B. Juurlink et al. (Hrsg.): PARS-Workshop,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2019 1

Influence of Discretization of Frequencies and Processor Allocation on Static Scheduling of Parallelizable Tasks with Deadlines

Sebastian Litzinger, Jörg Keller[1]

Abstract: Models for energy-efficient static scheduling of parallelizable tasks with deadlines on frequency-scalable parallel machines comprise moldable vs. malleable tasks and continuous vs. discrete frequency levels. We investigate the tradeoff between scheduling time and energy efficiency when going from continuous to discrete processor allocation and frequency levels. To this end, we present a tool to convert a schedule computed for malleable tasks on machines with continuous frequency scaling (P. Sanders, J. Speck, Euro-Par 2012) into one for moldable tasks on a machine with discrete frequency levels. We compare the energy efficiency of the converted schedule to the energy consumed by a schedule produced by the integrated crown scheduler (N. Melot et al., ACM TACO 2015) for moldable tasks and a machine with discrete frequency levels. Our experiments indicate that the converted Sanders Speck schedules, while computed faster, consume more energy on average than crown schedules. Surprisingly, it is not the step from malleable to moldable tasks that is responsible, but the step from continuous to discrete frequency levels.

Keywords: Static Scheduling; Frequency Scaling; Energy-efficient Schedules

1 Introduction

Repetitive tasks with deadlines often occur in embedded multicore systems, for example in streaming applications where each task is activated in a scheduling round [Me15]. The length of the round determines the throughput, such as the number of images that can be fed to an image processing application per second, and at the same time poses a deadline until which all the tasks must or should be executed, depending on the hardness of the deadline. While each task feeds output to its follow-up tasks according to the streaming task graph, this input can be considered to be transferred from one scheduling round to the next, so that the task invocations in one round are independent of each other.

If the application has a longer runtime or is frequently executed, it pays off to find a static schedule to execute the tasks within a round with minimum frequencies in order to lower the energy consumption for the given throughput. A static schedule typically has lower runtime overhead and at the same time better quality than a dynamic schedule, given that

[1] FernUniversität in Hagen, Faculty of Mathematics and Computer Science, 58084 Hagen, Germany first.last@fernuni-hagen.de

the task executions are predictable, which is the case in the above scenario. Minimizing energy consumption is of tremendous importance in embedded systems, as lower energy might mean a housing without a fan and thus lower production and maintenance cost. At least it means a lower bill from reduced energy supply and reduced cooling.

An interesting sub-case of this static scheduling scenario are parallelizable tasks, i. e. tasks that are parallel programs themselves. This is advantageous e. g. when the number of cores is larger than the number of tasks. Several approaches have proposed solutions to this problem, however starting from different assumptions. On the one hand, Sanders and Speck [SS12] have proposed an algorithm that computes a schedule under the assumption that the frequency can assume an arbitrary value, and not only a finite number of discrete levels, and under the assumption that the degree of parallelism of a task can vary during its execution (so-called malleable tasks), i. e. a task allocated to 5.3 cores runs on 6 cores for 30% of the time till the deadline, and on 5 cores for the remaining time. Finally, they assume that a (sequential) task can be interrupted and continued later on. On the other hand, crown scheduling [Me15] assumes that only a finite number of discrete frequency levels is available, that a task can only be allocated to an integral number of cores[2] (so-called moldable tasks) and that a task is not interrupted by another task once its execution has started. There are other differences as well (the Sanders Speck scheduler poses some restrictions on the power and speedup functions, while the crown scheduler allows arbitrary power and speedup profiles) which we will ignore here.

In the current work, we are interested in the trade-off between these two extremes. In particular, we investigate the hypothesis that to schedule a set of moldable tasks, we might first use the Sanders Speck scheduler as if the tasks were malleable, and convert the schedule into one for moldable tasks, remove preemption of sequential tasks, and step over to discrete frequencies. For all these steps, we observe how much the energy consumption of the resulting schedule increases, and compare the final result with the energy consumption of a crown schedule. The Sanders Speck scheduler (even including the converter) has a shorter runtime than crown scheduling, which solves a mixed-integer linear program. Our hypothesis, which we will test by experiments, is that a converted Sanders Speck schedule has a higher energy consumption than a crown schedule, so the user can trade scheduling time for energy consumption. In addition, by doing a sequence of conversion steps we can see which of the model differences has most influence, so that one may perform research on that difference in the future.

Surprisingly, the main difference does not seem to lie in the contrast between malleable and moldable tasks, but in the step from continuous to discrete frequencies. This might call for a reconsideration of the requirement that the discrete frequency level of a task need not be changed during task execution. [EK15] demonstrate that for tasks with sufficient runtime, a non-existing frequency might be simulated by using each of the two existing surrounding

[2] Crown Scheduling further assumes that a task can only be allocated to p processors where p is a power of 2, but it turned out [Me19] that this restriction only leads to slightly higher energy compared to requiring an integral number of cores.

discrete frequency levels for part of the execution time, with only a small increase in energy consumption.

The remainder of this article is structured as follows. In Section 2, we briefly review background information on energy-efficient scheduling and related work. In Section 3, we describe both schedulers used, present the routine to convert a Sanders Speck schedule for malleable tasks into a schedule for moldable tasks, and indicate how we discretize the frequency levels. In Section 4, we report on the experiments with which we test the above hypothesis, and in Section 5, we summarize and give an outlook to our future work.

2 Background

We consider the problem of scheduling a set of parallelizable tasks $T = \{t_1, \ldots, t_n\}$ to a set of homogeneous processors $P = \{P_1, \ldots, P_p\}$, where we can scale the frequency for each core independently. A task t_j is characterized by its workload τ_j, which might represent the number of processor cycles necessary to execute the task on one core, and its maximum width W_j, determining the maximum number of cores t_j can be executed on concurrently. A task-specific parallel efficiency function $e_j(q), 1 \leq q \leq p$, is defined as the speedup when running the task on q processors, divided by q, thus depending on the number of cores to which the task is allocated.[3]

If a task t_j is allocated to a noninteger number of processors – as is done by the Sanders Speck scheduler, which assumes tasks to be malleable – the expression $w_j + \gamma_j$ gives the total number of processors for t_j, w_j being an integer value and $\gamma_j \in [0, 1)$.

We compute a static schedule, i. e. we schedule prior to the actual execution. The schedule allocates each task to a set of processor cores and assigns an execution frequency and a start date. The schedule must be feasible, i. e. no core is ever allocated to more than one task at the same time. Furthermore, execution of the task set shall terminate before reaching a deadline M.

When a processor core is running at a frequency f, it draws electrical power $P(f)$. In the most general sense, P is non-decreasing in f. Next to the frequency, P depends on a number of other factors, such as the supply voltage, the instruction mix of the currently executed code and the temperature. We assume that voltage is always set to the minimum level possible for each frequency (and nowadays a single voltage level often serves many frequency levels), so the influence of voltage on power consumption is covered by the frequency parameter. The instruction mix is considered to be uniform for all tasks (but might be extended, cf. [HK17, LKK19]), and the temperature is assumed to be controlled to remain constant.

A task with workload τ that runs on q cores has a workload of $\frac{\tau}{q \cdot e(q)}$ on each core. Normally, it is assumed to run at one frequency f for its whole execution on q cores, so that the power

[3] The concept of maximum width is therefore introduced solely for convenience, as one could simply set $e_j(q) = 0$ for $q > W_j$.

consumption remains constant during the execution. The runtime $t(q)$ can be obtained by dividing core workload (number of cycles) by frequency (number of cycles per time unit). The energy consumption can be obtained by multiplying runtime and power consumption. The energy consumption of a schedule is the sum of the tasks' energy consumptions.

We employ a simple energy model, assuming dynamic power to be (proportional to) f^α [SS12]. Here, f denotes the processor's current operating frequency, and α is a constant depending on the actual hardware. To facilitate comparability between the different scheduling approaches, we assume that frequency scaling does not produce any time or energy overhead, and static power consumption as well as idle power are ignored here. The energy consumed by a processor executing a task on frequency f over a period M is thus $f^\alpha \cdot M$.

While literature on task scheduling is vast, two approaches are of particular interest for our current purposes. In [SS12], a static scheduler for malleable tasks is presented, which allows continuous frequency scaling and seeks to minimize energy consumption while meeting a set deadline. The Crown scheduler introduced in [Me15] is a static scheduler for moldable tasks. It also aims for minimization of energy consumption under deadline constraints, which is achieved via integer linear programming (ILP). Processor allocation, mapping, and frequency scaling are either performed separately or in a combined manner, promising further energy savings. Due to restrictions regarding allocation, mapping, and execution sequence, Crown scheduling allows for solving medium-sized problems by means of linear programming. The Crown scheduling technique is expanded in [MKK16] when static and idle power are taken into consideration and the concept of *core consolidation* is explored. In [XKD12], parallel tasks with deadlines and discrete frequencies are treated. Scheduling is performed via a *level-packing* approach, and for minimization of energy consumption, a 0-1 ILP is contrasted with a three-step heuristic consisting of the specification of each task's width, task scheduling, and frequency assignment.

3 Scheduling Approaches for Parallelizable Tasks

In this section, Sanders Speck scheduling is outlined in 3.1. Subsection 3.2 shows how to convert a Sanders Speck schedule, and 3.3 offers a recap of the crown scheduling technique.

3.1 Sanders Speck Scheduling

Sanders and Speck [SS12] assume that the tasks are malleable, i. e. that the scheduler can vary the number of cores used during execution of a task. For example, a task might be run on 4 cores in the time interval $[0; 0.3 \cdot M]$ and run on 5 cores in the time interval $[0.3 \cdot M; M]$. The number of allocated cores (also called the width) for task j is therefore[4]

[4] They prove that in their setting, other variations than using q and $q + 1$ cores for a task do not occur.

given as $w_j + \gamma_j$, where $w_j = 4$ and $\gamma_j = 0.3$ for this example. There are restrictions on the efficiency function, which according to their analysis seem to be met by many parallel algorithms.

The cores can be scaled to an arbitrary, continuous frequency $f \geq 0$, and the power consumption of a core is f^α, where typically $2 \leq \alpha \leq 3$. Adding a constant amount of static power is ignored, as it does not change the allocations that achieve minimum energy, as long as the cores are not switched off. Thus, transformations can be used to morph realistic frequency ranges into the dimensionless space used here and morph back to power consumption values for real processor architectures.

The algorithm given by Sanders and Speck computes a processor allocation $w_j + \gamma_j$ for each task j, such that $\sum_j (w_j + \gamma_j) = p$, the total energy spent in the computation is minimum, and the task set is executed until the deadline. From the allocation and the given parallel efficiency function, they are able to derive the operating frequency for each task. The mapping is as follows: a task with $w_j \geq 1$ gets w_j cores for the complete time till the deadline. All the γ_j parts, i. e. the times where a task gets an extra core, and the sequential tasks (where $w_j = 0$) are mapped to the remaining $p - \sum_j w_j$ cores by a wrap-around rule: A core is filled with these tasks. When it is allocated for a fraction δ till the deadline, and $\delta + \gamma_j > 1$, then this task gets $1 - \delta$ of the time on this core, and $\gamma_j + \delta - 1$ on the next core. This leads to another requirement: it must be possible to stop a task and continue it later on a different core.

To illustrate the mapping procedure, Figure 1 provides an example mapping of 3 tasks to 6 cores, with processor allocations of 3.2 (t_1, orange), 0.9 (t_2, green), and 1.9 (t_3, pink). Since $w_j > 1$ for the orange and pink tasks, these receive 3 and 1 cores, respectively, for the whole execution time. The sequentially executed green task is not considered at this point. As there are 6 cores in total, the γ_j are distributed across the remaining 2 cores in the next step. The orange task receives core 5 for 20% of the execution time. The green task, which is allocated 0.9 cores overall, is mapped to core 5 for the remaining 80% of total execution time and – by wrap-around – to core 6 for 10%. As one can gather from Figure 1, we now require preemption. The pink task is run on core 6 after t_2 to reach its total allocation of 1.9 cores. For the orange and pink tasks, we assume malleability since their width changes from 4 to 3 (orange) and 1 to 2 (pink) during the course of their execution.

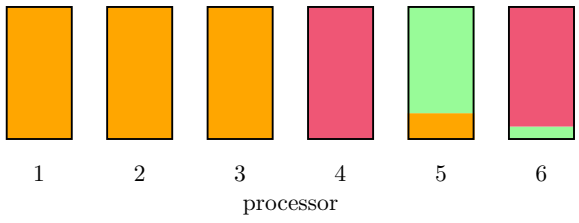

Fig. 1: Example mapping of 3 tasks to 6 cores

3.2 Converting a Sanders Speck Schedule

Converting a Sanders Speck schedule starts with moving from malleable to moldable tasks. Under this restriction, a task can either be executed in parallel on a fixed integer number of processors (i. e. $w_j > 1, \gamma_j = 0.0$), or sequentially on a single core (i. e. $w_j = 1, \gamma_j = 0.0$ or $w_j = 0, 0 < \gamma_j < 1$).

In order to achieve this, one first splits the task set T into $T_P = \{t_j \in T : w_j + \gamma_j > 1\}$ and $T_S = T \setminus T_P$. In the first step, only tasks in T_P are considered. To begin with, we determine the sum of the parallel tasks' processor allocations under the Sanders Speck schedule, $\pi_{total} = \sum_{t_j \in T_P} w_j + \gamma_j$, as well as the number of processors solely dedicated to a single parallel task, $\pi_{single} = \sum_{t_j \in T_P} w_j$. Computing $u = nint(\pi_{total}) - \pi_{single}$ gives you the number of cases in which processor allocation shall be rounded up, where $nint()$ signifies the nearest integer function.[5] One now proceeds as follows: The tasks in T_P are sorted by descending γ_j. Then, for the first u tasks in that order, set $w_j = w_j + 1, \gamma_j = 0.0$. For the remaining $|T_P| - u$ parallel tasks, set $\gamma_j = 0.0$. The processor allocations for parallel tasks now are integer values, and the total amount of cores utilized is $nint(\pi_{total})$. On a side note, it may be the case that a task which would be executed in parallel under the Sanders Speck schedule runs sequentially under the converted schedule.

Going back to the example from Section 3.1, we have $T_P = \{t_1, t_3\}$ and $T_S = \{t_2\}$ since only the green task is executed sequentially. We get $\pi_{total} = w_1 + \gamma_1 + w_3 + \gamma_3 = 3 + 0.2 + 1 + 0.9 = 5.1$ and $\pi_{single} = w_1 + w_3 = 3 + 1 = 4$, which yields $u = nint(\pi_{total}) - \pi_{single} = nint(5.1) - 4 = 5 - 4 = 1$. Sorting the tasks in T_P by descending γ_j gives us t_3, t_1. We now set $w_j = w_j + 1$ and $\gamma_j = 0.0$ for the first u tasks in that order, i. e., for t_3 we assign $w_3 = 2$, $\gamma_3 = 0.0$. For the remaining $|T_P| - u$ parallel tasks, we set $\gamma_j = 0.0$, i. e., for t_1 we now have $\gamma_1 = 0.0$ (and w_1 stays at 3).

The next step is to compute the sequential tasks' processor allocations. The number of cores available for the execution of sequential tasks is $p - nint(\pi_{total})$. The tasks in T_S are now mapped to the remaining processors via a binpacking approach. To this end, they are sorted by descending τ_j and subsequently are assigned to the (at the respective time) least occupied bin, i. e. core.[6] Note that bin size can easily be computed as $f_{max} \cdot M$, which represents the maximum workload a processor can handle up to the deadline M running on its maximum operating frequency f_{max}. After the binning step, one can immediately compute the processor allocation for each task $t_j \in T_S$: If a task t_j is the only one running on a given processor, set $w_j = 1, \gamma_j = 0.0$. Otherwise, a task is allocated a fraction of a

[5] Naturally, u could be computed differently, e. g. one could set $u = \lceil \pi_{total} \rceil - \pi_{single}$ or $u = \lfloor \pi_{total} \rfloor - \pi_{single}$, or try both ways and see which resulting schedule yields lower energy consumption. In any case, as long as there are sequential tasks, it has to be ensured that there is at least one core left for the execution of sequential tasks, i. e. if $\lceil \pi_{total} \rceil = p$, one must set $u = \lfloor \pi_{total} \rfloor - \pi_{single}$.

[6] As the processor's operating frequency for each task is not carried over from the Sanders Speck schedule but is computed anew after processor allocation has been performed, it cannot serve as a sorting criterion. Consequently, a task's workload is used to best represent its size. We expect mapping the sequential tasks to processors in order of descending workload to lead to a reasonable load balancing.

processor corresponding to its share of the processor's total workload, $\gamma_j = \frac{\tau_j}{\sum_{t_k \in T_i} \tau_k}$, where T_i denotes the set of tasks to be executed on processor P_i, and $t_j \in T_i$. For these tasks, set $w_j = 0$.

In our example from Section 3.1, there is just one sequential task and we have $p - nint(\pi_{total}) = 6 - 5 = 1$. Thus, t_2 is allocated an entire core, which gives us $w_2 = 1, \gamma_2 = 0.0$ under the conversion. Figure 2 shows the resulting mapping after conversion. As all parallel tasks' allocations are now integer values, malleability is not required anymore. Beyond that, the mapping of any sequential task to a single core, where it is executed in one go, renders preemption expendable.

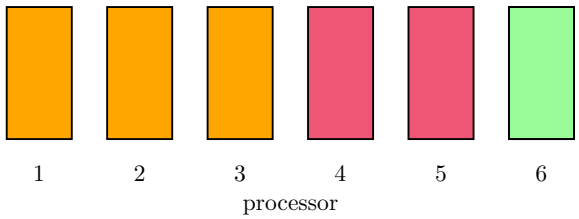

Fig. 2: Mapping after applying the conversion procedure to the example depicted in Figure 1

As processor allocation is now completed for all $t_j \in T$, one can compute the frequency for each task as

$$F_j = \begin{cases} \frac{\tau_j}{\gamma_j \cdot M} & \text{if } w_j = 0, \\ \frac{\tau_j}{e_j(w_j) \cdot w_j \cdot M} & \text{if } w_j \geq 1. \end{cases}$$

This allows calculation of the energy consumption for each $t_j \in T$ (note that $w_j = 0 \Leftrightarrow \gamma_j \neq 0$):

$$E_j = F_j^{\alpha} \cdot (w_j + \gamma_j) \cdot M.$$

Now that processor allocation has been carried out, frequency discretization can be employed in order to obtain energy consumption values under the further restriction that cores feature a set of discrete frequency levels $F = \{f_1, \ldots, f_s\}$ they can run on. We further assume that a core's operating frequency can be changed only between task executions. As before, we first consider T_P: Here, frequency discretization is fairly easy to perform: Since each $t_j \in T_P$ is the only task allocated to its respective processor(s), one has no choice but to increase frequency to the closest frequency level: $F_j = \min\{f \in F : f > F_j\}$. Lowering F_j to the closest frequency level $f < F_j$ instead would incur a deadline violation.

For T_S, frequency discretization does not necessarily imply increasing the operating frequency for each $t_j \in T_S$ to the next possible value. On the contrary, one should aim for reducing the frequency for as many tasks as possible so as to improve energy consumption. To facilitate this, in a first step all $t_j \in T_S$ are treated as described above for parallel tasks: F_j is increased to $\min\{f \in F : f > F_j\}$. Afterwards, for each processor P_i executing tasks

from T_S, the operating frequency is decreased by one level, task by task, until the deadline cannot be met, starting with the last task mapped to P_i (which is the least bulky one due to the binpacking performed beforehand). The last assignment of frequencies to tasks adhering to the deadline is then adopted.

If one allows frequency scaling during task execution, energy consumption can be reduced since a given frequency F_j can be simulated by running on $f_{l_j} = \max\{f \in F : f < F_j\}$ for $c \cdot M$ and on $f_{h_j} = \min\{f \in F : f > F_j\}$ for $(1-c) \cdot M$, $c \in [0,1]$. This procedure forms a generalization of the above approach, where frequency scaling is performed this way for all $t \in T_P$ with $c = 0$. Choosing c individually for all $t \in T$ on the other hand will most likely have a positive impact on energy consumption. This is done as follows:

$$c_j = -\frac{F_j - f_{h_j}}{f_{h_j} - f_{l_j}}.$$

The calculation of a task's energy consumption in this scenario is then pretty straightforward:

$$E_j = (f_{l_j}^{\alpha} \cdot c_j + f_{h_j}^{\alpha} \cdot (1 - c_j)) \cdot (w_j + \gamma_j) \cdot M.$$

3.3 Crown Scheduling

A different static scheduling approach for moldable tasks with discrete frequencies is the *crown scheduler* [Ke13, MKK16]. Here, we consider the *integrated* version, where processor allocation, mapping, and frequency scaling is performed in a combined fashion by solving an integer linear program (ILP). In order to ease computation, the P_i are assigned to a hierarchy of processor groups as in Figure 3: The largest group, P^1, comprises all processors, which is then divided into two equally sized subgroups. These subgroups each are divided into two equally sized subgroups, and so on, with the smallest groups containing one processor only. This group structure can be applied for core counts which are powers of 2, and the resulting number of groups is $2p - 1$. Tasks are then mapped to processor groups, which run at a specified frequency for the duration of a task's execution.

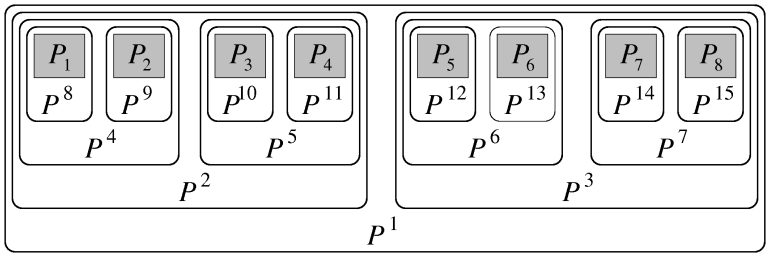

Fig. 3: Processor groups for $p = 8$, taken from [Ke13]

The optimization problem therefore yields $n \cdot (2p-1) \cdot s$ binary decision variables $x_{j,i,k}$, $x_{j,i,k} = 1$ signifying that task t_j shall be executed on processor group P^i operating at frequency f_k. The target function to be minimized computes the total energy consumption

$$E = \sum_{j,i,k} x_{j,i,k} \cdot \frac{\tau_j \cdot f_k^{\alpha-1}}{e_j(p_i)}.$$

The term p_i denotes the number of cores in processor group P^i. Several constraints apply in order to guarantee a valid schedule. First, every task shall be scheduled only once:

$$\forall j : \sum_{i,k} x_{j,i,k} = 1.$$

Furthermore, to ensure the deadline is met, the total runtime of all tasks mapped to a core P_l must not exceed the deadline:

$$\forall l : \sum_{j} \sum_{i \in G_l} \sum_{k} x_{j,i,k} \cdot \frac{\tau_j}{p_i \cdot f_k \cdot e_j(p_i)} \leq M,$$

where G_l denotes the set of all groups core P_l belongs to. Finally, the maximum number of cores allocated to a task t_j shall be its maximum width W_j:

$$\forall j : \sum_{i, p_i > W_j} \sum_{k} x_{j,i,k} = 0.$$

Applying an ILP solver to the optimization problem then yields a mapping of tasks to processor groups as well as the respective processors' operating frequencies.

4 Experiments

In this section, we compare the resulting energy consumption when scheduling task sets via the various approaches presented in Section 3. Our experiments are based on synthetic task sets of varying cardinality and tasks' maximum widths as in [Ke13, MKK16]: A task set comprises 10, 20, 40, or 80 tasks and displays a low ($W_j \in \{1, \ldots, p/2\}$), average ($W_j \in \{p/4, \ldots, 3p/4\}$), high ($W_j \in \{p/2, \ldots, p\}$), or maximum ($\forall j W_j = p$) degree of parallelism[7], or it contains sequential tasks only ($\forall j W_j = 1$). For each combination of cardinality and degree of parallelism, 10 different task sets are considered, thus yielding a total of 200 different task sets for the evaluation of the previously introduced scheduling techniques.

[7] All W_j for low, average, and high degrees of parallelism are determined randomly based on a uniform distribution.

The number of cores is set to 32, and the set of discrete frequencies is $\{1.0, 2.0, 3.0, 4.0, 5.0\}$. Furthermore, $\alpha = 3.0$ and the parallel efficiency of task t_j is defined as in [Ke13]:

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

The parameter q is the number of cores t_j is executed on. The deadline M is determined as in [Ke13]:

$$M = \frac{\sum_j \frac{\tau_j}{p \cdot f_{max}} + 2 \sum_j \frac{\tau_j}{p \cdot f_{min}}}{2},$$

where f_{min} and f_{max} are the processors' minimum and maximum operating frequencies.

Energy consumption values are computed for (cf. Figure 4 for the visualization of results):

- the Sanders Speck schedule (reference),
- the Sanders Speck schedule converted to moldable tasks without preemption (blue bar),
- the converted schedule with discrete frequencies (purple bar),
- the converted schedule with discrete frequencies and one-time frequency scaling during task execution (pink bar),
- the crown schedule (yellow bar).

We deployed a C implementation of the Sanders Speck scheduler, a Python implementation of the converter tool, and for crown scheduling, the Gurobi 8.1.0 solver was adopted in conjunction with the gurobipy module for Python. The Sanders Speck scheduler as well as the converter tool were executed sequentially on an AMD Ryzen 7 2700X, while the Gurobi solver ran in 16 threads on 8 cores with a 5 minute timeout for each ILP. It took the Sanders Speck scheduler ≈ 0.3 s to compute schedules for the 200 synthetic task sets, and the conversion process lasted another ≈ 56 s, while the crown scheduler required ≈ 366 min to deliver the results.[8] It should be noted though that the time to solve the ILPs varied heavily among the task sets: Roughly half the schedules could be computed in < 1 s, $\approx 87\%$ in < 10 s, $\approx 92\%$ in < 1 min, and 4 ILPs could not be solved to optimality until the 5 minute timeout occurred.

From Figure 4 it becomes clear that moving from malleable to moldable tasks has a minor impact on energy consumption over all task set types and cardinalities. Subsequent frequency

[8] The execution times given are the sums of user and system times, while the timeout refers to real (wall clock) time.

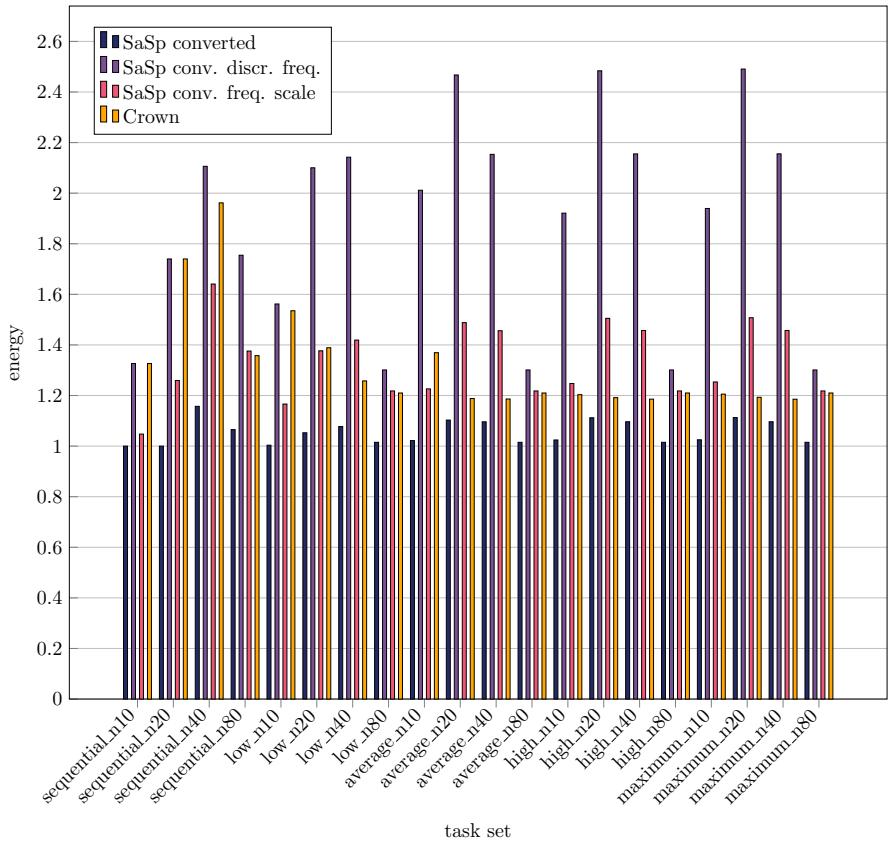

Fig. 4: Energy consumption for different scheduling techniques and synthetic task sets relative to energy consumption for the Sanders Speck scheduler

discretization on the other hand leads to a considerable increase in energy consumption, up to 2.5x compared to the energy consumption of a corresponding Sanders Speck schedule. The effect of frequency discretization is particularly severe for high degrees of parallelism. This is due to the discretization strategy, which permits scaling to lower frequencies for sequentially executed tasks but enforces moving to a higher frequency for tasks running in parallel. The significantly lower relative energy consumption values for high cardinality task sets over all degrees of parallelism support this explanation, since despite their potential for parallel execution, many tasks will have to be run sequentially due to the large number of tasks.

It can also be observed that in many cases the converted Sanders Speck schedule with discretized frequencies leads to a substantial growth in energy consumption over a crown schedule – the exceptions being high cardinality and low maximum width task sets. This is

unfortunate, as both the converter and the crown scheduler essentially operate based on the same constraints. In some scenarios, the former might be preferable though, as producing a converted Sanders Speck schedule is less computationally intensive than creating a crown schedule.

A good deal of the negative effect of frequency discretization can be mitigated when frequency scaling during task execution is permitted. In many cases, the resulting schedule performs better or as good as a crown schedule. It must be pointed out though that the additional overhead of frequency scaling is not reflected in the current findings.

As a final observation, which is not afforded by Figure 4, the absolute energy consumption values hardly differ when the degree of parallelism exceeds the average category. This applies to all scheduling techniques. Here, a preliminary conjecture would be that deadline constraints prevent exploitation of the higher potential for parallelism.

5 Conclusions

We have presented a tool that converts a schedule for malleable tasks on a machine with continuous frequency scaling into one for moldable tasks on a machine with discrete frequency levels. By applying this converter to schedules computed by the Sanders Speck scheduler, we could demonstrate the tradeoff between scheduling time (lower for converted Sanders Speck schedules) and energy-efficiency (better for crown schedules). The average scheduling times are 0.28 s vs. 110 s, while the average energy consumption is $\approx 28\%$ higher for converted Sanders Speck schedules. By doing the conversion in two steps, we see that the crucial point is not the switch from malleable to moldable tasks, but the change from continuous to discrete frequency levels. Hence it might be worthwhile to investigate in future research the influence of frequency switch during task execution, which would allow to "simulate" a continuous frequency f by running a task partly on surrounding discrete frequency levels f_1 and f_2 with $f_1 < f < f_2$.

Acknowledgments

We are very grateful to Christoph Kessler for inspiring our line of research and providing helpful comments.

References

[EK15] Eitschberger, Patrick; Keller, Jörg: Energy-Efficient Task Scheduling in Manycore Processors with Frequency Scaling Overhead. In: Proc. 23rd Euromicro Int. Conf. Parallel, Distributed, and Network-Based Processing (PDP 2015). pp. 541–548, 2015.

[HK17] Holmbacka, Simon; Keller, Jörg: Workload Type-Aware Scheduling on big.LITTLE Platforms. In (Ibrahim, Shadi; Choo, Kim-Kwang Raymond; Yan, Zheng; Pedrycz, Witold, eds): Algorithms and Architectures for Parallel Processing. Springer International Publishing, Cham, pp. 3–17, 2017.

[Ke13] Kessler, Christoph W.; Melot, Nicolas; Eitschberger, Patrick; Keller, Jörg: Crown scheduling: Energy-efficient resource allocation, mapping and discrete frequency scaling for collections of malleable streaming tasks. In: 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation. pp. 215–222, 2013.

[LKK19] Litzinger, S.; Keller, J.; Kessler, C.: Scheduling Moldable Parallel Streaming Tasks on Heterogeneous Platforms with Frequency Scaling. In: Proc. 27th European Signal Processing Conference (EUSIPCO 2019). To appear September 2019.

[Me15] Melot, Nicolas; Kessler, Christoph; Keller, Jörg; Eitschberger, Patrick: Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Manycore Systems. ACM Trans. Archit. Code Optim., 11(4):62:1–62:24, 2015.

[Me19] Melot, Nicolas; Kessler, Christoph; Eitschberger, Patrick; Keller, Jörg: Co-optimizing Core Allocation, Mapping and DVFS in Streaming Programs with Moldable Tasks for Energy Efficient Execution on Manycore Architectures. In: Proc. 19th International Conference on Application of Concurrency to System Design (ACSD 2019). To appear 2019.

[MKK16] Melot, Nicolas; Kessler, Christoph W.; Keller, Jörg: Improving Energy-Efficiency of Static Schedules by Core Consolidation and Switching Off Unused Cores. In: Parallel Computing: On the Road to Exascale (Proc. ParCo 2015). pp. 285–294, 2016.

[SS12] Sanders, Peter; Speck, Jochen: Energy Efficient Frequency Scaling and Scheduling for Malleable Tasks. In (Kaklamanis, Christos; Papatheodorou, Theodore; Spirakis, Paul G., eds): Euro-Par 2012 Parallel Processing. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 167–178, 2012.

[XKD12] Xu, H.; Kong, F.; Deng, Q.: 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. pp. 98–103, 2012.