global deadline of the workflow are derived from physical requirements and / or control theory. Once the automation task has been modeled according to the modular decomposition from Figure 1a, these global deadlines are attached to the workflow, as shown in Figure 1b. Because the target area of embedded systems often requires platform specific implementations for each functional module and the modules themselves make use of sensors and actuators the assignment of jobs to machines can be viewed as a design-time decision performed by a skilled engineer. For a given assignment, the worst case execution time (WCET) of each task in the workflow can be measured or estimated. This leads to the situation shown in Figure 1c where the global period and deadline of the workflow are known and the machine placement and WCET of each workflow task have been determined. Afterwards, the heuristics take over: First, local timing constraints are derived from the global end-to-end deadline, taking into account the task placement on different machines and the TDMA-slot assignment in the network connecting the devices. The individual workflow tasks



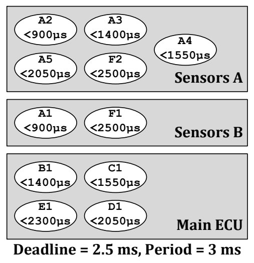
(a) Task in the embedded application



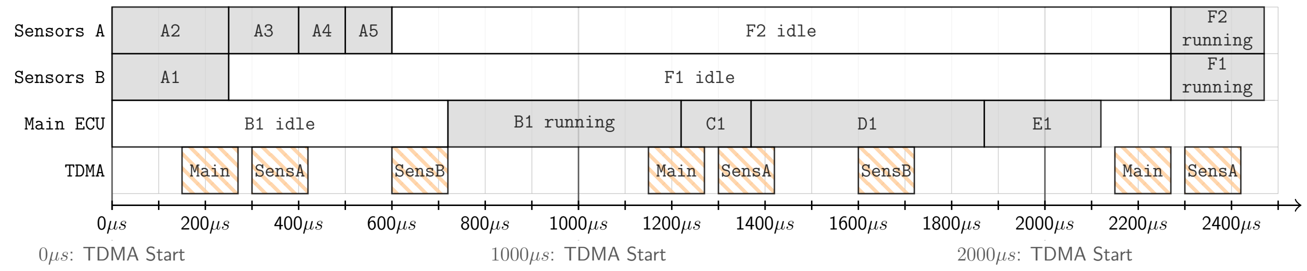
(b) Workflow layout and global timing constraints



(c) Task assignment and task WCET on assigned machines



(d) Local timing constraints as input for scheduling algorithm



(e) Simulation of a three machine schedule for the workflow in sub figures (a) - (d), generated by heuristics. Grey blocks denote task execution, white blocks show time spent waiting for messages from preceding tasks. Hatched blocks show the TDMA-slots.

Fig. 1: A complex workflow in an adaptive cruise control scenario

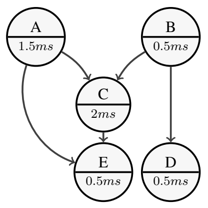


Fig. 2: Example graph

Node	ED	ED*	EQS	EQF	PD	Slice Pure	Slice Norm	Slice Adapt-G	Slice Adapt-L
A	7.00 ms	7.50 ms	3.67 ms	4.00 ms	3.33 ms	3.50 ms	3.75 ms	3.67 ms	3.77 ms
B	7.00 ms	7.00 ms	3.67 ms	4.00 ms	3.33 ms	3.50 ms	3.75 ms	3.67 ms	3.77 ms
C	9.50 ms	9.50 ms	7.83 ms	9.00 ms	6.67 ms	7.50 ms	8.75 ms	8.08 ms	7.83 ms
D	9.50 ms	10.00 ms	7.83 ms	9.00 ms	6.67 ms	10.00 ms	10.00 ms	10.00 ms	10.00 ms
E	10.00 ms	10.00 ms	10.00 ms	10.00 ms	10.00 ms	10.00 ms	10.00 ms	10.00 ms	10.00 ms

TABLE I: Deadlines assigned to the tasks of the example graph shown to the left. Overall workflow deadline was $10\,ms$, individual task deadlines are shown below the task name in the graph.

are automatically annotated with these constraints resulting in the deadlines shown in Figure 1d. What remains is essentially a distributed scheduling problem in which suitable start times for the jobs on each machine have to be determined to satisfy the global timing requirements. In the final step, the system is verified through discrete event simulation, the output of this simulation step for our example is shown in Figure 1e.

It is apparent that in communicating systems with tight timing requirements the network configuration plays an essential role in finding valid schedules. We cannot simply place an upper limit on the communication delay and add it to the WCET of each task as that would prevent us from finding a feasible schedule in Figure 1e and many other situations. Instead, timing information about each individual TDMA slot has to be considered during the schedule synthesis. As shown in our example, the slots may be distributed irregularly, for example when an application is sharing the same communication medium with a legacy application and was only granted the time slots that were previously unused. Another example would be communication protocols like Flexray which set aside a portion of each cycle for lower priority traffic, rendering the corresponding time slots unsuitable for hard real-time use. We therefore consider the available TDMA slots as an input to our schedule synthesis instead of trying to find a suitable slot assignment for a given schedule.

The problem is NP-hard, leaving two divergent approaches: exhaustive search through satisfiability solvers or employing heuristics in the hope that they will lead to a feasible result. Both approaches view the automation task as a network of communicating tasks which are annotated with global constraints. We show that a combination of deadline assignment and scheduling heuristics delivers feasible schedules with a high chance.

IV. Heuristics and SMT formulation

This section provides an intuitive, high level overview of the employed deadline assignment heuristics and formulation of the satisfiability problem. For readability reasons, detailed formal definitions for are postponed to Appendix A. Our previous work [14] has identified Pott's heuristic [9] as the best choice for subsequent scheduling. Non-preemptive Earliest Deadline First (EDF) is a good alternative.

In general the problem is modeled as a directed, acyclic graph of tasks which we call a workflow. The edges in the graph represent data flow between individual tasks. Each task is annotated with a worst case execution time (WCET). The range of possible start and completion times of the task can be constrained by setting a task release time and deadline. Each workflow is annotated with a global deadline and a period after which the workflow is repeated.

The overall goal is to find a suitable task ordering for a given assignment of tasks to machines. Tasks on a single machine cannot overlap or be preempted. Both local deadlines (on the task level) and release times as well as global deadlines (on the workflow level) have to be fulfilled by this task ordering. A task cannot be started before all transitively preceding tasks have been completed.

A. Simple Heuristics

The heuristic approach solves the task ordering problem in two steps. The workflow, with its global deadline and period, and a mapping of tasks to machines are the input for the heuristic approach. In the first step a deadline assignment heuristic attaches local deadline and release-time constraints to each task. Afterwards, Potts' heuristic generates a task ordering which is validated through simulation. This section describes the employed deadline assignment heuristics. The first set of heuristics we implemented was described by Kao and Garcia-Molina [7] for soft real-time tasks with serial-parallel dependencies. We adapt these heuristics to general DAGs by grouping tasks together by their levels, meaning the distance of a task from a root or leaf of the workflow graph. See Appendix A for more details. Figure 2 shows an example workflow with a deadline of $10\,ms$ and a total execution time of $5\,ms$. The deadlines generated by the heuristics described in this section are shown in Table I.

The simplest heuristic is called **Effective Deadline** (ED), which assigns a deadline to each task that is equal to the global deadline minus the execution time of all succeeding levels. It is greedy in the sense that it assigns all available slack to the root tasks of the DAG. Slack is defined as the time span between task release time and deadline minus the task's or level's WCET. This can be seen in the example table where all of the available slack ($5\,ms$) is assigned to tasks in the first level from the top (A and B) which means the following levels, comprising of tasks C and D in level 2 and task E in level 3, are assigned no slack at all. **Equal Slack** (EQS) tries to avoid ED's bias by assigning equal amounts of slack to the individual levels of a workflow. In the running example there are $5\,ms$ of total slack available which are distributed over 3 levels, leaving $1.66\,ms$ per level. These are added to the WCET of all tasks in the level plus the deadline of the previous level. **Equal Flexibility** (EQF) follows a similar strategy, but it scales the amount of slack assigned to a job proportionally to its length. Larger levels receive more slack in EQF. In the example, level 1 receives 40% of the slack because it makes up 40% of the total WCET in the workflow. The **Proportional Deadline** (PD) heuristic follows a different strategy. After dividing the graph into n levels it divides the global deadline into n parts of equal length which are assigned to each level. In the example graph with 3 levels, the second level from the top is assigned a deadline that corresponds to $\frac{2}{3}$ of the global deadline.

B. The Slicing Technique

Jonsson and Shin [8] presented a set of deadline assignment heuristics based on the slicing technique, which works by identifying the critical path through a DAG based on one of several path metrics. The global deadline is then distributed by assigning non-overlapping execution windows (slices) to the jobs on the critical path. Algorithm 1 in the Appendix shows the deadline distribution algorithm as defined in the original paper. In the following we will outline the different metrics used for finding the critical path in a DAG.

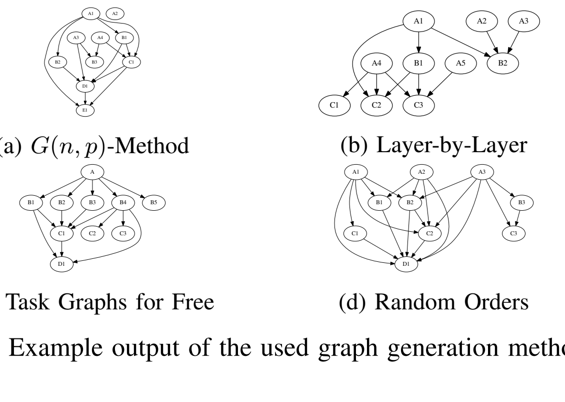
The **Pure** (Slice-Pure) metric is similar to the EQS heuristic in so far as it distributes the available slack equally between all jobs on the critical path, analogously, the **Normalized** (Slice-Norm) metric is similar to the EQF heuristic and scales the assigned slack with the task length. Table I demonstrates the outcome: The slicing technique will identify the path $A \prec C \prec E$ as the critical path in the workflow. The total slack available on this path is $6\,ms$, which results in $2\,ms$ of slack being allotted to tasks A, C, E. The next critical path in the graph is $B \prec D$ where $9\,ms$ of slack would be available. However, B must still finish before C, resulting in the same deadline ($3.5\,ms$) being assigned to B and all of the leftover slack being added to D. Slice-Norm works after the same principle but the assigned slack is scaled by the task length. The **Globally Adaptive** (Slice Adapt-G) metric only scales the task execution time if the length of a job is over a certain threshold. As in the original paper, we use the mean task execution time [8]. In the globally adaptive metric, the length of a task is then scaled by a constant factor that depends on the number of machines, on which the workflow is executed, and a global metric that measures the degree of parallelism in the workflow. In the example case, tasks A and C would be scaled by the factor 1.5 (for details see Appendix A or the original paper [8]), because they meet or exceed the mean execution time of $1\,ms$. Working with these virtual execution times, the total remaining slack to distribute along the path is $10\,ms - (1.5\,ms * 1.5) - (2\,ms * 1.5) - 0.5\,ms = 4.25\,ms$. Thus, the resulting deadline for task A is $(1.5\,ms * 1.5) + \frac{4.25\,ms}{3} \approx 3.67\,ms$. In contrast, the **Locally Adaptive** (Slice Adapt-L) metric scales the job execution length based on the local level of task parallelism instead of globally. For example, the degree of parallelism for A is 2 because there are no data dependencies with tasks B and D. As with the Adapt-G metric, jobs A and C are assigned a longer virtual execution time and the remaining slack is distributed along the critical path.

C. Formulations for SMT and MIP solvers

In addition to the heuristics, we also formulated the problem for a Satisfiability Modulo Theory (SMT) solver and a Mixed Integer Programming (MIP) solver. So far, we have neglected to model the network configuration and have ignored it as an input parameter. In tight control loops that are comparable in size to a TDMA-cycle, ignoring the network delay leads to overly pessimistic results. Since the solver should always find a solution, if one exists, it needs detailed network information for its scheduling decisions. Our model of a TDMA-configuration consists of a number of TDMA-Slots with a fixed slot start and end time. Each slot is assigned to exactly one machine and is repeated after one TDMA cycle period. A TDMA slot may carry messages from multiple tasks on one machine to receivers on all other machines, i.e. we assume a broadcast semantic. Tasks on the same machine communicate without time delay. The solver receives a workflow and a TDMA configuration as input. It then tries to find start times for each job in the workflow so that the global deadline is adhered to. Notice that this skips the deadline assignment step and directly tackles the problem of task ordering. Deadlines and release times for execution on real hardware are derived from this task ordering. Appendix A-C gives more detail on the TDMA model and our SMT and MIP constraints.

V. Evaluation

Since industrial use cases span a wide range of potential layouts of the resulting task graphs, we rely on synthetic benchmarks, based on several well-known graph generation methods [15]. The **Erdős-Rènyi** $G(n,p)$ method generates an unbiased DAG out of all possibilities, therefore our benchmarks contain this form of task graph weighted with 50%. The **Layer-by-Layer** method allows specifying the maximum depth of the graph and was developed specifically for validation of scheduling algorithms. It is contributing 20% to the overall number of test cases. Similarly, **Task Graphs for Free** (TGFF) is another method of generating task graphs for the validation of scheduling methods. It is also weighted with 20%. The **Random Orders** method generates a partial order (i.e. a DAG) by intersecting several total orders, which are constructed by shuffling the nodes of the graph. This method generates graphs with all transitive edges and is used for generating the last 10% of the test cases. Figure 3 shows examples of the graphs generated by these methods.



(a) $G(n,p)$-Method

(b) Layer-by-Layer

(c) Task Graphs for Free

(d) Random Orders

Fig. 3: Example output of the used graph generation methods

We generated a total of 1 160 000 *feasible* workflows, in the aforementioned ratio, with either 16, 32, 48 or 64 tasks which were randomly assigned to either 2, 4 or 8 machines, which were connected via TDMA. The feasibility of the workflows was verified through the SAT solvers. To avoid bias in the evaluation, the amount of feasible workflows is largely the same for each combination of machine count and task count, i.e. there are roughly 100 000 workflows for each combination. The only exception is the combination of 8 machines with 64 workflows where we could only generate 60 000 feasible workflows[1] due to time constraints. The workflows all have a common deadline and period of 10 ms, which is also the length of a TDMA round. TDMA slots of 120 μs length (the time needed to transmit 1500 bytes, which is the largest allowed UDP packet size, over 100 Mbit Ethernet) were assigned in round-robin style to the machines. The workflows were then run through the heuristics pipeline, described in Section III, to determine which percentage of feasible solutions a heuristic can find. We call this measure the *efficiency* of the heuristic. A value of 100% means that a heuristic was able to solve all of the same problems that the SAT-solvers[2] determined as feasible, a value of 50% means is solved halve of the problems. The efficiency of the SAT-approach naturally is 100%, therefore it is not shown in the performance plots below.

Figure Figure 4 a shows the performance of the heuristics plotted against the size of the workflow. In addition to the individual heuristics, we are also displaying the combined metric, meaning the percentage of workflows that was solved by at least one of the heuristics. The figures show that the slicing technique [8], does not exhibit a high efficiency for generalized workflows in our context. Its performance can only be considered competitive for workflows generated with the Layer-by-Layer or Random-Orders methods, as shown in Figure 4 b. Figure 4 c shows the performance of the heuristics for graphs with varying edge density. The x-axis in shows the percentage of the theoretically possible edges being present in a given workflow. 100% would mean a fully connected DAG. We see that the connectedness of a graph is no deciding factor in the overall performance of the heuristics. We attribute the slight increase in performance to biases stemming from the fact that the average number of machines and the average number of tasks is decreasing for feasible workflows with a high edge chance. A similar argument can be made for the height of a workflow, expressed as the number of levels in Figure 4 d. Here we see the slicing heuristics performing worse with increasing workflow length while the simple heuristics [7] are showing stable performance. This graph is again slightly biased through a decreasing average number of machines but this should be counterbalanced through the increased average number of tasks in deeper workflows. Figure 4 e shows that the number of machines on which a workflow is allocated has an impact on the efficiency of the heuristic approach. This stems from the fact that more network communication is required while the average wait time for the next TDMA-slot is increasing. This can be mitigated by playing tasks in such a manner that the network communication is minimized, but we assigned tasks to machines in a random manner in this experiment. Figure 4 e also demonstrates that a combination of different cheap heuristics is beneficial. For the two machine case, it could be argued that a single efficient heuristic should be enough. But it is apparent, in the eight machine case,

[1]There are only 10 000 instead of 50 000 feasible workflows of the Erdős-Rènyi $G(n, p)$ variant.

[2]Microsoft Z3, version 4.3 (http://z3.codeplex.com/) as SMT solver and Gurobi, version 6.0 (http://www.gurobi.com/) as MIP solver

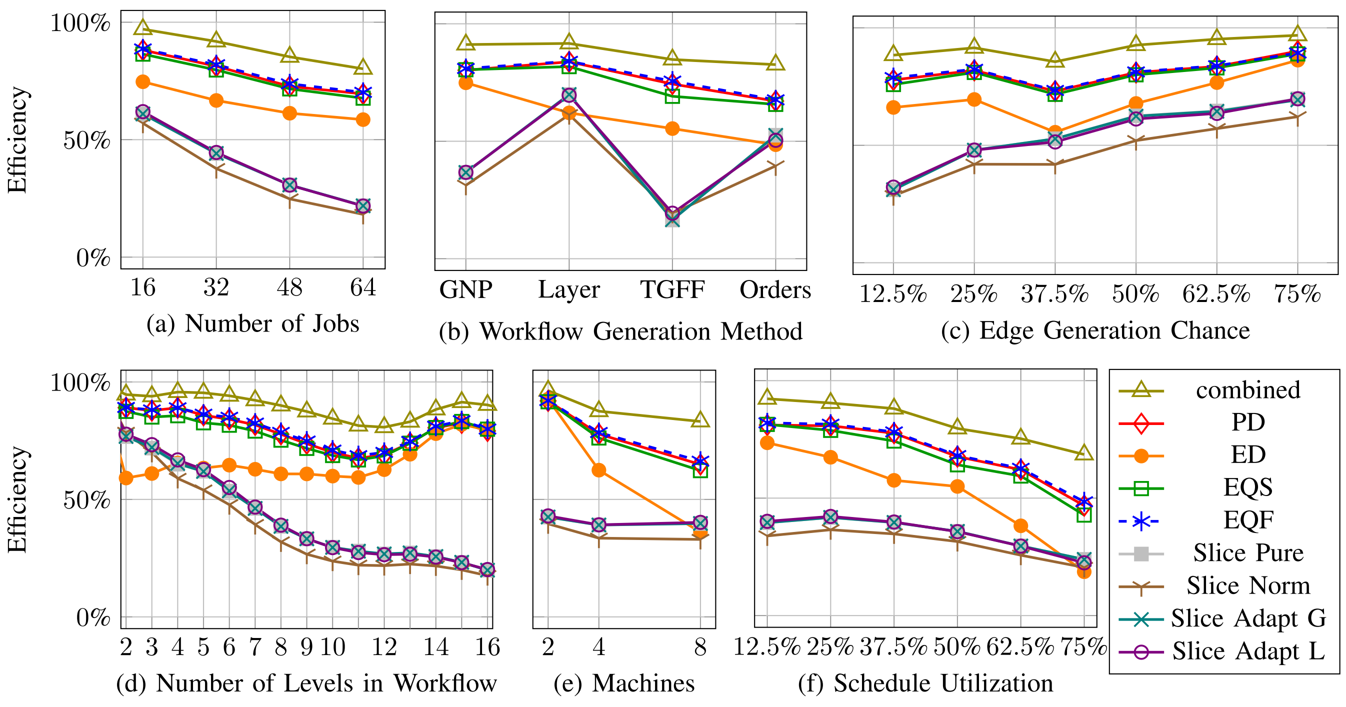


Fig. 4: Evaluation of deadline assignment heuristics

Workflow Type	EQS	EQF	PD	Slice Pure	Slice Norm	Slice Adapt-G	Slice Adapt-L	Gurobi	Z3
16 feasible	0.06 ms	0.06 ms	0.06 ms	0.19 ms	0.19 ms	0.20 ms	0.21 ms	20.34 ms	0.27 s
16 infeasible	0.07 ms	0.07 ms	0.07 ms	0.20 ms	0.17 ms	0.21 ms	0.22 ms	17.74 ms	0.07 s
32 feasible	0.14 ms	0.14 ms	0.14 ms	0.60 ms	0.54 ms	0.64 ms	0.75 ms	154.56 ms	18.99 s
32 infeasible	0.16 ms	0.16 ms	0.16 ms	0.70 ms	0.62 ms	0.74 ms	0.84 ms	68.96 ms	24.99 s
64 feasible	0.39 ms	0.38 ms	0.39 ms	3.95 ms	3.70 ms	4.19 ms	5.06 ms	2,086.95 ms	– s
64 infeasible	0.47 ms	0.45 ms	0.48 ms	26.40 ms	24.85 ms	27.48 ms	30.52 ms	572.97 ms	– s

TABLE II: Geometric means of the heuristics' run time. The figures for the SMT-solver (Z3) are from a comparable, but not identical, set of workflows.

that the overall efficiency of the heuristics is not dropping as steeply as the efficiency of the individual heuristics. The overall schedule utilization, defined as the sum over all task processing times divided by the global workflow deadline and the number of machines, is also a influencing factor in the heuristics' efficiency. This is shown in Figure 4 f. The actual decrease should be a bit steeper as the figure is biased towards a smaller number of machines with a larger schedule utilization while the workflow size is stable.

In general, the EQF heuristic performs best. The simple approach of the PD heuristic, which results in a topological sorting of the tasks in the workflow after scheduling, performs second best with EQS following by a small margin. ED follows by a more visible margin before the slicing techniques follow by a wider gap. Out of these, the locally and globally adaptive methods as well as the pure metric perform best, with a little gap to the normalized slicing method. The combined approach naturally performs best, but in which order should the heuristics be applied? Figure 5 shows the percentage of unique solutions which were not found by any other heuristic. Only 15% of the solutions could only be found via SAT-solver. It is our opinion that the heuristics should be applied in sequence of their uniqueness, meaning that the second best heuristic (PD) should be tried before the best heuristic (EQF), because it offers comparable performance but has a higher chance to find an unique solution. Similarly, the globally adaptive slicing technique is almost completely subsumed by the locally adaptive technique.

If the goal is an interactive system to support developers in their decision making, as stated in the introduction, the run time of an approach based on a SMT-solver is prohibitive. Table II shows the geometric mean of the run time of all presented approaches measured over a separate data set with 3 000 workflows with the same mix as described before. There are two orders of magnitude between the approaches based on heuristics and the SMT-solver (Z3). We cannot present average run time figures for workflows with more then 32

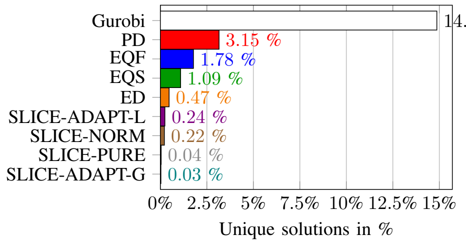


Fig. 5: Percentage of solutions not found by other heuristics

jobs because we had to cap the execution time of the SMT-solver after 30 minutes to find a sufficiently large base of feasible test cases for the heuristics. The MIP-solver (Gurobi) is considerably faster. However, Figure 6 shows that there is a large spread in the run time distribution the MIP-solver. The PD heuristic has relatively stable runtime characteristics because it is not influenced by the number of edges in the workflow. The slicing heuristic, on the other hand, is largely influenced by the workflow structure. This cannot yet be seen for small workflows, but larger ones lead to a large spread in runtime, to the point that the MIP-solver can even be faster, in the best case, than the heuristic, in the worst case.

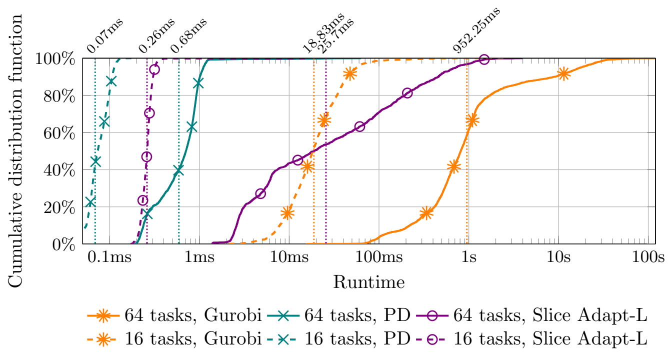


Fig. 6: Cumulative run time distribution of selected heuristics and the MIP solver over 3 000 separate workflows. Vertical lines denote geometric means.

VI. Conclusions and Future Work

In this paper we presented a heuristics based approach for automated schedule synthesis for distributed hard real-time systems. Through extensive evaluation, we have shown that an approach based on a combination of several heuristics has a very high efficiency – defined as the capacity to find a feasible system wide schedule if one exists – and was able to solve over 85% of all feasible test cases. The heuristics proved to be, on average, orders of magnitude faster than an approach based on an MIP satisfiability solver. We have ruled out SMT-solvers as suitable tools, because they are much slower than a state of the art MIP solver. Heuristics finish in milliseconds where the solver may need several minutes. For larger problem instances the run time of a MIP solver becomes prohibitive, but the success chances of the heuristics are reduced. However, we have only evaluated general heuristics from the soft real-time domain so far. In future work we plan to make use of the explicit network timing information by developing our own network aware heuristics.

Appendix A
Formal Definitions

The appendix formally defines our implementation of well known deadline assignment heuristics [7], [8]. Section IV provides a high-level overview of the heuristics explained here. The basis is a workflow $\mathfrak{W} = \langle \mathcal{J}, \mathcal{G}, D_{\mathfrak{W}}, P_{\mathfrak{W}} \rangle$ with a job set $\mathcal{J}$, the graph $\mathcal{G}$ formed by the jobs, a global deadline $D_{\mathfrak{W}}$, and a period $P_{\mathfrak{W}} \geq D_{\mathfrak{W}}$ after which the execution of the job set is repeated. The job set $\mathcal{J} = \{j_0, j_1, \ldots, j_n\}$, contains the individual jobs j_i. Each job has a (worst case) execution time written as $|j_i|$, a start time $j_i.S$ and completion time $j_i.C = j_i.S + |j_i|$. The release time $j_i.R$ and deadline $j_i.D$ constitute constraints on the earliest start time and latest completion time, respectively. The graph $\mathcal{G} = \langle \mathcal{J}, \prec \rangle$ places constraints on the task ordering. A precedence relation between two jobs j_i and j_j is written as $j_i \prec j_j$ if j_i is a direct predecessor of j_j and as $j_i \prec\!\!\!\prec j_j$ if j_i is a transitive predecessor of j_j, e.g. $j_i \prec \ldots \prec j_j$. $\mathcal{G}$ forms a directed, acyclic graph. Each job is assigned to a machine $m_i \in \mathcal{M}$ by the function $\mu_j(j_i) : \mathcal{J} \to \mathcal{M}$. μ_j is defined prior to the deadline assignment problem.

Let $in(j_j)$ be the in-degree of a job, i.e. the number of direct predecessors $j_i \prec j_j$ it has and $out(j_i)$ the out-degree counting the number of direct successors. $\mathcal{R} = \{j_i \in \mathcal{J} | in(j_i) = 0\}$ is then the set of jobs with in-degree 0, i.e. the roots of the DAG. $\mathcal{L} = \{j_i \in \mathcal{J} | out(j_i) = 0\}$ constitutes the leafs of the DAG. Additionally, let $\lceil j_i \prec\!\!\!\prec j_j \rceil$ be the length of the longest path from j_i to j_j. The length of $j_i \not\prec\!\!\!\prec j_j$ is ∞, the length of a job j_i to itself is 0. We then define the top-level $j_i.\hat{L}$ of job j_i as $min(\lceil j_r \prec\!\!\!\prec j_i \rceil)$ for $j_r \in \mathcal{R}$. Analogously the bottom-level $j_i.L^b$ of job j_i is $min(\lceil j_r \prec\!\!\!\prec j_i \rceil)$ for $j_l \in \mathcal{L}$. The expression $\hat{\mathcal{J}}_{\odot} = \{j_j | j_i, j_j \in \mathcal{J} \wedge j_j.\hat{L} \odot j_i.\hat{L}\}$ describes a set of jobs which fulfill a condition on their level $\hat{L}$ where $\odot$ can be any binary operator, e.g. $\hat{\mathcal{J}}_{>} = \{j_j | j_i, j_j \in \mathcal{J} \wedge j_j.\hat{L} > j_i.\hat{L}\}$.

A. Simple Heuristics

Effective Deadline (ED) heuristic:

$$j_i.D = D_{\mathfrak{W}} - \sum_{j \in \hat{\mathcal{J}}_{>}} |j| \tag{1}$$

Equal Slack (EQS) heuristic:

$$j_i.D = j_i.R + \sum_{j \in \hat{\mathcal{J}}_{=}} |j| + \frac{D_{\mathfrak{W}} - j_i.R - \sum_{j \in \hat{\mathcal{J}}_{\geq}} |j|}{max(\hat{L}) - \hat{L}_i + 1} \tag{2}$$

$$j_i.R = max(0, j_p.D) \text{ for } j_p \in \hat{\mathcal{J}}_{<} \tag{3}$$

Equal Flexibility (EQF) heuristic:

$$j_i.D = j_i.R + \sum_{j \in \hat{\mathcal{J}}_{=}} |j| + \frac{\left(D_{\mathfrak{W}} - j_i.R - \sum_{j \in \hat{\mathcal{J}}_{\geq}} |j|\right) * \sum_{j \in \hat{\mathcal{J}}_{=}} |j|}{\sum_{j \in \hat{\mathcal{J}}_{\geq}} |j|} \tag{4}$$

$$j_i.R = max(0, j_p.D) \text{ for } j_p \in \hat{\mathcal{J}}_{<} \tag{5}$$

Proportional Deadline (PD) heuristic:

$$j_i.D = \frac{1 + L_i^b}{1 + max(L^b)} * D_{\mathfrak{W}} \tag{6}$$

B. Slicing technique

Algorithm 1 Slicing algorithm by Jonsson and Shin [8]

```
1: function SLICING
2:     $\mathcal{J}_{working} \leftarrow \mathcal{J}$
3:     while $\mathcal{J}_{working} \neq \emptyset$ do
4:         Find critical path $\phi$ in $\mathcal{G}$ that minimizes metric R
5:         Distribute deadline $D_\phi$ of $\phi$ to all jobs in $\phi$
6:         for all $j_i \in \phi$ do
7:             for all $j_p : j_p \prec j_i$ do
8:                 $j_p.D = j_i.R$
9:             for all $j_s : j_i \prec j_s$ do
10:                $j_s.R = j_i.D$
11:        $\mathcal{J}_{working} \leftarrow \mathcal{J}_{working}$ without $\phi$
```

In the following we will outline different metrics R used for finding ϕ in Algorithm 1. $|\phi|$ is the number of jobs in the path ϕ.

Slice-Pure metric:

$$R_{Pure} = \left(D_\phi - \sum_{j \in \phi} |j| \right) / |\phi| \tag{7}$$

$$j_i.D = j_p.D + |j_i| + R_{Pure} \text{ for } j_p \in \phi : j_p \prec j_i \tag{8}$$

Slice-Norm metric:

$$R_{Norm} = \frac{D_\phi - \sum_{j \in \phi} |j|}{\sum_{j \in \phi} |j|} \tag{9}$$

$$j_i.D = j_p.D + |j_i| * (1 + R_{Norm}) \text{ for } j_p \in \phi : j_p \prec j_i \tag{10}$$

Globally adaptive slicing (Slice Adapt-G) metric:

$$|j_i^{virt}| = \begin{cases} |j_i| & \text{if} |j_i| < thres \\ |j_i| * (1 + k_G * \xi / |\mathcal{M}|) & \text{if} |j_i| \geq thres \end{cases} \tag{11}$$

$$j_i.D = j_p.D + |j_i^{virt}| \text{ for } j_p \in \phi : j_p \prec j_i \tag{12}$$

Locally adaptive slicing (Slice Adapt-L) metric:

$$|j_i^{virt}| = \begin{cases} |j_i| & \text{if} |j_i| < thres \\ |j_i| * (1 + k_L * |\Psi_i| / |\mathcal{M}|) & \text{if} |j_i| \geq thres \end{cases} \tag{13}$$

$$j_i.D = j_p.D + |j_i^{virt}| \text{ for } j_p \in \phi : j_p \prec j_i \tag{14}$$

C. SMT Formulation

The TDMA configuration is represented as $\mathfrak{T} = \langle \mathcal{T}, \mu_t, P_{\mathfrak{T}} \rangle$, where $\mathcal{T} = \{t_0, \ldots, t_n\}$ is a set of TDMA slots with length $|t_i|$, start time $t_i.S$ and completion time $t_i.C = t_i.S + |t_i|$. Each slot is assigned to a single machine by the function $\mu_t(t_i) : \mathcal{T} \to \mathcal{M}$. The TDMA cycle is repeated after period $P_{\mathfrak{T}}$. Jobs in $\mathfrak{W}$ either communicate locally if they are on the same machine, or they send a message in a TDMA-slot that is assigned to the same machine.

The SMT solver receives workflow $\mathfrak{W}$ and TDMA configuration $\mathfrak{T}$ as input and finds start times $j_i.S$ for each job $j \in \mathfrak{W}.\mathcal{J}$ so that the global deadline $D_{\mathfrak{W}}$ is adhered to. The constraints for the SMT-solver are: Equation 15, which requires that all task start times have to be non negative, completion times have to be smaller than the global deadline and the completion time must be the start time plus the job execution time.

$$\forall j_i \in \mathcal{J} : j_i.S \geq 0 \wedge j_i.C \leq D_{\mathfrak{W}} \wedge j_i.C = S_i.j + |j_i| \quad (15)$$

Equation 16, which requires that all tasks on the same machine do not overlap.

$$\begin{aligned} \forall j_i, j_j \in \mathcal{J} : j_i.C \geq j_j.S \vee j_j.C \geq j_i.S \\ \vee\ j_i = j_j \vee \mu_j(j_i) \neq \mu_j(j_j) \end{aligned} \quad (16)$$

And the final set of equations expresses the precedence constraints. Equation 17 expresses that two subsequent jobs on the same machine do communicate locally and not via TDMA while Equation 18 states that subsequent jobs on different machines have to communicate via a TDMA slot assigned to the first job's machine. Equation 19 requires that either the local or global precedence constraints are fulfilled.

$$\begin{aligned} \textit{prec-local}(j_i, j_j) : \mu_j(j_i) = \mu_j(j_j) \\ \wedge\ \tau(j_i) = \emptyset \wedge\ j_j.S \geq j_i.C \end{aligned} \quad (17)$$

$$\begin{aligned} \textit{prec-global}(j_i, j_j) : \mu_j(j_i) \neq \mu_j(j_j) \\ \wedge\ \mu_t(\tau(j_i)) = \mu_j(j_i) \\ \wedge\ j_i.C \leq \tau(j_i).S \\ \wedge\ j_j.S \geq \tau(j_i).C \end{aligned} \quad (18)$$

$$\begin{aligned} \textit{prec} : \forall j_i, j_j \in \mathcal{J} : j_i \prec j_j \implies \textit{prec-local}(j_i, j_j) \\ \vee\ \ \textit{prec-global}(j_i, j_j) \end{aligned} \quad (19)$$

D. LP Formulation

The Linear Program formulation is similar to the SMT approach and follows a formulation based on completion time variables by Queyranne [16] for scheduling task graphs with release times and deadlines. The LP variables are thus $C_i = j_i.C$ for all jobs $j \in \mathcal{J}$ and binary variables $y_{ij} = 1$ that denote that job j_i is scheduled before job j_j. Note that $j_i \prec\!\!\!\prec j_j \implies y_{ij} = 1$ but $y_{ij} = 1 \not\Longrightarrow j_i \prec\!\!\!\prec j_j$, i.e y_{ij} only denotes precedence in the schedule produced by the solver. Additionally we introduce the binary variables $u_{ik} = 1$ that indicates that job j_i is sending its data via TDMA-slot t_k. Additionally the formulation makes use of the upper bound $M = \Sigma|j|$, meaning the sum of all processing times.

Equation 20 and 21 define the value ranges for the completion time variables and the precedence variables.

$$\forall j_i \in \mathcal{J} : C_i \geq |j_i| \wedge C_i \leq D_{\mathfrak{W}} \quad (20)$$

$$\forall j_i, j_j \in \mathcal{J} : y_{ij} \in \{0, 1\} \quad (21)$$

The objective of the LP solver is to minimize the sum of the completion times: $min(\Sigma C_j)$. The constraints in Equation 22 prohibit two tasks from overlapping. This means that for every two tasks on a machine, one of them needs to finish before the start of the other. Two tasks on different machines may overlap at will and thus there are no restrictions placed on their corresponding y-variables.

$$\begin{aligned} \forall j_i, j_j \in \mathcal{J} :\ \mu_j(j_i) = \mu_j(j_j) \implies \\ C_i + |j_j| \leq C_j + M * y_{ji} \\ \wedge\ C_j + |j_i| \leq C_i + M * (1 - y_{ji}) \end{aligned} \quad (22)$$

The last set of constraints expresses communication over the TDMA-network. If two tasks are in a direct predecessor relation, but not running on the same machine, they need to communicate over the network. This means firstly that the preceding job j_i has to finish before an TDMA slot assigned to its machine. This is implied in the first two (in-)equalities in Equation 23. Lastly, the receiving job j_j can only start after the slot, which transports the message from j_i, has ended.

$$\begin{aligned} \forall j_i, j_j \in \mathcal{J} : j_i \prec j_j \wedge \mu_j(j_i) \neq \mu_j(j_j) \implies \\ \sum_{t_k \in \mathcal{T}} u_{ik} = 1 \\ \wedge\ C_i \leq \sum_{t_k \in \mathcal{T}} u_{ik} * t_k.S \\ \wedge\ C_j - |j_i| \geq \sum_{t_k \in \mathcal{T}} u_{jk} * t_k.C \end{aligned} \quad (23)$$

References

[1] F. Jammes, B. Bony, P. Nappey, A. Colombo, J. Delsing, J. Eliasson, R. Kyusakov, S. Karnouskos, P. Stluka, and M. Till, "Technologies for SOA-based distributed large scale process monitoring and control systems," in *38th Annual Conference of the IEEE Industrial Electronics Society (IECON)*, 2012.

[2] S. Voss and B. Schatz, "Deployment and scheduling synthesis for mixed-critical shared-memory applications," in *20th IEEE International Conference and Workshops on the Engineering of Computer Based Systems (ECBS)*, 2013.

[3] D. S. Johnson and M. Garey, *Computers and Intractability: A guide to the theory of NP-completeness*. Freeman&Co, San Francisco, 1979.

[4] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok, "Real time scheduling theory: A historical perspective," *Real-Time Systems*, vol. 28, no. 2-3, 2004.

[5] R. I. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys (CSUR)*, vol. 43, no. 4, 2011.

[6] D.-T. Peng, K. Shin, and T. Abdelzaher, "Assignment and scheduling communicating periodic tasks in distributed real-time systems," *Software Engineering, IEEE Transactions on*, vol. 23, no. 12, 1997.

[7] B. Kao and H. Garcia-Molina, "Deadline assignment in a distributed soft real-time system," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 8, no. 12, 1997.

[8] J. Jonsson and K. G. Shin, "Robust adaptive metrics for deadline assignment in distributed hard real-time systems," *Real-Time Systems*, vol. 23, no. 3, 2002.

[9] C. Potts, "Analysis of a heuristic for one machine sequencing with release dates and delivery times," *Operations Research*, no. 6, 1980.

[10] M. Di Natale and J. A. Stankovic, "Dynamic end-to-end guarantees in distributed real time systems," in *15th IEEE Real-Time Systems Symposium (RTSS)*, 1994.

[11] D. Marinca, P. Minet, and L. George, "Analysis of deadline assignment methods in distributed real-time systems," *Computer Communications*, vol. 27, no. 15, 2004.

[12] G. Buttazzo, E. Bini, and Y. Wu, "Partitioning real-time applications over multicore reservations," *Transactions on Industrial Informatics*, vol. 7, no. 2, 2011.

[13] N. Kandasamy, J. P. Hayes, and B. T. Murray, "Dependable communication synthesis for distributed embedded systems," *Reliability Engineering & System Safety*, vol. 89, no. 1, 2005.

[14] T. Kothmayr, A. Kemper, A. Scholz, and J. Heuer, "Machine ballets dont need conductors: Towards scheduling based service choreographies in a real-time SOA for industrial automation," in *8th IEEE International Workshop on Service-Oriented Cyber-Physical Systems in Converging Networked Environments (SOCNE)*, 2014.

[15] D. Cordeiro, G. Mounié, S. Perarnau, D. Trystram, J.-M. Vincent, and F. Wagner, "Random graph generation for scheduling simulations," in *3rd ICST International Conference on Simulation Tools and Techniques (SIMUTools)*, 2010.

[16] M. Queyranne, "Structure of a simple scheduling polyhedron," *Mathematical Programming*, vol. 58, no. 1-3, 1993.