Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Crown-scheduling of sets of parallelizable tasks for robustness and energy-elasticity on many-core systems with discrete dynamic voltage and frequency scaling[☆]

Christoph Kessler[a,*], Sebastian Litzinger[b], Jörg Keller[b]

[a] *Linköping University, Sweden*
[b] *FernUniversität in Hagen, Germany*

ARTICLE INFO

Keywords:
Adaptive task scheduling
Robustness of schedules
Moldable parallel tasks
Crown scheduling
Energy optimization

ABSTRACT

Crown scheduling is a static scheduling approach for sets of parallelizable tasks with a common deadline, aiming to minimize energy consumption on parallel processors with frequency scaling. We demonstrate that crown schedules are robust, i. e. that the runtime prolongation of one task by a moderate percentage does not cause a deadline transgression by the same fraction. In addition, by speeding up some tasks scheduled after the prolonged task, the deadline can still be met at a moderate additional energy consumption. We present a heuristic to perform this re-scaling online and explore the tradeoff between additional energy consumption in normal execution and limitation of deadline transgression in delay cases. We evaluate our approach with scheduling experiments on synthetic and application task sets. Finally, we consider influence of heterogeneous platforms such as ARM's big.LITTLE on robustness.

1. Introduction

Static scheduling of parallelizable task sets on parallel machines has been investigated for decades, and the advent of frequency scaling has led to scheduling approaches that e. g. try to minimize energy consumption for a given throughput, i. e. deadline until which each task must be executed.

The static schedulers assume that the workload of each task is known exactly, but small variations might occur. By the *robustness* of a schedule we understand by which fraction β the deadline of this schedule will be violated when the runtime of a single task is increased by a fraction α, which in turn increases its makespan. Deadline violation is more likely if the deadline is tight, but sometimes, the discrete frequency levels lead to a gap between makespan and deadline so that makespan extension can be – at least partly – compensated by filling this gap.

For a schedule, we are interested in minimum, maximum and average deadline transgression relative to extension of a task (average formed over all tasks). As an example, we consider a task set with two tasks of similar workload scheduled onto two cores until the deadline. The tasks could remain sequential and be executed one on each core until the deadline (with suitable frequency), or the tasks could be parallelized, whereupon we assume perfect speedup here for simplicity of presentation. Then the tasks could be executed one by one, each on both cores, with the same frequency. Thus, two different schedules are possible for such a task set. While the energy consumption is the same for both schedules, the robustness is different, cf. Fig. 1: In the first schedule, the deadline is surpassed by time αM. In the second schedule, the deadline is exceeded by time $\alpha M/2$. Thus, scheduling decisions influence robustness. Furthermore, parallelization of tasks may improve robustness in addition to normal uses like better load balancing or reduced energy.

Robust schedules help to close the gap between static and dynamic scheduling: if task runtimes are known up to a fraction α, and robustness is $\beta < \alpha$, then it suffices to schedule for a deadline which is a fraction $1/(1+\beta) < 1$ of the real deadline, to be sure to meet the real deadline, even if a task's runtime could increase to $1+\alpha$ of its nominal value. The better the robustness, i.e. the smaller β, the smaller the energy to be invested to meet the deadline without having to resort to online (re-)scheduling. As an unfortunate example, in schedules generated according to Sanders and Speck [2] each task that is parallelized uses all its allocated cores until the deadline, so that for those tasks $\beta = \alpha$, which cannot be considered robust.

There is an online alternative to scheduling for tighter deadline. When a task has longer runtime than expected, the runtime system can

☆ This article is an extended version of our conference paper [1] at PDP'20, March 2020.
* Corresponding author.
E-mail addresses: christoph.kessler@liu.se (C. Kessler), sebastian.litzinger@fernuni-hagen.de (S. Litzinger), joerg.keller@fernuni-hagen.de (J. Keller).

https://doi.org/10.1016/j.sysarc.2021.101999
Received 7 June 2020; Received in revised form 11 November 2020; Accepted 1 January 2021
Available online 10 January 2021

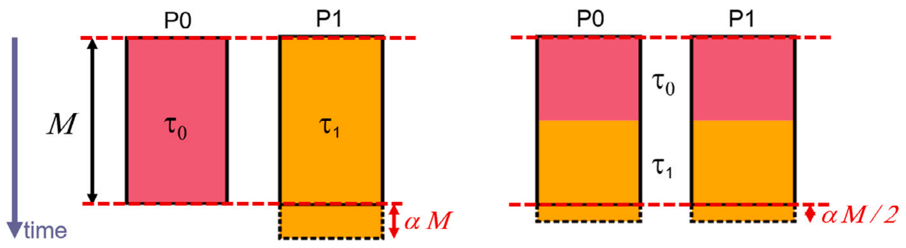

Fig. 1. Makespan extension due to a factor α delay of task τ_1: by αM for sequential execution (left) and by $\frac{\alpha}{2}M$ for parallel execution with 2 processor cores (right).

accelerate tasks scheduled later than the prolonged task, to still meet the deadline M, or an only slightly extended deadline $M+C$ for a given small $C > 0$, at a moderate increase in energy consumption. We call this property *elasticity* (for $C = 0$), or in its generalized form, C-*elasticity* (for $C > 0$) of a schedule. Even when cores are considered to run at maximum operating frequency, the turbo frequency is usable for a short period of time for such a case. Furthermore, if the deadline is not too tight, the most energy-efficient frequency can be the next-to-maximum frequency, e.g. in ARM's big.LITTLE [3].

In this work, which is a significantly extended [1] version of our conference paper [1], we make the following contributions:

- We present a formal notation of robustness and C-elasticity of crown schedules and demonstrate with benchmark task sets how robust crown schedules are.
- We present an integer linear program (ILP) to compute for a given static crown schedule and a given delayed task, which tasks should be accelerated, i. e. run at a higher frequency level, to still meet the deadline after a prolonged task with a minimum addition in energy. We also present a dynamic rescaling heuristic to choose these tasks, to be able to quickly choose at runtime, and evaluate its quality by comparing with the ILP results. Note that only the DVFS levels (of applicable tasks) are adapted dynamically, while the other aspects of the schedule (core allocation, mapping, ordering) are static and remain fixed.
- We present an ILP that considers normal execution and all delay cases together and finds a schedule that, given α and β, finds a schedule that achieves the desired degree of robustness without re-scaling and with minimum additional energy consumption during normal execution.
- We investigate how using a heterogeneous platform such as ARM's big.LITTLE improves robustness compared to a similar but homogeneous platform.

The remainder of this article is structured as follows. In Section 2, we give background information on task scheduling, in particular crown scheduling, and related work. In Section 3, we define robustness and present algorithms to optimally and heuristically choose tasks for acceleration by frequency increase, to meet the deadline in case of a slow task while spending as little additional energy as possible. In Section 4, we investigate tradeoff between additional energy consumption in normal execution and achieving desired robustness and elasticity in delay cases. Section 5 reports on our benchmark task sets and the results obtained for robustness and elasticity by energy increase. In Section 6 we investigate influence of platform heterogeneity on robustness. Finally, Section 7 concludes and gives an outlook on future work.

[1] The following contents has been added since the conference paper [1]: the concept of C-elasticity in Section 3, the extension towards robustness-aware scheduling (new Section 4), additional experiments on robustness-aware scheduling, experiments with task sets based on real applications and experiments with larger task set and machine sizes in Section 5, generalization towards robustness on heterogeneous architecture, taking ARM big.LITTLE as example (new Section 6). Also, the discussion of related work has been extended and the notation has been clarified.

2. Background and related work

2.1. Machine model

As machine model we consider a generic parallel machine with $p > 1$ processor cores $\{P_0, \ldots, P_{p-1}\}$, where the execution frequency for a core can be switched individually within a given set of K discrete frequency/voltage levels $F = \{f_0 = f_{min}, f_1, \ldots, f_{K-1} = f_{max}\}$.

When executing a task at frequency f_i, a core draws power $power(f_i)$, so that execution of a task with λ clock cycles results in a runtime of λ / f_i and an energy consumption of $E = power(f_i) \cdot \lambda / f_i$. Our power consumption model only takes the frequency as a parameter, but not voltage, temperature nor instruction mix (on which it also depends). We assume that for each frequency, the least possible voltage level is used, that temperature is controlled by cooling, and that the tasks' instruction mixes are sufficiently similar. The first two assumptions have been tested in experiments, and in Litzinger et al. [4] we have shown how to extend the model to task type-awareness. Currently, we do not yet consider power consumption while a core is idle, but plan to study this in future work. For a heterogeneous architecture, the workload of a task may depend on the type of the core that the task runs on, as different core types may have different instruction set architectures, or different microarchitecture even for similar instruction set architecture. Similarly, the power consumption and the range of frequencies of a core depend on its type.

2.2. Application model

The application is modeled as an iterative computation where each iteration or *round* consists of a set of n independent (and possibly parallelizable) tasks $\{\tau_0, \ldots, \tau_{n-1}\}$. Such a computation structure might result, for example, from the iteration over the steady-state pattern of a software-pipelined streaming task graph [5], see Fig. 2. Each task τ_j performs *work* λ_j, i. e., the execution of τ_j on a single core takes λ_j clock cycles. Task τ_j could use up to W_j cores; for $W_j = 1$ the task is sequential, for $W_j > 1$ it is parallelizable up to W_j cores, where we assume *moldable* parallel tasks, i. e., the number of cores to use needs to be decided before task execution, in contrast to *malleable* tasks, where a task can decide or change the number of cores it uses during its execution. Given an individual parallel efficiency function e_j, the resulting execution time of task τ_j using q cores (all at same frequency $f \in F$), for $1 \le q \le W_j$, is

$$t_j(q, f) = \frac{\lambda_j}{f \cdot q \cdot e_j(q)} .$$

For a heterogeneous platform, we assume that a task only runs on cores of a single type. Still, the runtime will depend on the core type because the workload will be core type-specific.

2.3. Scheduling

A static *schedule* for the n tasks (instances) of one round of the computation allocates to each task τ_j a number of cores w_j, maps it to a subset of w_j cores of P for execution, decides a start time for τ_j,

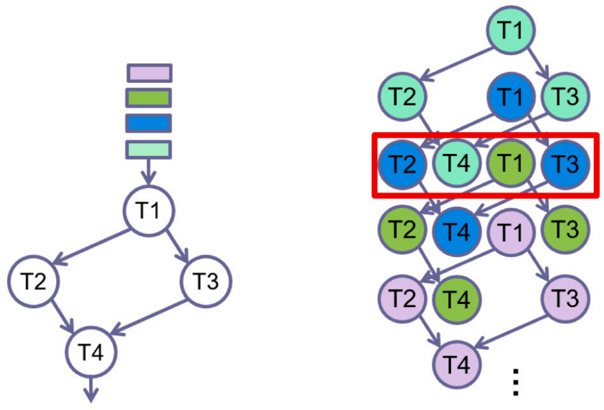

Fig. 2. Left: A streaming task graph with 4 tasks processing a stream of input data packets. — Right: The repeating steady-state pattern (red box) of the software-pipelined execution of the streaming tasks contains independent task instances within each iteration. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

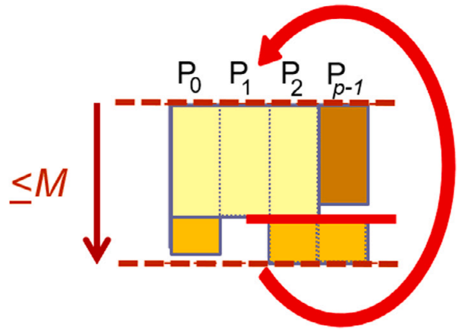

Fig. 3. An arbitrary schedule for one round, for $n = 4$ moldable parallel tasks mapped to $p = 4$ cores. The color coding indicates the execution frequencies selected for each task (darker means higher frequency, thus more power-consuming). The makespan of the schedule must be within the limit M. Note the idle time due to external fragmentation between the two tasks assigned to the last core, caused by starting a subsequent parallel task on a partially overlapping core subset (red bar). Such internal synchronization points within the schedule can cause local frequency rescaling decisions (e.g., for the three-core task mapped to cores 0, 1, 2) to propagate to unrelated cores (core 3), i.e., have global effects, adding to the complexity of the overall optimization problem for general schedules. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

which must be identical on each core in that subset, and selects an execution frequency $f_j \in F$ for the task. As a core can only execute one task at a time and tasks are non-preemptive, i. e., a started task will not be interrupted by another a task, a core serially executes all tasks assigned to it in the start time order given by the schedule, and tasks should start as soon as all required cores are available. Hence, each schedule S implies a fixed *makespan* t_S, i. e., the overall time for executing all n tasks of one round as prescribed in S. See Fig. 3 for an example.

We request that our application needs to keep a certain application-specific throughput X (e. g., X image frames processed per second), which translates into an ideal or maximum makespan or deadline $M = 1/X$ per round (e. g., processing of one image frame). A schedule S with makespan shorter than M is of no use, and hence scheduling instead minimizes the overall energy consumed under the constraint $t_S \leq M$. A number of techniques for energy-optimizing scheduling of moldable parallel task sets on parallel machines with DFVS-scalable processors have been proposed in the literature; see e. g. Melot et al. [6] for a survey and experimental comparison.

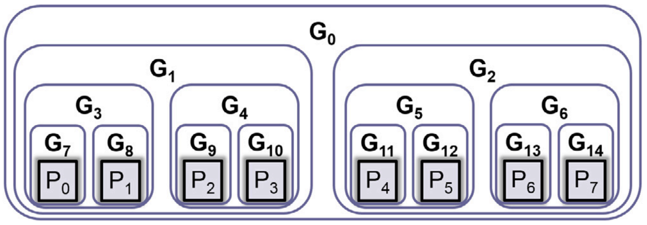

Fig. 4. Crown scheduling group hierarchy for 8 cores.

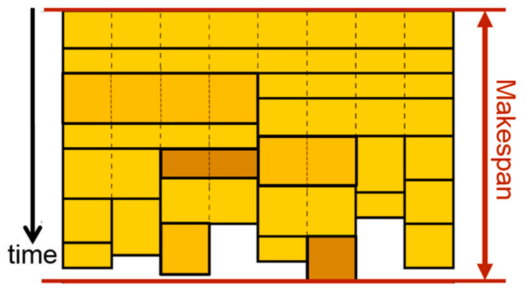

Fig. 5. A crown schedule for a machine with 8 cores. The tasks' shading represents operating frequency levels.

2.4. Crown scheduling

Crown scheduling, see Melot et al. [6], is a scheduling technique which maps tasks to predefined groups of cores, and at the same time sets the cores' operating frequency for each task. For p cores, there are $2p-1$ groups $G_0, \ldots, G_{2p-2}$ organized in a particular group hierarchy, cf. Fig. 4: One group (the root group) contains all cores, and for each group with > 1 core, there are two groups evenly splitting the cores the group comprises. That way, a core is a member of multiple groups (for any parallel machine), and a task can be allocated powers of 2 as core count. This restriction eases the computation of a schedule significantly since the solution space is considerably narrowed.

To obtain a feasible schedule from a mapping of tasks to core groups, tasks are executed in order of non-increasing width w_j, cf. Fig. 5.

Due to the above mentioned restriction, mapping and frequency selection can effectively be performed via solving an ILP, facilitating the inclusion of aspects such as performance or energy efficiency. In the present context, we aim to optimize for low energy consumption, cf. Section 2.3. The crown scheduler thus solves the following ILP with $(2p-1)\cdot n \cdot K$ decision variables $x_{i,j,k}$, where $x_{i,j,k} = 1$ signifies that task τ_j is executed in group G_i by its p_i cores at frequency level f_k:

$$\begin{aligned}
\text{min} \quad & E = \sum_{i,j,k} x_{i,j,k} \cdot t_j(p_i, f_k) \cdot power(f_k) \cdot p_i \\
\text{s.t} \quad & \forall j : \sum_{i,k} x_{i,j,k} = 1 \\
& \forall j : \sum_{i : p_i > W_j, k} x_{i,j,k} = 0 \\
& \forall l : \sum_{i \in G^{(l)}, j, k} x_{i,j,k} \cdot t_j(p_i, f_k) \leq M.
\end{aligned}$$

Here, $power(f_k)$ is a core's average power consumption at frequency level f_k, which could easily be extended to task-specific power consumption $power_j(f_k)$. The constraints ensure that each task is executed exactly once, and that a task τ_j is not allocated more than W_j cores (p_i being the number of cores in group G_i). The last type of constraints guarantees that no core is allocated more work than it can handle until the common deadline M (where $G^{(l)}$ denotes the set of groups core P_l is a member of). In order to obtain a complete schedule, the resulting mapping of tasks to core groups and the selected operating frequencies must be accompanied by a task execution order for each group since there may be more than one task mapped to a particular group. While the execution order does not matter if all goes well, a schedule's robustness can and most likely will be sensitive to changes in this respect.

For a heterogeneous architecture with 2^c core types, each with the same number $p/2^c$ of cores, the first $2^c - 1$ groups remain empty. The core type then is uniquely defined by the core group, so that power, workload and time are additionally indexed by the core group index i.

2.5. Related work

Streaming task graphs[2] have been investigated at least since Kahn introduced Kahn process networks [8]. Lee and Messerschmidt [9] considered static scheduling for applications revealing such synchronous data flow, in particular signal processing. In that community, also the name *actor networks* is used [10]. Tasks can be sequential or parallelizable.

Moldable and malleable task scheduling has been considered in the literature for both makespan [11] and energy [12] optimization, both for streaming task graphs [13] and for graphs of one-shot tasks [14]. While *moldable* tasks use a fixed, discrete number of cores from start, *malleable* tasks can dynamically add or release cores at arbitrary points of time, which leads to a more continuous optimization problem. Resource allocation, mapping and DVFS together is considered in very few works on moldable task scheduling [6,13] and malleable task scheduling [2]. However, none of these considered robustness of schedules nor other fault tolerance aspects.

Robustness of schedules against unforeseen delays of tasks has been considered in the scheduling theory literature for the case of *sequential* tasks on single and multiple cores. In most works, the task mapping in the original schedule is kept after an unforeseen delay and the timing is modified by postponing subsequent task start times (”right-shifting” along the time axis). For example, Jorge Leon et al. [15] define robustness measures for schedules in job-shop scheduling as a weighted sum of the expected value of the changed makespan and of the difference from the original schedule's makespan, and analyze for the case of a single task experiencing an unforeseen delay. For sequential tasks with dependencies, [16] define the robustness of a schedule as the expected value of the makespan for a given (narrow) random distribution of task delays, arguing that the average impact of a delay is smallest for schedules where most tasks are not on a critical path. Our triplet measure for robustness as presented in Section 3 can be considered as a generalization of the robustness measures used in these works.

Robustness of schedules for task graphs is a major concern in practical real-time scheduling. As worst-case execution time estimates are often far too conservative and real task execution times tend to average close to their best case time, (soft) real-time scheduling works, in practice, rather with average times and adds some slack time and adaptivity mechanisms to a schedule to account for time variability and increase a schedule's robustness against occasional task delays [17]. Canon and Jeannot [18] give an experimental comparison of different robustness metrics in real-time scheduling of directed acyclic graphs (DAGs) of sequential tasks and also propose different heuristics to maximize robustness. Stochastic analysis of DAG schedules and their robustness by propagating representations of task delay probability distributions has been considered in a number of works [19,20] and a number of robust scheduling heuristics for DAGs has been proposed in the literature, e.g. Adyanthaya et al. [21], Lombardi et al. [22]. In this work we consider the makespan of crown schedules, thus basically independent tasks (within one round) yet with relative ordering constraints especially for parallel tasks and with a common (soft) deadline, which can be considered as a special case.

For independent moldable jobs, Srinivasan et al. [23] consider the *on-line* scheduling problem to be solved by batch schedulers for HPC cluster systems if they were given the choice of resource (here, #nodes) allocation from a given range for each submitted job. They study the robustness of different on-line scheduling algorithms (not of individual schedules) in terms of their average quality change compared to a baseline rigid scheduler when varying job parameters such as scalability or system parameters such as current system load. They experimentally find by simulations based on log files from real supercomputer centers that existing greedy and proportional allocation strategies do not uniformly perform better in all scenarios (job categories, scalabilities, system loads), while the work per job is assumed to be fixed. DVFS and energy consumption are not considered in Srinivasan et al. [23]. In our work we instead consider off-line scheduling, robustness with respect to task delays, and DVFS as an additional knob to turn to reduce the impact of task delays by rescaling while preserving mapping and resource allocation in the schedule.

We are not aware of any work taking robustness of schedules into account for the case of *off-line* scheduling *moldable parallel* tasks considered here, in particular not in the context of energy optimization for target systems with discrete DVFS.

Our techniques leverage not only the slack in lower-frequency tasks within the scope of the delayed task for delay compensation, but also the remaining idle time towards the deadline M, which might be unavoidable esp. due to the discreteness of the frequency and core number assignments. Another way to exploit such idle time towards M consists in dynamically switching back and forth between two crown schedules, a conservative one and one that slightly exceeds M, to remove such idle times for improved average energy efficiency [24].

3. Energy-elastic crown schedules

3.1. Robustness metric

From now on, let us assume that we are already given a fixed crown schedule S for the n tasks, which is based on the given task work parameters λ_j. Assume now that a task τ_j is running longer than expected, i. e. that its length t_j grows to $t_j(1+\alpha)$, where $\alpha > 0$. Then, all tasks scheduled after τ_j on the same core(s) will be delayed by the same amount of time αt_j, as a core can only execute one task at a time, and a crown schedule has no gaps between tasks. Such cascaded delays *will* lead to a larger makespan exceeding the deadline M if the delayed task τ_j is located on a *critical path* in S. Otherwise, such delays may happen or not, depending on whether there is sufficient idle time (slack) in the schedule S after the last task on this core(s) to accommodate the delayed tasks without exceeding the deadline M. Please note that we model the prolongation of runtime independent of the parallelization of a task. If the prolongation follows the same parallel efficiency as the task as a whole, then α affects the task's workload λ_j in the same way as the runtime. There can be multiple reasons for such prolongation. For example, the task's input data may be more imbalanced than assumed by the application, or one core running the task may be affected by some operating system noise. We have chosen parameterization by α instead of an absolute delay in order to study how much imbalance a schedule can withstand, independent of the size, i.e. workload, of a task.

We do not require a probability of the task runtime increase, which would necessitate to specify task runtime distributions. Instead of having a probability that the deadline is not violated, we are interested in the delay of one task that can be tolerated. A stochastic treatment (see probabilistic robustness later in this section) could be done by interpreting the processor groups as a binary tree, and the tasks in one group as a chain, thus forming a particular DAG of tasks whose runtime

[2] Task graphs are also called workflows in the Grid community [7].

distribution can be computed from the task runtime distributions, cf. e.g. [25].

By the delay of one task, some tasks in the original schedule S may accordingly be started later in time, resulting in a slightly modified schedule S'. In some cases, the revised schedule's makespan $t_{S'}$ will then exceed the original schedule's makespan t_S. If $t_{S'}$ is larger than the deadline M, then there are several possible ways to quantify this impact. We can express the lateness of the new makespan $t_{S'} - M$ relative to the deadline M, i.e.

$$\beta_j = \frac{t_{S'} - M}{M} \tag{1}$$

or $t_{S'} = (1 + \beta_j)M$. Alternatively, we can express the lateness relative to the delayed task's runtime, i.e.

$$\beta_j = \frac{t_{S'} - M}{t_j}. \tag{2}$$

If $t_{S'} \leq M$, we define $\beta_j = 0$ in both cases.

In both cases, β_j is bounded by α, because $t_{S'} - M \leq \alpha t_j$ (the schedule cannot be delayed more than the task causing this) and $t_j \leq M$ (each task in the normal schedule is processed by the deadline). The bound is tight if p sequential tasks of similar workload are scheduled onto p cores with a runtime that equals the deadline.

Which definition of β_j, i.e. which *delay impact model* is more appropriate depends on the *delay cause model*, i.e. the model according to which the task delays appear. For a *uniform* delay cause model (UDC), where the task suffers a delay independent of the task's size, e.g. because of data imbalance, the *deadline-based* delay impact model (DDI) according to Eq. (1) is chosen. For a *time-based* delay cause model (TDC), where the task suffers a delay dependent on its workload, e.g. because of OS noise, the *time-weighted* delay impact model (TDI) according to Eq. (2) is chosen.

In order to normalize the deadline transgression relative to α, i.e. to have a value in $[0;1]$, and to express that the robustness is better for higher values, we will use *relative robustness* as a measure for the impact of a delay of task τ_j by α

$$R_j(\alpha) = 1 - \frac{\beta_j}{\alpha}$$

in the remainder of this article.

We define the *robustness* of a schedule S with respect to α as

$$R_S(\alpha) = \left(\max_j R_j(\alpha),\ \sum_j R_j(\alpha)/n,\ \min_j R_j(\alpha) \right),$$

i.e. as the tuple describing the minimum, average and maximum relative robustness over all tasks τ_j, given α. To indicate the impact model, we index R with DDI or TDI, but skip this when the model is clear from the context.

One might call this form of robustness *first order* robustness and consider in future work also higher order robustness, where several tasks might increase their workload, or probabilistic robustness where tasks increase their workload according to independent distributions.

The value of robustness will decrease with α. For small α, the runtime increase of a task, if it is not on the critical path determining the makespan, might be small enough to not increase the makespan over the deadline, or only to a small extent, i.e. R will be close to 1. For large α, the runtime increase αt_j will be such that the makespan increase β will tend towards α, i.e. R will be close to 0.

For any task j in schedule S and task delay factor α, we can compute β_j by computing the makespan of the revised schedule S' with the same allocation, mapping, frequency scaling, and task ordering as in S, with the only difference that task τ_j is now longer and some of the (later) tasks may accordingly be shifted forward in time. If we interpret the schedule as an acyclic directed graph with n nodes for the tasks, and an edge connecting any tasks u to v iff task v is executed on some core after u has been terminated, then the makespan is the length of the longest path from any source to sink.[3]

For a crown schedule S, this graph is a tree that corresponds to the group structure, and thus the β_j can be computed faster as follows. We read the task set description and the schedule description, establish to which group $g(j)$ in the crown each task τ_j has been mapped, and compute the runtime of each group. As the group hierarchy forms a balanced binary tree, we can thus also define start and end times for each group. For each group G_i, we can also compute the maximum end time $t_S(G_i)$ of any leaf group depending on G_i. The makespan is then $t_S := t_S(G_0)$ for the root group G_0. Fig. 6 illustrates the above calculations for an example featuring 7 core groups.

In order to compute β_j for given α, we observe that if τ_j's runtime t_j is increased by αt_j, then this increase will delay all descendant groups, i. e., also $t_S(G)$ will increase by this time. Hence, if $t_S(G) + \alpha t_j \leq M$, then $\beta_j = 0$ and the makespan constraint will not be violated, and otherwise the makespan limit M will be exceeded by β_j according to Eqs. (1) and (2) in the DDI and TDI models, respectively. In this manner, all β_j and the robustness can be derived for different values of α.

The definition can be extended to collections of schedules, where beyond the absolute min and max over all schedules also the average over the schedules' min's and the average over their max's can be computed. When we compute the average, we also compute the standard deviation to see the spread.

3.2. C-Elasticity metric for crown schedules

The *elasticity* of a (crown) schedule denotes the minimum, average and maximum amount of delay per core that the schedule can fully compensate for by modifying the DVFS levels for some of the tasks that follow a delayed task and still make the given deadline M, in general at the expense of higher energy cost as now some of these tasks may have to run at higher frequency levels.

Given a crown schedule S and a task d, let $scope_S(d)$ denote the set of all tasks j that start in S after d finished and that would have to be shifted in time if d was delayed. For a crown schedule, $scope_S(d)$ consists of all tasks following the delayed task d in the core group of d, plus all tasks starting later on any core of this group (see Fig. 8 for an illustration). It is a unique property of crown schedules that tasks outside $scope_S(d)$ will not be affected by a delay of d (see also the illustration of a non-crown schedule in Fig. 3 where this property does not hold).

We introduce a generalized metric, C-*elasticity*, which tolerates, for a given parameter $C > 0$, a deadline transgression by C. Hence, elasticity as defined above is 0-elasticity. Intuitively, C-elasticity reflects the minimum, average or maximum slack time within $scope_S(d)$ after each possibly delayed task d, i.e., accumulated possible time gains over the tasks $j \in scope(d)$ with still up-scalable frequency levels $f_j < f_{\max}$ and any possibly remaining idle time on cores in $group(d)$ towards the relaxed deadline $M + C$. Formally, we define the C-elasticity of a given crown schedule S as

$$E_S(C) = \left(E_S^{\min}(C),\ E_S^{avg}(C),\ E_S^{\max}(C) \right)$$

with

$$E_S^{avg}(C) = \sum_d \left(\sum_{j \in scope(d)} (f_{\max} - f_j) \cdot t_j \cdot |group(j)| + \sum_{q \in group(d)} (M + C - t_S(q)) \right) / n$$

where $t_S(q)$ denotes the accumulated length of all tasks running on core q in S (recall that idle times in a crown schedule only occur at the end of the round).

[3] If we add artificial unique source and sink nodes that connect to the source and sink tasks, respectively, only one longest path must be computed.

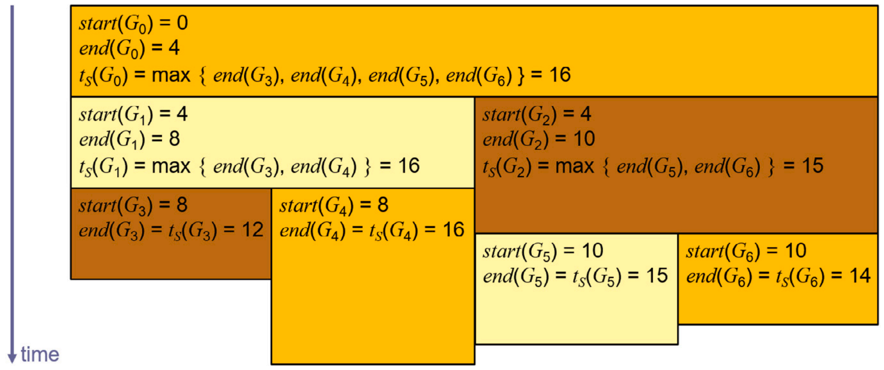

Fig. 6. Calculation of the groups' start and end times as well as maximum leaf end times for an example Crown schedule featuring 7 core groups.

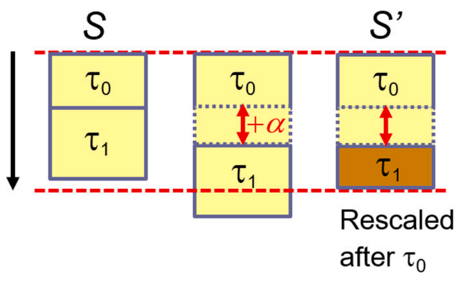

Fig. 7. Left: Schedule S (here, for a single core) without task delays. Time flows downwards. — Middle: Deadline transgression by the α-delayed task τ_0 in schedule S. — Right: dynamic adaptation to schedule S' by frequency rescaling of τ_1.

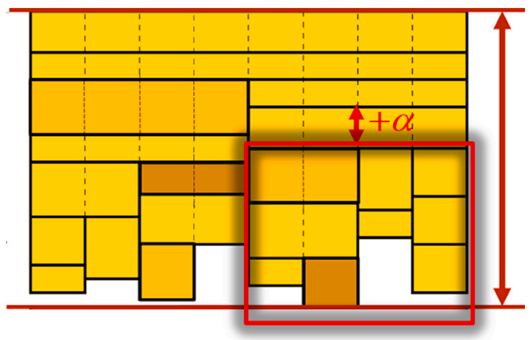

Fig. 8. A crown schedule with the rescaling scope (red frame) for a delayed task.

$E_S^{\min}(C)$ and $E_S^{\max}(C)$ are defined similarly, where $\sum_d$ is replaced by $\min_d$ and $\max_d$ respectively.

In the remainder of this section, we will now investigate how we can exploit the elasticity for improving the robustness of a given crown schedule S by leveraging dynamic frequency rescaling for the remaining tasks after a delay has been observed for a task. We present a fast rescaling heuristic and also provide an ILP-based optimal solution for comparison. We will later investigate ways to construct crown schedules that are, ab initio, more elastic, and thus, more robust.

3.3. Heuristic crown schedule adaptation

In order to (partly) compensate for the additional time from a task with increased workload, we perform at runtime, i.e. after the delayed task τ_d has completed and thus α_d and β_d are known, a fast greedy rescaling of the tasks that start after the delayed task τ_d has finished. The rescaling algorithm can run on one of the cores of the delayed task; the delay α is artificially extended by the runtime overhead of the rescaling step. In some cases (some of) the remaining tasks can be run at higher frequency to "absorb" the delay entirely or partially; this will then be done, regardless of the increased energy cost. After rescaling and makespan extension by β_d we obtain an *adapted schedule* S', which differs from S only by the modified t_d and by the frequency levels of zero or more other tasks. See Fig. 7 for an example.

The rescaling algorithm is presented in Algorithm 1. In case the delay leads to a deadline violation, it first determines for which tasks an increase in operating frequency could have an impact on the schedule's makespan. Suitable tasks are those which are executed after τ_d in $g(\tau_d)$'s offspring groups $offspring(g(\tau_d))$, i. e. groups encompassed by $g(\tau_d)$ according to the group hierarchy (including $g(\tau_d)$ itself). See also Fig. 8 for illustration of this rescaling scope. Only a faster execution of tasks in these groups can compensate for the delay of τ_d. A group order is established on the relevant groups, first by descending width, and for groups of the same width by descending annotation. A group G's annotation is the time span from the beginning of the first task's execution in G until the termination of the latest task in G's leaf groups, i. e. groups which belong to G's offspring and contain one core. Note that the group that τ_d is mapped to always has the first position in the order thus specified. To obtain a list of suitable tasks, the relevant groups are examined in the given order, their respective tasks added to the list in order of increasing frequency. Task order within a group is not defined by the crown schedule itself. Here, we support five variants: order by increasing task ID, order by increasing or decreasing frequency, and order by increasing or decreasing workload. Task order can be independently specified for $g(\tau_d)$ and its child groups. There are reasons for each of those orders. If tasks with large workload come first, then delays in the last tasks will only have small effect on the makespan. On the other hand, if a large task is delayed, there are only small tasks after it to make up for the delay. If tasks with high frequencies come first, then the last tasks can be increased in frequency. Hence it might depend on the taskset and schedule which order is best, thus also an order like task ID that is oblivious to above arguments might be worthwhile.

Ordering groups and tasks in the indicated manner aims at increasing the operating frequency for few tasks only and with little overhead regarding energy consumption. The list of suitable tasks is then traversed and for each task, the operating frequency is increased by one level if not yet at maximum, and only if a positive impact on the makespan can be observed. The algorithm terminates as soon as the modifications are sufficient to match the deadline, or after all suitable tasks have been considered. In the latter case, rescaling may have yielded a schedule violating the deadline.

Please note that the presentation in Algorithm 1 is conceptual, and that a more efficient implementation can be provided, which e.g. does not use *clone* but rescales the schedule in-place, to minimize overhead at runtime. In fact, only the modified DVFS levels of rescaled tasks need be stored, one at a time, and a single copy is sufficient. Note that the set of rescaling candidates ($\mathcal{T}_d$) could consist of up to $n-1$ tasks (namely, if already the first task at root group level is delayed). Assuming that groups' execution times are stored together with the schedule,[4] the initial makespan recalculation of $t_{S'}$ after the task delay can be done in time $O(\lg p)$, as up to $\lg p$ groups' accumulated execution times are affected by the delayed task. Sorting $\mathcal{G}$ can be done in time $O(p \lg p)$. Apart from these operations, Algorithm 1 mainly consists of two `foreach` loops, where the first loop prepares an ordered list of rescaling candidate tasks, which altogether can be done in time $O(n \lg n)$, and the second one traverses the candidate list and incrementally rescales the schedule, one task at a time, in each case updating the makespan times of the up to $\lg p$ groups affected by the rescaled task to recalculate *makespan*. Hence, the second loop takes time $O(n \lg p)$, and the entire Algorithm 1 runs in time $O((n+p)(\lg n + \lg p))$.

3.4. Optimal crown schedule adaptation

In order to evaluate the performance of our heuristic rescaling algorithm, we compute an energy-minimal schedule adaptation, as long as successful rescaling is at all possible. The following ILP computes a rescaled crown schedule minimizing energy consumption:

$$\begin{aligned}
\min\quad & E \\
\text{s.t}\quad & \forall j : \sum_{i \neq g(j),k} x_{i,j,k} = 0 \\
& \forall j : \sum_{k} x_{g(j),j,k} = 1 \\
& \forall j \notin mapped_S(offspring(g(\tau_d))) : \sum_{i,k \neq freq_S(j)} x_{i,j,k} = 0 \\
& \forall j \notin mapped_S(offspring(g(\tau_d))) : \sum_{i} x_{i,j,freq_S(j)} = 1 \\
& \forall j \in mapped_S(g(\tau_d)), j < d : \sum_{i,k \neq freq_S(j)} x_{i,j,k} = 0 \\
& \forall j \in mapped_S(g(\tau_d)), j < d : \sum_{i} x_{i,j,freq_S(j)} = 1 \\
& \forall l : \sum_{i \in G^{(l)},j,k} x_{i,j,k} \cdot t_j(p_i, f_k) \leq M.
\end{aligned}$$

Here, $G^{(l)}$ is the set of groups containing core l, $mapped_S(i)$ gives the tasks mapped to group i and $freq_S(\tau_j)$ the frequency level assigned to τ_j in the original crown schedule S. Furthermore, we assume that within each group, tasks are executed in order of task number (which effectively permits any task order by numbering the tasks accordingly). The first two types of constraints ensure that the mapping of tasks to groups is preserved. The next two types of constraints lock the frequencies at their current values for all tasks in irrelevant groups, i. e. those not belonging to $offspring(g(\tau_d))$. The following two types of constraints achieve the same for all tasks in $g(\tau_d)$ which are executed prior to τ_d. The constraints shown use $j < d$, i.e. ordering tasks according to increasing task ID. Similar to the heuristic, tasks can also be ordered according to frequency or workload, as those are also already specified by the underlying crown schedule.

```
input : deadline $M$, schedule $S$, delayed task $\tau_d$, $\alpha$, sorting orders
        A and B
output: a rescaled crown schedule $S'$
$S' \leftarrow S.clone()$ with task times updated given the delay $(1+\alpha)$ of
task $\tau_d$;
$t_{S'} \leftarrow makespan(S')$;
if $t_{S'} > M$ then
    $\mathcal{G} \leftarrow offspring(g(\tau_d))$;
    sort $\mathcal{G}$ by descending width; for same width, sort by
    descending annotation value;
    foreach group $G \in \mathcal{G}$ do
        if $G = g(\tau_d)$ then
            initialize list of suitable tasks $\mathcal{T}_d$ with tasks in
            $mapped_S(G)$ executed after $\tau_d$, sorted by order A;
        else
            append all tasks in $mapped_S(G)$ to $\mathcal{T}_d$ sorted by order B;
        end
    end
    $S'' \leftarrow S'.clone()$;
    foreach $\tau \in \mathcal{T}_d$ do
        if $freq_{S''}(\tau) = f_k < f_{max}$ then
            modify $S''$ by increasing $freq_{S''}(\tau)$ from $f_k$ to $f_{k+1}$;
            $t_{S''} \leftarrow makespan(S'')$;
            if $t_{S''} < t_{S'}$ then
                $S' \leftarrow S''$;
                $t_{S'} \leftarrow t_{S''}$;
                if $t_{S'} \leq M$ then
                    break;
                end
            end
            $S'' \leftarrow S'.clone()$;
        end
    end
end
```

Algorithm 1: Heuristic rescaling algorithm to accommodate the extended runtime of task τ_d. Sorting orders A and B can be according to increasing task ID, increasing or decreasing task frequency, or increasing or decreasing workload.

Finally, the last type of constraints ensures that no core is allocated more work than it can process until the deadline M. Note that $t_d()$ here uses the modified λ_d value for the delayed task τ_d.

If this optimization problem does not yield a feasible solution we thereby learn that a successful rescaling is impossible (and therefore our heuristic rescaling algorithm cannot succeed, either). On the other hand, if a feasible solution is found we know that rescaling can produce a schedule under which the deadline will be met. We can then judge the heuristic rescaler's performance by whether it encounters a feasible solution — and if so, we can compare the corresponding energy consumption values. Note that while the rescaling heuristic increases the frequency level of a task by at most one, the optimal rescaler may choose freely from the entire set of available frequency levels.

4. Robustness-aware crown scheduling

So far, we have identified and improved robustness starting from a *given* schedule in which no task was prolonged, and that minimized energy consumption while meeting the deadline. However, there may be scenarios where we target a robustness that cannot (in all cases) be achieved when starting from this schedule. In those scenarios, we might rather wish to have a schedule for the non-delayed case with a slightly higher energy consumption but that allows to achieve the targeted robustness.

[4] For incrementally recomputing *makespan*, we only need to update changed group times, not the absolute start- and finish times as in Fig. 6. Formally, we use a binary heap structure organized in 2 arrays *gtime*, *Gtime*, each with $2p-1$ floats, for bottom-up recomputing of *makespan*. Let $gtime(i)$ denote the time of all tasks mapped to (exactly) group i, for $i = 1, \ldots, 2p-1$. Then the makespan of all tasks in a group including its subgroups is inductively defined as $Gtime(i) = gtime(i) + \max(Gtime(2i), Gtime(2i+1))$ for non-leaf groups $i < p$, and $Gtime(i) = gtime(i)$ for leaf groups $i \geq p$. Hence, the makespan of the entire crown schedule is $makespan = Gtime(1)$. With these values in place, we only need, whenever the time of any task j changes, to update the $gtime$ of the group j it is mapped to, plus the $Gtime$ entries on the path "upwards the crown" from that group's index to the root group $i = 1$.

A simple means to get such a schedule is to reduce the deadline M to $M - C$ when computing the schedule for the non-prolonged case. Here C is computed from the maximum makespan $t_{S'}$ that occurs in case of any task prolongation, and the maximum makespan extension $\tilde{C}$ that shall be tolerated:

$$C = t_{S'} - M - \tilde{C} .$$

While this procedure is sufficient, and eliminates the need for any frequency re-scaling in case of a delayed task, it will typically not lead to the minimum energy with which this result can be achieved. Thus, it only serves as a bound on what is achievable.

A better approach is to consider robustness already when computing the crown schedule for the normal case. As a simple observation, we note that the crown schedule with minimum energy often is not unique. Thus one can first compute a crown-optimal schedule via ILP and thus learns the minimum energy E_{min} needed for the crown schedule of a given task set. Then one can compute the crown schedule with minimum energy *and* optimal robustness. This is done again via ILP using the energy E in a constraint $E \le E_{min}$, and by adding constraints

$$\forall l, d : \sum_{i \in G^{(l)}, j \neq d, k} x_{i,j,k} \cdot t_j(p_i, f_k) + \sum_{i \in G^{(l)}, k} x_{i,d,k} \cdot t_d(p_i, f_k)(1+\alpha) \le (1+\beta_d) \cdot M .$$

The target function to be minimized then is

$$B = \sum_j \beta_j / (\alpha \cdot n) ,$$

i.e., $B = 1 - \bar{R}_s(\alpha)$. By using an additional variable β, constraints $\forall d : \beta_d \le \beta$ and minimizing β, we can alternatively optimize worst case robustness.

If we do not insist on the minimum energy and use target function

$$\varepsilon \cdot E + (1 - \varepsilon) \cdot B ,$$

we can weigh additional energy in the normal case against improvement in robustness. For $\varepsilon \to 0$, we approach the situation sketched at the beginning, where all β_j will be 0, at the cost of a higher energy (which is still as low as possible).

For ε still close to 1, robustness is already improved by using a little more energy in the normal case. Yet, the efforts for frequency rescaling in case of the prolonged execution time of a task will also be reduced, but still be treated separately from the normal case.

As a further alternative, we might also give thresholds for $\max_j \beta_j$ and/or $\sum_j \beta_j / n$ and minimize energy.

Trading energy in the normal case against deadline extension or energy increase in a delay-case seems like a waste of energy, except in case of hard real-time deadlines. However, if we have $n = 10^3$ tasks and each task experiences a delay or prolongation with probability $p = 10^{-3}$, the chance of a round where no task experiences a delay is reduced to

$$(1-p)^n = \left(1 + \frac{-1}{10^3}\right)^{10^3} \approx e^{-1} \approx 36\% ,$$

as $(1 + c/x)^x \to e^c$. Thus, rounds where at least one task experiences a delay can become the majority. Still, a complete discussion of this topic is outside the scope of this paper.

Treating the energy for the normal case, the robustness in case of a delay and the effort for frequency rescaling together, i.e. C-elasticity, can also be achieved within a single ILP. In the following, we extend this notion to C_d-elasticity, i.e. the tolerable deadline transgression (after re-scaling) may depend on the delayed task d. We start with the ILP constraints of the original Crown scheduler, which model the delay-free case:

$$\begin{aligned} E &= \sum_{i,j,k} x_{i,j,k} \cdot power_j(f_k) \cdot p_i \cdot t_j(p_i, f_k) \\ &= \sum_{i,j,k} x_{i,j,k} \cdot power_j(f_k) \cdot p_i \cdot \lambda_j / (p_i \cdot e_j(p_i) \cdot f_k) \\ &= \sum_{i,j,k} x_{i,j,k} \cdot \lambda_j \cdot power_j(f_k) / (e_j(p_i) \cdot f_k) \end{aligned} \tag{3}$$

$$\text{for each task } j : \sum_{i,k} x_{i,j,k} = 1 \tag{4}$$

The next constraint, enforcing given maximum core allocations, may also be skipped if $e(p_i)$ is anyway close to 0 for $p_i > W_j$:

$$\text{for each task } j : \sum_{i:\ p_i > W_j,\ k} x_{i,j,k} = 0 \tag{5}$$

$$\text{for each core } l : \sum_{i \in G^{(l)}, j, k} x_{i,j,k} \cdot \lambda_j / (p_i \cdot e_j(p_i) \cdot f_k) \le M \tag{6}$$

We extend this ILP model for the case of single task delays by introducing new variables: $y_{i,j,k,d} = 1$ iff task j is assigned to group i at frequency k and task d is delayed (by a factor $(1+\alpha)$). Then the resulting energy for this case is:

$$\begin{aligned} E_d = & \sum_{i,\ j \neq d,\ k} y_{i,j,k,d} \cdot \lambda_j \cdot power_j(f_k) \ / \left(e_j(p_i) \cdot f_k\right) \\ & + \sum_{i,k} y_{i,d,k,d} \cdot \lambda_d \cdot (1+\alpha) \cdot power_d(f_k) / \left(e_d(p_i) \cdot f_k\right) \end{aligned} \tag{7}$$

and for the makespan in the delay case we add the following (depending on the choice of $C_d > 0$, relaxed) constraint

$$\begin{aligned} & \text{for each core } l, \text{ for each task } d : \\ & \sum_{i \in G^{(l)},\ j \neq d,\ k} y_{i,j,k,d} \cdot \lambda_j / (p_i \cdot e_j(p_i) \cdot f_k) \\ & + \sum_{i \in G^{(l)},\ k} y_{i,d,k,d} \cdot \lambda_d \cdot (1+\alpha) / \left(p_i \cdot e_d(p_i) \cdot f_k\right) \ \le \ M + C_d \end{aligned} \tag{8}$$

Here, the C_d model the tolerable deadline extension if task d is delayed and frequency re-scaling is applied. $C_d = 0$ models the case that no deadline extension is possible, and that the normal case must be adapted, i.e. accelerated, such that all delay cases will terminate execution of all tasks within the deadline, if frequency re-scaling is applied.

Then the optimization target is

$$\min \ \varepsilon \cdot E + (1 - \varepsilon) \cdot \sum_d E_d \tag{9}$$

Please note that the C_d could even be made variable and be part of the optimization target, instead of the E_d.

Hence, if ε is close to 1, then the schedule for the normal case will be close to optimal schedule. For larger ε we trade higher energy in the delay-free case for reduced energy in the delay cases. For a break-even average case, $(1-\varepsilon)\varepsilon$ should reflect the probability of execution rounds with single task delays in relation to rounds with no task delays.

As only the frequency may be changed, we can restrict

$$\text{for each } i, j, d : \quad \sum_k y_{i,j,k,d} = \sum_k x_{i,j,k} \tag{10}$$

then each task must be at the same group as before.

In order to forbid that tasks not in $scope(d)$, i.e., tasks that run before task d (including d) in d's group, or that are not in an offspring group of d's group, could change their frequency, we introduce binary variables $z_{j,d}$ where $z_{j,d} = 1$ iff task j runs in d's group after d, or in an offspring group. Then we require that

$$\text{for each group } i, \text{ task } j, \text{ freq. level } k, \text{ task } d : \quad y_{i,j,k,d} \le x_{i,j,k} + z_{j,d} \tag{11}$$

Hence, if j is not running in group i, then all $y_{i,j,*,*}$ are 0 because of Constraint (10). If j is running in group i, then a different frequency than in the normal case is only possible if $z_{j,d} = 1$. Constraint (10) still ensures that only one frequency is used.

First, we exclude tasks[5] $j \le d$ that are in the same group as d (and thus not running after d):

$$\text{for each } d,\ j \le d,\ i : \quad 1 - z_{j,d} \ \ge \ \sum_k \left(x_{i,j,k} + x_{i,d,k}\right) - 1 \tag{12}$$

[5] For simplicity of presentation we assume here again the default ordering of the tasks within each group by the task index. A generalization to different orderings within a group can be done for workload as described in Section 3.4.

i.e., if j and d are both running in i, then the right hand side will be 1, and thus $z_{j,d}$ is forced to 0.

Tasks $j > d$ in the same group are allowed to change frequency:

$$\text{for each } d, j > d, i: \quad z_{j,d} \geq \sum_k \left(x_{i,j,k} + x_{i,d,k}\right) - 1 \tag{13}$$

This constraint could be skipped for an optimal solution, but it is helpful for a solution after ILP solver timeout.

Now, exclude tasks j that are not in an offspring group of d's group:

$$\text{for each } d,\ i,\ i' \notin \mathit{offspring}(i),\ j: \quad 1 - z_{j,d} \geq \sum_k \left(x_{i,d,k} + x_{i',j,k}\right) - 1 \tag{14}$$

i.e., if d runs in group i and j in i', then the right hand side is 1, and thus $z_{j,d}$ is forced to 0.

Tasks in an offspring group of d's group i are allowed to change frequency:

$$\text{for each } d,\ i,\ i' \in \mathit{offspring}(i),\ j: z_{j,d} \geq \sum_k \left(x_{i,d,k} + x_{i',j,k}\right) - 1 \tag{15}$$

Introduction of the variables y with four indices considerably increases the number of variables in the ILP. Thus, it might only be usable for small n and p.

5. Evaluation

We have conducted extensive experiments in order to evaluate the various approaches to improving the robustness of schedules presented in Section 3. To give a short overview, the following experiments were performed, and will be addressed in detail below:

- compute a crown schedule's robustness and perform rescaling via ILP and heuristic,
- investigate the effect of task ordering within the delayed task's group (and its offspring groups in case of heuristic rescaling),
- jointly minimize energy consumption and optimize robustness,
- optimize robustness for a given (optimal) energy consumption,
- jointly minimize energy consumption for the non-delayed and delayed cases,
- jointly minimize energy consumption for the non-delayed case and makespan extension in the delayed cases,
- compute a crown schedule's robustness and perform rescaling via ILP and heuristic for real applications,.
- compare the examined rescaling techniques for larger task set and machine sizes.

Most of the experiments are based on 30 task sets of different sizes (8, 16, 32 tasks, 10 of each size). Schedules were computed for machine sizes of 8 and 16 cores. Each task's workload is an integer randomly chosen from $[1, 100]$, and its maximum width an integer randomly chosen from $[1, 16]$, both based on a uniform distribution. To compute the energy consumption, we have used real power consumption values from Holmbacka and Keller [3] for the ARM big.LITTLE architecture. So far, we assume a homogeneous machine, thus we have performed all computations with the power values for the big cores. As parallel efficiency function we have chosen from Melot et al. [6]

$$e_j(q) = \begin{cases} 1 & \text{for } q = 1, \\ 1 - 0.3\frac{q^2}{(W_j)^2} & \text{for } 1 < q \leq W_j, \\ 0.000001 & \text{for } q > W_j, \end{cases}$$

where τ_j is executed on q cores. The deadline M is determined similar to Melot et al. [6]:

$$M = \frac{\sum_j \frac{\lambda_j}{p \cdot f_{max}} + \sum_j \frac{\lambda_j}{p \cdot f_{min}}}{2}.$$

Table 1
Relative robustness and β values, grouped by α, averaged over all task sets and setups.

	α	$\beta_{\min}$	β_{avg}	$\beta_{\max}$	R_{avg}	$R_{\min}$
Average values	0.05	0.000	0.012	0.035	0.765	0.305
	0.10	0.000	0.030	0.077	0.695	0.234
	0.15	0.001	0.052	0.119	0.656	0.204
	0.20	0.003	0.074	0.163	0.629	0.187
	0.25	0.006	0.098	0.206	0.609	0.175
	Average				0.671	0.221
Standard deviation	0.05	0.000	0.005	0.011		
	0.10	0.001	0.010	0.017		
	0.15	0.003	0.014	0.024		
	0.20	0.009	0.018	0.030		
	0.25	0.012	0.023	0.038		

Here, f_{min} (f_{max}) is the minimum (maximum) processor operating frequency. In accordance with Holmbacka and Keller [3], we set $f_{min} = 0.6\,\text{GHz}$ and $f_{max} = 1.6\,\text{GHz}$.

For each of our experiments, we have implemented the respective tools in Python. For tools solving ILPs, we have employed the Gurobi 8.1.0 solver with the gurobipy Python module, and set a 5 min timeout. All computations have been executed on an AMD Ryzen 7 2700X processor with 8 physical cores and SMT.

5.1. Robustness and rescaling

To begin with, we have computed a regular crown schedule for each of the 30 task sets and recorded makespan as well as energy consumption. In a second step, we have determined the schedules' robustness for $\alpha \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$. Table 1 shows the values for the examined values of α averaged over all task set and machine sizes. As a general observation, robustness decreases with increasing α. The $\beta_{\min}$ are often close to zero, indicating that many times there are tasks whose delay does not cause a significant makespan extension. Average worst cases – characterized by $\beta_{\max}$ – see a makespan surpassing the deadline by up to $\frac{1}{5}$ for high values of α, whereas β_{avg} in these cases amounts to $\approx 10\%$. For small $\alpha \leq 0.1$, the makespan extension beyond the deadline is moderate over all examined setups: $\approx 3\%$ or less on average. Relative to α, the β_{avg} and $\beta_{\max}$ values increase with growing α, which is indicated by R_{avg} and $R_{\min}$. For the considered values of α, they average at 0.671 and 0.221, respectively. Even for worst cases, the execution is thus delayed by less than α and average relative robustness exceeds 0.6 for all values of α considered here. We also provide standard deviation values, broken down by α. It becomes immediately clear that dispersion is very low throughout, although it increases with growing α. When looking at relative robustness and β values grouped by machine and task set size (not shown in Table 1), one notices significantly better robustness values for $n = 32$ and $p = 8$ (e. g. avg. $\beta_{\text{avg}} = 0.061$ and avg. $\beta_{\max} = 0.136$ for $\alpha = 0.25$). This indicates that in setups with a high n/p ratio, one can expect higher robustness of crown schedules.

Finally, we have applied the heuristic rescaling algorithm as well as the ILP rescaler providing an optimal solution. The primary concern here is to determine in how many cases rescaling yields a feasible schedule, i. e. one which respects the deadline. The resulting energy overhead of the modified schedule over the original one when rescaling terminates successfully is another interesting aspect. What is more, we have applied a rescaler based on a genetic algorithm to facilitate comparability to established heuristic optimization techniques. Due to the nature of the problem at hand it is not advisable to permit unconstrained modification of solution candidates. In such a case, the vast majority of newly created individuals will represent infeasible solutions as either a task's frequency is modified despite actually being fixed, or the deadline is violated, or both. Therefore, the probability of an infeasible solution occurring must be lowered. This is done by

Table 2
Number of infeasible solutions and percentage of infeasible solutions for optimal, heuristic, and genetic algorithm rescalers, and comparison of rescalers regarding feasible solutions (values accumulated over all α values and task sets), grouped by machine and task set size.

# cores	# tasks	Optimal		Heuristic			Genetic algorithm		
		# inf.	% inf.	# inf.	% inf.	v opt.	# inf.	% inf.	v opt.
8	8	240	0.600	258	0.645	1.075	240	0.600	1.000
	16	409	0.511	492	0.615	1.203	409	0.511	1.000
	32	516	0.323	669	0.418	1.297	516	0.323	1.000
	Total	1165	0.416	1419	0.507	1.218	1165	0.416	1.000
16	8	216	0.540	228	0.570	1.056	216	0.540	1.000
	16	440	0.550	487	0.609	1.107	440	0.550	1.000
	32	893	0.558	1043	0.652	1.168	893	0.558	1.000
	Total	1549	0.553	1758	0.628	1.135	1549	0.553	1.000
Total		2714	0.485	3177	0.567	1.171	2714	0.485	1.000

Table 3
Number of infeasible solutions and percentage of infeasible solutions for optimal, heuristic, and genetic algorithm rescalers, and comparison of rescalers regarding feasible solutions (values accumulated over all machine sizes, task set sizes, and task sets), grouped by α value.

α	Optimal		Heuristic			Genetic algorithm		
	# inf.	% inf.	# inf.	% inf.	v opt.	# inf.	% inf.	v opt.
0.05	408	0.364	456	0.407	1.118	408	0.364	1.000
0.10	495	0.442	575	0.513	1.162	495	0.442	1.000
0.15	557	0.497	655	0.585	1.176	557	0.497	1.000
0.20	613	0.547	719	0.642	1.173	613	0.547	1.000
0.25	641	0.572	772	0.689	1.204	641	0.572	1.000
Total	2714	0.485	3177	0.567	1.171	2714	0.485	1.000

- designing the mutation operator such that only those frequencies can be modified which in fact may be modified,
- initializing the population not with random individuals but with the original solution,
- dispensing with a recombination operator.

Application of the mutation operator consists in setting a task's frequency level to a random level from the range of available frequency levels (based on a uniform distribution), for all tasks where frequency changes are permitted and with a given probability. The evaluation function computes the resulting schedule's energy consumption for a solution candidate, where infeasible schedules are assigned an arbitrary high value. An individual's fitness is the additive inverse of its evaluation result, amounting to an optimization towards minimal energy consumption among feasible schedules.

Since performing a hyperparameter optimization for the genetic algorithm is out of this paper's scope, we simply worked with two different sets of parameter values. For the second run, we significantly increased the mutation operator application probability as well as the probability of a single mutation (i. e. a frequency change for a single task) but lowered population size and number of generations. In the end, this led to more feasible solutions being found as well as a shorter runtime of the genetic algorithm. We have implemented the genetic algorithm with the DEAP framework [26].

Tables 2 and 3 display the number of infeasible solutions as well as the fraction of infeasible solutions for optimal, heuristic, and genetic algorithm rescalers. They also contain information about the number of infeasible solutions the heuristic and genetic algorithm rescalers produce in relation to the optimal rescaler. In Table 2, the values are grouped by machine and task set size, whereas Table 3 provides values grouped by α. Looking at Table 2, one can observe that the fraction of infeasible schedules from rescaling decreases with growing task set size for the setup with 8 cores, while the opposite effect arises for the 16-core setup. Furthermore, the smaller machine size shows a larger influence of task set size. The relative performance of the heuristic rescaler versus the optimal rescaler does not seem to correlate with machine size but improves for smaller task sets. Presumably, this is due to the lower number of tasks per group, which implies fewer opportunities to make up for a delay and prevents the optimal rescaler from capitalizing on its strengths. Interestingly, the relative performance figure is worst for the combination of 8 cores and 32 tasks — the setup which produced the best robustness values. The genetic algorithm rescaler shows a strong performance in terms of feasible solutions found, it matches the optimal rescaler's for all combinations of task set size and machine size.

Table 4
Average rescaling times for the three examined techniques.

# cores	Average rescaling time (ms)				
	Optimal	Heuristic	vs. opt.	Genetic algorithm	vs. opt.
8	31.71	0.39	0.01	251.43	7.93
16	65.98	0.50	0.01	315.69	4.78
Total	48.84	0.44	0.01	283.56	5.81

From Table 3 one can gather that the fraction of infeasible schedules obtained when rescaling grows with increasing α. This is not surprising since a greater delay requires a greater capacity for rescaling. The performance of the heuristic rescaler with regard to the optimal rescaler varies between 12% and 20% more infeasible solutions generated, with an average of 17%. The best relative performance of the heuristic rescaler can be observed for small values of α. As has already become clear from Table 2, the genetic algorithm rescaler performs on a par with the optimal rescaler.

When looking at the energy overhead rescaling causes compared to the original schedule one must take into account that a certain increase in energy consumption is to be ascribed to the delay and thus higher runtime of τ_d. Nonetheless, the energy overhead is very low ($< 5\%$ on average), so one can conclude that rescaling comes about in an energy-efficient manner. What is more, the differences in energy overhead between the heuristic and the optimal rescaler are minuscule (in cases where there are any at all). The same applies to the genetic algorithm rescaler.

Lastly, another important aspect regarding the rescalers is their execution time. If a task delay occurs unexpectedly at runtime, the rescaler must quickly deliver a result in order to avoid a deadline violation. Table 4 provides average rescaling times for the three examined techniques. In comparison to the optimal rescaler the genetic algorithm is roughly 5x slower on average, while the heuristic rescaler is about two orders of magnitude faster making it the prime choice in the envisioned scenario when round lengths are rather short. As we have seen, the significantly lower rescaling time comes at the expense of a lower count of feasible rescaling results. Interestingly, the machine size has a more pronounced impact on rescaling time for the optimal rescaler than for the two other methods. In Section 5.8, we will investigate whether this also holds on a larger scale.

5.2. Task ordering experiments

For both rescalers, task ordering may have an impact on results. Regarding the optimal rescaler computing an ILP, the execution order within the delayed task's group is important as tasks running prior to the delayed task cannot be executed at a higher frequency anymore. This consideration also applies to the heuristic rescaler. One further aspect here is how the list of suitable tasks is sorted. As detailed by Algorithm 1, we distinguish sorting of the suitable tasks within the delayed task's group from sorting of the suitable tasks in offspring groups of the delayed task's group. We have examined 5 different task orders: by index (representing an arbitrary task order), by ascending workload (wlasc), by descending workload (wldesc), by ascending frequency (freqasc), and by descending frequency (freqdesc). Table 5 provides information on the relative performance when assuming the

Table 5
Optimal rescaler number of infeasible solutions for different task orders relative to ordering by index. Data is based on figures for each combination of machine size, task set size, and α.

	freqasc	freqdesc	wlasc	wldesc
min.	0.89	0.93	0.73	0.85
avg.	1.02	1.00	0.94	1.06
max.	1.15	1.15	1.07	1.34

examined execution orders. The data shown here are computed over figures for each combination of machine size, task set size, and value of α. It can be gathered that ordering tasks by frequency (either ascending or descending) does not produce a significant effect on the number of cases where rescaling affords a feasible solution. Ordering tasks by workload on the other hand leads to noticeably more (or fewer) feasible solutions when attempting rescaling. One can conclude here that assuming an execution order by ascending workload will increase the chances of successful rescaling.

In order to evaluate sorting of tasks in the offspring groups of the delayed task's group for the rescaling heuristic, we compared for each combination of machine size, task set size, α, and execution order in the delayed task's group the number of infeasible solutions when sorting the tasks in the offspring groups by one of the 5 criteria. Sorting in the delayed task's group was always by execution order. Interestingly, for all 5 examined sorting criteria the average number of infeasible solutions was 1.16 relative to the optimal rescaler's number of infeasible solutions, with minimum values at 1.00 and maximum values at 1.74. A possible explanation could be that oftentimes, there are hardly any tasks in the delayed task's offspring groups and thus, the criterion by which they are sorted does not make any difference. The heuristic's most unfavorable relative performance values of around 1.7 occur for large n/p ratios, high α, and freqdesc or wldesc execution order in the delayed task's group. In absolute figures, the heuristic produced between 16 089 and 16 159 infeasible solutions (out of 28 000), depending on the sorting criterion for the offspring groups. While producing roughly 19% more infeasible solutions than the optimal rescaler with 13 580 infeasible solutions, rescaling can still be carried out successfully in $\approx 42\%$ of the examined cases (optimal rescaler: $\approx 52\%$).

To get an impression of the solution quality when rescaling is successfully performed, we have compared energy consumption computed for the execution of the resulting schedules. Of course, to facilitate a meaningful comparison, only cases where a feasible solution was found by both rescalers and for each offspring groups sorting criterion can be considered. Based on these cases, the rescaling heuristic achieves an energy consumption of 1.003 on average compared to the optimal rescaler, with worst case values of 1.061, regardless of the offspring groups sorting criterion. The takeaway here is that if the heuristic rescaler manages to encounter a feasible solution, it will most likely be on a par with the optimal rescaler's in terms of energy consumption.

5.3. Joint optimization of robustness and energy

When jointly minimizing energy consumption and robustness, both energy consumption and β values occur in the objective function, cf. Section 4. The importance of each can then be controlled by the ϵ parameter. In our experiments, we have determined average energy consumption and average β values for $\alpha \in \{0.05, 0.1\}$ and $\epsilon \in \{1, 0.9, 0.8, 0.7, 0.6, 0.5\}$. Considering that an ILP had to be solved for each combination of machine size, α, and ϵ for each of the 30 task sets, the total number of ILPs computed was 720. While not all of these could be solved to optimality prior to the 5 min timeout, the optimality gap was always < 0.02, so any solution is at least near-optimal. For each combination of machine size, α, and ϵ, we have computed average values over the 10 task sets for $\beta_{\min}$, β_{avg}, $\beta_{\max}$,

Table 6
Average values for $\beta_{\min}$, β_{avg}, $\beta_{\max}$, and energy consumption, relative to $\epsilon = 1$.

ϵ	$\beta_{\min}$	β_{avg}	$\beta_{\max}$	Energy consumption
0.9	0.010	0.533	0.759	1.003
0.8	0.003	0.302	0.553	1.011
0.7	0.000	0.190	0.413	1.016
0.6	0.000	0.131	0.322	1.020
0.5	0.000	0.095	0.250	1.023

Table 7
Average $\beta_{\min}$, β_{avg}, and $\beta_{\max}$ after minimizing energy consumption and after minimizing sum of betas for a given energy budget.

	avg. $\beta_{\min}$	avg. β_{avg}	avg. $\beta_{\max}$
min E	0.0070	0.0284	0.0632
min $\sum_j \beta_j$ for given E	0.0002	0.0209	0.0550
min $\sum_j \beta_j$ rel. to min E	0.0232	0.7354	0.8699

and energy consumption. We could then obtain these values relative to $\epsilon = 1$ and again average over all combinations of machine size, task set size, and α to determine how the approach enables us to trade energy efficiency for larger robustness. Table 6 displays the results. It becomes obvious that a small increase in energy consumption of less than 3% can substantially lower β values, with β_{avg} going down to 10% for $\epsilon = 0.5$ compared to $\epsilon = 1$, and $\beta_{\max}$ being reduced to 25%. Even for $\epsilon = 0.9$ and an increase in energy consumption by 0.3% one can bring down β_{avg} by $\approx 50\%$.

5.4. Optimization of robustness for given energy budget

As mentioned in Section 4, one may first minimize energy consumption and in a second step choose the solution yielding the best robustness among the energy-optimal ones. To this end, we constrain the total energy consumption to the minimal values obtained from the experiments presented in the preceding paragraph for $\epsilon = 1$ and minimize $\sum_j \beta_j$. Oftentimes, no feasible solution is found before the timeout occurs, therefore we provide the solver with the energy-minimal solution from the previous ILP as a starting point. That way, 77 out of 120 ILPs could be solved to optimality, while for the remaining 43 the largest optimality gap was less than 0.04. From Table 7 we can gather that $\beta_{\min}$ can be reduced to almost zero, while β_{avg} is lowered by $\approx \frac{1}{4}$ on average, and $\beta_{\max}$ is brought down to 87% at no additional energy cost. We can see that robustness may vary significantly among the energy-optimal solutions.

5.5. Joint minimization of energy consumption for non-delayed and delayed cases

Due to the high complexity of the ILP and the large number of parameters, we have performed these experiments with only one task set per size (8, 16, and 32 tasks, respectively). We have chosen $\alpha \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$, $\epsilon \in \{1.0, 0.9, 0.8, 0.7, 0.6, 0.5\}$, $C \in \{0.25\alpha, 0.5\alpha, 0.75\alpha\}$ with $C_d = C, d = 0, \ldots, n-1$. For each combination of task set, machine size, α, ϵ, and C an ILP was solved for a total of 540 ILPs. As it turns out, the solutions for $n = 32$ are far from the optimum (average optimality gap 0.24), they will thus be disregarded in the following. For the combination of $n = 8$, $p = 16$, $\alpha = 0.25$, and $C = 0.25\alpha$ there does not exist a feasible solution (this of course holds for any value of ϵ), so our analysis will focus on the remaining 354 solutions, 117 of which are optimal.

Table 8 shows both energy consumption for the delay-free case as well as energy consumption when a delay occurs, averaged over all tasks. The values displayed are averaged over all experiments for the respective task set, machine size, and ϵ. They are provided relative to the energy consumption of a schedule obtained via regular crown

Table 8
Energy consumption values for the delay-free case and energy consumption for the delayed cases (averaged over all tasks) relative to energy consumption for a regular crown schedule, averaged over all experiments for the respective ϵ value.

# tasks	# cores	ϵ	E delay-free case	avg. E delayed cases	avg. opt. gap
8	8	1.0	1.248		0.000
		0.9	1.254	1.327	0.069
		0.8	1.267	1.321	0.089
		0.7	1.265	1.320	0.095
		0.6	1.267	1.320	0.107
		0.5	1.269	1.320	0.109
	16	1.0	1.553		0.000
		0.9	1.554	1.612	0.038
		0.8	1.554	1.612	0.051
		0.7	1.555	1.611	0.055
		0.6	1.555	1.611	0.057
		0.5	1.555	1.611	0.062
16	8	1.0	1.000		0.000
		0.9	1.005	1.011	0.002
		0.8	1.010	1.015	0.003
		0.7	1.009	1.015	0.003
		0.6	1.009	1.015	0.004
		0.5	1.013	1.018	0.004
	16	1.0	1.013		0.017
		0.9	1.045	1.053	0.067
		0.8	1.061	1.067	0.085
		0.7	1.060	1.064	0.088
		0.6	1.065	1.070	0.094
		0.5	1.104	1.105	0.120

scheduling. It can be noticed immediately that the energy overhead with respect to a regular crown schedule heavily varies with task set and machine size. For $n = 16$, it hardly exceeds 10% even for the lowest ϵ examined, while for $n = 8$, it may reach $> 50\%$. In most cases, the average energy consumption for the delayed case is only marginally higher than for a situation where no delay occurs. Theoretically, average energy consumption for the delayed cases should drop with decreasing ϵ. Here, this does not always hold, probably due to the high complexity of the scheduling problem: the average optimality gap shows a tendency to grow with decreasing ϵ. Thus, while gradually shifting focus to the energy consumption under delays, the obtained solutions may actually be worse as they are further from the optimum. Presumably, extending the timeout would lead to better solutions and hence lower energy consumption when a task is delayed.

5.6. Joint minimization of energy for non-delayed and elasticity for delayed cases

The experimental setup here included the following parameter values:

- one task set per size (8, 16, and 32 tasks),
- $\alpha \in \{0.05, 0.1, 0.15, 0.2, 0.25\}$,
- $\epsilon \in \{1.0, 0.5, 0.1\}$,

with $\epsilon \cdot E + (1-\epsilon) \cdot C$ as objective function to be minimized. Again, no task-specific C_d were considered. The experimental results are provided in Table 9. For each combination of task set size, machine size, and ϵ, C as well as energy consumption for the delay-free case are provided, averaged over all α values examined here. In addition, the (averaged) optimality gap is provided. For $\epsilon = 1$, only energy consumption in the delay-free case is minimized, which should therefore be equal to the energy consumption for a regular crown schedule. As can be noted, this is not the case for $n = 32$ and for $n = 16, p = 16$. Here, an optimal solution could not be obtained prior to the timeout, for $n = 32, p = 8$, the solution quality is exceptionally bad and considerably worse than for the other ϵ values, when the optimization problem should be more complex. When the deadline extension C is considered in the objective function, i. e. for all $\epsilon < 1$, one notices that in many cases, substantially

Table 9
Energy consumption values for the delay-free case relative to energy consumption for a regular crown schedule and deadline extension C, averaged over all examined α values.

# tasks	# cores	ϵ	C	E delay-free case	opt. gap
8	8	1.0	2.000	1.000	0.000
		0.5	1.167	1.000	0.000
		0.1	0.665	1.008	0.000
	16	1.0	3.000	1.000	0.000
		0.5	1.674	1.001	0.000
		0.1	1.011	1.009	0.000
16	8	1.0	1.000	1.000	0.000
		0.5	0.025	1.000	0.000
		0.1	0.025	1.000	0.000
	16	1.0	1.000	1.013	0.018
		0.5	0.004	1.011	0.016
		0.1	0.007	1.008	0.014
32	8	1.0	1.000	1.291	0.226
		0.5	0.007	1.016	0.016
		0.1	0.001	1.020	0.020
	16	1.0	1.000	1.084	0.084
		0.5	0.004	1.141	0.130
		0.1	0.000	1.121	0.114

shorter deadline extensions can be achieved at the expense of very little energy overhead: for $n = 16$, one can almost match the original deadline for $< 1\%$ increase in energy consumption. For $n = 32$, a similar situation seems possible, although for $p = 16$, a comparable solution could not be found (the optimality gap is quite large here, so the existence of a significantly better solution is conceivable). Generally, for larger task set sizes and core numbers, the problem grows in complexity and cannot be solved (near-)optimally prior to the timeout. Another insight gained here is that one cannot make universal assumptions about specific ϵ values. While for $n \in \{16, 32\}$, $\epsilon = 0.5$ leads to schedules with virtually no extension over the original deadline, the deadline extension for $n = 8$ is still large, and even for $\epsilon = 0.1$, it does not drop to anywhere near zero.

5.7. Rescaling experiments with real applications

In addition to the experiments with synthetic task sets, we have conducted experiments with three real applications: mergesort, H.263 encode, and stereo depth estimation (SDE). The mergesort application consists of 15 tasks, each of which is executed sequentially. The corresponding task graph is shown in Fig. 9. The SDE application can be found in [27]. It reads two images, converts them to grayscale, applies sobel filtering, computes the sum of absolute differences, and writes an output image. Fig. 10 displays the task graph. The H.263 encode application is included in the Dataflow Benchmark Suite (DFBench) [28]. It is composed of 9 tasks and 16 communication channels as depicted in Fig. 11. We had to prune edges in the task graph in order to obtain an acyclic graph here. Due to the small number of tasks in the SDE and H.263 applications, we have performed the rescaling experiments for a 4-core machine.

Table 10 presents relative robustness and β values for the applications under consideration. We can see that on average, robustness figures are better than for the synthetic task sets (average $R_{\text{avg}} = 0.671$, average $R_{\min} = 0.221$) for all three applications. The highest robustness can be observed for the H.263 encoder application. As can be expected, robustness again decreases for growing α, with the exception of $R_{\min}$ remaining constant for the mergesort application.

From Table 11 we can gather that the number of infeasible solutions when rescaling is significantly lower than for the synthetic task sets (48.5% infeasible solutions for the optimal rescaler, 56.7% for the heuristic). What is more, the relative performance of the heuristic rescaler varies to a high extent: for the SDE application, it performs on a par with the optimal rescaler, while it produces nearly double (or

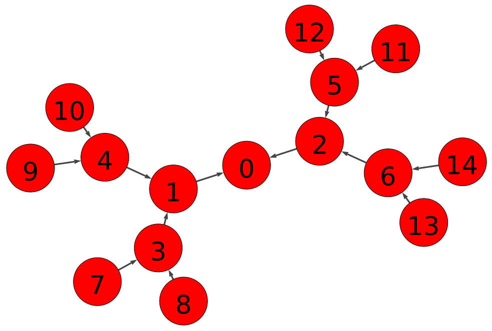

Fig. 9. Task graph for the mergesort application.

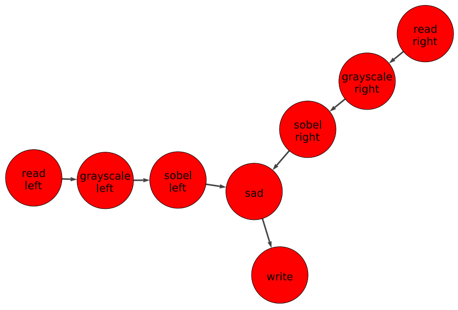

Fig. 10. Task graph for the SDE application.

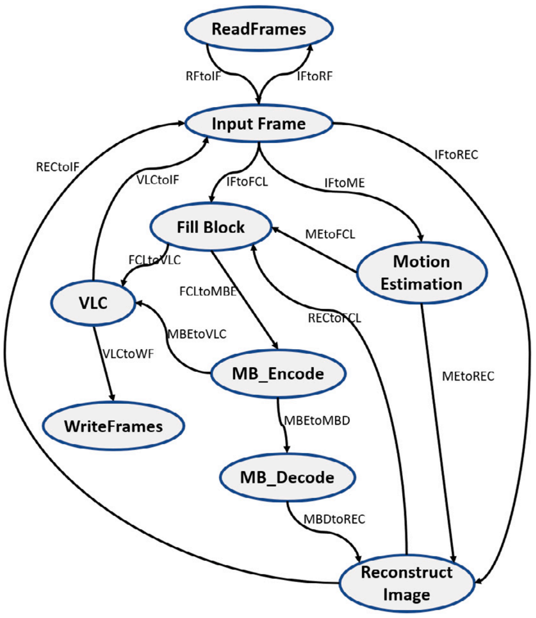

Fig. 11. Task graph for the H.263 encode application.

even more) infeasible solutions for the other two applications. All in all, it seems to depend on the concrete application to a high degree whether rescaling can be successfully performed in the majority of cases.

Finally, Table 12 provides the average and maximum energy overhead for the rescalers' schedules with regard to a regular crown schedule. Average overhead values are very similar to the figures obtained for the synthetic task sets (1.6%–2.2% overhead, depending on machine size and rescaler, cf. Table 18). Maximum energy overhead was 14.2%–38% for synthetic task sets, whereas for the real applications, we do not observe an overhead of $> 10\%$, and again the values differ among the three applications with SDE featuring the lowest overhead.

Table 10
Relative robustness and β values for the three examined applications.

Application	α	β_{min}	β_{avg}	β_{max}	R_{avg}	R_{min}
mergesort	0.05	0	0.007	0.036	0.867	0.273
	0.1	0	0.014	0.073	0.861	0.273
	0.15	0	0.022	0.109	0.851	0.273
	0.2	0	0.032	0.145	0.839	0.273
	0.25	0	0.042	0.182	0.830	0.273
	Average				0.849	0.273
SDE	0.05	0	0.006	0.023	0.884	0.536
	0.1	0	0.020	0.068	0.803	0.317
	0.15	0	0.036	0.113	0.758	0.243
	0.2	0	0.053	0.159	0.735	0.207
	0.25	0	0.070	0.204	0.721	0.185
	Average				0.780	0.298
H.263 encode	0.05	0	0.003	0.024	0.946	0.516
	0.1	0	0.007	0.058	0.927	0.417
	0.15	0	0.012	0.092	0.917	0.384
	0.2	0	0.018	0.127	0.911	0.367
	0.25	0	0.027	0.161	0.891	0.357
	Average				0.918	0.408

Table 11
Number of infeasible solutions and percentage of infeasible solutions for both rescalers, and comparison of rescalers regarding feasible solutions for the three considered applications.

Application	α	Optimal		Heuristic		rel. to opt.
		# inf.	% inf.	# inf.	% inf.	
mergesort	0.05	2	0.133	2	0.133	1.00
	0.1	2	0.133	4	0.267	2.00
	0.15	2	0.133	6	0.400	3.00
	0.2	4	0.267	6	0.400	1.50
	0.25	4	0.267	7	0.467	1.75
	Total	14	0.187	25	0.333	1.79
SDE	0.05	2	0.250	2	0.250	1.00
	0.1	3	0.375	3	0.375	1.00
	0.15	3	0.375	3	0.375	1.00
	0.2	3	0.375	3	0.375	1.00
	0.25	3	0.375	3	0.375	1.00
	Total	14	0.350	14	0.350	1.00
H.263 encode	0.05	0	0.000	0	0.000	
	0.1	0	0.000	0	0.000	
	0.15	0	0.000	1	0.111	
	0.2	0	0.000	2	0.222	
	0.25	3	0.333	5	0.556	1.67
	Total	3	0.067	8	0.178	2.67

Table 12
Average and maximum energy overhead over a regular crown schedule for both rescalers and all three considered applications.

Application	Optimal rescaler		Heuristic rescaler	
	avg. E	max. E	avg. E	max. E
mergesort	1.019	1.098	1.011	1.053
SDE	1.006	1.024	1.006	1.024
H.263 encode	1.017	1.090	1.015	1.069

5.8. Rescaling experiments with larger task set and machine sizes

In this section, our intention is to observe the rescaling techniques' behavior when increasing task set and machine sizes. To this end, we have created 10 synthetic task sets with 64 tasks and another 10 task sets with 128 tasks. This time, the tasks' maximum width is a randomly chosen integer from $[1, 64]$, based on a uniform distribution. Scheduling was performed for machine sizes of 32 and 64 cores. Everything else remained as described at the beginning of Section 5. Again, three rescalers were considered: optimal rescaler solving an ILP, heuristic rescaler, and genetic algorithm rescaler.

Table 13
Number of infeasible solutions and percentage of infeasible solutions for optimal, heuristic, and genetic algorithm rescalers, and comparison of rescalers regarding feasible solutions (values accumulated over all α values and task sets), grouped by machine and task set size.

# cores	# tasks	Optimal		Heuristic			Genetic algorithm		
		# inf.	% inf.	# inf.	% inf.	v opt.	# inf.	% inf.	v opt.
32	64	1747	0.546	2045	0.639	1.171	1749	0.547	1.001
	128	1899	0.297	2457	0.384	1.294	1899	0.297	1.000
	Total	3646	0.380	4502	0.469	1.235	3648	0.380	1.001
64	64	1835	0.573	2064	0.645	1.125	1835	0.573	1.000
	128	3437	0.537	4035	0.630	1.174	3437	0.537	1.000
	Total	5272	0.549	6099	0.635	1.157	5272	0.549	1.000
Total		8918	0.464	10601	0.552	1.189	8920	0.465	1.000

Table 14
Percentage of infeasible solutions for the optimal and heuristic rescalers, for $p = 8$ and various n/p values.

n/p	1	2	4	8	16
% inf. optimal	0.600	0.511	0.323	0.154	0.076
% inf. heuristic	0.645	0.615	0.418	0.210	0.114

Table 15
Average rescaling times for the three examined techniques.

# cores	Average rescaling time (ms)				
	Optimal	Heuristic	vs. opt.	Genetic algorithm	vs. opt.
32	639.34	1.75	0.003	1272.01	1.990
64	1415.51	2.64	0.002	1899.86	1.342
Total	1027.42	2.19	0.002	1585.93	1.544

As shown in Table 13, the results are very similar to those for smaller task set and machine sizes. For most combinations, the optimal rescaler delivers between 50% and 60% infeasible solutions, an exception being the scenario where we have $n = 128$, $p = 32$. Here, the n/p ratio is the largest one examined in our experiments so far, and the optimal rescaler achieves $\approx$ 70% feasible solutions. For the smaller task set and machine sizes, a larger n/p ratio yielded significantly better results as well. The heuristic rescaler again performs a bit worse in terms of feasible solutions found ($\approx$ 19%), while there are only two cases where the optimal rescaler discovers a feasible solution and the genetic algorithm rescaler does not.

In order to further study the influence of the n/p ratio on solution feasibility we have performed additional experiments with $p = 8$ for the larger task sets. Table 14 displays the percentage of infeasible solutions for the optimal and heuristic rescalers and various n/p values. Apparently, both techniques produce even fewer infeasible solutions for the larger n/p values, as is to be expected.

Table 15 provides the rescaling times for all three techniques. It becomes clear that the heuristic's relative runtime advantage has increased ($\approx$ 0.2% compared to the optimal rescaler vs. $\approx$ 1% for the smaller task set and machine sizes). The genetic algorithm rescaler also gains on the optimal rescaler, and although still being slower, there is a clear tendency to a more moderate runtime increase with growing machine size.

All in all, the experiments in the current section have delivered plausible results. With regard to the fraction of feasible solutions, the n/p ratio is the crucial factor: a larger ratio implies a higher potential to compensate for delays due to the larger number of tasks being considered on average. The absolute number of tasks or cores on the other hand is of little relevance. When looking at rescaler runtimes, it is easy to conceive that expensive techniques such as ILP solving struggle with growing problem complexity to a greater extent than heuristic procedures.

Table 16
Relative robustness and β values, grouped by α, averaged over all task sets and setups for the heterogeneous environment.

	α	$\beta_{\min}$	β_{avg}	$\beta_{\max}$	R_{avg}	$R_{\min}$
Average values	0.05	0.000	0.010	0.033	0.806	0.334
	0.10	0.000	0.026	0.074	0.739	0.263
	0.15	0.001	0.045	0.115	0.699	0.232
	0.20	0.001	0.066	0.158	0.671	0.212
	0.25	0.004	0.088	0.201	0.650	0.198
	Average				0.713	0.248

6. Heterogeneous platforms and robustness

So far, we have assumed that all cores of the machine to be used are identical. However, many processors nowadays are heterogeneous, i.e. they comprise several types of cores that differ in power profile and execution speed. For example, ARM's big.LITTLE architecture comprises the big cores which are optimized for execution speed by a superscalar microarchitecture, at the price of higher power consumption, and the LITTLE cores which are optimized for power and energy efficiency by a simpler microarchitecture, at the price of a reduced execution speed compared to a big core for the same code at the same frequency. All cores can be frequency-scaled.

Use of a combination of two core types gives a scheduler further options to arrange tasks in time and speed, and thus allows a better energy consumption while meeting the same deadline. Similarly, we expect that the robustness will profit from a heterogeneous platform.

The crown scheduler is suitable for heterogeneous architectures [4], and both the presented re-scaling heuristic and the ILPs from Section 3 work as well for heterogeneous platforms.

To test our hypothesis, we performed experiments considering a heterogeneous system assuming a machine modeled after the ARM big.LITTLE architecture with as many big as LITTLE cores. The required power consumption values were again adopted from [3], where one can also find execution time multipliers to accommodate for the LITTLE cores' lower computational capabilities. We now set $f_{max} = 1.4\,\text{GHz}$, again cf. [3]. It should be noted that there does not exist a feasible solution in all cases for $n = 8, p = 16$ under the chosen parameter settings. Table 16 presents relative robustness and β values for all examined α, averaged over all task sets and setups. In comparison to the homogeneous environment, cf. Table 1, we can notice a slightly higher robustness throughout. While we had overall robustness values of $R_{\text{avg}} = 0.671$ and $R_{\min} = 0.221$ for a homogeneous machine, we now obtain $R_{\text{avg}} = 0.713$ and $R_{\min} = 0.248$. Dispersion (not shown here) is mostly lower than for the homogeneous environment. In 83% of the cases, $\beta_{\min}$ was less than or equal to the value for the homogeneous setting (71% for β_{avg}, 63% for $\beta_{\max}$). Apparently, heterogeneity can serve as a means of advancing robustness.

From Table 17 one can gather that the trend to more infeasible solutions for higher values of α unsurprisingly also applies to the heterogeneous environment. What is more, for each value of α and both rescalers, fewer feasible solutions are encountered than in the homogeneous scenario. Thus, while having a positive impact on robustness, heterogeneity at the same seems to impede rescaling. The relative performance of the heuristic compared to the optimal rescaler is better for the heterogeneous system.

Finally, we have computed the average energy overhead for the schedule after rescaling compared to the original crown schedule. Of course, this can only be done for the instances of successful rescaling, i. e., where a feasible solution exists. Table 18 shows the number of feasible solutions and energy overhead values for both environments. As already mentioned, rescaling terminates successfully more often in the homogeneous scenario. For $p = 16$, we also have a lower energy overhead on average and for worst cases. For $p = 8$, the energy overhead in the heterogeneous scenario is a bit lower on average, and

Table 17
Number of infeasible solutions and percentage of infeasible solutions for both rescalers, and comparison of rescalers regarding feasible solutions (values accumulated over all machine sizes, task set sizes, and task sets), grouped by α value for the heterogeneous environment.

α	Optimal		Heuristic		rel. to opt.
	# inf.	% inf.	# inf.	% inf.	
0.05	518	0.463	550	0.491	1.062
0.10	648	0.579	691	0.617	1.066
0.15	714	0.638	765	0.683	1.071
0.20	763	0.681	815	0.728	1.068
0.25	799	0.713	858	0.766	1.074
Total	3442	0.615	3679	0.657	1.069

Table 18
Comparison of number of feasible solutions and energy overhead when applying rescaling for the homogeneous and for the heterogeneous environment.

Environment	p	Rescaler	# feas. sol.	avg. E overhead	max. E overhead
Homogeneous	8	opt	1635	1.022	1.380
		heur	1381	1.022	1.132
	16	opt	1251	1.018	1.142
		heur	1042	1.016	1.144
Heterogeneous	8	opt	1070	1.015	1.113
		heur	933	1.016	1.095
	16	opt	1008	1.086	1.589
		heur	908	1.081	1.589

worst case figures are better, especially for the optimal rescaler. It is not surprising that the optimal rescaler can produce a larger energy overhead since it covers more cases: when the heuristic rescaler cannot find a feasible solution, the optimal rescaler may do so but at a possibly large energy overhead.

7. Conclusions

We have investigated the robustness and elasticity of crown schedules, i. e. static schedules for parallelizable tasks on parallel machines with frequency scaling, given a deadline and minimizing energy consumption. We demonstrated with synthetic benchmark tasksets that a runtime increase of one task by a fraction α leads to a makespan exceeding the deadline by a fraction 0.319α on average and 0.779α at maximum (values averaged over all task sets and α values considered). This is much better than a transgression by a fraction α, which is much more likely for other schedulers of parallelizable tasks, such as Sanders and Speck [2].

Furthermore, we presented a heuristic to compensate for this increase by running other tasks on a higher frequency level. The heuristic's runtime is very short, so that it can be applied during the schedule's execution. We evaluated the heuristic by comparison with the optimal solution featuring minimum additional energy consumption. To compute this minimum, we presented an integer linear program. We find that for the synthetic benchmark tasksets, the heuristic causes minuscule additional energy consumption with regard to the optimum solution, and finds a feasible solution in $\approx 84\%$ of the cases where one exists.

We also presented a generalized ILP model for robustness-aware crown scheduling that allows to trade some energy efficiency for improved robustness of the generated schedule, and evaluated it with synthetic benchmarks. The results show that a minor increase in energy consumption of well below 5% can lead to considerably better robustness values. Further experiments have demonstrated that among the energy-optimal schedules, robustness may vary to some extent, making a subsequent optimization of robustness seem attractive.

Future work will comprise to investigate further variants of the rescaling heuristic, and to use this heuristic as part of a scheduler that solves the problems of task allocation, mapping and frequency scaling in subsequent steps. Furthermore, the comparison could be extended to crown schedules computed by heuristics [6], not only crown schedules computed by an ILP. Finally, an experimental comparison which includes the re-scaling overhead at runtime and the power consumption of idle cores will be performed.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] C. Kessler, S. Litzinger, J. Keller, Robustness and energy-elasticity of crown schedules for sets of parallelizable tasks on many-core systems with DVFS, in: Proc. 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP'20, Västerås, Sweden, 2020, IEEE, 2020, pp. 136–143.
[2] P. Sanders, J. Speck, Energy efficient frequency scaling and scheduling for malleable tasks, in: Euro-Par 2012 Parallel Processing, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 167–178.
[3] S. Holmbacka, J. Keller, Workload type-aware scheduling on big.LITTLE platforms, in: Algorithms and Architectures for Parallel Processing, Springer Int. Publ., 2017, pp. 3–17.
[4] S. Litzinger, J. Keller, C.W. Kessler, Scheduling moldable parallel streaming tasks on heterogeneous platforms with frequency scaling, in: 27th European Signal Processing Conference, EUSIPCO 2019, A Coruña, Spain, September 2–6, 2019, IEEE, 2019, pp. 1–5, http://dx.doi.org/10.23919/EUSIPCO.2019.8903180.
[5] M.I. Gordon, W. Thies, S.P. Amarasinghe, Exploiting coarse-grained task, data, and pipeline parallelism in stream programs, in: J.P. Shen, M. Martonosi (Eds.), Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21–25, 2006, ACM, 2006, pp. 151–162, http://dx.doi.org/10.1145/1168857.1168877.
[6] N. Melot, C. Kessler, J. Keller, P. Eitschberger, Fast Crown scheduling heuristics for energy-efficient mapping and scaling of moldable streaming tasks on manycore systems, ACM Trans. Archit. Code Optim. 11 (2015) 62:1–62:24.
[7] J. Yu, R. Buyya, A taxonomy of workflow management systems for grid computing, J. Grid Comput. (2005) 171–200.
[8] G. Kahn, The semantics of a simple language for parallel programming, in: Proc. IFIP Congress on Information Processing, North-Holland, 1974, pp. 471–475.
[9] E.A. Lee, D.G. Messerschmitt, Static scheduling of synchronous data flow programs for digital signal processing, IEEE Trans. Comput. 36 (1987) 24–35, http://dx.doi.org/10.1109/TC.1987.5009446.
[10] J. Boutellier, O. Silvén, M. Raulet, Scheduling of CAL actor networks based on dynamic code analysis, in: Proc. IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011, IEEE, 2011, pp. 1609–1612, http://dx.doi.org/10.1109/ICASSP.2011.5946805.
[11] J. Blazewicz, M. Machowiak, J. Weglarz, M.Y. Kovalyov, D. Trystram, Scheduling malleable tasks on parallel processors to minimize the makespan, Ann. Oper. Res. 129 (2004) 65–80, http://dx.doi.org/10.1023/B:ANOR.0000030682.25673.c0.
[12] K. Li, Energy efficient scheduling of parallel tasks on multiprocessor computers, J. Supercomput. 60 (2012) 223–247, http://dx.doi.org/10.1007/s11227-010-0416-0.
[13] H. Xu, F. Kong, Q. Deng, Energy minimizing for parallel real-time tasks based on level-packing, in: 2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, 2012, pp. 98–103.
[14] Y.-K. Kwok, I. Ahmad, Static scheduling algorithms for allocating directed task graphs to multiprocessors, ACM Comput. Surv. 31 (1999) 406–471.
[15] V. Jorge Leon, S.D. Wu, R.H. Storer, Robustness measures and robust scheduling for job shops, IIE Trans. 26 (1994) 32–43.
[16] L. Bölöni, D.C. Marinescu, Robust scheduling of metaprograms, J. Sched. 5 (2002) 395–412.
[17] A.F. Mills, J.H. Anderson, A multiprocessor server-based scheduler for soft real-time tasks with stochastic execution demand, in: 17th IEEE Int.l Conf. Embedded and Real-Time Computing Systems and Applications, 2011, pp. 207–217, http://dx.doi.org/10.1109/RTCSA.2011.30.
[18] L.-C. Canon, E. Jeannot, Evaluation and optimization of the robustness of DAG schedules in heterogeneous environments, IEEE Trans. Parallel Distrib. Syst. 21 (2010) 532–546.
[19] S. Manolache, P. Eles, Z. Peng, Schedulability analysis of applications with stochastic task execution times, ACM Trans. Embedded Comput. Syst. 3 (2004) 706–735.

[20] S. Adyanthaya, Z. Zhang, M. Geilen, J. Voeten, T. Basten, R. Schiffelers, Robustness analysis of multiprocessor schedules, in: Proc. Int. Conf. on Embedded Computer Systems: Architecture, Modeling, and Simulation, SAMOS-XIV, 2014, pp. 9–17.
[21] S. Adyanthaya, M. Geilen, T. Basten, J. Voeten, R. Schiffelers, Iterative robust multiprocessor scheduling, in: Proc. RTNS, 2015, pp. 23–32.
[22] M. Lombardi, M. Milano, L. Benini, Robust scheduling of task graphs under execution time uncertainty, IEEE Trans. Comput. 62 (2013) 98–111.
[23] S. Srinivasan, S. Krishnamoorthy, P. Sadayappan, A robust scheduling strategy for moldable scheduling of parallel jobs, in: Proc. Int. Conf. on Cluster Computing, CLUSTER'03, IEEE, 2003, pp. 92–99.
[24] C.W. Kessler, S. Litzinger, J. Keller, Adaptive crown scheduling for streaming tasks on many-core systems with discrete DVFS, in: U. Schwardmann, C. Boehme, D.B. Heras, V. Cardellini, E. Jeannot, A. Salis, C. Schifanella, R.R. Manumachu, D. Schwamborn, L. Ricci, O. Sangyoon, T. Gruber, L. Antonelli, S.L. Scott (Eds.), Euro-Par 2019: Parallel Processing Workshops - Euro-Par 2019 International Workshops, Göttingen, Germany, August 26–30, 2019, in: Lecture Notes in Computer Science, vol. 11997, Springer, 2019, pp. 17–29, http://dx.doi.org/10.1007/978-3-030-48340-1_2, Revised Selected Papers.
[25] J. Keller, S. Litzinger, W. Spitzer, Probabilistic runtime guarantees for statically scheduled taskgraphs with stochastic task runtimes, in: Proc. 17th International Conference on High Performance Computing & Simulation, HPCS 2019, IEEE, 2019.
[26] F.-A. Fortin, F.-M. De Rainville, M.-A. Gardner, M. Parizeau, C. Gagné, DEAP: Evolutionary algorithms made easy, J. Mach. Learn. Res. 13 (2012) 2171–2175.
[27] I. Hautala, From Dataflow Models to Energy Efficient Application Specific Processors (Ph.D. thesis), Oulu University, Finland, 2019.
[28] DFbench, Dataflow benchmark suite (DFbench), 2010, http://www.es.ele.tue.nl/dfbench/.

Christoph Kessler received the Ph.D. degree in computer science from Universität des Saarlandes, Saarbrücken, Germany, in 1994.

He is a Professor at the Department of Computer and Information Science (IDA) of Linköping University, Linköping, Sweden.

His main research interests are in the areas of parallel computing and compilers, especially models and frameworks for high-level parallel programming, program parallelization, optimization and code generation.

Sebastian Litzinger received a master's degree in philosophy from University of Tübingen and a master's degree in computer science from FernUniversität in Hagen, Germany.

Currently, he is a PhD student at the Parallelism & VLSI research group at FernUniversität in Hagen.

His research interests are energy-efficient task scheduling for parallel machines, the application of machine learning techniques to task scheduling, and neural architecture search.

Jörg Keller received the Ph.D. degree in computer science from Universität des Saarlandes, Saarbrücken, Germany, in 1992.

He is a professor at the Faculty of Mathematics and Computer Science of FernUniversität in Hagen, Germany.

His research interests include energy-efficient parallel computing, security and cryptography and fault tolerant computing.