Real-time Avionics Optimization

Mathematische Optimierung von Echtzeitsystemen im Flugzeugdesign

Friedrich Eisenbrand, Martin Niemeier, École Polytechnique Fédérale de Lausanne, Switzerland,
Martin Skutella, José Verschae, Andreas Wiese, Technische Universität Berlin

Summary We report on the solution of a difficult optimization problem which arises in avionics industry. When constructing the on-board controlling-network of an airplane, the engineers need to solve a computationally highly complex problem. The goal is to assign periodic tasks to the processors on the plane and define a schedule for each processor. Current state-of-the-art approaches to tackle the problem are by far not powerful enough to solve instances of real-world size. With the help of the powerful algorithm engineering paradigm we analyzed the mathematical properties of the scheduling problem and designed sophisticated software based on the structural insights. We were able to design a model that outperformed current state-of-the-art approaches by several orders of magnitude. In particular, we could solve industrial size real-world instances to optimality. Our methods lead, for the first time, to an industrial strength tool to schedule aircraft sized instances. ►►► **Zusammenfassung** In modernen Flugzeugen übernimmt der Bordcomputer einen wichtigen Teil der Flugkontrolle. Dementsprechend ist es wichtig, dass der Computer zu jeder Zeit exakt laut Spezifikation arbeitet. Einen Bordcomputer zu entwickeln ist eine hochgradig komplexe Aufgabe. Unter anderem muss folgenes Problem gelöst werden. Periodisch wiederkehrende Tasks (Programmabläufe und Berechnungen) müssen auf sinnvolle Weise den im Flugzeug vorhandenen Recheneinheiten zugeteilt werden. Für jede Recheneinheit muss dann ein gültiger Schedule berechnet werden, d. h. es muss sichergestellt werden, dass jede Task die ihr laut Spezifikation zustehende Rechenintervalle nutzen kann, ohne in Konflikt mit anderen Tasks derselben Recheneinheit zu treten. Allein dieses Problem, eine gültige Zuteilung mit Schedules zu berechnen, ist selbst mit Hilfe von modernen Rechnern sehr schwierig zu lösen. Alle bekannten algorithmischen Ansätze sind zu ineffizient, um Systeme industrieller Komplexität zu berechnen. Wir beschreiben, wie sich ein tieferes Verständnis der mathematischen Struktur des Problems nutzen lässt, um ein vollkommen neuartiges mathematisches Modell des Problems abzuleiten. Im Rahmen des „Algorithm-Engineering"-Paradigmas haben wir das neue Modell implementiert. Das daraus resultierende Programm ist um Größenordnungen schneller als alle bisher bekannten Ansätze. Insbesondere können wir erstmals Instanzen lösen, die ein gesamtes Flugzeug beschreiben, wodurch unser Programm zu einem industrietauglichen Werkzeug wird.

Keywords F [Theory of Computation]; G [Mathematics of Computing]; J [Computer Applications]; real-time scheduling, algorithm engineering, avionics, on-board computer, integer programming, IP-model, mathematical modelling ►►► **Schlagwörter** Real-time Scheduling, Echtzeitsysteme, Algorithm Engineering, Flugzeugbau, Bordcomputer, Integer Programming, IP-Modell, mathematische Modellierung

1 Introduction

Modern airplanes use sophisticated computer systems for steering and controlling the plane, communicating with the air-traffic controller, and many other tasks. For example, fly-by-wire control systems or autopilots would not be possible without the help of sophisticated on-board computing networks. In order to guarantee flight safety, the network must operate according to its specifications at all times. While for a home computer a small discrepancy between expected and actual behavior is perfectly acceptable, for a network that controls an aircraft it is not. Even a slight delay in the execution of a program might result in serious and even fatal problems.

On a home computer, several *tasks* share computing resources like the processors, hard-disk, memory or I/O-devices. Managing the tasks and assigning resources to them is the primary objective of the *operating system*. One important challenge for the operating system is the fact that tasks with a variety of resource requests arrive and that these resources should be distributed in a fair way among the tasks. In such a setting *dynamic effects* are unavoidable. The following is an example of such a dynamic effect. Everyone watching movies on a PC from time to time has experienced short stalls of the movie. These are usually due to other tasks that require a resource that is also used by the software playing the movie.

Such effects are now responsible for the following problem: The *worst case execution time* of a task depends on other tasks on the system in a dynamic way. Estimating worst-case execution times under circumstances as described above is very difficult. And even worse, if a software gives an estimate, can you be sure that this estimate is correct? Is there a *certificate of correctness* of the estimate?

Such dynamic effects are unacceptable in safety-critical applications such as fly-by-wire. This is why such safety-critical systems need to pass a *certification process* that *proves* its validity. To be compliant with the official safety requirements, the scheduling of tasks on the on-board network of an airplane has to adhere to a strict and static protocol. The distribution of tasks under this protocol, where resource and communication constraints have to be satisfied is a computationally very challenging problem that, however, needs to be frequently solved by avionics companies. The design of an efficient solver for this task is the focus of our project.

In the following, we describe the static protocol and provide an illustrating example. First of all, control tasks should ideally be run continuously over time. This however would result in blocking a resource, like a processor, by one single task. Therefore, this continuous activity is approximated by running tasks *periodically*. The formal model is as follows. A *task* is a tuple (c_i, p_i), where c_i is the amount of *time* (milliseconds) that is required by the task i and p_i is its *period*. Every integer multiple of the period the task has to be run without interruption. Those values are fixed. However, the scheduler has the freedom to assign the first time when the task has to be run. This *offset* is between 0 and p_i. Suppose we have only one processor. We define a schedule by computing this offset a_i for each task. Once the offset has been specified, the slots on which this task will be using the processor are irrevocably fixed. No two tasks are allowed to use the same processor simultaneously. The goal is to assign the given tasks to the processors and to define a schedule for each processor while obeying the above constraints.

Figure 1 shows an example with two schedules for three tasks, two of them depicted with stripes and the third one dotted. The upper part shows an infeasible schedule for them. It is infeasible because at the position denoted by the box the two striped tasks need the processor simultaneously. This is not allowed. However, the schedule shown in the lower part is feasible. Notice that eventually the schedule repeats itself and hence only a limited part (the part shown in the picture) is of interest.

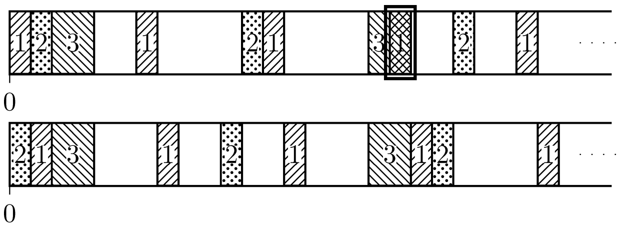

Figure 1 An infeasible and a feasible schedule.

On an airplane one has dozens of such processors that communicate with each other on a *network*. The complete problem is then twofold.

- Assign each task to a processor, and
- Assign an offset to each task.

At the same time, memory and bandwidth constraints have to be met. All-together this problem is highly challenging from a computational complexity viewpoint. This means that *modeling* the problem in a language interpretable by state-of-the-art solvers is not a problem. However, the state-of-the-art solvers are not able to *solve* the model. Our contribution to this field is a new algorithm based on *integer and linear programming* that tackles industrial-sized instances. More precisely, we solved this problem being guided by the paradigm of algorithm engineering.

First, we analyzed the problem from a mathematical perspective and deduced structural properties that allowed us to rule out many sub-optimal assignments efficiently. Based on this, we designed a suitable formulation and implemented a prototype for solving large avionics instances. Finally, we did experiments for evaluating the current methods. In our iterations, the quality of the developed software improved step by step. The first approach was a purely straightforward implementation and was not even capable of solving instances of very modest size. Deriving a mathematical lemma allowed us to design a formulation which performed better. However, it was still unable to solve instances of real-world size. Having a closer look on real-world instances from our industrial partner, we identified two important structural properties. These properties allowed us to prove that there are always solutions which are well-structured (we elaborate on this in Sect. 2). By focusing on such well-behaved solutions, we designed a completely new formulation. It is much more sophisticated than the previous two approaches and performs better by several orders of magnitude. In particular, it was able to solve industrial size instances to optimality.

In this article we give an overview of the computational results. More detailed and more formal explanations can be found in [1; 2].

Integer Programming

All our problem formulations are expressed as integer linear programs (IP). Linear programming is a mathematical model to optimize a linear objective function subject to a set of linear *constraints*. Given a set of *variables*, a linear constraint is an inequality where the right hand side is simply a constant number, and the left hand side consists of the sum of a subset of variables, each of them having a constant number as coefficient. The objective function is a sum similar to the left hand side of the linear constraints. Restricting to *linear* constraints is crucial for solving those models efficiently. In an integer linear program, some of the variables are additionally constrained to take only integer values, maybe even only zero or one.

Our methods are based on integer programming models. To give an example, variables can model decisions like whether a certain task is assigned to a certain processor or not. Consider two binary variables $x_{i,j}, x_{i,j'}$ which model whether some task τ_i is assigned to two available processors j and j'. Setting $x_{i,j} := 1$ and $x_{i,j'} := 0$ models that τ_i is assigned to processor j whereas $x_{i,j} := 0$ and $x_{i,j'} := 1$ models that τ_i is assigned to processor j'. To model the logical constraint that τ_i is assigned to at most one of the two processors we can add the linear constraint

$$x_{i,j} + x_{i,j'} \leq 1 \,.$$

In fact, all logical constraints of our scheduling problem can be modeled by linear constraints as we explain later. The linear objective function will always be the number of used processors which we seek to minimize. Then computing a solution to the IP, i. e., an assignment of values to our variables such that all linear constraints are satisfied, immediately gives a solution to our scheduling problem.

There are several good reasons to use integer programming. First of all, there is very sophisticated software available to solve integer programs. In particular, the solvers compute solutions which are provably optimal. Also it is very easy to incorporate additional constraints like memory limitations which arise in addition to the core problem by adding new linear constraints to the formulation. Alternatively, one could design heuristics which compute some solution but do not give any quality guarantee. However, in our case such methods turned out not to cope well with real-world instances from the industry [1; 2].

Although there are sophisticated integer program solvers available like CPLEX or GUROBI, solving an IP can be a difficult and computationally challenging task even for state-of-the-art computers. Whether an IP can be solved or not highly depends on a good problem formulation. Obtaining strong formulations requires a thorough mathematical understanding of the underlying structure of the problem as we will see in the sequel.

2 Three Problem Formulations

As mentioned above, we undertook several iterations of the algorithm engineering cycle. For all our formulations here we discuss only the case that there is one single processor for which a schedule has to be found. This can easily be generalized to a setting with multiple processors [1].

Time-Indexed Formulation

Our first formulation was a straightforward approach which resulted in the *time-indexed formulation*. For each task, there are many possible choices for a suitable time-offset. Each schedule chooses exactly one time-offset for each task. In the time-indexed formulation, we introduce one binary variable $x_{i,t}$ for each combination of a task i and a time-offset t. This variable models whether task i is assigned the offset t. Then we add inequalities which exclude forbidden pairs of choices. Like above, this means that for any two choices which contradict each other, our inequalities ensure that at most one of them is taken. For instance, if two tasks i and i' collide if they are assigned offsets t and t', respectively, then we add the constraint

$$x_{i,t} + x_{i',t'} \leq 1 \,.$$

Finally, we add inequalities which ensure that every task is assigned to some time-offset.

The time-indexed formulation is a valid IP-formulation for our problem but it suffers from the great number of variables. It performed fairly well on very small instances with only 10 tasks. However, already on most instances with 20 to 30 tasks it did not finish within a reasonable time bound.

Congruence Formulation

Since the time-indexed formulation was not even able to handle instances of medium size, further structural insights were needed. In fact, one reason of the poor performance of the above formulation was that for each possible offset for a task we introduced new binary variables. Instead, it seems much more reasonable to introduce one single variable for each task which represents its start offset. In order to be able to formulate with such variables that no two tasks on the same processor execute a job at the same time, we used a structural lemma [3]. The lemma proves an exact mathematical characterization on when two tasks on the same processor collide, depending on their offsets, their periods, and processing times. Using this lemma, we were able to formulate linear inequalities which ensure that no two tasks on the same processor collide.

This approach results in the *congruence-formulation*. It contains a much smaller number of variables than the time-indexed formulation. As we will see below, it performs strictly better than the time-indexed formulation. However, it is still not capable to solve instances of real-world size. We believe that the reason for this is that it contains a (linearized) modulo-operator. Such constructions are usually difficult to handle for IP-solvers, even if the instance contains only a modest number of variables.

Bin-Formulation

As the performance of the congruence formulation is not sufficient to solve industrial size problems, we had a closer look at typical problem instances that arise in real world. Our goal was to reveal additional properties that can be exploited to improve efficiency. We found two notable properties. The most important one is that the periods in real world instances are *harmonic*, i. e., for every pair of periods p_i and p_j, one is an integer multiple of the other. Another property is that the ratio of the largest period to the smallest period is rather small compared to the number of tasks (20 vs. 200). The first property allows for a new interpretation of the problem with a better tractable structure. The second property ensures that this structure can be handled efficiently, as its complexity depends largely on that ratio.

A crucial observation that leads to the new interpretation is the following. If a feasible schedule exists for a given task set, then there exists a schedule with a very special structure as we will explain now. Let us assume that the two properties from above hold and consider a feasible single processor schedule. This means that every task j has an offset a_j so that the tasks executed with these offsets do not collide. Figure 2 illustrates an example of a schedule with 7 tasks of three different periods. Tasks one and two are of period 8, tasks three and four of period 16 and the remaining three tasks of period 32.

We can transform the schedule such that a task of smallest period, say task 1, has offset $a_1 = 0$ by shifting all offsets by the same amount. Here we shift the whole schedule "right" and obtain a new feasible schedule which is shown in Fig. 3.

Task 1 now plays a special role. It divides the timeline into intervals whose length is given by the tasks period p_1. We call these intervals *bins*. In Fig. 4 we highlight the bins.

Why are these bins important? Note that every execution of every task has to start and end within the same bin. Otherwise it would collide with task 1. Due to the periodic nature of the problem and the fact that every task period is an integer multiple of the interval length, each task appears in bins following a very regular pattern: A task of smallest period will be executed in *every* bin. A task whose period is twice the smallest period will be executed in every second bin. Similarly, a task j whose period p_j is an ℓ multiple of p_1, i. e., $p_j = \ell \cdot p_1$ will be executed in every ℓ-th bin. However this appearance pattern can be shifted. For example for the two tasks of period 16 in Fig. 4, task 3 appears in every odd bin while task 4 appears in every even bin. This shift is determined by the task offset a_j. As the pattern for each task is given by their period, the only freedom in designing the schedule is to determine the shift of the pattern. The number of choices (regarding the bins in which a task is executed) depends only on the period of a task: For a task j with period p_j we have p_j/p_1 many choices.

This observation already reveals a lot of structure, but one can do even more. Observe that every bin looks exactly the same as far as tasks of smallest period are concerned. For an input of tasks of period up to $\ell \cdot p_1$, every ℓ-th bin looks exactly the same. Moreover, for any two tasks the one with the larger period either appears in a subset of bins the other task appears in, or they never share a bin at all. This motivates to represent the schedule in form of a tree, which we call a *bin tree*. The tree has as many levels as we have different periods, say k many. Each node of the tree consists of a bin. The root node on level 1 corresponds to the smallest period and contains all tasks of period p_1. A bin on level ℓ contains only tasks of period less than or equal to the ℓ-th smallest period. The bins on level ℓ represent the possible choices (pattern shifts) for a task with the ℓ-th period. Each of these tasks must choose one bin. Every node except for the root also has one parent bin, which corresponds to those tasks of smaller periods that appear in a superset of bins in the schedule. The nodes inherit all tasks from its parent bins. This ensures that tasks of different periods that appear in a common bin in the schedule always share a bin in the bin-tree as well. The bins on the last level correspond to the actual bins from the schedule as illustrated in the picture above. (However, the order is different.) See Fig. 5 for an illustration of the bin tree that corresponds to the schedule from above.

How does this abstraction help us? A powerful structural insight which we will not prove here, is that one can ignore the actual execution slots of the tasks within the bins completely, only the fact whether a task appears in a bin or not is important. Observe that in every feasible schedule, the bins in the bin tree are not "overloaded", i. e., the sum of execution times of tasks appearing in the same bin never exceeds the bin size. The strength of the bin-tree abstraction is that this criterion is not only necessary but also sufficient for constructing a feasible schedule: Given an assignment of tasks to bins in the bin-tree on their respective levels, such that (following the inheritance rules) no bin on the last level is overloaded, then one can compute feasible offsets efficiently with a simple algorithm.

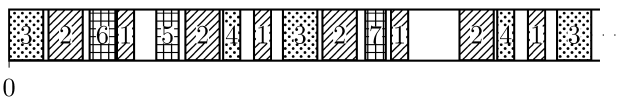

Figure 2 A schedule for an instance with 7 tasks.

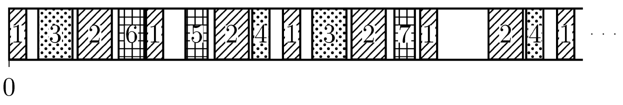

Figure 3 Another schedule obtained by "shifting" the previous one.

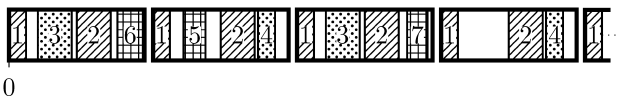

Figure 4 Bin representation of the previous schedule.

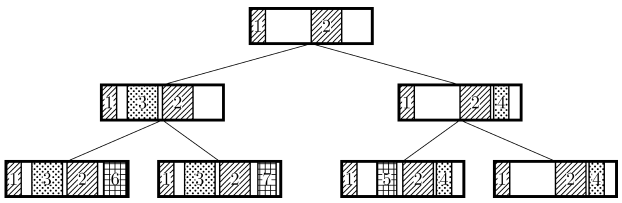

Figure 5 Bin tree representation of the schedule from Fig. 3.

This allows us to interpret our scheduling problem as an *assignment problem*: Assign tasks to bins in the bin tree such that no bin is overloaded. These assignment problems can be handled more efficiently in practice. The complexity of the representation is dominated by the number of bins in the bin-tree, i. e., the choices of the shift of the assignment pattern. As mentioned previously, the number of choices for a task is given by the ratio of its period and the smallest period q_1. At this point, our second observation about the characteristics of real-world instances comes into play: The ratio of largest period to smallest period is low. For that reason, the complexity of the model is well bounded for real-world instances.

We now show how the described theoretical structure can be turned into a compact model which is appropriate to solve real-world instances. As mentioned before, we need to determine which bin in the tree a task i is assigned to. For this it is enough to see which bin of level r contains task i, where q_r is the period of task i. Hence, we introduce a variable $z_{i,b}$ that represents if task i is assigned to bin b (which belongs to level r in the tree). The variable $z_{i,b}$ equals to one if task i is assigned to bin b, and equals to zero otherwise. For example, if we consider the solution given by Fig. 5, where b corresponds to the third bin (from left to right) of the last level, then $z_{5,b} = 1$. Also $z_{5,b'} = 0$ for any other bin b'. The model has to ensure that each task is assigned to exactly one bin. This can be captured with the following linear constraint

$$\sum_{\text{bin }b} z_{i,b} = 1 \quad \text{for each task } i\,. \tag{1}$$

This constraint ensures that our solution will assign each task to some bin, however this is not enough to imply that our solutions will be feasible: it may happen that two jobs use the processor at the same time, or equivalently by the criterion explained before, some bin could be overloaded. Hence, we need to add another constraint to restrict to solutions that do not overload any of the bins. For this recall that if task i is assigned to bin b, then this task will appear in all of the descendants of this bin. Thus, to guarantee that a bin b of the last level of the tree is not overloaded we can impose the following constraint.

$$\sum_{b'} \sum_{\text{task } i} c_i \cdot z_{i,b'} \le q_1\,, \tag{2}$$

where the first sum is taken over all of bins b' that are an ancestor of b.

Summarizing, we have that an assignment of tasks to bins given by the variables $z_{i,b}$ represents a feasible schedule on a single processor if and only if all variables are either 1 or 0 and they satisfy constraints (1) and (2).

3 Computational Results

Out of the three formulations mentioned above, only the last one is powerful enough to solve the industrial size problems in our tests. We illustrate this in the following, where we compare computationally the running times for solving each of the three models on the same test instances. This comparison is done over extensions of the models described in the previous section to the multiple processor case. For the tests, we set the objective function to minimize the number of processors needed to process all the tasks.

We generated random instances by perturbing real-world instances. The instances are generated by drawing a random subset of 10, 20, 30, 40, 50 or 60 tasks from the largest industrial instance we had at our disposition, and perturb some of their parameters randomly.

In these real world instances additional side constraints like memory and bandwidth restrictions are also incorporated into the model. All these extra conditions can be addressed in a straightforward manner by adding extra linear inequalities to each of the models.

All computations are performed on a two-processor computer with Intel Xeon 2.66 GHz CPUs with 8 GB of RAM, running Linux. We used CPLEX release version 12.1.0.

In Figs. 6 and 7 we compare how the different methods perform. Figure 6 shows the percentage of the random instances which are able to finish within 30 minutes and where the system does not run out of memory. Figure 7 compares the average running times for solving the model for the different number of tasks. As the differences in running time are so drastic, to improve readability the plot is shown in semi-logarithmic scale.

In Fig. 6 we can see that the congruence formulation and time-indexed formulation are not able to solve most of the instances within the time horizon of 30 minutes, even when the number of tasks is relatively small. Considering that real-world instances usually have at least 90 tasks, we conclude that these approaches are inappropriate. Figure 7 confirms this statement by showing how

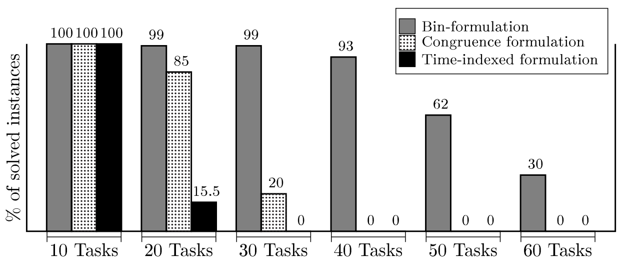

Figure 6 Performance of the models: Percentage of instances solved within 30 minutes on instances of various sizes.

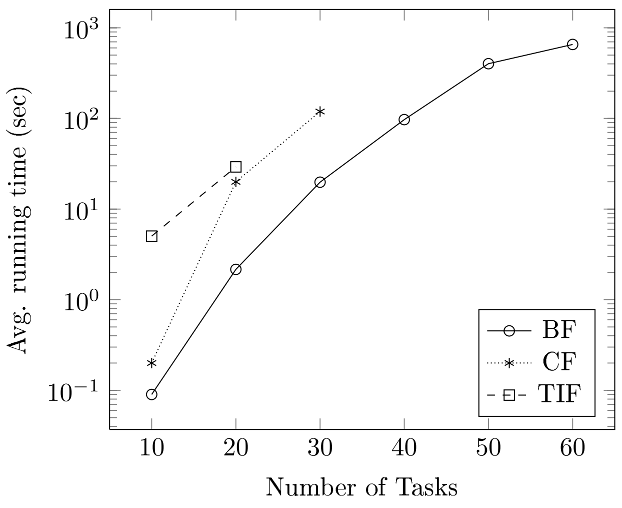

Figure 7 Performance of the models: Average time needed to solve instances of different sizes. BF: bin-formulation, CF: congruence-formulation, TIF: time-indexed formulation.

the running time of these formulations grows with the number of tasks in the instance.

4 Conclusion

Complex optimization problems arising in industry often need sophisticated optimization tools to be solved. In particular, heuristics and other straightforward approaches are often inappropriate to handle such problems since they can be unreliable: They can return suboptimal solutions or they could even not find a feasible solution.

In this article we studied such an optimization problem arising in avionics, for which heuristics are particularly inappropriate. We resorted to integer programming tools to model and solve the problem. Guided by the algorithm engineering paradigm, we proposed several such models. These models used insights about the real world instances as well as several mathematical tools developed to understand the problem thoroughly. These insights were crucial to formulate the problem appropriately, and finally find a compact formulation that is capable of handling the real-world instances.

References

[1] F. Eisenbrand, K. Kesavan, R. S. Mattikalli, M. Niemeier, A. W. Nordsieck, M. Skutella, J. Verschae, and A. Wiese. Solving an avionics real-time scheduling problem by advanced IP-methods. In: Proc. of the 18th Annual European Conf. on Algorithms (ESA), LNCS 6346, pages 11–22, 2010.

[2] F. Eisenbrand, N. Hähnle, M. Niemeier, M. Skutella, J. Verschae, and A. Wiese. Scheduling periodic tasks in a hard real-time environment. In: Proc. of the 37th Int'l Colloquium on Automata, Languages, and Programming (ICALP), LNCS 6198, pages 299–311, 2010.

[3] J. Korst, E. Aarts, J. K. Lenstra, and J. Wessels. Periodic multiprocessor scheduling. In: Proc. of PARLE'91 Parallel Architectures and Languages Europe, LNCS 505, pages 166–178, 1991.

Received: March 4, 2011

Prof. Dr. Friedrich Eisenbrand is a full professor of mathematics at EPFL. His main research interests are algorithms and complexity, integer programming, geometry of numbers, and applied optimization. Before joining EPFL in 2008, Friedrich Eisenbrand was a full professor at University of Paderborn. He received an Alexander von Humboldt professorship, the Heinz Maier-Leibnitz award and the Otto Hahn medal.

Address: École Polytechnique Fédérale de Lausanne, Station 8, 1015 Lausanne, Switzerland, Tel.: +41-21-6932560, Fax: +41-21-6935840, e-mail: friedrich.eisenbrand@epfl.ch

Dipl.-Math., Dipl.-Inf. Martin Niemeier received his diplomas in Mathematics and Computer Science from the University of Paderborn in 2008. Since then, he has been a Ph.D. student at EPFL under supervision of Prof. Dr. Friedrich Eisenbrand. His research interests lie in Combinatorial Optimization, in particular on approximation algorithms and scheduling problems.

Address: École Polytechnique Fédérale de Lausanne, Station 8, 1015 Lausanne, Switzerland, Tel.: +41-21-6932739, Fax: +41-21-6935840, e-mail: martin.niemeier@epfl.ch

Prof. Dr. Martin Skutella is a full professor of mathematics at TU Berlin and at the DFG research center MATHEON in Berlin. He received his Ph.D. in Mathematics from TU Berlin in 1998. After several stays abroad, including a guest professorship at MIT in Cambridge (USA), and a sr. researcher position at MPI for Computer Science, he became full professor in Dortmund in 2004 before he moved on to Berlin in 2007.

Address: Technische Universität Berlin, Institut für Mathematik, Straße des 17. Juni 136, 10623 Berlin, Germany, Tel.: +49-30-31478654, Fax: +49-30-31425191, e-mail: martin.skutella@tu-berlin.de

Math. Ing. José Verschae received his Mathematical Engineering degree in 2009 (graduated with honors), and his B.Sc. degree in Engineer (Major in Mathematics) in 2002, both at Universidad de Chile. Since 2009 he is a Ph.D. student in the Institute of Mathematics at TU Berlin. His main area of research is Combinatorial Optimization, with an emphasis in optimization under uncertainty and Scheduling.

Address: Technische Universität Berlin, Institut für Mathematik, Straße des 17. Juni 136, 10623 Berlin, Germany, Tel.: +49-30-31478656, Fax: +49-30-314-25191, e-mail: verschae@math.tu-berlin.de

Dr. Andreas Wiese received his diploma in Mathematics at the TU Berlin in 2008. Under the supervision of Prof. Dr. Martin Skutella, he received his Ph.D. in 2011. His research interests lie in the area of Combinatorial Optimization, focusing on approximation algorithms, scheduling, and realtime scheduling.

Address: Technische Universität Berlin, Institut für Mathematik, Straße des 17. Juni 136, 10623 Berlin, Germany, Tel.: +49-30-31427448, Fax: +49-30-31425191, e-mail: wiese@math.tu-berlin.de