Crown Scheduling: Energy-Efficient Resource Allocation, Mapping and Discrete Frequency Scaling for Collections of Malleable Streaming Tasks

Christoph W. Kessler
Nicolas Melot
Dept. of Computer and Information Science
Linköping University
58183 Linköping, Sweden
⟨firstname⟩.⟨lastname⟩@liu.se

Patrick Eitschberger
Jörg Keller
Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
⟨firstname⟩.⟨lastname⟩@fernuni-hagen.de

***Abstract*—We investigate the problem of generating energy-optimal code for a collection of streaming tasks that include parallelizable or malleable tasks on a generic manycore processor with dynamic discrete frequency scaling. Streaming task collections differ from classical task sets in that all tasks are running concurrently, so that cores typically run several tasks that are scheduled round-robin at user level in a data driven way. A stream of data flows through the tasks and intermediate results are forwarded to other tasks like in a pipelined task graph. In this paper we present *crown scheduling*, a novel technique for the combined optimization of resource allocation, mapping and discrete voltage/frequency scaling for malleable streaming task sets in order to optimize energy efficiency given a throughput constraint. We present optimal off-line algorithms for separate and integrated crown scheduling based on integer linear programming (ILP). We also propose extensions for dynamic rescaling to automatically adapt a given crown schedule in situations where not all tasks are data ready. Our energy model considers both static idle power and dynamic power consumption of the processor cores. Our experimental evaluation of the ILP models for a generic manycore architecture shows that at least for small and medium sized task sets even the integrated variant of crown scheduling can be solved to optimality by a state-of-the-art ILP solver within a few seconds.**

I. Introduction

We consider the steady state of a streaming task application where all streaming tasks are active simultaneously. Each task repeatedly consumes some amount of input, does some computation, and produces output that is forwarded to another task (or to memory if it is a final result). Streaming task applications are widespread in embedded systems, in particular as applications for multiprocessor systems on a chip (MPSoC). Examples include image processing and video encoding/decoding pipelines, and other applications operating on large data volumes, e.g. streamed mergesort or mapreduce task graphs.

A streaming task application can be modeled by a static streaming task graph, where the nodes represent the tasks, but are not annotated with runtimes but with average computational rates. An edge between tasks u and v does not indicate a precedence constraint but denotes a communication of outputs from producer task u to become inputs of consumer task v, and is annotated with an average bandwidth requirement. As the communication is typically done in the form of fixed-size packets, one can assume that the communication rate indicates how often a packet is transferred, and the computation rate indicates how much computation is done to turn an input packet into an output packet.

As the workloads of different tasks may vary around the given averages and as there are normally more tasks than cores on the underlying machine, several tasks might run concurrently on the same core. In this case, a scheduling point is assumed to occur after the production of a packet, and a round-robin non-preemptive user-level scheduler normally is sufficient to ensure that each task gets its share of processor time, provided the core has enough processing power, i.e. is run at a frequency high enough to handle the total load from all tasks mapped to this core.

Here we consider *streaming task collections*, i.e., we model the tasks' computational loads only. Our results might thus be applied to *streaming task graphs* only as long as the latencies of producer-consumer task communications can be mostly hidden by pipelining with multi-buffering and if on-chip network link bandwidths are not oversubscribed.

We assume that our underlying machine consists of p identical processors, which can be frequency-scaled independently. We consider discrete frequency levels. We do not consider voltage scaling explicitly, but as most processors auto-scale the voltage to the minimum possible for a given frequency, this is covered as well.

We allow for *malleable* (multithreaded, aka. parallelizable) tasks that can internally use a parallel algorithm involving multiple processors (possibly using communication and synchronization via shared memory or message passing) in order to achieve parallel speedup. We make no assumptions about the parallel efficiency functions of malleable tasks; these are parameters of our model. Malleable, partly malleable and sequential tasks might be mixed.

We are interested in (1) allocating processing resources to tasks and (2) mapping the tasks to processors in such a way that, after (3) suitable task-wise frequency scaling, overall power consumption during one full round of the round-robin scheduler is minimized. We refer to this combined optimization problem (1–3) shortly as *energy-efficient scheduling*.

In this respect, we make the following contributions:

978-1-4799-1170-7/13/$31.00 ©2013 IEEE

- We introduce the *crown structure* as a constraint on resource allocation, which reduces the complexity of allocation, mapping and discrete frequency scaling considerably, namely from p to $O(\log p)$ possible task sizes for allocation and from 2^p to $O(p)$ possible processor subsets for mapping. We show that this constraint makes the exact solution of the considered NP-hard optimization problem feasible even for medium problem sizes.
- We present *crown scheduling*, a novel method for combining resource allocation and mapping of malleable streaming tasks to parallel cores with dynamic scheduling. This allows to mix malleable and sequential streaming tasks in a round-robin data-driven schedule such that starvation of malleable tasks using many cores simultaneously is avoided.
- We show how to apply discrete frequency scaling of the tasks in a given crown schedule to optimize its energy efficiency for a given throughput constraint.
- We show how the optimization of the crown schedule can be combined with optimization of discrete frequency scaling to obtain an integrated optimal solution.
- We show that crown schedules are especially flexible for dynamic rescaling to adapt to fluctuations in streaming task load at runtime, in order to save further energy e.g. in situations where a task may not be data-ready in a schedule round and has to be skipped.
- We give implementations as ILP models and evaluate them for a generic manycore architecture. We show that the schedule quality improves by integration of the subproblems, and that even the integrated formulation can be solved to optimality for small and most medium sized task sets within a few seconds.

The remainder of this paper is organized as follows: Section II introduces the general concepts and central notation. Section III considers the separate optimization of crown resource allocation/mapping and subsequent frequency/voltage scaling, while Section IV provides the integrated solution of crown resource allocation, mapping and scaling in the form of an integer linear programming model that can provide an optimal solution for small and medium sized problems. Section V addresses dynamic crown rescaling. Section VI presents experimental results. Section VII generalizes the crown structure to better adapt its constraints to given architectural structures and task set properties in a preceding crown configuration phase. Section VIII discusses crown vs. non-crown scheduling. Section IX discusses related work, and Section X concludes the paper and proposes some future work.

II. Crown Scheduling of Malleable Streaming Tasks

We consider a generic multicore architecture with p identical cores. For simplicity of presentation, we assume in the following that $p = 2^L$ is a power of 2.[1]

[1]A generalization towards (non-prime) p that are not powers of 2, such as $p = 48$ for Intel SCC, derives from a recursive decomposition of p into its prime factors a corresponding multi-degree tree structure instead of the binary tree structure described in the following for organizing processors in hierarchical processor groups.

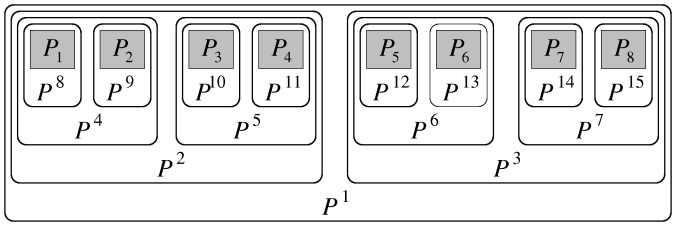

Fig. 1. Processor groups in a binary crown over 8 processors.

The set of processors $P = \{P_1, ..., P_p\}$ is hierarchically structured into $2p - 1$ *processor subgroups* by recursive binary partitioning as follows: The *root group* P^1 equals P; it has the two child subgroups $P^2 = \{P_1, ..., P_{p/2}\}$ and $P^3 = \{P_{p/2+1}, ..., P_p\}$, four grandchildren groups $P^4 = \{P_1, ..., P_{p/4}\}$ to $P^7 = \{P_{3p/4+1}, ..., P_p\}$ and so on over all $L + 1$ tree levels, up to the *leaf groups* $P^p = \{P_1\}$,..., $P^{2p-1} = \{P_p\}$, see also Figure 1. Unless otherwise constrained, such grouping should also reflect the sharing of hardware resources across processors, such as on-chip memory shared by processor subgroups. Let G_i denote the set of all groups P^q that contain processor P_i. For instance, $G_1 = \{P^1, P^2, P^4, P^8, ..., P^p\}$. Let $p_i = |P^i|$ denote the number of processors in processor group P^i. Where it is clear from the context, we also write i for P^i for brevity.

We consider a set $T = \{t_1, ..., t_n\}$ of n malleable, partly malleable or sequential streaming tasks, where each task t_j performs *work* τ_j and has a *maximum width* $W_j \geq 1$ and an *efficiency function* $e_j(q) > 0$ for $1 \leq q \leq p$ that predicts the parallel efficiency (i.e., parallel speedup over q) with q processors. For *malleable tasks*, W_j will be ∞, i.e., unbounded; for *partly malleable tasks*, W_j can be any fixed value > 1, and for *sequential tasks* $W_j = 1$. For all tasks t_j we assume that $e_j(1) = 1$, i.e., no parallelism overhead when running on a single processor.[2] Where clear from the context, we also write j as shorthand for t_j.

Resource allocation assigns each task t_j a *width* w_j with $1 \leq w_j \leq \min(W_j, p)$, for $1 \leq j \leq n$. As additional constraint we require that w_i be a power of 2, and thus could be mapped completely to one of the $2p-1$ processor subgroups introduced above.

A *crown mapping* is a mapping of each task t_j with assigned width w_j to one of the processor subgroups in $\{P^1, ..., P^{2p-1}\}$ of a size matching w_j. For each processor group P^i, $1 \leq i \leq 2p-1$, let T^i denote the set of tasks $t_j \in T$ that are mapped to P^i. For each processor P_i, $1 \leq i \leq p$, let T_i denote the set of tasks $t_j \in T$ mapped to P_i, i.e., to any $P^i \in G_i$. Conversely, let R_j denote the group to which task t_j is mapped.

For each processor P_i, we now order the tasks in each T_i in decreasing order of width, e.g., by concatenating the elements of all $T^k \in G_i$ in increasing order of group index k. The relative order of the tasks within each processor group must

[2]In principle, partial malleability might be also expressed by manipulating the e_j functions, but in this paper we use the upper limits W_j for this purpose. — We may assume that the efficiency functions e_j are monotonically decreasing (where adding resources would increase time, they are ignored), although this assumption is not strictly necessary for the techniques of this paper.

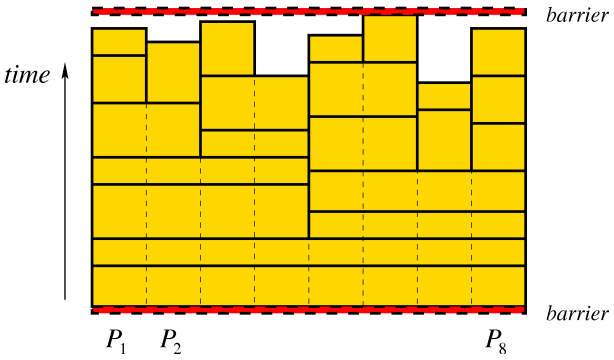

Fig. 2. A crown schedule across 8 processors.

be kept the same across all its processors, e.g., in increasing task index order. We call such a sequence $\langle t_{s_1}, ..., t_{s_{l_s}} \rangle$ a *crown schedule*.

A *crown scheduler* is a user-level round-robin scheduler running on each processor P_i, $1 \leq i \leq p$, that uses a crown schedule for the (instances of) tasks in T_i to determine their order of execution. A crown scheduler works in *rounds*. Each round starts with a global barrier synchronization across all p processors. Thereafter the tasks are executed round-robin as specified in the chosen crown schedule of the tasks in T_i. Tasks that are not data ready are skipped. Then the next round follows. The stable state of a round in a crown-based streaming computation across all processors, which we also call "the crown", will involve (almost) all tasks in T as they all have data to process. See Figure 2 for an illustration.

The use of global barrier synchronization and the processing of tasks in decreasing order of width is required to make sure that all processors participating in the same malleable task (instance) start and stop approximately at the same time, allowing for efficient interprocessor communication and avoiding idle times, except for the round-separating barrier. Barriers between tasks (even of different widths) within a crown are not required because the processors of the subgroup assigned to executing the subsequent task will automatically start quasi-simultaneously after finishing the previous common task where they also participated; a malleable task is either data-ready or not, which means it will be skipped either by all or none of the processors in the assigned processor group.

If no task with width p occurs, the global barrier (i.e., at root group level 1) can be relaxed into separate subgroup-wide barriers (i.e., at higher levels in the hierarchy). The largest occuring width determines the (minimum) barrier size needed to start up a new crown.

The constraint to use powers of 2 as task widths makes the mapping and scaling problems considerably simpler; we can capture it as a bin packing problem (see below). If arbitrary task widths were allowed, we would obtain a strip packing problem [1] instead.

III. Phase-Separated Energy-Efficient Crown Scheduling

In this section we discuss separate optimization algorithms for each of the subproblems crown resource allocation, crown mapping and crown scaling. An integrated and more expensive approach will be discussed in Section IV. An overview of these approaches is given in Figure 3.

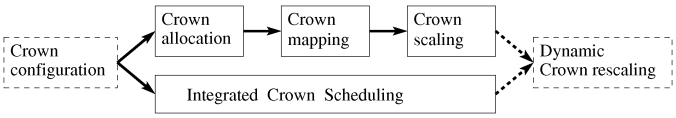

Fig. 3. Overview of step-wise vs. integrated crown scheduling. The methods shown by solid boxes are detailed in this paper.

A. Heuristic Resource Allocation Algorithms

The group structure constraint also allows to partly decouple the problems of resource allocation and mapping. In a phase-decoupled solution, the resource allocation must be determined before mapping.

For now, we could apply simple heuristic resource allocation policies such as: (R1) assign each task j width $w_j = 1$; (R2) assign each task j its maximum width, $w_j = \min(W_j, p)$; (R3) assign each task j a proportional width $w_j = \min(W_j, p\lceil\!\lceil \tau_j / \sum_k \tau_k \rceil\!\rceil)$; etc., where τ_j denotes the work of task j and $\lceil\!\lceil y \rceil\!\rceil$ denotes the nearest power of 2 larger than or equal to y.

B. Heuristic Crown Mapping Algorithms

In the subsequent mapping step, we first sort by decreasing width. The tasks of a certain width can be mapped to the processor groups of that width by a suitable greedy heuristic, e.g. Earliest Finishing Time first (EFT) which sorts the tasks by decreasing work, and then assigns them one by one in an attempt to balance the workload across all processors.

C. Optimal Crown Resource Allocation

Let Q_z, where $z = 0, ..., \log_2 p$, be the set of indices of all the tasks which will be assigned width $w_j = 2^z$. Then task t_j ($j \in Q_z$) has runtime $\hat{\tau}_j = \tau_j / (e_j(w_j) w_j)$. The runtime of these tasks on p processors is

$$R_z = \sum_{j \in Q_z} \frac{\hat{\tau}_j}{p/2^z} \tag{1}$$

as there are $p/2^z$ processor groups of size 2^z each. Note that we assume as a simplification that all tasks with the same width can be distributed over the processor groups in a balanced fashion.

Optimal resource allocation chooses the Q_z in a way to minimize

$$\sum_{z=0}^{\log p} R_z$$

i.e., we minimize the total runtime assuming that the processors run at a fixed frequency and the tasks of one width can be distributed optimally. To do this by linear optimization, we introduce binary variables $y_{j,z}$ with $y_{j,z} = 1$ iff task j gets width $w_j = 2^z$. Then Eqn. 1 transforms to

$$R_z = \sum_j \frac{\hat{\tau}_j y_{j,z}}{p/2^z}$$

and we need the constraint

$$\forall j : \sum_z y_{j,z} = 1 \, .$$

D. Optimal Crown Mapping Algorithm

For an optimal crown mapping, we treat all widths separately, i.e., we solve $\log_2 p$ smaller optimization problems. For the tasks t_j of a given width $w_j = 2^z$, we try to distribute them over the p/w_j processor groups available such that the maximum load over any group is minimized.

In order to formulate the mapping of the tasks of a given width 2^z as an integer linear program, we minimize a variable $\mathit{maxload}_z$ (note that the target function here is simple) under some constraints. To express the constraints, we use binary variables $y_{j,q}$ where j runs (in consecutive numbering) over all tasks with width 2^z and q runs over all considered $p/2^z$ processor groups $P^{p/w_j}, \ldots, P^{2p/w_j-1}$. Hence, $y_{j,q} = 1$ iff task j is mapped to processor group q.

The constraints are as follows. Task j is mapped to exactly one processor group among those that have the right width w_j:

$$\forall j \text{ with width } w_j = 2^z : \quad \sum_{q=p/w_j}^{2p/w_j-1} y_{j,q} \;=\; 1$$

and the maximum load is determined as follows:

$$\forall q = p/w_j, \ldots, 2p/w_j - 1 : \; \sum_j \hat{\tau_j} \cdot y_{j,q} \leq \mathit{maxload}_z$$

E. Voltage/Frequency Scaling of Crowns

Any given fixed rate of producing output packets can now be translated into "crowns per second" for the steady state of the pipeline. Its inverse M is the upper bound for the makespan of one round of the crown schedule across all processors. We assume that when running all cores at maximum speed f_1 all the time, the resulting makespan will be $\hat{M} \leq M$. The gap $M - \hat{M}$ and any processor-specific idle times at the "upper end" of the crown can then be leveraged for voltage/frequency scaling to obtain better energy efficiency. We call this "scaling the crown".

We assume that voltage/frequency scaling for individual cores and tasks is supported by the target architecture, and that the (average) dynamic power consumption of a processor running at frequency f is proportional to f^α for some technology-dependent constant $\alpha \approx 3$. Static power consumption is assumed to be linearly proportional to the execution time. For a concrete machine, there is no real need to derive an analytical power model. It is sufficient to use results from power measurements of the target machine at the available frequencies [2].

For a given crown, energy efficiency can be optimized by each processor running each of its tasks t_j requiring work τ_j at a frequency F_j chosen from a given set $\{f_1, ..., f_s\}$ of s discrete (voltage and) frequency levels $k = 1, ..., s$, such that it still meets the throughput constraint

$$\forall i = 1, \ldots, p : \quad \mathit{time}_i \;:=\; \sum_{j \in T_i} \frac{\tau_j}{F_j w_j e_j(w_j)} \;\leq\; M$$

and the overall energy usage

$$\sum_{j=1}^{n} \frac{F_j^{\alpha-1} \tau_j}{e_j(w_j)} \;+\; \zeta \sum_{i=1}^{p} (M - \mathit{time}_i)$$

is minimized, subject to the additional constraint that a group of processors must use the same voltage/frequency level for a common task. The parameter ζ models the importance of static power consumption during idle time in relation to the dynamic power; here we assume that the processors switch to the lowest possible voltage/frequency f_s wherever (expected) idle time $M - \mathit{time}_i$ exceeds a processor-specific threshold time.[3]

IV. Integrated Energy-efficient Crown Scheduling

Mapping a crown from a given resource allocation and a-posteriori scaling of the resulting crown may lead to suboptimal results compared to co-optimizing resource allocation, mapping and discrete scaling from the beginning.

The integration of resource allocation, crown mapping and crown scaling is modeled as a bin packing problem: The $2p - 1$ processor groups with their s discrete scaling levels are represented by s sets of $2p - 1$ bins, the bins in each set S_k having a (time) capacity equivalent to the frequency f_k at level $k = 1, \ldots, s$. The cost $f_k^{\alpha-1}(\tau_j/p_i)/e_j(p_i)$ of a bin i in set S_k is equivalent to the power consumption of a processor at frequency f_k.

From the bin packing model we construct a linear program that uses $(2p-1) \cdot s \cdot n$ binary variables $x_{i,k,j}$ where $x_{i,k,j} = 1$ denotes that task j has been assigned to bin i of set S_k, i.e. to processor group P^i running at frequency level k.

We require that each task be allocated and mapped to exactly one bin:

$$\forall j = 1, \ldots, n : \quad \sum_{i=\max(p/W_j,1)}^{2p-1} \sum_{k=1}^{s} x_{i,k,j} = 1 \; .$$

and forbid mapping a task with limited width W_j to a bin i corresponding to an oversized group:

$$\forall j = 1, \ldots, n : \quad \sum_{i=1}^{\max(p/W_j,1)-1} \sum_{k=1}^{s} x_{i,k,j} = 0 \; .$$

Another constraint asserts that no processor is overloaded and the throughput constraint M is kept:

$$\forall i' = 1, ..., p : \; \mathit{time}_{i'} \;:=\; \sum_{i \in G_{i'}} \sum_{k=1}^{s} \sum_{j=1}^{n} x_{i,k,j} \cdot \frac{\tau_j}{p_i e_j(p_i) f_k} \leq M$$

Then we minimize the target energy function

$$\sum_{i=1}^{2p-1} \sum_{k=1}^{s} f_k^{\alpha-1} \cdot \sum_{j=1}^{n} x_{i,k,j} \cdot \frac{\tau_j}{e_j(p_i)} + \zeta \sum_{i=1}^{p} (M - \mathit{time}_i)$$

where again ζ weighs static energy consumed during residual idle times against dynamic energy (first term).

[3] Note that idle time due to load imbalance cannot generally be optimized away by discrete scaling. As an example, consider a twin group, say $\{P_1, P_2\}$, where one of the processors (say P_1) gets assigned a sequential task but not the other one. As long as continuous scaling is not possible, one of the two processors might incur some idle time.

Note that the coefficients $f_k^{\alpha-1}$, τ_j, p_i, $e_j(p_i)$, ζ are constants in the linear program. Note also that the time and energy penalty for frequency switching is not modelled.

One sees that $(2p-1)\cdot s\cdot n$ variables and $p+n$ constraints are needed to allocate, map and scale n tasks onto p cores with s frequency levels. For problem sizes where it is not possible any more to employ an ILP solver such as gurobi or CPLEX, there exist heuristics for the bin packing problem with fixed costs [3], [4].

V. Dynamic Crown Rescaling

The above methods for crown resource allocation, mapping and scaling optimize for the steady state of the pipeline and there in particular for the ("worst") case that all tasks are data ready and will execute in each round. In practice, however, there might be cases where individual tasks are not yet data ready at the issue time and thus are to be skipped for the current round by the crown scheduler.

If the skipped task j executes at root group level (using all processors), it does not lead to additional imbalance and all processors simply continue with their next task in the crown schedule.

If however the skipped task j was mapped to a subgroup (or even a singleton group) P^i with $|P^i| < p$, skipping j reduces $time_l$ by $\tau_j/(F_j w_j e(w_j))$ which may create considerable load imbalance in the current round[4], which could be remedied by rescaling some "down-crown" tasks of the processors in P^i that follow j in the crown schedule. Note that such rescaling must be consistent across all processors sharing a malleable task, and hence the concurrent search for candidates to rescale must be deterministic. The crown structure can be leveraged for this purpose.

Dynamic rescaling of the current round is an on-line optimization problem that should be solved quickly such that its expected overhead remains small in relation to the saved amount of time. Hence, simple heuristics should be appropriate here. We developed several iterative, linear-time crown rescaling heuristics but skip their presentation here due to lack of space and leave this issue to future work.

VI. Evaluation

We have implemented the three steps of the step-wise approach, namely (locally) optimal crown resource allocation, mapping and frequency scaling, each as an ILP model, and compare it with the integrated approach of Section IV, which is likewise implemented as an ILP model. Hence, in total there are 4 ILP-based optimizers:

- *Crown allocation:* optimizes the resource allocation for each task (see Section III-C). It is statically calculated to the minimum between the total amount of cores available and the task width. A constraint forces allocations to be powers of b (default: $b = 2$) cores; by disabling it we can get a non-crown allocation.
- *Crown mapping:* optimizes the mapping given an allocation. By disabling the power-of-b constraint we can get a non-crown mapping.
- *Crown scaling:* computes the most energy saving frequency setting each task must run at (meaning all involved cores running it) given an allocation, a mapping and target makespan.
- *Integrated Crown Scheduling*: optimizes energy consumption by setting both resource allocation, mapping and frequency given a target makespan.

For evaluation we first use generated synthetic workloads with malleable tasks whose behavior corresponds to the model parameters τ_j, e_j, M etc.

We use randomly generated task sets with $n = 10$, 20, 40, 80 tasks, where the work parameters τ_j follow either a tridiagonal or a uniform random distribution; the upper limit for τ_j values is 40. The work parameter τ_j of these tasks is chosen randomly within a range given by minimum and maximum value and with two different probability distributions (uniform and tridiagonal). For the maximum width W_j, five different settings are considered: *serial* ($W_j = 1\ \forall j$), *low* ($W_j \in \{1, ..., p/2\}$ with uniform random distribution), *average* ($W_j \in \{p/4, ..., 3p/4\}$ with uniform random distribution), *high* ($W_j \in \{p/2, ..., p\}$ with uniform random distribution), and *random* ($W_j \in \{1, ..., p\}$ with uniform random distribution). For a generated task set, the makespan constraint M is determined as the arithmetic mean between the lower bound $\sum \tau_j/(p \cdot \textit{max\_frequency})$ and $2\sum \tau_j/(p \cdot \textit{min\_frequency})$.

The scalability behavior of these tasks is controlled by the maximum width parameter W_i and the efficiency functions e_j defined over $\{1, ..., W_j\}$, which are concave and decreasing as given by

$$e_j(q) = 1 - R\frac{g(q)}{g(W_j)} \quad \text{for } q \le W_j,$$

for a given constant $0 < R < 1$ and a superlinear function $g(q) \in \omega(q)$.[5] Hence, $e_j(q)$ decays concavely from 1 at $q = 1$ downto $1 - R$ at $q = W_j$. In the experiments we used $R = 0.3$ and $g(q) = q^2$.

For each setting (task number, max-width, distributions of τ_j and W_j), 10 different task sets are generated and we report results averaged over these in the following.

For the evaluation we assumed a generic manycore architecture with $p = 1, 2, 4, 8, 16, 32$ cores; clock frequencies could be chosen at $s = 5$ levels with $f_k = k \cdot \textit{min\_frequency}$. For an early energy estimation of the computed (crown) schedule on a generic architecture we use the energy model introduced above with $\alpha = 3$ and $\zeta = 0.1$ (i.e., dynamic energy dominates).

For ILP solving with AMPL 10.1 and Gurobi 5.1 on a standard Linux PC we set a 5 minute timeout per ILP instance. In total, about 200 parameter combinations were generated with 10 task set instances each to be solved with the four ILP optimizers. Due to its simplicity, the crown allocation ILP model could be resolved completely by the AMPL frontend.

Figures 4 and 5 show the predicted quality (energy usage) of the generated crown schedules of the step-wise and the integrated ILP-based optimization methods by the number of cores and tasks, respectively, for all task sets of type *average*. Almost all ILP instances could be solved to optimality within

[4] We conservatively assume that processors in $P \setminus P^i$ will not skip any task, and we consider the mapping as fixed, i.e., do not try to migrate tasks dynamically to deal with such imbalances.

[5] For all $q > W_j$, we assume $e_j(q) = \epsilon$ where $0 < \epsilon \ll 1$.

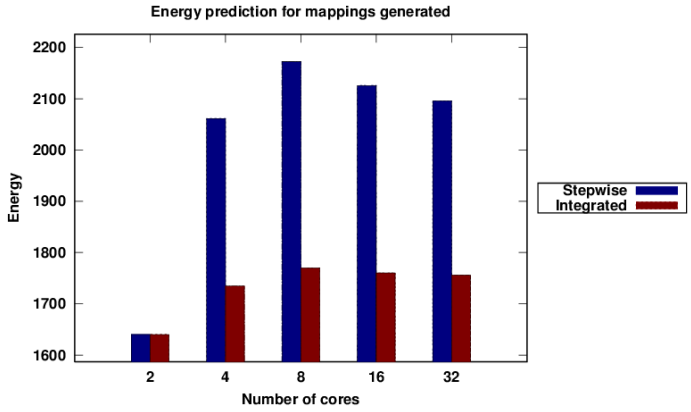

Fig. 4. Average energy quality of crown schedules by number of cores (with $n = 80$ tasks of type *average*), generated with the stepwise and integrated ILP based methods.

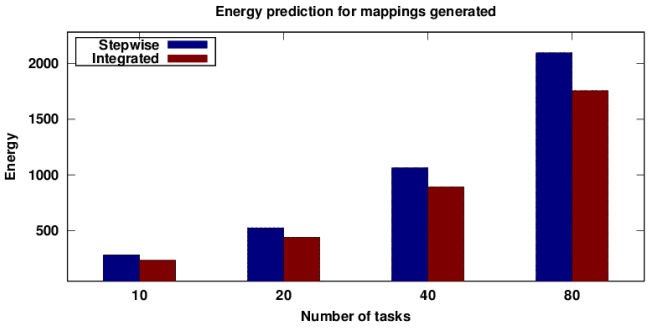

Fig. 5. Average energy quality of crown schedules by number of tasks (with $p = 32$ cores), generated with the stepwise and integrated ILP based methods.

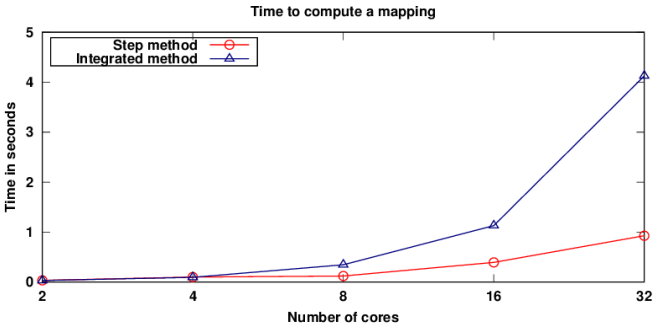

Fig. 6. Averaged optimization times by the stepwise and integrated ILP based methods for generating crown schedules of $n = 80$ tasks of type *average*, by number of cores.

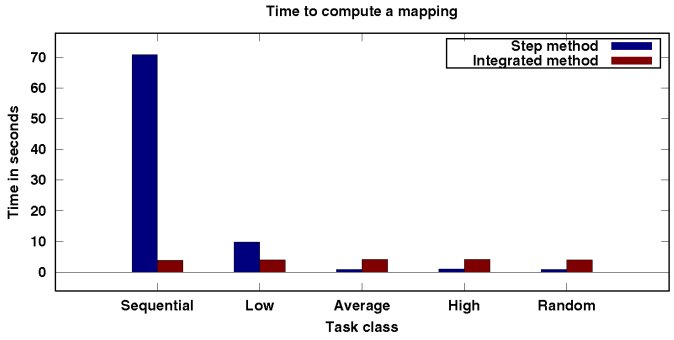

Fig. 7. Averages of optimization times by the stepwise and integrated ILP based methods, by degree of parallelism in tasks.

the given 5-minutes timeout and typically within a few seconds per instance (see Figures 6 and 7).

We can make the following observations:

- Schedule quality (energy predicted by the formula used for optimization) is generally better for the integrated compared to the step-wise ILP-based methods. Even if the integrated approach sometimes hit the 5 minutes timeout and consequently did not return a provably optimal solution, the resulting schedule quality is still better than solutions computed with the stepwise approach.
- The main features that drive solving time appear to be the number of tasks in a task graph and the number of processor groups (which is, thanks to the crown structure constraint, linear in p), see also Figure 6.
- The integrated formulation uses more ILP variables and takes usually longer time than the step-based approach where the average parallelism degree of the malleable tasks is medium to high, but outperforms the stepwise method for sequential and low-parallelism tasks even in optimization time, see Figure 7.

We also observed that the stepwise method benefits (in schedule quality) from additionally taking up load balance into the objective function, which however adds considerably to the solving time especially of the mapping step, and makes it 1–2 orders of magnitude slower than the integrated method which still produces higher quality crown schedules.

Finally, we considered task collections for some concrete computations: parallel FFT, parallel mergesort and parallel reduction. We observed that the integrated method clearly outperformed the stepwise method in schedule quality for FFT and reduction, which is due to the stepwise method choosing too high widths and thus more constraining mapping and frequency scaling. For mergesort, which has sequential (merge) tasks only, the difference is marginal.

VII. Crown Configuration

Investigating the possible performance penalty of the restriction of processor group sizes to powers of 2, we consider the following worst-case example for crown mapping.

Given $T = \{t_1, t_2\}$ as follows:

- Task t_1 is sequential and performs work $\tau_1 = 1$.
- Task t_2 is fully parallel ($e(q) = 1\ \forall q \in \{1, ..., p\}$) and performs work $\tau_2 = p - 1$, where p is the overall number of processors. We assume as before that p is a power of 2.

There are only 2 possibilities (modulo symmetries in time and processor space) for crown resource allocation and mapping:

- **Mapping 1:** Task t_1 is mapped to one processor, w.l.o.g. to P_1. Task t_2 is parallelized with width p and mapped to P^1 using all processors. Then, the makespan of the crown is $2-1/p$, and the idle time on $P_2, ..., P_p$ cannot be remedied by subsequent scaling. (If there is a gap towards M, all scaling should be spent on t_2.)

- **Mapping 2:** Task t_1 is mapped to one processor, w.l.o.g. to P_1, task t_2 to group $P^3 = \{P_{p/2+1}...P_p\}$. Then, the makespan is $2 - 2/p$; $P_2, ..., P_{p/2}$ have nothing to do at all, and P_1 has some idle time that could be remedied by scaling.

In both cases almost 50% of the resources are wasted, mostly because of the crown group structure (and the extreme task set—only 2 tasks, very unequal load), and scaling does not really help with that.

This is a worst-case scenario because adding more sequential or more parallel tasks to T would not make the relative amount of non-scalable idle time larger in an optimal crown mapping. Hence, under the assumption that the task set were not known statically, the default balanced binary crown configuration described in this paper would still be $(2-1/p)$-competitive with respect to resource utilization.

There are two workarounds to handle such extreme situations. First, in some cases the task set is the result of a (design-time) calculation, e.g., adaptive unfolding of a recursive template for task graphs [5]. In such cases, the large task(s) could be unfolded further to give more flexibility to the crown mapper.

A more general solution is to allow *asymmetric crowns*. The decision to split a processor group into two equal-sized subgroups was arbitrary, mostly motivated by symmetric organization of on-chip resources and easy indexing. But conceptually one could of course allow "odd" subgroup sizes, given that we know that the task set is likewise "odd". In other words, we could *configure* the crown structure appropriately based on an analysis of the task set (preprocessing phase), before doing crown resource allocation, mapping and scaling. Developing such analysis is a topic for future work. In the above example scenario, we could have split the root group into a single-processor lane for sequential tasks whereas the remaining $p-1$ processors form a large subgroup suitable for very wide parallel tasks, leading to perfect load balance.

VIII. Crown vs. Non-Crown Scheduling

In this section we consider a worst-case example that compares Crown scheduling to non-Crown scheduling.

Consider a system with p processors (where p is a power of 2), maximum makespan $M = p$, and $n = 2p - 1$ tasks $j = 1, ..., n$ defined as follows:

$$\tau_j := \begin{cases} \text{for odd } j: & (j+1)/2 \\ \text{for even } j: & p - j/2 \end{cases}$$

$$e_j(q) := \begin{cases} \text{for odd } j: & \begin{cases} 1 & \text{for all } q \le (j+1)/2, \\ \epsilon & \text{for all } q > (j+1)/2 \end{cases} \\ \text{for even } j: & \begin{cases} 1 & \text{for all } q \le p - j/2, \\ \epsilon & \text{for all } q > p - j/2. \end{cases} \end{cases}$$

where ϵ, $0 < \epsilon \ll 1$ is a tiny positive value.

Energy-efficient resource allocations will thus not assign more processors (PE) than can be used with efficiency 1 for any task. A non-crown-constrained greedy allocation step will allocate exactly to the cut-off limit because this minimizes energy usage of each task and the makespan constraint still can be achieved with p processors:

A greedy non-crown allocation and subsequent mapping will, following the task numbering, always pair an odd and the next even task together to fill a unit-time slot of all p available processors:

```
Time 0:    Task 1 (on 1 PE)    || Task 2 (p-1 PE);
Time 1:    Task 3 (on 2 PE)    || Task 4 (p-2 PE);
...        ...                    ...
Time p-2:  Task 2p-3 (p-1 PE) || Task 2p-2 (1 PE);
Time p-1:  Task 2p-1 (p PE).
```

resulting in makespan p.

This will require a barrier synchronization after every time step because PEs coming from different tasks $\{j, j+1\}$ with j odd, are joining into a subsequent parallel task from $\{j+2, j+3\}$. Hence we would need $p - 1 = \Theta(n)$ subsequent barriers. The accumulated barrier time overhead also adds to makespan and will require additional frequency scaling to meet the makespan constraint, increasing energy cost beyond the energy overhead of the barriers.

Crown scheduling can do with a single barrier at time 0, now starting with Task $n = 2p - 1$. If resource allocation assigns the maximum possible width to to all tasks (heuristic R2), then for each possible width $w_z = 2^z < p$ there are $2w_z$ tasks t_j with that width assigned, and with runtimes $\tau_j = w_z + k$, where $k = 0, \ldots, w_z - 1$ (two tasks with each runtime, one with odd and one with even task index j). The total load of tasks with width $w_z < p$ is $L_z = 2\sum_{k=0}^{w_z-1}(w_z + k) = 2w_z^2 + w_z(w_z - 1) = 3w_z^2 - w_z$. Additionally, there is one task with width p.

For $\sqrt{p} \le w_z < p$, the number of tasks with width w_z is at least twice as large as the number p/w_z of processor groups of that width, and a snakewise assignment of the tasks (considered in decreasing order of runtime) over the processor groups leads to a balanced load.

If we now consider the tasks with width less than $\sqrt{p}$ (there are at most $0.5\log_2(p)$ of those widths), then their combined load is $\sum_{z=0}^{\lceil 0.5\log_2(p)\rceil - 1} L_z < 3p$, i. e., rather small compared to the total load p^2 of this worst-case example. Also, if the mapper would assign one task per processor group, and would map the task of width w_z with maximum parallel runtime $2 - 1/w_z$ to a group comprising processor 0, then the total runtime of the schedule would be less than $p - 3 + \log_2(p)$: the majority $p^2 - 3p$ of the workload is balanced, for the remaining $0.5\log_2(p)$ widths there is at most one task per group with parallel runtime at most 2.

As a consequence, even for this worst-case scenario, already the non-integrated crown scheduler (and thus also the integrated crown scheduler) finds a makespan that is close to the optimum makespan p and thus there is not much need for frequency scaling to achieve makespan p and associated energy overhead.

IX. Related Work

Energy-aware allocation, mapping and scheduling problems are being researched intensively in scheduling theory. For instance, Pruhs *et al.* have investigated on-line and offline algorithms for energy-efficient resource allocation and scheduling [6] and dynamic frequency scaling [7] of malleable tasks with arbitrary parallel efficiency functions on multiprocessors with dynamic frequency scaling.

Energy efficient static scheduling of task collections with parallelizable tasks onto multiprocessors with frequency scaling has been considered by Li [8]. However, the number of processors allocated to each task is fixed and given as part of the problem scenario, and continuous (not discrete) frequency scaling is used.

Sanders and Speck [9] consider the related problem of energy-optimal allocation and mapping of n independent *continuously malleable* tasks with monotonic and concave speedup functions to m processors with *continuous* frequency scaling, given a deadline M and a continuous convex energy usage function E_j for each task j. Continuous malleability means that also a fractional number of processors (e.g., using an extra processor only during part of the execution time of a task) can be allocated for a task; this is not allowed in our task model where allocations, speedup and energy functions are discrete. They propose an almost-linear work algorithm for an optimal solution in the case of unlimited frequency scaling and an approximation algorithm for the case where frequencies must be chosen between a given minimum and maximum frequency. It is interesting to observe that the assumptions of continous malleability and continuous frequency selection make the integrated problem much easier to solve.

Related approaches for throughput or energy efficient (or multi-objective) mapping of complex pipelines have been developed mainly by the MPSoC community for HW/SW synthesis, e.g. by using genetic optimization heuristics [10].

Hultén et al. [11] and Avdic et al. [12] have considered mapping of streaming task applications onto processors with fixed frequency with the goal of maximizing the throughput. First investigations of energy-efficient frequency scaling in such applications have been done by Cichowski et al. [13].

X. Conclusions and Outlook

We have presented crown scheduling, a new technique for static resource allocation, mapping and discrete frequency scaling that supports data-driven scheduling of a set of malleable, partly malleable and sequential streaming tasks onto manycore processors in order to support energy-efficient execution of on-chip pipelined task graphs.

We have presented heuristics and integer linear programming models for the various subproblems and also for an integrated approach that considers all subproblems together, and evaluated these with synthetic benchmarks. Our experimental results show that the complexity reduction imposed by the crown structure constraint, reducing the number of allocatable processor group sizes from p to $O(\log p)$ and of mappable processor groups from 2^p to $O(p)$, allows for the solution of even medium-sized instances of the integrated optimization problem within a few seconds, using a state-of-the-art integer linear programming solver. The crown structure also minimizes the number of barrier synchronizations necessary.

We have shortly described the increased flexibility for dynamic rescaling due to the crown structure constraint, which allows to easily adapt to fluctuations in streaming task load at runtime.

We have also sketched ways to relax the restrictions of the crown shape, and showed that, in the worst case, an optimal crown schedule is off from an optimal general schedule by a factor of $2 - 1/p$ in overall resource utilization.

Future work will broaden the experimental evaluation, include measurements of the resulting code on a concrete many-core system such as Intel SCC or Kalray MPPA, and in particular add a quantitative comparison to non-crown schedulers, thereby quantifying the average loss (or maybe gain) imposed by the crown structure constraint. Dynamic rescaling of crown schedules will be implemented and evaluated, too.

We will also complement the optimal methods in our implementation by various heuristics to be able to address very large task sets, and investigate optimal crown configuration based on static analysis of the task set. Also, additional constraints on scaling might be considered, such as power domains comprising whole processor subsets as in the case of Intel SCC where frequencies can only be set for groups of 8 processors. Finally, the energy cost of communication between tasks needs to be taken up in the mapping problem.

Acknowledgment

The authors thank Intel for providing the opportunity to experiment with the "concept-vehicle" many-core processor "Single-Chip Cloud Computer". C. Kessler acknowledges partial funding by Vetenskapsrådet and SeRC. N. Melot acknowledges partial funding by the CUGS graduate school at Linköping University.

References

[1] J. Augustine et al., "Strip packing with precedence constraints and strip packing with release times," in *Proc. SPAA'06*. ACM, 2006, pp. 180–189.

[2] P. Cichowski, J. Keller, and C. Kessler, "Modelling power consumption of the Intel SCC," in *Proceedings of the 6th MARC Symposium*. ONERA, July 2012, pp. 46–51.

[3] T. G. Crainic, G. Perboli, W. Rei, and R. Tadei, "Efficient heuristics for the variable size bin packing problem with fixed costs," CIRRELT, Tech. Rep. 2010-18, 2010.

[4] M. Haouari and M. Serairi, "Heuristics for the variable sized bin-packing problem," *Computers & Operations Research*, vol. 36, pp. 2877–2884, 2009.

[5] M. Hashemi et al., "FORMLESS: Scalable utilization of embedded manycores in streaming applications," in *Proc. LCTES'12*. ACM, June 2012, pp. 71–78.

[6] J. Edmonds and K. Pruhs, "Scalably scheduling processes with arbitrary speedup curves," *ACM Trans. Algorithms*, vol. 8, no. 3, p. 28, 2012.

[7] H.-L. Chan, J. Edmonds, and K. Pruhs, "Speed scaling of processes with arbitrary speedup curves on a multiprocessor," *Theory Comput. Syst.*, vol. 49, no. 4, pp. 817–833, 2011.

[8] K. Li, "Energy efficient scheduling of parallel tasks on multiprocessor computers," *J. of Supercomputing*, vol. 60, no. 2, pp. 223–247, 2012.

[9] P. Sanders and J. Speck, "Energy efficient frequency scaling and scheduling for malleable tasks," in *Proc. Euro-Par 2012, LNCS 7484*, 2012, pp. 167–178.

[10] N. Nedjah, M. V. Carvalho da Silva, and L. de Macedo Mourelle, "Customized computer-aided application mapping on NoC infrastructure using multiobjective optimization," *J. of Syst. Arch.*, vol. 57, no. 1, pp. 79–94, 2011.

[11] J. Keller, C. Kessler, and R. Hulten, "Optimized on-chip-pipelining for memory-intensive computations on multi-core processors with explicit memory hierarchy," *Journal of Universal Computer Science*, vol. 18, no. 14, pp. 1987–2023, 2012.

[12] K. Avdic, N. Melot, J. Keller, and C. Kessler, "Parallel sorting on Intel Single-Chip Cloud Computer," in *Proc. A4MMC Workshop on Applications for Multi- and Many-core Processors at ISCA-2011*, 2011.

[13] P. Cichowski, J. Keller, and C. Kessler, "Energy-efficient mapping of task collections onto manycore processors," in *Proc. 5th Swedish Workshop on Mulitcore Computing (MCC 2012)*, Nov. 2012.