Journal of Signal Processing Systems (2022) 94:849–864
https://doi.org/10.1007/s11265-022-01787-y

Integrating Energy-Optimizing Scheduling of Moldable Streaming Tasks with Design Space Exploration for Multiple Core Types on Configurable Platforms

Jörg Keller[1] · Sebastian Litzinger[1] · Christoph Kessler[2]

Received: 2 October 2021 / Revised: 4 February 2022 / Accepted: 19 June 2022 / Published online: 30 June 2022

Abstract
Design space exploration of a configurable, heterogeneous system for a given application with required throughput searches for a combination of cores or softcores with different architectures that can be accommodated within the given ASIC or FPGA area and that achieves the required throughput and optimizes power consumption. For a soft real-time streaming application, modeled as a task graph with internally parallelizable streaming tasks, this requires assigning a core type and quantity and DVFS frequency level to each task, which implies task runtime and energy consumption, and mapping and scheduling the tasks, such that the throughput requirement is met. We tightly integrate such static scheduling for stream processing applications with design space exploration of the best heterogeneous core combination, and solve the resulting combined optimization problem by an integer linear program (ILP). We evaluate our solution for different numbers of core types on synthetic and application-based task graphs, and demonstrate improvements of up to 34.8% for ARM big and LITTLE cores, and 70.5% for 3 different core types.

Keywords Design space exploration · Task scheduling · Energy efficiency

1 Introduction

Data stream processing is an important computation paradigm in embedded (and edge) computing, where a continuous stream of data elements coming from data sensors should, due to its high volume and velocity, be processed as close to the data source as possible. An example is the preprocessing of continuously arriving camera raw data or vehicle sensor data. Such devices are often constrained in power usage (e.g., battery driven and/or using passive cooling only), therefore the throughput requirements often can only be fulfilled by using a heterogeneous multicore platform offering also a type of cores with a special architecture optimized for low power consumption. For a configurable platform such as an application-specific integrated circuit (ASIC) or a field-programmable gate array (FPGA), also the number of each type of (soft) core can (and must) be determined during design space exploration (DSE), to find a collection of cores that for the given area and throughput constraints optimizes the power consumption.

Stream processing programs are usually expressed as a graph of persistent streaming tasks that read packets of data from their input channels, process one packet at a time, and write a packet of output data to output channels, thus forwarding it to data consumer tasks or to the program's result channel(s). By providing sufficient FIFO buffering capacity along all channels (thus following the Kahn Process Network model [2]), the streaming program execution can be software-pipelined such that all instances of streaming tasks for different data packets in the same *round* in the steady state of the pipeline are independent and thus can execute concurrently (see Fig. 1). On a many-core system, these can then be scheduled to different cores or core groups so that the makespan for one round of the steady-state loop is kept

This article is an extended version of [1].

✉ Sebastian Litzinger
sebastian.litzinger@fernuni-hagen.de

Jörg Keller
jorg.keller@fernuni-hagen.de

Christoph Kessler
christoph.kessler@liu.se

[1] FernUniversität in Hagen, Hagen, Germany

[2] Linköping University, Linköping, Sweden

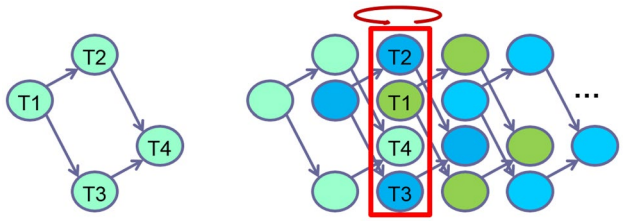

Figure 1 Left: A streaming task graph with four streaming tasks. Right: The red box shows the steady state of the software-pipelined streaming program execution, where all task instances in one round are independent, i.e. belong to different instances of the streaming task graph or are independent tasks from one task graph instance.

low enough to meet the throughput requirement and the workload is well balanced. Streaming tasks perform a certain amount of work per input packet and can be internally parallel, i.e., run on multiple cores to speed up one instance of their execution. For example, a *moldable* task can use any number of cores that must be determined before the task is executed, in contrast to malleable tasks which are parallelizable tasks that can change their degree of parallelism during execution [3]. *Crown scheduling* [4] is a static scheduling approach for moldable tasks that has been extended to heterogeneous platforms and multiple possible target functions such as maximum throughput for given power budget besides the usual energy minimization per round (thus achieving minimum average power) for given throughput [5]. In all these investigations, the platform was fixed, i.e., the number of cores of each type was given. While the scheduler had the freedom not to use some cores, their chip area could not be used for cores of a different type, which might have been helpful in further optimization, e.g., using more energy-efficient cores instead of high-performance cores if the task structure demands or allows this.

Such tradeoff is possible in a configurable platform such as an FPGA or ASIC where multiple (soft) cores can be implemented and the number of cores of each type is not predetermined (although the maximum combined area of the cores is). To achieve this, we extend the integer linear program (ILP) of the crown scheduler to allow the solver to determine the core counts of different core types. Thus, we integrate design space exploration with the scheduler to generate an energy-optimal configuration of heterogeneous (soft) cores for an application with given throughput constraints and a platform with a given area.

We evaluate our proposal with a benchmark suite of 40 synthetic task sets and 3 task sets from real-world applications for different numbers of core types. We start with 2 core types and use ARM's big.LITTLE in its original manifestation, i.e. A7 and A15 cores. We use power profiles for different task types and core types derived from measurements on a real platform. We find that our integrated approach improves energy efficiency compared to a fixed platform and that in many cases, the optimal number of LITTLE cores is notably larger than the number of big cores, i.e. different from real platforms. Moreover, our ILP solver is only marginally slower than for a fixed platform, so that we improve DSE time compared to heuristic DSE approaches like steepest descent or tabu search over the configuration space, which call energy computation (which in turn needs at least some approximation to scheduling) at each configuration point considered. We also perform scheduling experiments for three core types, adding ARM A72 as further core type, and again find an energy improvement of 50.3% averaged over all task sets, with only moderately increased runtime. Furthermore, we free the crown scheduler from using only core group sizes that are powers of two, which allows to use e.g. 3 cores of one core type more efficiently. In particular, we make the following contributions[1]:

- We extend crown scheduling from a fixed platform structure to a variable platform structure, i.e. numbers of each type of (soft) cores in a configurable architecture, in order to find the platform configuration *and* schedule that lead in combination to the energy-optimal execution. The presented approach works for any number of different core types.
- We demonstrate the potential for improving energy efficiency over crown-optimal schedules for a fixed platform by evaluation with a multitude of synthetic task sets with realistic task types, several task sets from real-world applications, and power profiles for ARM big.LITTLE types of cores. Additional experiments with three core types demonstrate the feasibility of the general approach beyond two core types.
- We demonstrate how to generalize this approach to different optimization targets such as minimum area solution for a given power budget and throughput, or a Pareto front of minimum power budgets for different maximum chip sizes of the configurable architecture, given a required throughput. By requiring that not only the sum of the core areas meet the chip size, but also cores with rectangular geometry fit into the chip rectangle, we provide a lower bound on usable core counts and refine such Pareto front, to illustrate that real floorplanning will be somewhere in between those extremes.
- We generalize the crown scheduler by still using a binary partitioning of cores into groups, but allowing core group sizes that are not powers of 2. This may help to improve

[1] This article is an extended version of [1]. In particular, we add experiments with more than 2 core types, contrast upper bound on usable cores with a lower bound, generalize the crown scheduler to non-power of 2 group sizes, and analyze possible threats to validity.

task assignment for core types where the core count is odd, like 3 or 5.

The remainder of this article is organized as follows: Sect. 2 introduces the processor, task and power models, and the crown scheduling principle. Section 3 explains how to combine the design space exploration for a configurable platform with task scheduling to achieve more accurate predictions. Scheduling results are presented in Sect. 4. Section 5 presents generalizations of our optimization, while Sect. 6 restricts core counts by geometric constraints to provide lower bounds on achievable core counts. Section 7 shows how to free the crown scheduler from the limitation to core group sizes of powers of two, and illustrates its helpfulness. Section 8 discusses possible threats to validity of this study, while related work is discussed in Sect. 9. Section 10 concludes and suggests future work.

2 Background

2.1 Architecture Model

We consider a generic multicore or manycore CPU with p cores with dynamic discrete voltage and frequency selection (DVFS). The cores can either be all of the same type (as in standard server multicore CPUs) or of different types (such as in ARM's big.LITTLE). We assume a configurable platform of a given area, i.e., we consider ASIC cores or FPGA soft cores of known area.[2]

A core can execute at most one task (instance) at a time, and can switch the voltage/frequency level between tasks among a given fixed set of K discrete voltage/frequency levels with frequencies f_k, $k = 1, ..., K$, where $f_{\min} = f_1 < f_2 < ... < f_K = f_{\max}$, and the voltage is, for each k, auto-co-scaled to the lowest level still supporting the chosen frequency f_k. For heterogeneous platforms with cores of different types t, the frequencies $f_{k,t}$ and even their number K_t might depend on the core type.

While idle, a core consumes a base power $P_{idle,t}$. When executing a task j at some DVFS level k, it consumes a power $\text{power}(t, k, \text{tty}(j))$ that depends on k and on the task type $\text{tty}(j)$, and, in the case of a heterogeneous architecture, also on the core type t. The *task type* $\text{tty}(j)$ of each task j can be roughly characterized by a small number of predefined discrete classes in order to distinguish e.g. computationally intensive from memory-access intensive tasks, as these react differently to frequency scaling. Idle power and task power can, for each DVFS level, task type and core type, either be sampled on the target system using microbenchmarks for the different task types, or be estimated using a theoretical power model. In the following, we will subtract $P_{idle,t}$ from $\text{power}(t, k, \text{tty}(j))$, so that the power drawn by a core of type t executing a task j is

$$\text{power}(t, k, \text{tty}(j)) + P_{idle,t}.$$

2.2 Task Model

Each task j, for $j = 1, ..., n$, performs work $\text{work}(j)$, corresponding to the time of executing the task on a single (reference-type) core at unit frequency, i.e., executing the task at frequency f_k results in a runtime $\text{work}(j)/f_k$. This runtime includes the phase where the task writes its output either to a memory shared by all cores, or via an on-chip network to the local memory of the core that runs the follow-up task.

We consider *moldable* parallel tasks, i.e. parallelizable tasks that can use any number of cores assigned prior to execution, internally employing a parallel algorithm to share the work, in contrast to malleable tasks where the degree of parallelism might change during execution [3]. We make no assumptions about monotonicity of parallel speedup nor absence of speedup anomalies. Instead, the actual relative speedup of task j with $q > 1$ cores on the target system, denoted by $\text{speedup}(j, q)$, can again be found out by microbenchmarking or be estimated from a theoretical cost model for the deployed parallel algorithm. Obviously, for all j we have $\text{speedup}(j, 1) = 1$. Inherently sequential tasks j can be modeled by simply setting $\text{speedup}(j, q) = 1$ for $q > 1$. If a task j has a maximum width or degree of parallelism $W(j)$, then we can e.g. set $\text{speedup}(j, q) = \text{speedup}(j, W(j))$ for $q > W(j)$, i.e. the runtime does not shrink anymore if we use more than $W(j)$ cores. One might even use a lower speedup to account for additional overhead with more cores. For architectures with different core types, we index the tasks' speedup tables also by the core type t, i.e., $\text{speedup}(j, q, t)$, where $\text{speedup}(j, 1, t) = 1$ in general only holds for the reference core type $t = 1$, while for other core types, $\text{speedup}(j, 1, t)$ denotes the relative performance between core type t and the reference core type for the task type of task j.

The time of executing task j at DVFS level k on q cores of core type t is then

$$\text{Time}(j, k, q, t) = \frac{\text{work}(j)}{f_{k,t} \cdot \text{speedup}(j, q, t)}.$$

[2] The cores of same type typically share common L2 on-chip caches and/or other common "uncore" on-chip infrastructure, thus forming a *core cluster*, as for example in ARM big.LITTLE architectures. Simplifying the area model, we assume here that such additional area is proportional in size to the number of cores of each type, and factor it proportionally into the core area resp. overall chip area. See also the lower-bound discussion in Sect. 6. An extension of our area model with explicit un-core area parameters is left to future work.

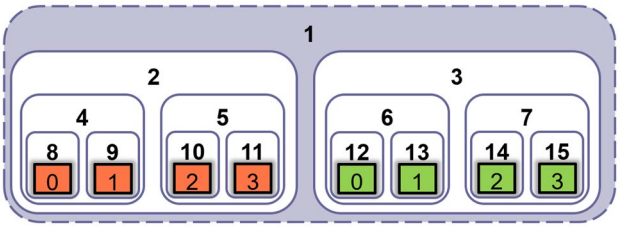

Figure 2 A heterogeneous crown for a standard big.LITTLE configuration with 4 big cores (orange) and 4 LITTLE cores (green). Note that the root group (1) does not exist, i.e., cannot be assigned (parallel) tasks.

2.3 Crown Scheduling

Crown scheduling [4, 6] is a static scheduling technique for the steady state of the software-pipelined streaming task graph where all tasks are active processing data. It considers the subproblems of core allocation to moldable tasks, DVFS level selection, mapping of tasks to specific groups of cores and ordering of task execution in time together, and introduces artificial constraints on core allocation and task sequencing to reduce the integrated optimization problem's complexity without sacrificing significant optimization potential in practice [7].

A key property is the hierarchical organization of the set of p cores by recursive binary partitioning into $2p - 1$ core groups jointly referred to as the *crown*, see Fig. 2 for an example of a balanced binary crown of 8 cores (ignore the colors/shading for now), which consists of 15 core groups in total: The root group of all cores (group 1) is decomposed into two child groups (groups 2 and 3) of half the size each, which are further split into four grandchild groups (groups 4–7) and so on, until we arrive at p leaf groups containing one core each. Note that the crown is exponentially smaller than the power set of p cores with $2^p - 1$ core groups.

In a crown schedule, moldable tasks can only be mapped to one of the core groups in the crown. This also implies that the core allocation to each moldable task has to be a power of two. Moreover, a crown schedule is constructed so that each core executes, within each iteration of the steady-state pattern of the software pipeline, the tasks mapped to it in (the same) order of non-increasing core allocation. This allows all cores mapped to a parallel task to simultaneously start their execution, so that the parallel algorithm running in the task does not incur additional delays. Hence, there is no internal fragmentation within the schedule; any idle times due to residual load imbalances only occur at the end of a round. Optimal and heuristic algorithms for crown scheduling have been presented in earlier work [4, 6].

Crown scheduling can be generalized [5] to heterogeneous multicore CPUs with different core types such as ARM big.LITTLE, which in our setting combines four A15 ("big") cores with four A7 ("LITTLE") cores. In this case, the top-level subdivision splits the overall heterogeneous core set such that internally homogeneous subsets are obtained as its child groups, and the heterogeneous root group is excluded as a possible mapping target, see Fig. 2. An alternative view, which might be more appropriate for more than two core types, is that each core type has a crown of its own.

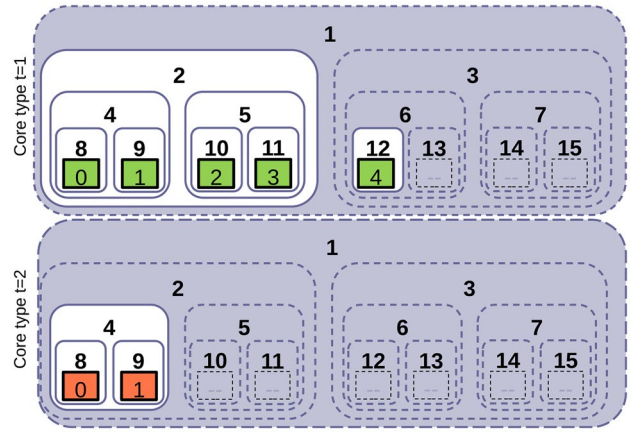

Figure 3 Crown structures for $T = 2$ different core types. As there are five cores of type 1, core groups 2, 4, 5, 8, 9, 10, 11, 12 exist for $t = 1$ (green, top), and with only two cores of type 2, we have core groups 4, 8, 9 for $t = 2$ (red, bottom). Tasks will only be assigned to the actually instantiated core groups colored in white.

3 Combining Design Space Exploration with Scheduling

Up to now, crown scheduling assumes a platform with a fixed number of cores of each core type, and a fixed mapping of core indices and core types. In order to integrate DSE with scheduling, we assume that we can give an upper bound p on the maximum possible number of cores of any core type t. More exactly, we will set p to be the smallest power of 2 larger than this maximum number. The number T of core types is a constant of arbitrary value, i.e. the combined DSE and scheduling works for any number of core types. This is illustrated by the fact that the following ILP[3] to solve the DSE and scheduling optimization only sums over all core types t, or uses constraints for all core types t. We denote the number of cores of each type t by an integer variable p_t. For the big.LITTLE example with $T = 2$, variables p_1 and p_2 which denote the number of LITTLE and big cores, respectively.

Each core type t gets a crown of its own with groups $i = 1 \ldots, 2p - 1$ and cores with indices $l = 0, \ldots, p - 1$. For

[3] Available from https://github.com/sglitzinger/idses.

core type t, a group i with maximum core index $\max(i)$ is only existing if $\max(i) < p_t$, cf. Fig. 3.

For $T = 2$ core types and fixed values of p_1 and p_2, the ILP below resembles the ILP from [5]. Yet, as p_1 and p_2, or in general p_t, are variables in the current setting where scheduling is combined with DSE, we will define further constraints for them, and minimize the objective function over all possible combinations of these variables.

We use binary variables $x_{i,j,k,t}$ where $x_{i,j,k,t} = 1$ iff task j is mapped to core group i of core type t at frequency $f_{k,t}$. Furthermore, we use binary variables $ex_{i,t}$ where $ex_{i,t} = 1$ iff core group i does not exist for core type t.

We then aim to solve the following optimization problem:

$$\min \sum_{i,j,k,t} x_{i,j,k,t} \cdot \text{time}(i,j,k,t) \cdot \text{power}(t,k,\text{tty}(j)) \cdot \text{size}(i) + \sum_t (p_t \cdot P_{idle,t}) \cdot M \tag{1}$$

$$\text{s.t.} \quad \forall j \quad \sum_{i,t} \sum_{k \le K_t} x_{i,j,k,t} = 1 \tag{2}$$

$$\forall j \quad \sum_{i,t} \sum_{k > K_t} x_{i,j,k,t} = 0 \tag{3}$$

$$\forall i,j,t \quad \sum_k x_{i,j,k,t} \le 1 - ex_{i,t} \tag{4}$$

$$\forall t,l \quad \sum_{i \in G_l, j, k} x_{i,j,k,t} \cdot \text{time}(i,j,k,t) \le M \tag{5}$$

$$\forall i,t \quad \max(i) - ex_{i,t} \cdot p < p_t \tag{6}$$

$$\forall i,t \quad \max(i) + (1 - ex_{i,t}) \cdot p \ge p_t \tag{7}$$

$$\sum_t p_t \cdot A_t \le A_{Total}. \tag{8}$$

Constraint 2 ensures that each task j is mapped to exactly one core group of one core type but only to an existing frequency level by constraint 3, and constraint 4 guarantees that tasks cannot be mapped to core groups that do not exist. For each core $l = 0, \ldots, p-1$, let G_l in constraint 5 denote the set of all core groups i that comprise l, whether they are instantiated or not. For given p, this set is known a priori. For example, in Fig. 3 we have $G_{13} = \{1, 3, 6, 13\}$. To denote task runtimes, we use $\text{time}(i,j,k,t) := \text{Time}(j,k,\text{size}(i),t)$. To constrain round duration, constraint 5 must hold, i.e., for each core l of each core type t, the total runtime of all tasks mapped to a core group in G_l must not exceed the deadline M.

To set variables $ex_{i,t}$, we use the constraints 6 and 7. If $\max(i) \ge p_t$, then $ex_{i,t}$ must be 1 because of constraint 6.

Table 1 Characteristics of the synthetic task sets.

number of tasks	10, 20, 40, 80
task types	Branch, Memory, FMult, SIMD, MatMul
task workloads (10^6 cyc.)	1 to 40 on LITTLE cores
max. parallelism degree	1 to 4, dep. on task type

By constraint 7, $ex_{i,t} = 0$ if $\max(i) < p_t$. Besides the obvious non-negativity constraints, the variables p_t need constraint 8, i.e., the sum of the core areas for the different core types cannot exceed the total FPGA area, where area here also might mean number of CLBs, depending on the FPGA architecture, and the total area might be discounted by a percentage to account for efficiency loss of some kind such as area for wiring. By the choice of p, this also implies $\sum_t p_t \le p$. The objective function seeks to minimize the sum of the energies needed to execute each task on the core group it is mapped to, plus the power at idle times. While the latter term was a constant for a fixed platform, i.e. not influencing the optimum, it is now variable and must be considered.

Please note that the given ILP can even be helpful for a fixed platform with P_t cores of each core type t, where the cores can be shutdown individually if they are not used. If we replace constraint 8 by $\forall t \ : \ p_t \le P_t$, then we have an improved variant for crown scheduling with core consolidation, cf. [8].

4 Scheduling Results

4.1 Experimental Settings

The reasoning behind our initial experiments was to facilitate a comparison to the results in [5] in order to assess the additional value of adapting the device's design to the application at hand. To this end, we adopted the experimental setup in [5], in particular: task sets, deadlines, core power consumption values, available core operating frequencies, relative performance figures for big vs. LITTLE cores, and speedup values[4].

We are aware that power consumption and frequencies will not be identical when switching from ASIC cores to soft cores on an FPGA. Still, we assume that the power consumption values (and operating frequencies) of different core types relative to each other might remain as they were.

[4] In [5], we used parallel efficiency, i.e. speedup over core count, instead of speedup and separated the difference in performance between core types from the speedup by using an additional variable $r_{i,j}$, yet the speedup can be uniquely computed from efficiency and $r_{i,j}$.

Table 2 Results for the current experiments on synthetic task sets with two core types (A7 and A15) in comparison to the original experiments in [5], same chip size.

task set size n		10	20	40	80	total
E_{total} curr. vs. orig. exp.	min.	65.2%	70.3%	74.5%	78.5%	65.2%
E_{total} curr. vs. orig. exp.	avg.	83.5%	79.2%	80.9%	80.4%	81.0%
E_{total} curr. vs. orig. exp.	max.	94.5%	86.4%	86.3%	83.9%	94.5%
#big vs. orig.	avg.	-2.2	-2.0	-2.6	-2.3	-2.3
#LITTLE vs. orig.	avg.	0.5	-0.6	0.3	-0.3	0.0
E_{idle}/E_{total} current	avg.	0.662	0.642	0.647	0.628	0.645
E_{idle}/E_{total} orig.	avg.	0.741	0.770	0.764	0.770	0.761

The original experiments were conducted for 40 task sets with 10, 20, 40, and 80 tasks, respectively (10 task sets of each cardinality). Tasks are of one of five possible types: MEMORY, BRANCH, FMULT, SIMD, or MATMUL. A task's type implies a certain average core power consumption when being executed as well as an affinity to one of the two available core types with regard to its runtime, meaning some task types run faster on LITTLE cores and some on the big ones. All this information is procured from [9] and summarized in Table 1. Furthermore, a task's maximum width is determined by its type in the following manner[5]:

$$W(j) = \begin{cases} 1, & \text{if tty}(j) \text{ is BRANCH,} \\ w_j \in \{2,4\} & \text{if tty}(j) \text{ is MEMORY or FMULT,} \\ 4, & \text{if tty}(j) \text{ is SIMD or MATMUL.} \end{cases}$$

As before, the operating frequencies to choose from are $\{0.6, 0.8, 1.0, 1.2, 1.4\}$ GHz, the theoretically available 1.6 GHz for the big cores is ignored. Parallel speedups for all tasks, i.e. all task types, on the reference core type (LITTLE) are given as 1, 1.8 and 3.44 for widths 1, 2 and 4, respectively, and are multiplied by the relative factors from [9, Table 4], to obtain speedups on big cores, cf. [5]. Deadlines are computed as

$$M = 0.6 \cdot \frac{\frac{\sum_j \text{work}(j)}{p \cdot f_1} + \frac{\sum_j \text{work}(j)}{p \cdot f_K}}{2},$$

with $p = 8$ to obtain deadlines for a platform with 4 big and 4 LITTLE cores.

For further details on the original experimental set-up please consult [5]. For these experiments, core idle power was ignored as under fixed round times and fixed core composition, energy consumption caused by idle power is a constant and therefore not relevant with regard to optimization. Now, core composition can vary, thus idle power must be taken into account, cf. the objective function in Eq. 1. To enable a meaningful comparison to the results in [5], we added the energy consumption caused by idle power for 4 big and 4 LITTLE cores over a round's duration to the original results.

The Gurobi 8.1.0 solver was employed to obtain ILP solutions. The respective computations were run on an AMD Ryzen 7 2700X with 8 physical cores and SMT under a 5 minute (wall clock) timeout. The implementation was carried out in Python utilizing the `gurobipy` module.

4.2 Scheduling Synthetic Task Sets

Primarily, we are interested in whether a chip of the same size as in [5] can lead to energy savings when altering core composition, i.e., whether jointly optimizing chip design and schedule can lower energy consumption for the execution of the target application. To gain any insight in this regard, we have repeated the original experiments with the ILP from Sect. 3 and the chip's size capped at the value required for the standard configuration of 4 big and 4 LITTLE cores. In the implementation described in [10], a big core[6] occupies an area of 19mm^2, while a LITTLE core's size is 3.8mm^2. For the area constraint 8, we therefore set $A_{Total} = 4 \cdot 19\text{mm}^2 + 4 \cdot 3.8\text{mm}^2 = 91.2\text{mm}^2$. As no more than $\left\lfloor \frac{91.2\text{mm}^2}{\min\{19\text{mm}^2, 3.8\text{mm}^2\}} \right\rfloor = 24$ cores can be placed on the chip, we further set $p = 32$. Also here, the absolute values will not be maintained when going from ASIC to FPGA, yet the relative size of cores, even if measured in configurable blocks (or resources), would rather be the same. Also, we are aware that using a purely additive area model is a simplification, but assume that the majority of the FPGA logic blocks will be spent for cores and not for communication in addition to the programmable interconnect, so that the inaccuracy is small.

Table 2 displays the results. It compares energy consumption values E_{total} from the original experiments to those obtained via our current experiments. Depending on task set size, we observe a 16.5–20.8% lower energy consumption on average. For single task sets, energy savings of up to 34.8% are possible. Variance drops with increasing task

[5] We did not express the maximum width explicitly in Sect. 2 but included it in the definition of speedup.

[6] In that paper, the area refers to a quad core including cache, so that one might divide the area by 4, if cache is evenly distributed over individual cores. We decided to keep the original figures.

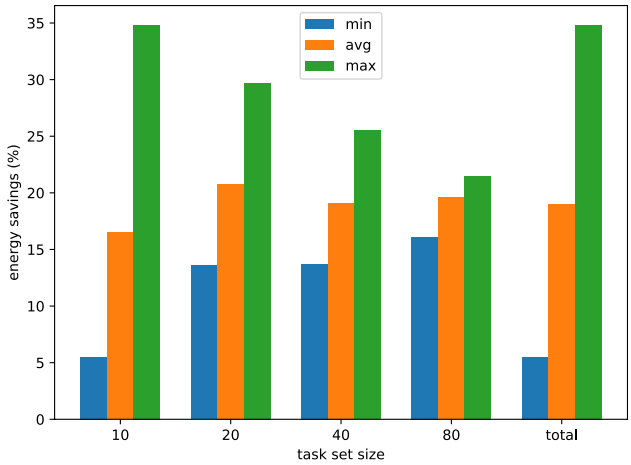

Figure 4 Energy savings in comparison to the original experiments on synthetic task sets for two core types (A7 and A15) and all examined task set sizes.

set size as indicated by growing minimum and decreasing maximum values. Figure 4 summarizes the energy savings with regard to the original experiments for all task set sizes. All ILPs could be solved either to optimality or solutions are very close to the optimum (the maximum MIPGap value over the 40 experiments is 0.0018).

It becomes clear that a considerable reduction of energy consumption is possible if chip size remains constant while the total number of cores as well as core composition may differ. In particular, the number of big cores was on average 2.3 lower than in the original experiments with fixed values for p_1 and p_2, i.e. the number of LITTLE and big cores, respectively. Interestingly, the total number of LITTLE cores did not change when averaged over all examined task sets. Figure 5 shows the distribution of deviation from the results in the original experiments in # cores for both big (top) and LITTLE (bottom) cores over all 40 task sets. The last two rows in Table 2 contain the fraction of the energy consumption caused by the chip's idle power. For the original experiments, the figures show that idle power[7] is responsible for > 75% of the total energy consumption. It is therefore not surprising that significant optimization potential may lie in altering the core composition for a given chip size. In our current experiments, the chip's idle power accounts for 62.8–66.2% of the total energy consumption. Jointly optimizing chip design and the target application's schedule enables a tradeoff between core count on the one hand and higher operating frequencies or heavier parallelization on the other, which proves to be beneficial with regard to energy efficiency.

[7] Please remember that also during task execution, power is composed from "idle" power and power on top, so that idle power mostly indicates static power.

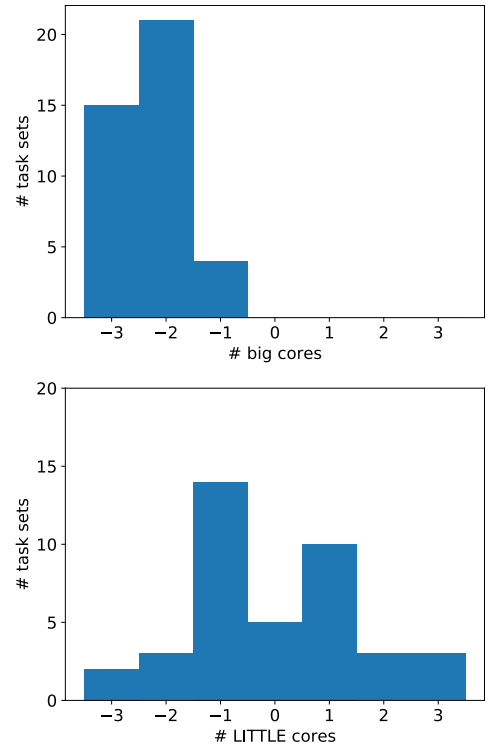

Figure 5 Distribution of deviation from the results in the original experiments in # cores for big (top) and LITTLE (bottom) cores over all 40 synthetic task sets.

As unused cores could simply be switched off, eliminating their idle power from the calculation, we scanned the original results from [5] for cores which had no tasks mapped to them. These could then be regarded as switched off, and the energy consumption caused by their idle power deducted from the total energy consumption. As it turned out, all 40 schedules had tasks mapped to each available core, so no cores could be switched off. In a way, this demonstrates the importance of considering chip design at scheduling. After all, our results can not only be employed to prescribe a specific core composition but also increase energy efficiency for a given chip design, where unused cores can be switched off to save energy.

Had our power model not considered a core's idle power, there would be no incentive for the solver to minimize the number of cores when a solution with optimal energy consumption has been reached. Thus, it is conceivable that including the total number of cores in the optimization

Table 3 Characteristics of the real application task sets.

application	number of tasks	parallelism
H.263 encode	9	seq.
SDE	8	seq. & par.
edge detection	6	seq. & par.

Table 4 Experimental results for three real-world applications with two core types.

application	H.263 encode	SDE	edge detection
E_{total} design & sched. vs. sched. only	64.9%	55.3%	62.0%
# big cores	3	2	2
# LITTLE cores	1	1	2

process might lead to solutions with equally optimal energy consumption but lower core count. By modifying the objective function, energy consumption and core count are jointly minimized:

$$\min \sum_{i,j,k,t} x_{i,j,k,t} \cdot \text{time}(i,j,k,t) \cdot \text{power}(t,k,\text{tty}(j)) \cdot \text{size}(i) + \gamma \cdot (p_b + p_L).$$

Of course, γ must be chosen such that optimality regarding energy consumption is given while core count still makes a relevant difference during optimization.

4.3 Scheduling Real Applications

In addition to the synthetic task sets considered in [5], we were interested in whether combined design space exploration and scheduling could offer any benefits when it comes to real applications. Therefore, we have performed the above experiments for three selected applications: H.263 encode detailed in [11], stereo depth estimation (SDE) from [12], and edge detection as described in [13]. The H.263 encode task set is comprised of 9 sequential tasks. SDE features 8 tasks, some of which are sequential while some can be parallelized to an arbitrary degree. This holds true as well for the edge detection application consisting of 6 tasks. Table 3 assembles information on the task sets while Table 4 shows the results for the three applications just presented. It becomes clear that the energy savings are more substantial than for the synthetic task sets, ranging from approximately 35% for the H.263 encode application to 45% for SDE. Optimizing chip design in conjunction with the schedule again pays off as is indicated by the specific chip compositions featuring less than four cores of each type for all three applications. In each case, we chose tight deadlines just about sufficient to produce a feasible schedule. This means that even from a performance-oriented point of view, the chip with four big and four LITTLE cores would not provide any advantages over the individual designs issued by our approach for the respective task sets. Due to the small number of tasks in each application, scheduling times were very low (< 1s) throughout, which is why we will omit any discussion to this effect here.

4.4 Experiments with 3 Core Types

In our approach outlined in Sect. 3, we allow for an arbitrary number of core types. In an attempt to demonstrate that this decision proves fruitful, we have repeated the scheduling experiments with an additional core type for a total of three different core types. As the data available in [9] covers only two core types (ARM A7 and A15), we have conducted our own measurements for a third core type, which we chose to be an ARM A72. It is implemented e.g. in the BroadcomBCM2711 chip, a quad-core processor featured in the latest instalment of the ubiquitous Raspberry Pi device, the Pi 4 Model B. For the measurements, we have adhered to the procedure detailed in [9]: per-core power consumption figures for the various task types and core operating frequencies were obtained by running the respective benchmark functions from the epEBench bench-mark [14] on all four cores, subsequently subtracting base power and dividing by 4. Relative performance characteristics were determined by executing a fixed number of loop iterations on a single core at 1 GHz for each of the benchmark functions on the A7 and A72 cores while measuring runtime. As no information on die size is available for the Broadcom BCM2711, we have derived an approximate value from the publicly available[8] data on the MediaTek Helio X20 chip, taking into account that it is manufactured in TSMC 20 nm in contrast to TSMC 28 nm for the BCM2711. Furthermore, the A72 implementation on the Raspberry Pi 4 B offers more extensive DVFS capabilities than the core types considered so far, with 10 discrete frequency levels in steps of 100 MHz from 600 MHz to 1.5 GHz.

Table 5 displays the results for the 40 synthetic task sets featuring five distinct task types from [5]. In comparison to the current experiments with two core types (A7 and A15, cf. Table 2), energy consumption can be brought down even further. The significantly lower E_{idle}/E_{total} value (43.3% vs. 64.5% on average for the experiments with two core types only) is due to the lower total number of cores placed on the chip (3.4 vs. 5.7 on average for the experiments with two core types only) as well as the lower base power for the A72. This is not the only reason for the more favorable energy consumption though: the fact that all three core types are made use of over all task set sizes in a large number of cases indicates that energy efficiency is furthered by the greater heterogeneity of the chip. Big cores (A15) were used for 26 out of the 40 task sets, LITTLE cores for 39, and an A72 core was placed on the chip for each of the cases examined here. In Fig. 6, one can see the distribution of deviation from the results in the original

[8] cf.https://en.wikichip.org/wiki/arm_holdings/microarchitectures/cortex-a72.

Table 5 Results for the current experiments with three core types (A7, A15, and A72) in comparison to the original experiments in [5], same chip size.

task set size n		10	20	40	80	total
E_{total} curr. vs. orig. exp.	min.	29.5%	41.9%	41.6%	47.0%	29.5%
E_{total} curr. vs. orig. exp.	avg.	50.6%	49.7%	48.1%	50.3%	49.7%
E_{total} curr. vs. orig. exp.	max.	73.3%	59.4%	54.4%	61.0%	73.3%
# big	avg.	0.7	0.6	0.5	0.8	0.7
#LITTLE	avg.	1.9	1.8	1.8	1.5	1.8
#A72	avg.	1.0	1.0	1.0	1.0	1.0
E_{idle}/E_{total} current	avg.	0.424	0.444	0.435	0.427	0.433
E_{idle}/E_{total} orig.	avg.	0.741	0.770	0.764	0.770	0.761

experiments in # cores for both big (top) and LITTLE (bottom) cores over all 40 task sets. Note that for each task set, one A72 core was placed on the chip. Finally, Fig. 7 visualizes the energy savings for all task set sizes versus the original experiments in [5] with two core types.

A natural assumption when designing a chip with a variety of cores of different types would be that a given core type is well suited to execute workloads of a particular kind, whereas others may run more efficiently on cores of a different type. It will therefore be interesting to look at how often certain task type/core type pairings have occurred in our experiments. This information is provided by Table 6, which by and large confirms our suspicions. For the compute-intensive task types FMult, SIMD, and MatMul, the A72 core is usually favored. This is not surprising as the A72 is a more recent development and some technological advancements can be assumed to have taken place. The vast majority of tasks of type Branch runs on the LITTLE cores, whereas the Memory tasks are roughly split between big and LITTLE cores. All in all, these results still do not merit the conclusion that one can simply schedule all tasks of a certain type to a particular core type and thus reach optimality w.r.t. energy efficiency with ease. Note that even if such a plain mapping procedure could be issued, the exact number of cores of each type would nonetheless be subject to further optimization.

The observation that the greater flexibility a third core types offers is well received by the optimizer and pays off in terms of energy efficiency also holds for the three real-world applications, as Table 7 shows. In each case, the optimal solution entails a different core type composition: big and A72 cores for the edge detection application, LITTLE and A72 cores for SDE, and for the H.263 encode task set, all three core types are picked in the chip design. In comparison to joint design space exploration and scheduling with two core types, we notice a further reduction of energy consumption for executing the resulting schedule, cf. Table 4.

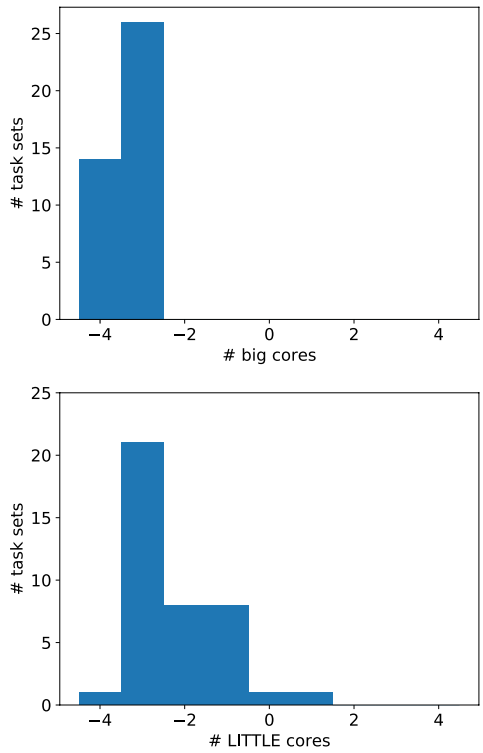

Figure 6 Distribution of deviation from the results in the original experiments in # cores for big (top) and LITTLE (bottom) cores over all 40 synthetic task sets and 3 core types.

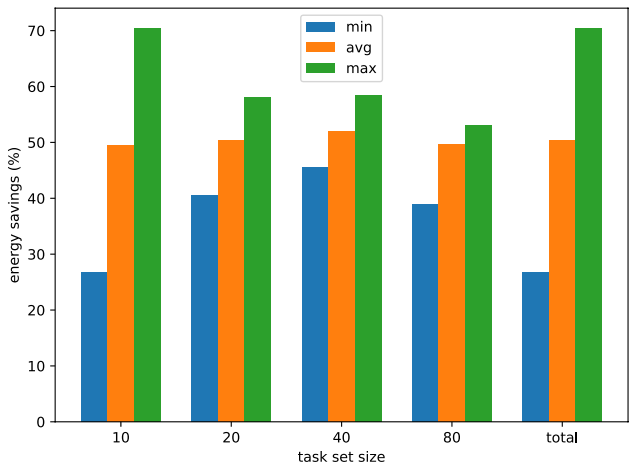

Figure 7 Energy savings for the experiments with three core types (A7, A15, A72) in comparison to the original experiments for two core types (A7 and A15) and all examined task set sizes.

Table 6 Occurrences of task type/core type mappings for the current experiments with three core types (A7, A15, and A72).

task type	big	# mapped to LITTLE	A72
MEMORY	128	142	22
BRANCH	0	264	36
FMULT	17	0	259
SIMD	3	0	282
MATMUL	0	4	343

4.5 Scheduling Time

Beyond scheduling quality we are concerned with the time it takes to compute a schedule (and potentially a core composition) for a given task set. Table 8 provides information on scheduling times for the original experiments without design space exploration in [5] as well as for our current experiments jointly optimizing chip design and application schedule under an area constraint ($A_{Total} \leq 91.2\text{mm}^2$), with 2 or 3 different core types. For small and moderately sized task sets, scheduling times are low throughout. For $n = 10$, they are notably lower in the original experiments though, which illustrates that considering chip design in the optimization process may lead to a higher computational effort than solely optimizing the schedule for low energy consumption. For the larger task sets ($n = 40$, $n = 80$), scheduling times rise significantly in all setups, indicating that the decisive factor is task set size when it comes to problem complexity. Here, the setup with three core types causes the lowest scheduling times. Over all task sets examined, the overhead in our current experiments under the area constraint is $\approx 37\%$ for the case with two core types. Unsurprisingly, the solver ran into the timeout more often as well (11 timeouts in total vs. 6 in the experiments in [5]). Nevertheless, the largest MIPGap value at timeout was 0.0018, thus we can consider all solutions (near-)optimal. For the setup with three core types on the other hand, the solver computes solutions notably quicker when considering all task set sizes examined here. It should be noted though that we have not performed any scheduling experiments for three core types without design space exploration, so we actually lack data which could permit any conclusions regarding the additional effort the optimization of chip design demands for the case where three core types are available. Figure 8 provides a visualization of the information on scheduling times in Table 8.

Table 7 Experimental results for three real-world applications with three core types.

application	H.263 encode	SDE	edge detection
E_{total} design & sched. (3 core types) vs. sched. only (2 core types)	42.5%	29.4%	37.9%
# big cores	1	0	1
# LITTLE cores	1	1	0
# A72 cores	1	1	1

5 Generalization

The design space for a streaming application on a configurable platform can be optimized in several dimensions: (1) energy for one round, (2) duration of one round, and (3) number of (soft) cores of each type (or total area of (soft) cores). So far, we have given fixed upper bounds for (2) and (3), and minimized the first target under those constraints. However, it is also possible to fix any other two and optimize the third. For (1) and (2), this is explained in [5], where a fixed platform is assumed, i.e., (3) is not a variable.

Here, we can change the optimization goal, e.g. to minimize the duration M of one round, by making M a real-valued variable, using objective function $\min M$, and changing the previous objective function in Eq. 1 to a constraint $E \leq P_{bud} \cdot M$, where P_{bud} is a given maximum (average) power budget, and E is a short-hand for the sum in 1. Similarly, the minimum area for given power budget and throughput can be found by using the left-hand side of constraint 8 as objective function, using the previous objective function in a constraint as given above, and letting M as a constant as in Sect. 3.

Moreover, it is possible to fix one of those dimensions and produce a Pareto front for the other two. We will exemplify this for the case of determining the minimum (average) power budget vs. chip area for a given throughput, i.e. for a given round length M. This can e.g. be used to find a suitably sized FPGA for a given application.

To do this, we solve a sequence of MILPs, starting with the maximum available chip area A_{max}, and using $A_{Total}(i) = A_{max} - i \cdot A_{LITTLE}$ with $i = 0, 1, 2, \ldots$ until[9] no feasible solution is found. Each solution comprises optimum core counts p_1 and p_2 and the minimum energy $E^\star$, which can be converted into an average power budget by dividing by M. Figure 9 depicts such a Pareto curve for one set of tasks ($n = 20$) with the deadline M from Sect. 4, giving maximum chip area in mm^2 and power in W as the axes, and marking each point in the Pareto curve with (p_2, p_1), i.e. starting with the number of big cores, for that solution. As we can see, for a wide range of the examined maximum die size values, core composition does not change as the most energy-efficient design can be accommodated within the given constraints. Presumably, adding more cores does more damage energy-wise than increasing operating frequencies

[9] A smaller step than the area A_{LITTLE} of the smaller core would not change anything.

Table 8 Scheduling times (sums of user and system time) and number of timeouts for the current experiments and for the experiments in [5].

	experiments in [5]		current experiments, 2 core types		current experiments, 3 core types	
n	avg. sched. time (s)	#timeouts	avg. sched. time (s)	#timeouts	avg. sched. time (s)	#timeouts
10	0.355	0	2.906	0	1.407	0
20	5.175	0	142.798	0	4.247	0
40	1702.486	3	2491.273	5	21.378	0
80	1671.283	3	1984.073	6	513.461	1
total	844.825	6	1155.262	11	135.123	1

or extending parallelization, even if there is plenty of unused area left on the chip. When maximum chip size starts to pose an effective limitation, big cores are substituted by one or several LITTLE cores. That way, average power consumption increases but feasible solutions can still be produced down to 34.2 mm^2 die size, when the chip only features LITTLE cores and average power consumption is $\approx$ 54% higher than for the energy-optimal solution under arbitrary chip size.

6 Lower Bound on Area Consumption

Constraint 8 which simply sums up the core areas is kind of a best case consideration, as it allows arbitrary, even unrealistic shapes of core layouts and arbitrary intertwining of layouts for different cores. Thus, the core counts for different core types possible via that constraint can be considered as an upper bound.

Here, we also provide a lower bound on core counts. We consider the different types' core layouts as rectangles of fixed size and shape (no rotation allowed), which must be placed into the FPGA area (also given as a rectangle)

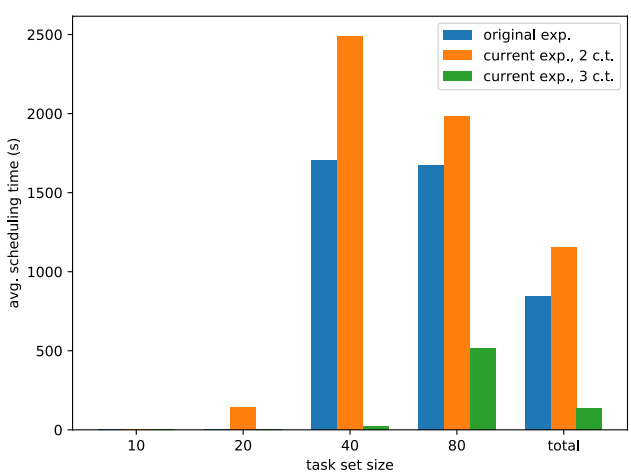

Figure 8 Comparison of scheduling times for the original experiments in [5] and for the current experiments on synthetic task sets with two and three core types, respectively, grouped by task set size.

without overlap. These requirements are rather harsh and thus comprise a kind of worst case (cf. e.g. Fig. 6 in https://www.design-reuse.com/articles/21583/processor-noc-fpga.html.)

By providing a Pareto curve similar to Fig. 9, the differences between lower and upper bound are visible, and a realistic solution will be inbetween.

Finding optimal, non-overlapping layouts for rectangles of different sizes is a hard problem [15, 16]. The solution can be approximated by a variant of strip packing, where the strip represents the FPGA and thus has a finite length, larger rectangles are only provided in the required counts, and smaller rectangles are placed as long as they fit the strip's length. In our case, we additionally require that cores of the same type be placed in a contiguous area of the chip, in order to account for shared on-chip infrastructure of a core-type cluster and also to avoid unnecessarily large intra-cluster core distances.

For two core types, the problem can be approximated by first tiling a given number p_2 of big cores in the lower left corner of the FPGA rectangle, and then finding the possible number of LITTLE cores by tiling them in the upper right corner of the FPGA rectangle, see example in Fig. 10. This is done for all possible arrangements of p_2 big cores in the lower left corner. The best result provides the maximum number $p_1(p_2)$ of LITTLE cores when p_2 big cores are already placed. By repeating this algorithm for $p_2 = 0, 1, 2, \ldots, p$, we get a list of possible core counts.

For the small areas available in our concrete case, we can even give exact figures. The die photo in [10] suggests that the aspect ratios of big and LITTLE cores are 5:3 and 2:1, respectively. The aspect ratio of the FPGA is assumed to be 1:1[10]. As the areas of all three have already been determined in Sect. 4, the side lengths of all three can be computed. Maximum core counts can be seen in Table 9. Reduction of FPGA size for bus area etc. has not been applied.

If an ILP is constructed for the worst case, only constraint 8 must be replaced by constraints

[10] cf.https://forums.xilinx.com/t5/Xcell-Daily-Blog-Archived/FPGAs-Not-Dead-Yet-Thank-you-Very-Much-Kevin-Morris/ba-p/387825.

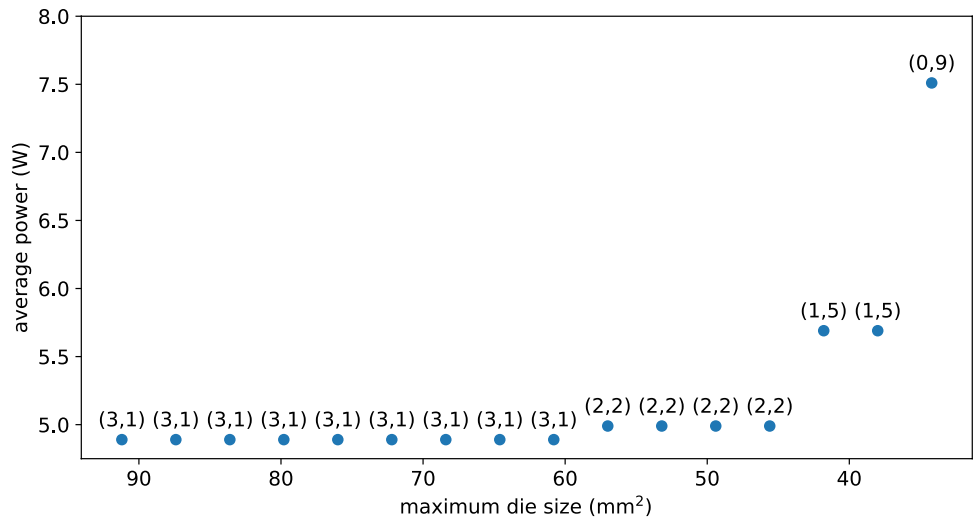

Figure 9 Pareto curve of power vs. chip area for an example task set. Each point on the curve is marked with the core counts (p_2, p_1, i.e. with the number of big and LITTLE cores, respectively,) for that solution. As energy consumption per round equals average power multiplied with the deadline of 3.2 ms, the Pareto curve for energy consumption has a similar shape, with a range of 8 to 25.6 mJ on the y-axis instead of 5 to 8 W.

$$p_2 \leq 2,$$
$$p_1 + 4 \cdot p_2 \leq 20.$$

The largest difference between best and worst case is the maximum number of big cores, which is 2 in the worst and 4 in the best case. For similar numbers of big cores, the maximum number of LITTLE cores is around 20% larger than in the best case.

As the number of LITTLE cores needed in Fig. 9 is below the maximum number even for the worst case, the lower bound curve is not different, but shorter as only 2 big cores can be used.

7 Using Core Types with Core Count other than a Power of 2

If the number p_t of cores of some core type t is not a power of 2, then the last core(s) can only be used to place tasks with small width, cf. Fig. 3. To improve this situation, some groups of larger width might be extended. This has been explored in the context of asymmetric binary core groups in [17], where however the decision where to split a group was done before actual scheduling, i.e. independent of the task set at hand. As it might depend on the task set and the task workloads if using different group sizes is helpful, it seems plausible to let the scheduler decide whether to do this.

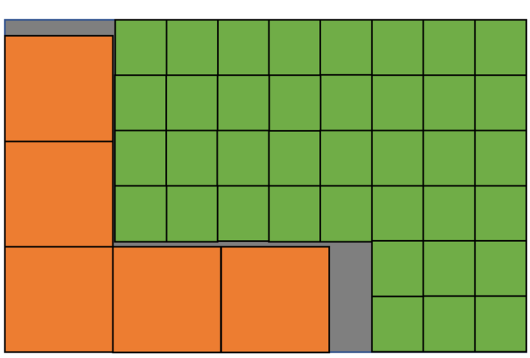

Figure 10 Exemplary tiling arrangement of 5 big cores in the lower left corner and maximum tiling of LITTLE cores in the upper right corner of the FPGA.

We still maintain a binary partitioning of the $p_t \leq p$ cores of each core type t into groups. However, within this framework the group sizes can be determined by the scheduler. We introduce binary variables $size_{i,t,s}$ and integer variables $low_{i,t}$ which indicate the size s of group i on core type t and give the index of this group's first core, respectively ($i = 1, \ldots, 2p-1, s = 0, \ldots, p$).

Then, the following constraints apply:

$$\forall i, t \quad \sum_s size_{i,t,s} = 1,$$

i.e. each group (on each core type) has exactly one size.

$$\forall t \quad \sum_s s \cdot size_{1,t,s} \leq p_t,$$
$$\forall t, i \text{ even} \quad \sum_s s \cdot size_{i,t,s} + \sum_s s \cdot size_{i+1,t,s} \leq \sum_s s \cdot size_{i/2,t,s},$$

i.e. the sum of sizes of two child groups is at most the size of the parent group, and the size of the root group is at most p_t.

$$\forall t \quad low_{1,t} = 0,$$
$$\forall t, i \text{ even} \quad low_{i,t} = low_{i/2,t},$$
$$\forall t, i \geq 3 \text{ odd} \quad low_{i,t} = low_{i/2,t} + \sum_s s \cdot size_{i-1,t,s}.$$

Table 9 Maximum number of LITTLE cores for given number of big cores in FPGA, in best and worst case.

# big	# LITTLE worst	# LITTLE best
0	20	24
1	16	19
2	12	14
3	N/A	9
4	N/A	4

We thus allow groups of size 0 (however no tasks can be mapped to them) and we allow to not use all cores in groups, e.g. if using 4 cores instead of 5 cores is better in Fig. 3.

A consequence of this flexibility is that the mapping, which groups comprise a certain core l, is not static anymore. Hence we introduce binary variables $map_{i,t,l} = 1$ iff group i on core type t comprises core l. This is the case if $low_{i,t} \leq l < low_{i,t} + \sum_s s \cdot size_{i,t,s}$. We thus exclude l if it is outside this range and by requiring that a sufficient number of cores is comprised, we force the variables for the proper cores to 1.

$$\forall t, i, l \quad l + (1 - map_{i,t,l}) \cdot p \geq low_{i,t},$$
$$\forall t, i, l \quad low_{i,t} + \sum_s s \cdot size_{i,t,s} > l - (1 - map_{i,t,l}) \cdot p,$$
$$\forall t, i \quad \sum_l map_{i,t,l} = \sum_s s \cdot size_{i,t,s}.$$

We connect the mapping to the tasks by introducing binary variables $y_{j,s,t,k,l} = 1$ iff task j runs on a group of size s comprising core l of core type t, using frequency level k.

Then constraint 5 can be reformulated as

$$\forall l, t \quad \sum_{j,s>0,k} y_{j,s,t,k,l} \cdot time(s, t, j, k) \leq M.$$

Note that time is dependent on the size, not on the group index itself. We also can reformulate the target function as

$$\min \sum_{j,s>0,t,k,l} y_{j,s,t,k,l} \cdot time(s, t, j, k) \cdot power(t, k, tty(j)) + \ldots$$

The binary variables $x_{i,j,t,k}$ now only serve to assign tasks to groups and frequency levels, i.e. $x_{i,j,t,k} = 1$ iff task j runs in group i on core type t at frequency level k.

Constraints 2 and 3 remain as they are.

We forbid to run tasks on groups of size 0:

$$\forall j, i, t \quad \sum_k x_{i,j,t,k} \leq 1 - size_{i,t,0}.$$

Constraints 4, 6, 7 are not needed anymore. Constraint 8 remains as is.

We still must connect variables x, y, and map to get a consistent solution, i.e. $y_{j,s,t,k,l} = 1$ iff $\exists i : x_{i,j,t,k} = 1 \wedge size_{i,t,s} = 1 \wedge map_{i,t,l} = 1$. As this implies computing the product of $size$ and x variables, we achieve linearization by introducing a further kind of decision variables $z_{i,j,s,t,k}$, where $z_{i,j,s,t,k} = 1$ iff task j runs in group i of size s on core type t and frequency level k. The additional constraints then read:

$$\forall j, i, s, t, k, l \quad y_{j,s,t,k,l} \geq x_{i,j,t,k} + size_{i,t,s} + map_{i,t,l} - 2,$$
$$\forall i, j, s, t, k \quad z_{i,j,s,t,k} \leq x_{i,j,t,k},$$
$$\forall i, j, s, t, k \quad z_{i,j,s,t,k} \leq size_{i,t,s},$$
$$\forall i, j, s, t, k \quad z_{i,j,s,t,k} \geq x_{i,j,t,k} + size_{i,t,s} - 1,$$
$$\forall j \quad \sum_{s,t,k,l} y_{j,s,t,k,l} = \sum_{i,s,t,k} s \cdot z_{i,j,s,t,k}.$$

The first constraint yields $y_{j,s,t,k,l} = 1$ if $\exists i : x_{i,j,t,k} = 1 \wedge size_{i,t,s} = 1 \wedge map_{i,t,l} = 1$, while the remaining four constraints serve to determine the number of $y_{j,s,t,k,l}$ set to 1 for each task, which collectively ensures that $y_{j,s,t,k,l} = 0$ whenever the above implication does not hold.

We illustrate the advantage of this relaxation with a small task set comprised from $n = 4$ tasks, and for the situation where only $p_t = 3$ cores of a single core type (big) are available, i.e. $p = 4$, for which the crown has $2p - 1 = 7$ groups, cf. left part of Fig. 2. The deadline is fixed to $M = 1$ second.

Task T1 has a workload work(1) $= 2.24 \cdot 10^9$ cycles and speedup speedup$(1, 1) = 1$ and speedup$(1, q) = 0.8q$ for $q \geq 2$. Thus, parallel execution of this task incurs some overhead, but there is a sufficient degree of parallelism so that parallelization for any (small) number of cores occurs with constant efficiency 0.8. The workload of this task is chosen such that it can complete on two cores at highest frequency 1.4 GHz until the deadline. A sequential execution of the task will not meet the deadline. If the task is run on $q = 3$ cores, then a frequency of 1 GHz is sufficient to meet the deadline, with a runtime of 0.9333 seconds.

The three other tasks are identical and sequential. Their workload is chosen such that those tasks can be executed in the remaining 0.0666 seconds after task T1 has completed at the lowest possible frequency 600 MHz, i.e., work$(j) = 0.04 \cdot 10^9$ for $j = 2, 3, 4$.

If task T1 is run on 2 cores, then the other three tasks will be run in sequence on the third core, cf. Fig. 11 (top). If task T1 is run on 3 cores, then the other three tasks will be run concurrently on the three cores after task T1 has completed, see Fig. 11 (bottom). However, this latter scenario is only possible with the above relaxation.

The energy of the 3 small tasks is the same in both scenarios. The energy for the large task is lower in the 3-core scenario, as the processor-cycle-product is the same in both scenarios, but the frequency and thus power consumption is lower than if the task runs on 2 cores.

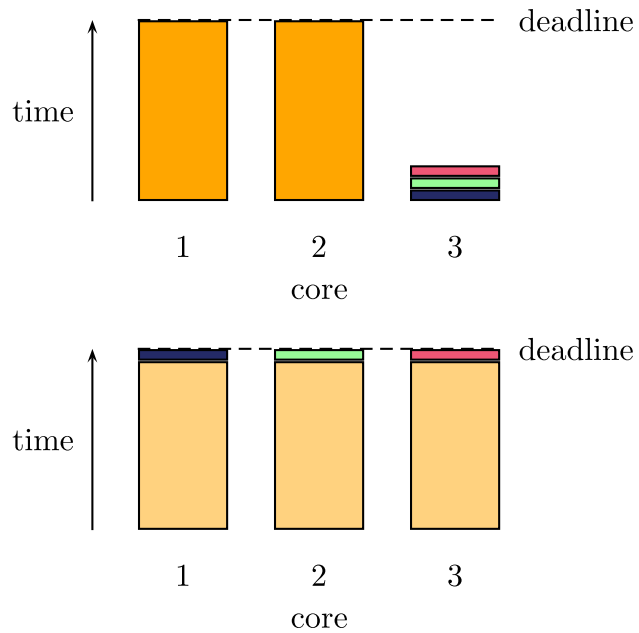

Figure 11 Energy-efficient schedules for example task set on three cores with allocations as powers of 2 (top) and with arbitrary allocations (bottom).

8 Threats to Validity

In this section, we discuss possible limitations of our method with regard to number of core types, use of rectangular layouts and influence of task interference on task runtimes.

While we only demonstrated applicability of our method to tightly integrate DSE and crown scheduling for 2 and 3 core types (cf. Sect. 4), larger numbers of core types are already foreseen in our ILP model and by extrapolating from Table 8 also seem computationally feasible for moderately sized task graphs.

Our assumption of rectangular layouts of the core types for the area constraint, motivated by publically available ASIC implementation data, is of course subject to the used hardware (FPGA vs. ASIC) and design tool (component libraries, placement, routing) technology, which we do not model in further detail, and thus we only considered an upper bound for area consumption and lower bound for accommodatable core counts, respectively, as described in Sect. 6. The real plasticity of pre-layouted core modules is expected to be somewhere in between this lower bound and the upper bound obtained by considering flat area limits only. The actual placement is considered a third-party plug-in to our method; in fact, the placements (as by the heuristic in Sect. 6) can be pre-computed offline for each possible configuration and then quickly looked up in the ILP model at solving time.

The task time parameters are derived by microbenchmarking on the concrete target core type for all possible core allocations and DVFS levels, but might, for execution on core subgroups, be sensitive to co-scheduled other tasks of different type that also compete for shared resources, in particular, main memory access bandwidth. This is mainly a problem for memory-intensive tasks. As crown scheduling does, by design, not constrain co-scheduling of tasks in disjoint core groups with respect to memory bandwidth, the microbenchmarking will have to co-schedule a task against the most memory-access-intensive task type available if used in scenarios where worst-case task execution times are critical.

9 Related Work

Design space exploration and scheduling have been considered for configurable platforms since decades, cf. e.g. [18] and the references therein, however in largely varying contexts and with widely differing goals.

Holzer et al. [19] and Duhem et al. [20] consider the combination of design space exploration and scheduling in the context of reconfiguration at runtime, while we consider static scheduling. Sarma and Dutt [21] consider design space exploration of heterogeneous chip multiprocessors with a given power or area budget for several given core types, with the goal of optimizing for energy, performance or a combination of these. In contrast to our work, they use sequential tasks with dynamic scheduling by the operating system scheduler, and DVFS is not considered. Uscumlic et al. [22] and Li et al. [23] consider design space exploration and scheduling via ILP and evolutionary computation, respectively, but in the application, i.e. for a fixed platform, while we target an architecture where different numbers and types of (soft) cores can be used.

The closest related work seems to be [24]. They combine modulo scheduling and design space exploration, and solve for performance and energy efficiency, i.e., they generate a Pareto front of solutions so that the designer has a choice. However, their tasks are instructions in a high-level hardware design language, and their resources are operation units like adders or multipliers, while our tasks are parallelizable programs in itself, and our resources are different types of cores.

10 Conclusion and Future Work

We have presented an approach to combine design space exploration and static scheduling for executing streaming task graph applications with parallelizable tasks on a configurable, heterogeneous platform, i.e. to select a collection of cores or softcores of different types that achieve the required throughput and minimize energy consumption per round, i.e. average power consumption. Our experiments indicate that

integrating the design space exploration with the scheduling does not notably increase scheduling time, and at the same time improves energy efficiency by 19% and 50% on average via selecting optimum core counts for two and three core types, respectively, within a given area available in a configurable architecture.

Moreover, we have exemplified that the approach can be generalized for other targets than energy efficiency, such as minimum area solution for a given power budget and throughput, and to produce a Pareto front relating two optimization goals like minimum power budgets for different ASIC or FPGA areas, given a required throughput. By determining core counts not only via area constraints but also geometric constraints, lower bounds on the number of usable cores can be found that indicate core counts that can be expected in a real implementation.

Future work will comprise to validate our experiments on a real configurable platform.

Moreover, the architecture area model (and the core packing heuristic) could be refined e.g. by introducing explicit area parameters and constraints to model the shared uncore area cost for core-type clusters.

Acknowledgements C. Kessler acknowledges partial funding by ELLIIT, project GPAI.

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Keller, J., Litzinger, S., & Kessler, C. (2021). Combining design space exploration with task scheduling of moldable streaming tasks on reconfigurable platforms. In S. Derrien, F. Hannig, P. C. Diniz, & D. Chillet (Eds.), *Applied Reconfigurable Computing. Architectures, Tools, and Applications - 17th International Symposium, ARC 2021, Virtual Event, June 29-30, 2021, Proceedings* (pp. 93–107). Springer volume 12700 of *Lecture Notes in Computer Science*.
2. Kahn, G. (1974). The semantics of a simple language for parallel programming. In *Proc. IFIP Congress on Information Processing* (pp. 471–475). North-Holland.
3. Leung, J. Y.-T. (Ed.) (2004). *Handbook of Scheduling*. Boca Raton, FL: Chapman & Hall/CRC.
4. Melot, N., Kessler, C., Keller, J., & Eitschberger, P. (2015). Fast Crown scheduling heuristics for energy-efficient mapping and scaling of moldable streaming tasks on manycore systems. *ACM Transactions on Architecture and Code Optimization*, *11*.
5. Litzinger, S., Keller, J., & Kessler, C. (2019). Scheduling moldable parallel streaming tasks on heterogeneous platforms with frequency scaling. In *Proc. 27th European Signal Processing Conference (EUSIPCO 2019)*.
6. Kessler, C. W., Melot, N., Eitschberger, P., & Keller, J. (2013). 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).
7. Melot, N., Kessler, C., Keller, J., & Eitschberger, P. (2019). 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), Aachen, Germany*. IEEE.
8. Melot, N., Kessler, C., & Keller, J. (2016). Improving energy-efficiency of static schedules by core consolidation and switching off unused cores. In *Proc. Int. Conf. on Parallel Computing (ParCo'15)* (pp. 285 – 294). IOS Press.
9. Holmbacka, S., & Keller, J. (2017). Workload type-aware scheduling on big.LITTLE platforms. In *Algorithms and Architectures for Parallel Proc.* (pp. 3–17). Springer.
10. Shimpi, A. L. (2013). Samsung Details Exynos 5 Octa Architecture & Power at ISSCC '13. https://www.anandtech.com/show/6768/. Accessed: 2021-03-01.
11. Eindhoven Technical University, Electronic Systems. (2010). Dataflow Benchmark Suite (DFbench). http://www.es.ele.tue.nl/dfbench/
12. Hautala, I. (2019). *From dataflow models to energy efficient application specific processors*. Ph.D. thesis Oulu University, Finland.
13. Litzinger, S., & Keller, J. (2021). Code generation for energy-efficient execution of dynamic streaming task graphs on parallel and heterogeneous platforms. *Concurrency and Computation: Practice and Experience*, *n/a*, e6072.
14. Holmbacka, S., & Müller, R. (2017). epEBench: True energy benchmark. In I. Merelli, P. Lio, & I. V. Kotenko (Eds.), *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)* (pp. 426–429).
15. Fekete, S. P., & Schepers, J. (2004). A combinatorial characterization of higher-dimensional orthogonal packing. *Mathematics of Operations Research, 29,* 353–368.
16. Garey, M. R., & Johnson, D. S. (1979). *Computers and intractability, a guide to the theory of NP-completeness*. Freeman.
17. Torggler, M., Keller, J., & Kessler, C. W. (2017). Asymmetric crown scheduling. In I. V. Kotenko, Y. Cotronis, & M. Daneshtalab (Eds.), *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2017, St. Petersburg, Russia, March 6-8, 2017* (pp. 421–425). IEEE Computer Society.
18. Maestre, R., Kurdahi, F., Fernandez, M., Hermida, R., Bagherzadeh, N., & Singh, H. (2001). Kernel scheduling techniques for efficient solution space exploration in reconfigurable computing. *Journal of Systems Architecture, 47,* 277–292.
19. Holzer, M., Knerr, B., & Rupp, M. (2007). Design space exploration for real-time reconfigurable computing. In *Proc. 41st Asilomar Conference on Signals, Systems and Computers* (pp. 1981–1985).
20. Duhem, F., Muller, F., Aubry, W., Le Gal, B., Négru, D., & Lorenzini, P. (2013). Design space exploration for partially reconfigurable architectures in real-time systems. *Journal of Systems Architecture, 59,* 571–581.
21. Sarma, S., & Dutt, N. (2015). Cross-layer exploration of heterogeneous multicore processor configurations. In *Proc. 28th Int. Conf. on VLSI Design and 14th Int. Conf. on Embedded Systems*. IEEE.

22. Uscumlic, B., Enrici, A., Pacalet, R., Gharbi, A., Apvrille, L., Natarianni, L., & Roullet, L. (2020). Design space exploration with deterministic latency guarantees for crossbar mpsoc architectures. In *Proc. IEEE International Conference on Communications (ICC)* (pp. 1–7).
23. Li, Z., Park, H., Malik, A., Wang, K.I.-K., Salcic, Z., Kuzmin, B., et al. (2017). Using design space exploration for finding schedules with guaranteed reaction times of synchronous programs on multicore architecture. *Journal of Systems Architecture, 74,* 30–45.
24. Oppermann, J., Sittel, P., Kumm, M., Reuter-Oppermann, M., Koch, A., & Sinnen, O. (2019). Design-space exploration with multi-objective resource-aware modulo scheduling. In R. Yahyapour (Ed.), *Euro-Par 2019: Parallel Processing* (pp. 170–183). Cham: Springer International Publishing.

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.

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 with the Parallelism & VLSI 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.

Christoph Kessler received the Ph.D. degree in computer science from the Universität des Saarlandes, Saarbrücken, Germany, in 1994. He is a Professor with 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.