Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Many-Core Systems

Nicolas Melot, Linköping University
Christoph Kessler, Linköping University
Jörg Keller, FernUniversität in Hagen
Patrick Eitschberger, FernUniversität in Hagen

Exploiting effectively massively parallel architectures is a major challenge that stream programming can help to face. We investigate the problem of generating energy-optimal code for a collection of streaming tasks that include parallelizable or moldable 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 may be forwarded to other tasks like in a pipelined task graph. In this paper we consider *crown scheduling*, a novel technique for the combined optimization of resource allocation, mapping and discrete voltage/frequency scaling for moldable streaming task collections in order to optimize energy efficiency given a throughput constraint. We first present optimal off-line algorithms for separate and integrated crown scheduling based on integer linear programming (ILP). We make no restricting assumption about speedup behavior. We introduce the fast heuristic *Longest Task, Lowest Group* (LTLG) as a generalization of the Longest Processing Time (LPT) algorithm to achieve a load-balanced mapping of parallel tasks, and the *Height* heuristic for crown frequency scaling. We use them in feedback loop heuristics based on binary search and simulated annealing to optimize crown allocation.

Our experimental evaluation of the ILP models for a generic manycore architecture shows that at least for small and medium sized streaming task collections even the integrated variant of crown scheduling can be solved to optimality by a state-of-the-art ILP solver within a few seconds. Our heuristics produce makespan and energy consumption close to optimality within the limits of the phase-separated crown scheduling technique and the crown structure. Their optimization time is longer than the one of other algorithms we test, but our heuristics consistently produce better solutions.

Categories and Subject Descriptors: C.2.2 [**Multicore and GPU programming**]: Parallel programming

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Multicore Many-core Parallel Energy Scheduling Mapping Frequency Scaling Streaming

ACM Reference Format:
Nicolas Melot, Christoph Kessler, Jörg Keller and Patrick Eitschberger, 2014. Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Many-Core Systems. *ACM* 9, 4, Article 39 (March 2010), 25 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

The increasing attention given to multi- and manycores architectures yields the need for programming models able to take easily profit of their power. Among many solu-

tions to the general parallel programming problem, the streaming approach is particularly adapted to coarse architectures and give opportunities toward the design of highly scalable parallel algorithms [Gordon et al. 2006; Lee and Messerschmitt 1987]. Examples include image processing and video encoding/decoding pipelines, and other applications operating on large data volumes, e.g. streamed mergesort or mapreduce. We consider the steady state of a software pipelined 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).

A streaming task application can be modeled by a static streaming task graph (or a Kahn Process Network [Kahn 1974]) where the nodes represent the tasks, that are not annotated with runtimes but with average computational rates. In the steady state, 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 workloads of different tasks may vary around the given averages and as there are normally more tasks than cores on the underlying machine, dynamic round-robin scheduling enables several tasks to 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 can still be applied to streaming task graphs if we assume that the latencies of producer-consumer task communications can be hidden by pipelining with multi-buffering from one scheduling round to the next, and that on-chip network links are not oversubscribed[1].

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 arbitrary core power models, which might comprise static and dynamic power consumption, and can be modeled analytically or based on measurements on real hardware. Beyond analytic power models like f^α, other models (e.g. derived from measurements) are also possible by replacing the corresponding terms in ILP formulations and heuristics.

A task instance produces a packet of output data from packets of input data. Its granularity depends both on the data packet size and on the complexity of the computation performed on it. In practice, we assume task instances (i.e., per packet) to take a time in the milliseconds range. This is coarse enough that the per-task user-level scheduling overhead, buffer management and frequency switching overhead can be considered reasonably low compared to task execution time. We allow for *moldable* (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 do not schedule tasks in a malleable way, i.e. a task cannot increase or decrease its number of allocated cores while running. Such task collections can still be scheduled by considering them as moldable tasks. We make no assumptions about the parallel efficiency functions of moldable tasks; these are parameters of our model. Moldable, partly moldable and sequential tasks might be mixed.

[1] Scheduling tasks under bandwidth constraints and optimizing latency is a matter of future work

We are interested in statically (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 given a throughput constraint and overall energy 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*. With this respect, we make the following contributions:

— 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 still considered NP-hard optimization problem feasible even for medium problem sizes. We argue that the resulting schedules are efficient and also 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 present *crown scheduling*, a novel method for energy-efficiency scheduling under the crown structure constraint.

— 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 give implementations of crown scheduling as Integer Linear Programming (ILP) models and evaluate them for a generic manycore architecture, showing that it can be solved to optimality (with regards to the crown structure) for small and most medium sized streaming task collections within a few seconds.

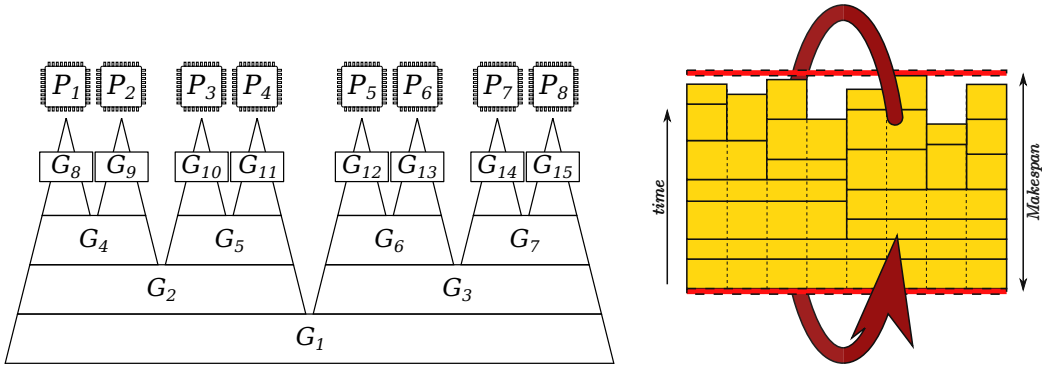
— We present efficient heuristics for crown allocation, mapping and discrete frequency scaling phases for large streaming task collections. We give a time complexity analysis for each of them and we show that the quality of their solution competes with optimal crown schedules for a much lower optimization time.

The remainder of this paper is organized as follows: Section 2 introduces the general concepts and central notation. Section 3 considers the separate optimization of crown resource allocation/mapping and subsequent frequency/voltage scaling, while Section 4 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 5 addresses dynamic schedule rescaling. Section 6 presents experimental results. Section 7 discusses related work, and Section 8 concludes the paper and proposes some future work.

2. CROWN SCHEDULING

A crown[2] is a recursive b-ary decomposition of the set of cores into subgroups down to singletons. These subgroups are to become the only possible mapping target for tasks. The base of a crown represents the full core set to run very parallel tasks. This inter-dependency decreases with the parallel degree of tasks and the increasing height in the crown. The crown structure aims to guarantee that cores belonging to a higher region of the crown (running less parallel tasks) can perform their work independently of cores belonging to another branch.

[2]The name is inspired by the shape of a crown, whose triangular spikes symbolize a decreasing degree of connectivity, see Fig. 1(a).



(a) The 15 processor groups in a binary crown over 8 processors, and their corresponding processors above.

(b) An (unscaled) crown schedule running in Round-Robin fashion.

Fig. 1. A group hierarchy and a possible corresponding unscaled crown schedule for 8 processors.

We consider a generic multicore architecture with p identical cores (processors). For simplicity of presentation, we assume in the following that $p = 2^{L\,(\text{see } 3)}$ is a power of $2\,^{(\text{see } 4)}$. All p cores can be individually and dynamically set to run at one frequency level from a finite set F.

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* G_1 equals P; it has the two child subgroups $G_2 = \{P_1, ..., P_{p/2}\}$ and $G_3 = \{P_{p/2+1}, ..., P_p\}$, four grandchildren groups $G_4 = \{P_1, ..., P_{p/4}\}$ to $G_7 = \{P_{3p/4+1}, ..., P_p\}$ and so on over all $L + 1$ tree levels, up to the *leaf groups* $G_p = \{P_1\}$,..., $G_{2p-1} = \{P_p\}$. 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 C_m denote the set of all groups that contain processor P_m. For instance, $C_1 = \{G_{2^z} : z = 0, \ldots, L\}$. Let $p_i = |G_i|$ denote the number of processors in processor group G_i. Where it is clear from the context, we also write i for G_i for brevity.

We consider a set $T = \{t_1, ..., t_n\}$ of n moldable, partly moldable 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 *moldable tasks*, W_j is ∞, i.e., unbounded; for *partly moldable 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[5]. 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 for crown allocation that w_j be a power

[3]We can define $p = b^L$ with $b > 1$ and $b \in \mathbb{N}$ for generality. However, our method yields more optimization opportunities as b decreases, for the same complexity. Therefore in this paper, we are only interested in the case where $b = 2$.

[4]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 processor groups .

[5]In principle, partial moldability 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, although this assumption is not strictly necessary for the techniques of this article.

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 $\{G_1, ..., G_{2p-1}\}$ of a size matching w_j. For each processor group G_i, $1 \leq i \leq 2p-1$, let $T^{(i)}$ denote the set of tasks $t_j \in T$ that are mapped to G_i. For each processor P_m, $1 \leq m \leq p$, let T_m denote the set of tasks $t_j \in T$ mapped to P_m, i.e., to any $G_i \in C_m$. Conversely, let R_j denote the group to which task t_j is mapped. *Crown scaling* attributes a running frequency level to each task. If task j is given frequency level $f_j \in F$, and mapped to group i, then all processors G_i must run at frequency f_j from when task j begins until task j finishes.

We define the execution time for tasks, groups and processors as functions of allocation, mapping and frequency settings given to all tasks. The execution time of a task j running on w processors at frequency f is defined as follows:

$$tt_j(w, f) = \frac{\tau_j}{f \cdot e_j(w) \cdot w} \tag{1}$$

Where clear from the context, we write tt_j for the execution time of task j with given width w_j and frequency f_j. We conveniently assume here that the running time of task j is proportional to the frequency inverse $1/f$, but the model allows extensions to sample the tasks' speedup function of frequency. Similarly, we consider that tasks use all w cores through all their entire execution time[6]. The static runtime of a processor group i, denoted gt_i, is the sum of execution times of all tasks mapped to this group. If they all run at frequency f, then we write $gt_i(f)$ as:

$$gt_i(f) = \sum_{j \in T^{(i)}} tt_j(w_j, f) \tag{2}$$

We write for convenience $gt_i = \sum_{j \in T^{(i)}} tt_j(w_j, f_j)$ for the execution time of group i, with all tasks j running at the frequency f_j they have been assigned. A processor m runs as long as the sum of all execution time of all groups it belongs to, assuming frequency f:

$$pt_m(f) = \sum_{i \in C_m} gt_i(f) \tag{3}$$

When clear from the context, we also write $pt_m = \sum_{i \in C_m} gt_i$ for the execution time of processor m, where all tasks j that it executes run at the frequency f_j that they are assigned. When we need to compare task j, group i or processor m by unscaled execution time, i.e. with no bias due to any task frequency settings, for instance to compute task's running frequency, we use the constant frequency $1Hz$ with $tt_j(w_j, 1)$ (Eq. 1), $gt_i(1)$ (Eq. 2) and $pt_m(1)$ (Eq. 3), respectively.

For each processor P_m, we order the tasks in T_m in non-increasing order of width, e.g., by concatenating the elements of all $T^{(i)} \in C_m$ in increasing order of group index i. The relative order of the tasks within each processor group must be kept the same across all its processors, e.g., in increasing task index order. We call such a p-tuple of processor schedules a *crown schedule*.

A *crown scheduler* is a user-level round-robin scheduler running on each processor P_m, $1 \leq m \leq p$, that uses a crown schedule for the (instances of) tasks in T_m to determine their order of execution. A crown scheduler works in *rounds*. Each round starts with a *conceptual* global barrier synchronization across all p processors[7]. Thereafter

[6]If not, then task j's efficiency function should not be modeled after its work τ_j, the number of processors w and a parallelization overhead, but instead derive the parallel time function tt_j of each task j from analysis or measurements.

[7]There is no need for an explicit barrier to separate rounds. For the prediction of makespan, a conceptual barrier can be assumed: As the cores involved in a parallel task (e.g., the first full-width parallel task

the tasks are executed round-robin as specified in the chosen crown schedule of the tasks in T_m. 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 involves (almost) all tasks in T as they all have data to process. See Figure 1(b) for an illustration.

The processing of tasks in decreasing order of width is required to make sure that all processors participating in the same moldable task (instance) start and stop approximately at the same time, allowing for efficient interprocessor communications and avoiding idle times, except at the end of a round. Between tasks (even of different widths) within a round, scheduled idle time ("external fragmentation") cannot occur because the processors of the subgroup assigned to the subsequent task automatically start quasi-simultaneously after finishing the previous task where they also participated; a moldable task is either data-ready or not, which means it is skipped either by all or none of the processors in the assigned processor group.

Crown scheduling computes crown allocation, mapping and frequency scaling for streaming tasks and minimizes energy consumption under a makespan constraint. We fix a target throughput for an application as the inverse of a time budget M to execute a round. Crown scheduling allows any analytic energy consumption model. Let $PC(f)$ be an arbitrary dynamic power function for the crown scheduler. For a concrete machine, it is sufficient to use results from power measurements of the target machine at the available frequencies [Cichowski et al. 2012]. In our experiments, we set $PC(f) = f^\alpha$ with $\alpha = 3$. We consider the energy spent at idle time as negligible, and also ignore the time and energy required to switch frequencies, as they are small against the time of a round. Equation 4 shows our energy consumption model.

$$energy = \sum_{j=1}^{n} tt_j \cdot w_j \cdot PC(f_j) \tag{4}$$

3. 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 is be discussed in Section 4. An overview of these problems is given in Figure 2.

We can reduce the overall energy usage by scheduling tasks to run at a frequency as low and uniform as possible through a pipeline stage [Li 2008]. Also, early phases of the phase-separated approach are unaware of their influence on the efficiency of subsequent phases. However the more idle time processors have between the target makespan and the execution of their last task, the lower frequency can run their task and the lower energy a pipeline stage consumes. Therefore, the allocation and mapping phases aim at optimizing for unscaled execution time and load-balancing. The frequency scaling phase scales frequencies as low as possible under the makespan constraint.

In this section, we call *crown-optimal* an optimal solution of the allocation, mapping or frequency scaling phases in a separate manner. Because they are solved separately, they do not participate in an optimal solution to the general scheduling problem studied in this article.

---

running on the root group at the beginning of a round) will task-internally communicate and synchronize with each other, the resulting waiting times that would occur if cores start the task at different times can conceptually be moved to the end of the preceding round, thus having the same effect as a zero-latency barrier separating the rounds.

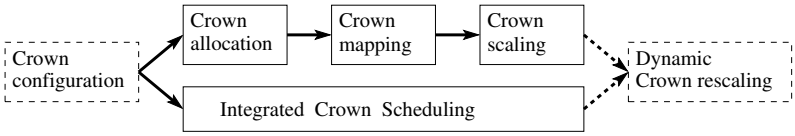


Fig. 2. Overview of phase-separated vs. integrated crown scheduling. The methods shown by solid boxes are detailed in this paper.

3.1. Crown-optimal Crown Resource Allocation

Let Q_z, where $z = 0, ..., \log_2 p$, be the set of indexes of all the tasks which are assigned width $w_j = 2^z$. Then task t_j ($j \in Q_z$) has unscaled runtime $\hat{\tau}_j = tt_j(w_j, 1)$ assuming that its processors run at fixed frequency 1. If we assume as a simplification that all tasks with the same width 2^z can be distributed over the $p/2^z$ processor groups in a balanced fashion, the runtime of these tasks on p processors is

$$R_z = \sum_{j \in Q_z} \frac{\hat{\tau}_j}{p/2^z} \tag{5}$$

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 $v_{j,z}$ with $v_{j,z} = 1$ iff task j gets width $w_j = 2^z$. Then Eq. 5 transforms to

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

and we need the constraint

$$\forall j : \sum_{z} v_{j,z} = 1 \ .$$

It is simple to find an allocation that minimizes the total runtime, under the assumption that the tasks can be distributed optimally, i.e., the total runtime is given by Eq. 5. Equation 5 is minimized iff all $\hat{\tau}_j v_{j,z}/(p/2^z)$ are minimized. Therefore, we compute for each task j the speedup $g_j(q) = e_j(2^q) \cdot 2^q$ for each $q \in [0..L]$ and select one q that maximizes g_j, from all $L = \log p$ possible allocations. Since we have n tasks, this takes at most $O(n \cdot \log p)$ time steps in total. Because all terms in the sum of Eq. 5 are independent and because we minimize each, the computed allocation is optimal. Because it computes an allocation such that each task runs individually for the shortest possible time, we refer to this algorithm as *fast allocation*.

3.2. Crown-optimal Task Mapping

For an optimal crown mapping given an allocation, we treat all widths separately, i.e., we solve $\log_2 p$ smaller optimization problems. For the tasks 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 processor 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 $maxload$ (note that the target function here is simple) under some constraints. To express the constraints, we use binary variables

$y_{j,i}$ where j runs (in consecutive numbering) over all tasks with width 2^z and i runs over all considered $p/2^z$ processor groups $G_{p/w_j}, \ldots, G_{2p/w_j-1}$. Hence, $y_{j,i} = 1$ iff task j is mapped to processor group i. The constraints are as follows. Task j is mapped to exactly one processor group among those that have the right width w_j:

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

We define T_i as the load mapped to processor i, that is the sum of the load of each group i is part of

$$\forall i \in 1..p : T_i = \sum_{q \in C_i} \sum_{j=1}^{n} \hat{\tau}_j \cdot y_{j,q}$$

And $maxload$ as the maximal T_i, i.e., the makespan:

$$maxload \geq \max_{i \in 1..p}(T_i)$$

3.3. Heuristic Task Mapping with Load Balancing

We describe the *Longest Task, Lowest Group* (LTLG) mapping heuristic for the load balancing problem modeled in Sec. 2. We consider the *height* of a processor group i as shown in Eq. 6 and illustrated in Fig. 3(a). The height of a group i is the maximum running time $\max_{m \in G_i} pt_m(f)$ of all its processors running at frequency f. We call for convenience $height_i$ the height of group i where all tasks j run at frequency f_j (Eq. 7). The LTLG heuristic discards all tasks of group 1 from the height of any group because group 1 cannot participate in any load imbalance ($height_{i=1} = 0$). As no frequency scaling is performed yet at the mapping stage (Fig. 2), LTLG computes the height of a group i for a constant frequency c ($height_i(c)$). Note that this is not a limitation of our LTLG heuristic as scaled tasks would only result in different a running time for each task to map.

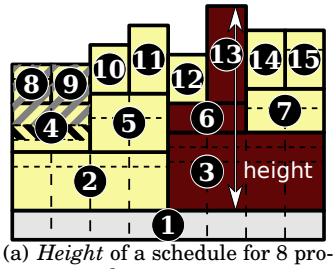
$$height_{i \in [1,2p-1]}(f) = gt_i(f) + \sum_{i'=2}^{\lfloor \log_2 i \rfloor} gt_{\lfloor \frac{i}{2^{i'}} \rfloor}(f) + \max(height_{2i}(f), height_{2i+1}(f)) \quad (6)$$

$$height_{i \in [1,2p-1]} = gt_i + \sum_{i'=2}^{\lfloor \log_2 i \rfloor} gt_{\lfloor \frac{i}{2^{i'}} \rfloor} + \max(height_{2i}, height_{2i+1}) \quad (7)$$

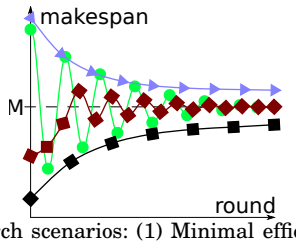
LTLG is based on three principles to produce better load-balanced mappings:

(1) The assignment of each task j to the group i of least $height_i$ among groups of size w_j limits load imbalance while building a schedule.
(2) The insertion of tasks of highest parallel running time first (with w_j processors) lowers the risk of creating load imbalance when adding the last tasks to a schedule.
(3) If tasks have the same parallel running time, inserting tasks of highest width first keeps lowest width tasks available to later fill in the schedule's holes.

The algorithm maintains $\log_2 p$ priority queues, one per group size. The priority of groups is defined by their least height, or their group number if both groups have the same current height (Eq. 8). Since tasks of size $w_j = p$ do not create load imbalance and as the root group is the only group of size p, we don't use a priority queue for it. The algorithm first sorts the tasks using Eq. 9 to compare them, with both their parallel running time and their width. Considering tasks in this order, we assign task j to the

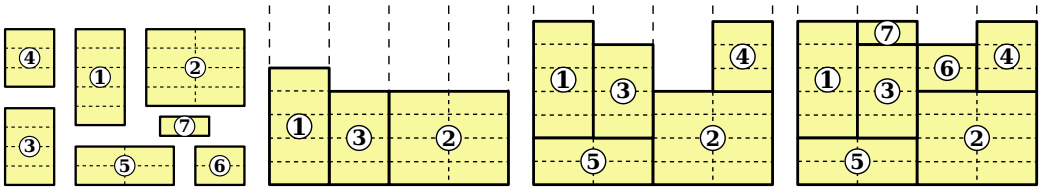


(a) *Height* of a schedule for 8 processors and processor groups 3, 6 and 13 and area (grey hash) to be rescaled if the back hashed task is delayed or skipped.



(b) Four binary search scenarios: (1) Minimal efficiency increase does not prevent valid schedules (bottom, plain black squares), (2) search for highest valid value from the beginning (mid-dark green dots), (3) minimal efficiency increase prevents valid schedules then search for higher valid value (dark red diamonds) and (4) Binary search fails to find any allocation (top, light blue triangles).

Fig. 3. Illustration of the *Height* of a schedule for eight processors and 4 possible scenarios of schedules' makespan evolution during binary search.

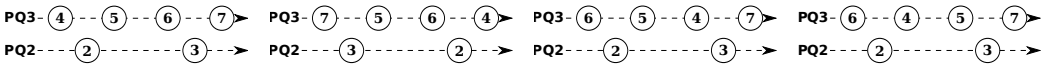


(a) Task set sorted and numbered by parallel running time and width as of principles 2 and 3.

(b) Map tasks 1, 2 and 3 to groups 4, 5 and 3 and start at t = 0 as no processor has any job scheduled yet.

(c) Map task 4 to group 7 (P_4) and task 5 to group 3 (P_1 and P_2), under tasks of lower parallel degree.

(d) Map tasks 6 and 7 to the sequential groups run by the least loaded processors.



(e) Initial state of priority queues. The left-most group in a level is the most prioritary.

(f) After inserting task 3. Groups 4 and 2 must be last of their queues as their height is the highest.

(g) Possible priority queues after inserting task 5.

(h) Possibe final priority queues. Groups 4, 5, 7, 2 and 3 could be ordered differently.

Fig. 4. Running example of 7 tasks mapped with LTLG yielding a makespan of 7. Tasks are sorted and numbered after their parallel running time and degree as described by principles 2 and 3. Tasks 1 to 7 with their current allocation run for 5, 4, 4, 3, 2, 2 and 1 time units. The reader may build schedules without principles 2 and 3 (or using their complement instead) to experience their contribution.

highest priority group i of size w_j as given by the priority queue for size w_j. Then we recompute the runtime gt_i of group i and its height, as well as for all its children groups recursively. If the new height of group i is the maximum height among all children of group $i' = \lfloor i/2 \rfloor$, then the height of i' is also updated, and this is repeated for i' until the root group (excluded) is reached.

At each insertion of task j to a group i, we recompute the group's runtime gt_i. This takes constant time as we can add τ_j to an existing gt_i value, initialized to 0. Then, we need to update the height of all $O(p)$ descendants groups of i as well as all $O(\log p)$ ancestor groups of i. Each group update requires a constant time update of the group starting time for descendant groups and of the maximum child height for ancestor groups. Finally, for each height recomputed, the group list for the corresponding level must be sorted to reflect the new height. This can take $O(\log p)$ time by removing and inserting again the group in the list. Each task insertion takes therefore $O(p \log p +$

$$\phi(i_1, i_2) = \begin{cases} height_{i_1} > height_{i_2} & \text{if } height_{i_1} \neq height_{i_2} \\ i_1 > i_2 & \text{otherwise.} \end{cases} \quad (8)$$

$$\phi(j_1, j_2) = \begin{cases} tt_{j_1}(w_{j_1}, 1) > tt_{j_2}(w_{j_2}, 1) & \text{if } tt_{j_1}(w_{j_1}, 1) \neq tt_{j_2}(w_{j_2}, 1) \\ w_{j_1} > w_{j_2} & \text{if } tt_{j_1}(w_{j_1}, 1) = tt_{j_2}(w_{j_2}, 1) \text{ and } w_{j_1} \neq w_{j_2} \\ j_1 > j_2 & \text{otherwise.} \end{cases} \quad (9)$$

$\log^2 p)$ time steps. As we insert n tasks, the time complexity of the LTLG heuristic is $O(n(p \log p + \log^2 p)) = O(np \log p)$.

3.4. Crown-optimal Voltage/Frequency Scaling of Schedules

We assume that when running all cores at maximum speed $F_{\max}$ all the time, the resulting makespan is $\hat{M} \leq M$. 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 schedule".

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 $F = \{F_1, ..., F_s\}$ of s discrete (voltage and) frequency levels, such that it still meets the throughput constraint. We require

$$\forall m = 1, \ldots, p : pt_m \ \leq \ M$$

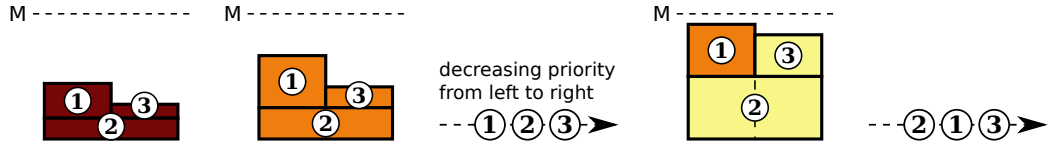
and the overall energy usage as defined by Eq. 4 is minimized, subject to the additional constraint that all processors of a group must use the same frequency level for a task.

3.5. Height Heuristic for Voltage/Frequency Scaling

The *Height* frequency scaling heuristic reduces individual tasks' frequencies so that the final schedule satisfies the throughput constraint M and tries to minimize energy consumption. In general, energy consumption is lower when all tasks run with the same frequency compared to running them at different frequencies for performing the same work in the same time interval [Li 2008], therefore we decrease moderately the frequency of all tasks together instead of decreasing aggressively the frequency for some of the tasks while greatly increasing the frequency of other tasks in order to satisfy the makespan constraint M. Our heuristic is an implementation of the Best-Fit heuristic for the bin packing problem, where each group is a bin and each bin's free space is the difference between the group's height and the target makespan.

The algorithm begins by assigning all tasks the maximum frequency $F_{\max}$ and computing the initial height $height_i(F_{\max})$ of all groups i. All tasks j are queued after their decreasing running time tt_j at frequency f_j (initially $\forall j : f_j = F_{max}$). We call $f_{j,cur}$ any frequency assigned to task j at any point in the algorithm. We call F_{next} the frequency variable that iterates over the set of frequencies F sorted decreasingly. For all tasks j in the task priority queue, if $f_{next} < f_{j,cur}$ and the time penalty resulting in running task j at frequency $f_{next} < f_{j,cur}$ is lower than, or equals the gap between the height $height_{i_j}$ of the group i_j (to which task j is mapped) and the target makespan M (that is, if $tt_j(w_j, f_{next}) - tt_j(w_j, f_{j,cur}) \leq M - height_{i_j}$), then we can assign task j the frequency f_{next}, update the height of group i_j as described in Sec. 3.3, and insert the newly scaled task into a new task priority queue for the next frequency iteration.

The initial frequency assignment to tasks takes $O(n)$ steps, the computation of the initial height for each group takes $O(n)$ for each $O(p)$ groups, hence $O(pn)$. Tasks and frequencies are sorted decreasingly in $O(n \log n)$ and $O(|F| \log |F|)$ time steps, respec-



(a) Initial schedule: all tasks run at $f = \max(\{1, 2, 3\}) = 3$.

(b) Scale tasks 1, 2 and 3 at $f_{next} = 2$.

(c) Task priority queue before states of Fig 5(a) and after Fig. 5(b).

(d) Only tasks 2 and 3 can scale further at $f_{next} = 1$.

(e) Final task priority queue .

Fig. 5. Running example of 3 tasks scaled with the Height heuristic for 2 processors of 3 frequency levels.

tively. In each nested loops iterations, taking the longest running time from the sorted task queue takes $O(1)$ time steps. Checking if running task j at frequency f_{next} doesn't make the schedule to cross the target makespan M takes $O(1)$ time steps if the height of group i_j is already computed. Updating the height of group i_j takes $O(p)$ time steps and adding the task in the new priority queue takes $O(\log n)$ time. Since we have $|F|$ frequency levels and n tasks, we run $n \cdot |F|$ iterations. Therefore the Height heuristic runs in $O(n + pn + n \log n + |F| \log |F| + pn \cdot |F| \log n) = O(|F| \cdot pn \log n)$ time steps.

3.6. Binary Search Allocation Heuristic

The methods described above for the allocation, mapping and frequency scaling phases rely on the mapping phase capability to compute task distribution close to optimal and produce a schedule whose runtime is the average load that each core receives (Eq. 5). However, we target the minimization of energy consumption under makespan constraint instead of runtime. As parallelization usually induces additional processing cost due to a task j's non-perfect efficiency function $e_j(q) \leq 1$, a too high parallelization consumes more energy. This suggests to give tasks as little parallelism as possible [Belkhale and Banerjee 1990].

We set a minimal acceptable efficiency value $e_{\min}$ for allocations of processors to all tasks, in order to limit parallelization and reduce its cost. We run the *fast allocation*, with the additional constraint that no task j should be allocated a number of cores that yields $e_j < e_{\min}$. The constraint is implemented as a $O(1)$ condition to discard any allotment q for a task j if $e_j(q) < e_{\min}$ in the search for q that minimizes $tt_j(q, 1)$. Then, we use the fast heuristics described in Sections 3.3 and 3.5 to check if the limited parallelization still allows our heuristics to compute a valid schedule. If so, we increase the minimal efficiency threshold by Δ and compute a new schedule. Otherwise we decrease the minimal efficiency by Δ. Finally, we update Δ to its half. We implement a binary search on $e_{\min}$ to maximize the minimal efficiency in the search space $(0, 1)$, with an initial threshold 0.5 and $\Delta = 0.25$ (Fig. 3(b)). The binary search stops when the minimal efficiency is higher than any possible value for e_{j,w_j} ($W_j > 1$) for all tasks j (Fig. 3(b) case 1) or when Δ becomes lower than the minimal efficiency difference between two allocations, for any task.

The binary search sequentializes tasks as far as task mapping and frequency scaling allow the production of valid schedules. However, more sequential tasks take longer time to execute. This can result in the frequency scaling phase to increase the task's frequency, which can replace the parallelization overhead by a much higher frequency cost, due to the fast increase of $PC(f) = f^{\alpha}$. During the binary search, we keep track of the energy consumption of every schedule generated and we keep the best. The best initial solution is computed with fast allocation (Sec. 3.1) with no efficiency constraint ($e_{min} = 0$). Before starting our binary search, we also try an allocation with a perfect efficiency constraint $e_{min} = 1$, that is only possible with sequential or perfectly scalable

tasks. If such an allotment results in a schedule running all tasks at the lowest possible frequency, then we select this solution and skip the search loop. Otherwise, we compare it to the initial best solution and begin the binary search iterations.

Figure 3(b) illustrates four possible search scenarios. In scenario 1, a progressive increment makes the final schedule's makespan to approach M, but it never reaches or crosses it. All allocations are valid but the best one can be any of them (bottom curve, plain black squares). In scenario 2, the initial $e_{\min}$ doesn't allow subsequent heuristics to compute a valid schedule. $e_{\min}$ is drastically reduced by initial Δ, and the resulting schedule's makespan is much lower than the target. The binary search method continues to halve Δ and increasing and decreasing $e_{\min}$ by Δ until Δ is too small and the allocation that yields the best schedule is kept (mid-dark green rounds). In scenario 3, the binary search increments $e_{\min}$ so the final schedule approaches M as in scenario 1 but as $e_{\min}$ increases, the heuristics begins to fail at computing a valid schedule. Binary search behaves then as in scenario 2, until Δ is too small (dark red diamonds). Finally in scenario 4, the binary search may always decrease $e_{\min}$ but still fails to produce any allocation that the heuristics could use to compute any valid schedule (top light blue triangles). In this case, we use the initial best solution computed with the fast allocation strategy (Sec. 3.1) and rely on LTLG and Height heuristics to compute a valid schedule, if possible.

The binary search algorithm explores the search space $(0, 1)$ for Δ. Each round runs the LTLG ($O(np \log p)$) and Height ($O(|F| \cdot pn \log n)$) heuristics and run therefore in $O(np(\log p + |F| \cdot \log n))$ time steps. We define the stop condition as Δ being smaller than the efficiency gap between two allocations for all tasks, which we calculate before beginning the binary search. The minimal allocation gap takes time $O(n \log p)$ to compute (see Sec.3.1 on allocation), where allocations that exhibit too low efficiency are pruned during search. If g is the minimal efficiency gap between two possible allocations for any task (a large value of g means that all tasks scale badly), then $\gamma = \lceil \log_2(\Delta/g) \rceil$ is the number of rounds in our binary search. We also compute twice a fast allocation (time $O(n \log p)$) as an initial solution and attempt with a perfect efficiency constraint. The overall complexity of our binary search is therefore $O(n \log p + 2(n \log p + np \log p + |F| \cdot pn \log n) + \lceil \log_2(\Delta/g) \rceil \cdot np(\log p + |F| \cdot \log n)) = O(\lceil \log_2(\Delta/g) \rceil \cdot np(\log p + |F| \cdot \log n))$.

3.7. Simulated Annealing

The binary search method described in Sec. 3.6 limits the allocation space exploration to solutions where all tasks share a common efficiency constraint. In particular, it does not permit any fine tuning through individual tasks' allotment adjustment. We use the fast LTLG mapping and height frequency scaling heuristics within a simulated-annealing meta-heuristic that optimizes the core allocation for each individual task. Our simulated annealing thus optimizes allocation for a task collection.

We define the *neighborhood* of an allocation as other allocations within a fixed *distance*. The distance between two allocations is a real in the interval $[0, 1]$, that defines a proportional amount of the n tasks, whose number of cores allotted differs between both allocations. The distance also defines the proportional variation of number of cores, as a power of 2, that run the same task in both solutions. Searching in the neighborhood of an allocation within a distance 0, is searching in the set of all allocations whose exactly one task is allocated the next higher or lower power-of-2 number of cores, compared to the allocation of origin. Searching in the neighborhood of an allocation within a distance 1 is searching in all possible allocations. A search within the neighborhood of an allocation always excludes the allocation of origin.

We begin with a schedule computed with binary search (Sec. 3.6). The LTLG mapping and height frequency scaling heuristics compute a schedule for this allocation

and we use the resulting energy consumption (Eq. 4, $PC(f) = f^{\alpha}, \alpha = 3$) as feedback to the simulated annealing allocation phase. In our experiment, we search a new solution within the neighborhood of the current allocation at a maximal distance 0.5 at each simulated annealing iteration. We allow at most 2 iterations before we decrease the simulated annealing temperature (starting at 8) by multiplying it by our cooling factor 0.6. For each temperature level, we accept at most 2 improving solutions. The simulated annealing stops when the simulated temperature reaches 0.9, or whenever it finds a correct solution where all tasks are sequential and run at the minimal frequency.

We have $\lceil \log(8/0.9)/\log(1/0.6) \rceil = 5$ iterations until the final temperature is reached. We allow at most 2 transformations per iteration, resulting in at most $10 = O(1)$ runs of the task mapping and frequency scaling phases. Because we use binary search as an initial solution, the overall time complexity is $O(np(\log p + |F| \cdot \log n + \lceil \log_2(\Delta/g) \rceil) + 10(pn(\log n + \log p)) = O(np(\log p + |F| \cdot \log n + \lceil \log_2(\Delta/g) \rceil))$.

4. INTEGRATED ENERGY-EFFICIENT CROWN SCHEDULING

Separate crown allocation, crown mapping and a-posteriori scaling of the resulting crown schedule may lead to suboptimal results compared to co-optimizing resource allocation, mapping and discrete scaling from the beginning. See Fig. 6 for an example.

The integration of crown allocation, mapping and scaling is modeled as follows: We construct an integer 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 group i and should run at frequency level F_k. We require that each task be allocated and mapped to exactly one group and frequency level:

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

and forbid the mapping of a task with width limit W_j to an oversized group:

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

Another constraint asserts that no processor is overloaded

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

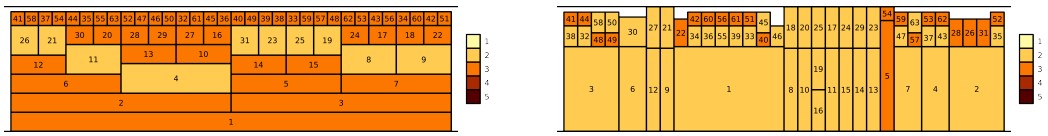
Then we minimize the target energy function (as derived from Eq. 4)

$$\sum_{i=1}^{2p-1} \sum_{k=1}^{s} PC(F_k) \cdot \sum_{j=1}^{n} x_{i,k,j} \cdot \frac{\tau_j}{e_j(p_i)}$$

Note that the coefficients τ_j, p_i and $e_j(p_i)$ are constants in the linear program. Note also that the time and energy penalty for frequency switching is not modeled. 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.

5. DYNAMIC CROWN RESCALING

The above methods for crown resource allocation, mapping and scaling optimize for the steady state of the pipeline and in particular for the ("worst") case that all tasks are data ready and execute in each round. In practice, however, there might be cases



(a) Optimal phase-separated crown schedule (b) Optimal integrated crown schedule

Fig. 6. Phase-separated and integrated solutions; the lighter the tasks, the lower their running frequency. Optimal allocation used in the phase-separated approach restricts the quality of the final solution. Frequencies are represented by colors and darkness: dark red is high frequency (5) and lighter yellow represents low frequency (1).

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, or where tasks are delayed due to cache misses, branch mispredictions or unexpected I/O latency.

If the skipped or delayed task j was mapped to a group (even a singleton group) G_i, for instance the black hashed task in Fig. 3(a), we call β the additional execution time of group i due to variable delays or tasks skipped. If task j is skipped, then the additional execution time is the negative opposite of tt_j ($\beta = -tt_j < 0$).

If $\beta < 0$ we can save more energy or if $\beta > 0$, the schedule may not meet the target makespan M. We can rescale "down-crown" all $n' \leq n$ remaining tasks of the processors in G_i that follow j in the crown schedule (gray hashed area in Fig. 3(a)). Note that such rescaling must be consistent across all processors sharing a moldable task, and hence the concurrent search for candidates to rescale must be deterministic. The crown structure can be leveraged for this purpose. Note that if a task of the root group is skipped or delayed, then the entire schedule needs to be rescaled.

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. Let R_i be the set of tasks in group i already executed ($j \notin R_i$). We can apply the Height heuristic (Sec. 3.5) with the subgroup G_i running tasks in R_i as the root group and target the makespan

$$M' = M - tt_j - \beta - \sum_{x=1}^{\lfloor \log_2 i \rfloor} gt_{\lfloor i/2^x \rfloor} - \sum_{j' \in R_i} tt_{j'}$$

for $p' = |G_i|$ processors ($p' \leq p$) and $n' < n$ tasks, taking $O(|F|p'n' \log n')$ time steps.

6. EXPERIMENTAL EVALUATION

We implemented the ILP models in AMPL [**?**], the LTLG mapping, the height frequency scaling, binary search and simulated annealing allocation heuristics in C++. We substitute the ILP-based mapping and frequency scaling phases of the phase-separated variant [Kessler et al. 2013b] with an optimal load-balanced ILP-formulated mapping phase, the LTLG mapping, and with the Height mapping heuristics, respectively. We obtain and compare 7 different crown schedulers:

(1) ILP allocation, ILP load-balanced mapping, ILP frequency scaling (Fast,Bal.ILP,ILP)
(2) ILP allocation, LTLG mapping heuristic, ILP frequency scaling (Fast,LTLG,ILP)
(3) ILP allocation, ILP load-balanced mapping, Height heuristic frequency scaling (Fast,Bal.ILP,Height)
(4) ILP allocation, LTLG mapping heuristic, Height frequency scaling heuristic (Fast,LTLG,Height)

(5) Binary search with LTLG mapping and Height frequency scaling heuristics (Bin,LTLG,Height)
(6) Simulated annealing allocation with initial binary search, LTLG mapping and Height frequency scaling heuristics (Bin,LTLG,Height Ann.)
(7) ILP integrated (Integ.)

We also implemented an ILP formulation from Xu et al. [2012], whose scheduler arranges tasks in levels of strips spanning across cores. Each level's height is defined by a *key* tasks, that is the first task mapped to its level. The rectangle of cores unused by the key task of a level is 2D-packed using some of the tasks not scheduled yet. Frequency scaling is performed through the rescaling of entire levels, that is the rescaling of all tasks mapped to this level, as a bin packing problem. Finally, we adapted and implemented the NLP (Non-Linear Programming) formulation from Pruhs et al. [2008] in AMPL and their heuristic in C++. Their technique schedules sequential tasks only (and therefore it never parallelizes any task j regardless of its maximal parallel degree W_j) and scales tasks' frequency with a continuous and unbounded frequency domain. For the NLP implementation, we perform frequency scaling using our ILP-based method (Sec. 3.4). For the heuristic, we use best-fit to assign discrete and bounded frequency levels to tasks.

For small and medium problems and machine sizes, we generate synthetic task collections as follows: we group synthetic task collections in different categories defined by the number of cores ($p \in \{1, 2, 4, 8, 16, 32\}$), the number of tasks (10, 20, 40 and 80 tasks), and tasks' maximum widths W_t: sequential ($W_t = 1$ for all t), low ($1 \leq W_t \leq p/2$), average ($p/4 \leq W_t \leq 3p/4$), high ($p/2 \leq W_t \leq p$) and random ($1 \leq W_t \leq p$). Tasks' maximum width are distributed uniformly. The target makespan of each synthetic task collection is the mean value between the runtime of an ideally load balanced task collection running at lowest frequency and at highest frequency

$$M_{syn} = \sum_{j=1}^{n} \frac{3 \cdot tt_j(1,1)}{8 \min(W_j, p) F_{\min}} + \frac{tt_j(\min(W_j, p), 1)}{8 \cdot F_{\min}} + \frac{3 \cdot tt_j(1,1)}{8 \min(W_j, p) F_{\max}} + \frac{tt_j(\min(W_j, p), 1)}{8 \cdot F_{\max}} \tag{10}$$

We use the same scheme to generate large problems and machine sizes, ranging from 500 to 2000 tasks and from 256 to 1024 processors for tasks of a random maximum width.

We also provide task collections of classic streaming algorithms: parallel FFT, parallel-reduction and parallel mergesort. FFT is characterized by $2 \cdot p - 1$ parallel tasks of a balanced binary tree. In level $l \in [0; \log_2 p]$ of the tree, there are 2^l data-parallel tasks of width $p/2^l$ and work $p/2^l$ so all tasks could run in constant time. The mergesort task collection is similar to FFT, but all its tasks are sequential. Parallel reduction involves $\log_2 p + 1$ tasks of maximum width 2^l and work 2^l for $l \in [1; \log_2 p + 1]$; they can also run in constant time. Because these task collections yield perfectly balanced task work and maximum widths, they are easy to schedule with a loose target makespan as defined for the synthetic task collections. We use a more constraining target makespan for concrete task collections. The makespan for FFT is

$$M_{FFT} = \sum_{j=2}^{n} \frac{tt_j(1, F_{\min})}{2 \cdot p} + \sum_{j=2}^{n} \frac{tt_j(1, F_{\max})}{2 \cdot p} \tag{11}$$

that is, the makespan of a synthetic task collection, ignoring task 1. The makespan for parallel mergesort

$$M_{msort} = \frac{\max_{\forall j}(\tau_j)}{\max F} \tag{12}$$

takes the root merging task and divide its work by the maximal frequency available. The makespan for parallel reduction task collections

$$M_{\textit{preduce}} = \sum_{j=1}^{n} \frac{\tau_j}{2 \cdot \min(p, W_j) \cdot F_{\min}} + \sum_{j=1}^{n} \frac{\tau_j}{2 \cdot \min(p, W_j) \cdot F_{\max}} \tag{13}$$

is the same as for synthetic collections, but we assume perfect scalability. All our synthetic and classic task collections have a minimal efficiency gap of $g = 0.023438$, yielding a complexity factor of $\gamma = \lceil \log_2(0.5/0.023438) \rceil = 5$ for our binary search method.

Finally, we use the technique of Gordon et al. [2006] to extract tasks' workload and parallel efficiency from the Streamit benchmark suite. We schedule all variants of the applications *audiobeam*, *beamformer*, *channelvocoder*, *fir*, *nokia*, *vocoder*, *BubbleSort*, *filterbank*, *perftest* and *tconvolve* from the streamit compile source package[8]. We use their compiler to obtain for each task j the task's estimated workload τ_j, the task's parallel degree W_j (either 1 or $+\infty$) and the task's communication rate we call ψ_j. We use ψ_j to compute tasks j's parallel efficiency $e_j(q)$ (Eq. 14).

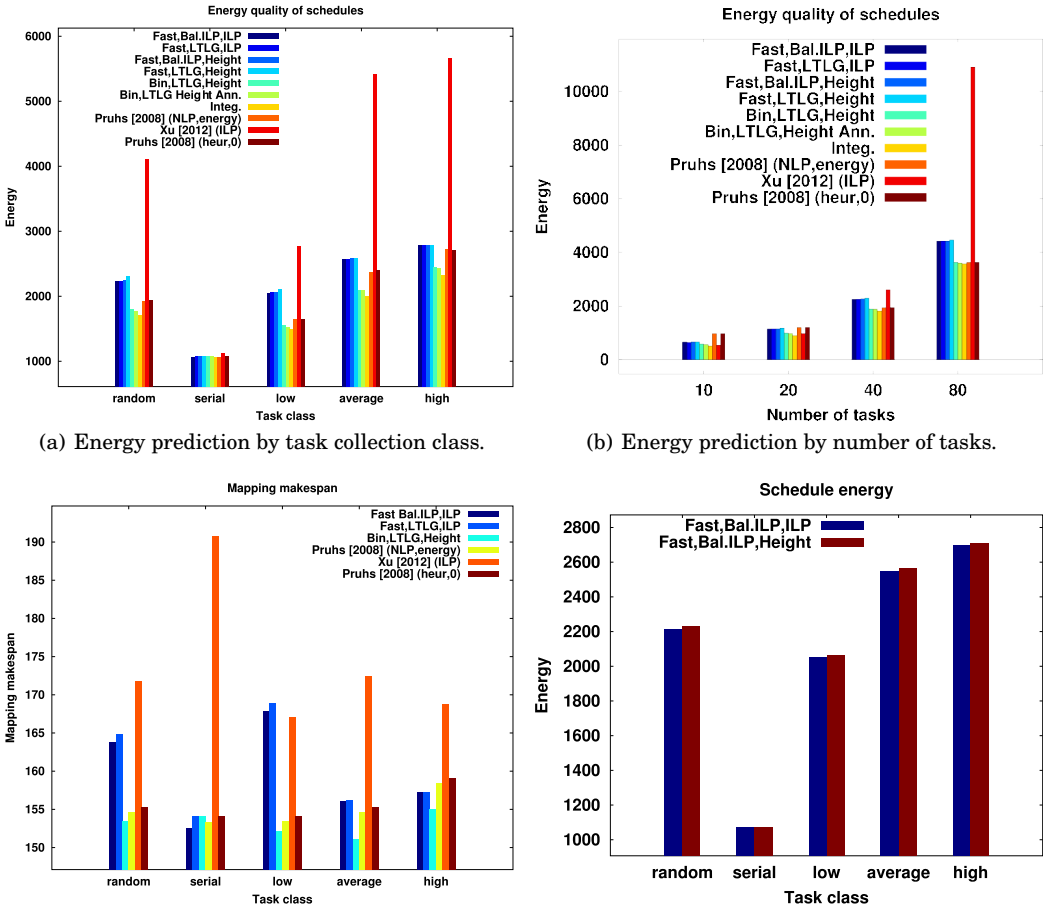
$$e_j(q) = \begin{cases} 1 & \text{if } q = 1 \\ \tau_j/(\tau_j + q \cdot \psi_j) & \text{if } q > 1 \text{ and } q \leq W_j \\ 10^{-6} & \text{otherwise} \end{cases} \tag{14}$$

and we use Eq. 10 to compute a target makespan.

We run both the ILP solver and heuristics on a quad-core i7 Sandy Bridge processor at 3GHz and 8GB of main memory. We use Gurobi 5.10 and ILOG AMPL 10.100 with 5 minutes timeout to solve the ILP models and we compile our C++ implementations with gcc 4.6.3 and -O3 optimization switch, on Ubuntu 12.04. Gurobi exploits 8 hardware threads and our heuristics are implemented as sequential programs.

We measure the overall scheduling quality (energy consumption) of all 7 crown schedulers as well as schedulers from Xu et al. [2012] and Pruhs et al. [2008]. Figures 7(a) and 7(b) show mean values for 10, 20, 40 and 80 tasks and across all width classes $\{Random, Serial, Low, Average, High\}$ for the synthetic task collection. Figures 8(b) and 9(b) show the average energy consumption by task collection classes for the classic streaming algorithms and the streamit benchmark, respectively. We can see that the integrated crown scheduler as well as scheduler heuristics based on binary search, LTLG, Height and simulated annealing combined produce consistently the best schedules. NLP and heuristics from Pruhs et al. [2008]' perform well on larger architectures and with sequential task sets (Figs.7(b) and 8(b)), but its solutions' quality drop when tasks' parallel degree increase (Figs. 7(a), and 8(b)). The level-packing solution from Xu et al. [2012] performs mostly poorly, except for small amount of tasks, or for sequential tasks. An exception is parallel reduction (Fig. 8(b)) for which it performs at least as good as our integrated, ILP-based crown scheduler. We scheduled a subset of our synthetic task collection set with random workload variations within 1% for each tasks and found no significant energy variations in schedules. Finally, we show for the classic streaming task collection and the streamit benchmark suite, the schedulers' ability to compute a valid schedule in Figs. 8(a) and 9(a). These figures show that Pruhs' techniques fail at scheduling parallel tasks under the makespan constraints we impose, while Xu's succeeds as much as our crown schedulers. For sequential task collections such as for mergesort, our crown scheduler has the same success rate as Pruhs', whereas Xu's approach fails more often.

[8]At the time of writing, the latest version was pushed to github on Aug 3, 2013. See groups.csail.mit.edu/cag/streamit/restricted/files.shtml and the source package for information about these applications.



(a) Energy prediction by task collection class.

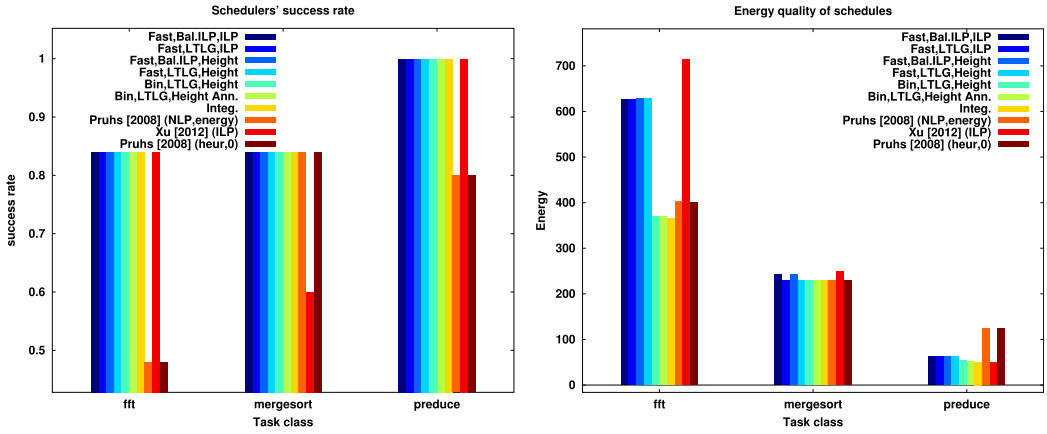
(b) Energy prediction by number of tasks.

(c) Mapping makespan by task collection class.

(d) Frequency scaling energy by task collection class.

Fig. 7. Projected energy per number of cores and tasks and efficiency of LTLG and Height heuristics for the synthetic task collection.

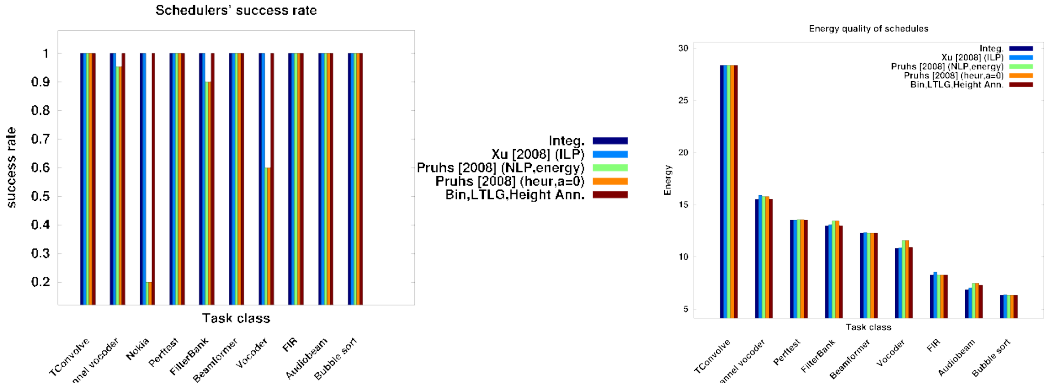
We also compare the makespan produced by our LTLG mapping heuristic to our crown-optimal load-balanced ILP mapping formulations (variant 1 above) as well as our binary search and simulated annealing heuristics and our implementations of Xu's and Pruhs's techniques (Fig 7(c)). Similarly, we compare in Fig 7(d) the efficiency of the frequency scaling ILP formulation and Height heuristic to optimize energy through frequency scaling. In order to ensure fairness, we use our crown-optimal ILP formulation to compute both mappings the frequency scaling implementations we compare need to scale. Figure 7(c) shows that our LTLG heuristic is very competitive with our crown-optimal ILP load-balancing formulation. The same figure suggests that the initial allocation influences greatly the mapping makespan. Our ILP and LTLG heuristic use the fast allocation (Sec. 3.1), that is allocating as many processors as possible to tasks. In contrast, Pruhs' NLP formulation and heuristic schedule sequential tasks. As a consequence, Pruhs' makespan are much lower for tasks that cannot run on many processors (but can run on a few) while the situation is more balanced with highly parallel tasks. The binary search always finds another allocation that allows lower makespan than either our crown heuristic with fast allocation or Pruhs' solutions.



(a) Scheduling success rate per task collection class.

(b) Energy prediction by task collection class.

Fig. 8. Scheduling success rate and projected energy for FFT, mergesort and parallel-reduction



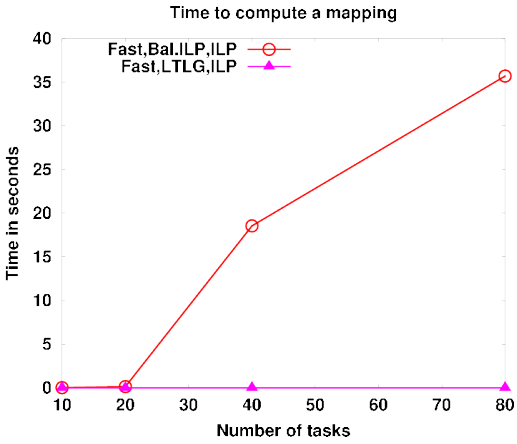
(a) Scheduling success rate per task collection class.

(b) Energy prediction by task collection class (logarithmic scale).

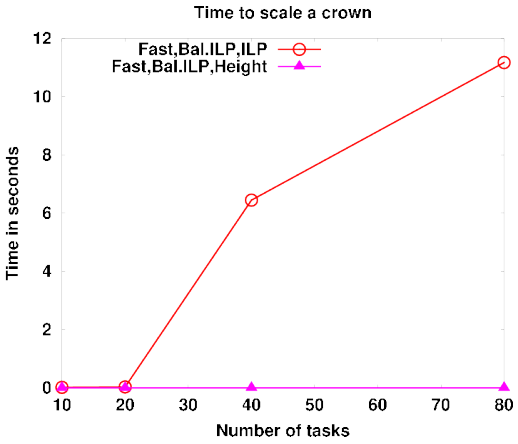
Fig. 9. Scheduling success rate and projected energy for FFT, mergesort and parallel-reduction

Again, the approach from Xu et al. [2012] yields much higher makespans. Figure 7(d) demonstrates the efficiency of our Height heuristic, compared to our crown-optimal, ILP-based frequency scaling method.

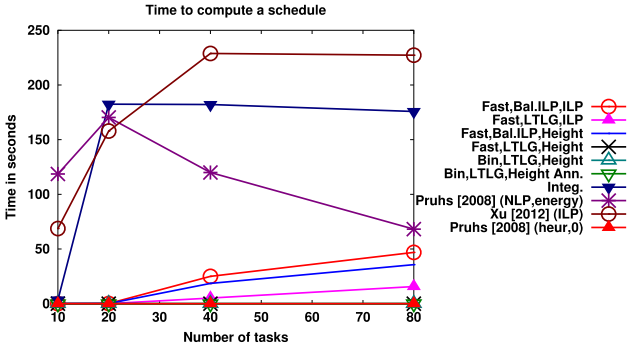
We provide the measured overall optimization time for all scheduler variants for the synthetic task collection set by number of tasks and number of cores (Figs. 10(c) and 10(f)), as well as the optimization time for both mapping and frequency scaling phases (Figs. 10(a), 10(d), 10(b) and 10(e)). We compare the measured optimization time of our heuristics and their time complexity by number of tasks and number of cores in Figs. 11 and 12 for small, medium and large problems of the synthetic task set. We can see that Pruhs, Xu's and our crown integrated ILP and NLP formulations largely dominate the overall optimization time. Phase-separated, crown-optimal ILP crown schedulers yield a much lower optimization time and none of the heuristics is visible in Figs. 10(c) and 10(f). The 6 figures in Fig. 10 demonstrate the LTLG and Height heuristics run much faster than their crown-optimal ILP counterparts, and that the phase-separated crown schedulers' optimization time is dominated by



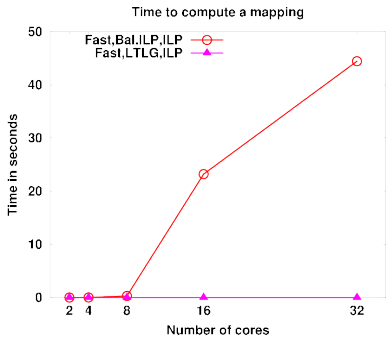
(a) Mapping phase runtime by number of tasks.



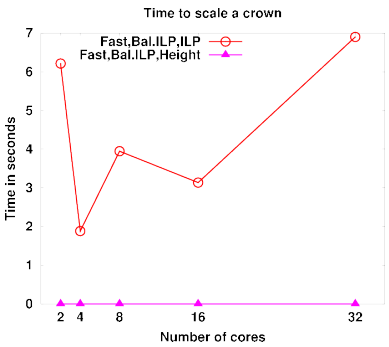
(b) Frequency scaling phase runtime by number of tasks.



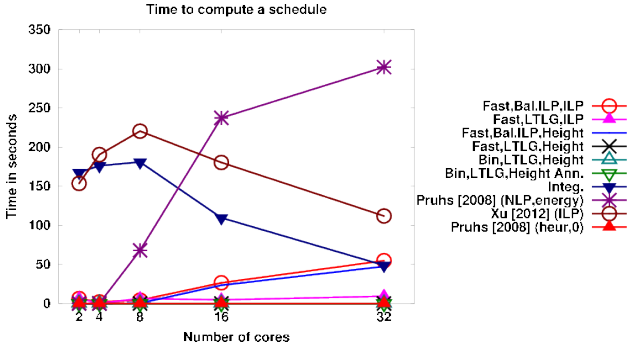
(c) Overall optimization time by number of tasks.



(d) Mapping phase runtime by number of cores.

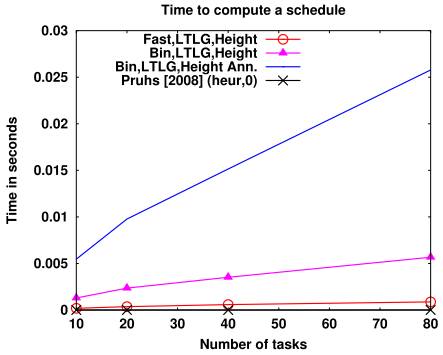


(e) Frequency scaling phase runtime by number of cores.

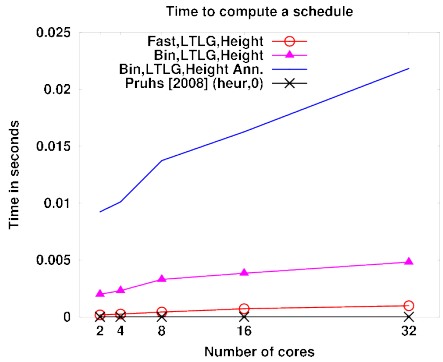


(f) Overall optimization time by number of cores.

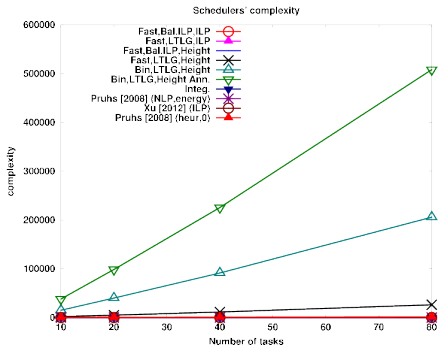
Fig. 10. Share of mapping and frequency scaling phases over overall optimization time.



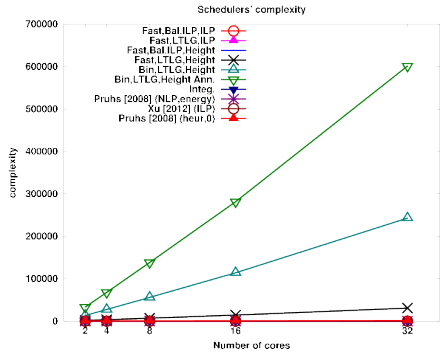
(a) Overall optimization time per number of tasks.



(b) Overall optimization time per number of cores.



(c) Overall complexity per number of tasks.

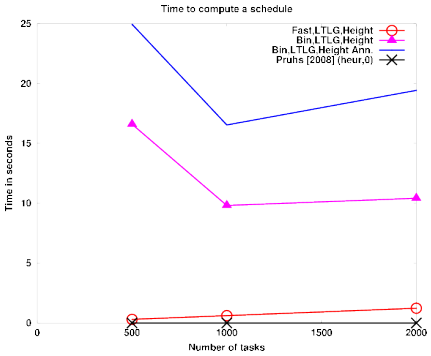


(d) Overall complexity per number of cores.

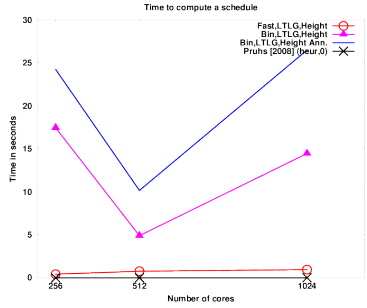
Fig. 11. Measured optimization time and time complexity of schedulers

the mapping phase. Figures 11(a), 11(b), 12(a) and 12(b) demonstrate that the optimization time for heuristics are dominated by the simulated annealing and our binary search crown schedulers. The phase-separated LTLG and Height crown heuristic as well as Pruhs' heuristic perform much faster. This can be explained by their lack of an elaborate allocation phase that our binary search and simulated annealing implement as a feedback loop. However, such simplistic allocation strategy makes them to produce worse solution, or to fail at producing any valid one. Finally, Figs. 11(a), 11(b), 11(c) and 11(d) show that our heuristics's optimization time follow the optimization time predicted by our complexity analysis. However, this is not true for our simulated annealing and binary search with the large synthetic task collection and target architectures (Figs. 12(a), 12(b), 12(c) and 12(d)). This can be due to the search interruption when a valid schedule is found where all tasks run sequentially at the lowest frequency available.

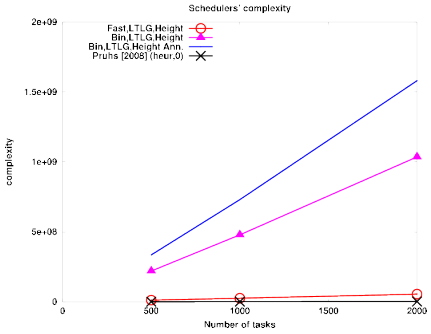
The crown constraints exhibits no restrictions to map sequential tasks to processors, but it allows the mapping of some parallel tasks. Hence, the solution space of Pruhs' mapping technique is strictly included in our crown mapping's solution space. Also, since we implemented the same ILP frequency scaling method for both schedulers, the overall solution space of crown schedulers includes the one of our implementation of Pruhs'. Therefore, when both formulations can be solved to optimality, the integrated crown scheduler will always find a solution at least as good as Pruhs'. Similarly, our LTLG heuristic for sequential tasks is reducible to LPT. Since we use LPT in our imple-



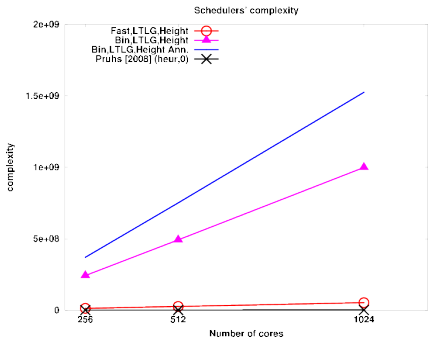
(a) Overall optimization time by number of tasks.



(b) Overall optimization time by number of cores.



(c) Overall complexity by number of tasks.



(d) Overall complexity by number of cores.

Fig. 12. Overall scheduling time and energy consumption of the resulting crown schedules for large task collections and massively parallel architectures

mentation of Pruhs' heuristic, its solutions cannot be better than LTLG's. Again, the Height heuristic is an implementation of Best-fit for the crown structure; therefore it has the same solution space as Pruhs' frequency scaling method. Since our LTLG and Height heuristics are reducible to Pruhs' mapping and frequency scaling methods, they can always find a solution at least as good.

We inspected schedules of the synthetic task sets produced by our best crown scheduler heuristic as well as the technique described by Pruhs et al. [2008]. We assume that tasks in our synthetic task set run sequentially and at the lower frequency available for between 1ms and 19ms, that is, 8μs with 32 cores at frequency 5. We assume further that switching frequency takes 50μs [Mazouz et al. 2014] and $5 \cdot 10^{-6}$ energy units. Finally, we assume that frequency switching can take place asynchronously, while a task is running[Mazouz et al. 2014]. We found that all frequency transitions can be hidden when tasks are processed or when processors are idle. In the case of a task running for a too short time to hide a necessary frequency transition, no significant additional time and energy was found.

7. RELATED WORK

Energy-aware allocation, mapping and scheduling problems are being researched intensively in scheduling theory. For instance, Edmonds and Pruhs [2012] consider an on-line scheduling algorithm for jobs arriving over time, minimize overall response time (the sum of distances between arrival and completion of each job). Jobs are paral-

lelizable with arbitrary speedup curves, but frequency scaling or energy does not play a role. Chan et al. [2011] use an analytic power model and continuous frequencies. Their algorithm does not statically know the work of a job, nor its speedup curve.

Energy efficient static scheduling of task collections with parallelizable tasks onto multiprocessors with frequency scaling has been considered by Li [Li 2012]. Their power model is defined as polynomially-reducible to f^α where $\alpha \geq 3$ and therefore comparable to the one we use in this paper. 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.

Xu et al. [2012] use a level-packing approach, which means that they might need as many conceptual barrier synchronizations as they have levels, i.e. up to the number of jobs. They use also discrete frequency levels, arbitrary power profiles of cores, moldable tasks with individual, arbitrary speedup curves and ILP. As their ILP approach needs many more variables than ours, they do not use a solver but only evaluate their heuristics.

Sanders and Speck [Sanders and Speck 2012] 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 continuous malleability and continuous frequency selection make the integrated problem much easier to solve. Pruhs et al. [2008] solve the problem optimally for sequential tasks with continuous frequencies and power model f^α.

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 [Nedjah et al. 2011]. Kessler et al. [2013a] elaborate on the relation between throughput requirements of streaming applications and deadline-based scheduling of independent jobs. They also show that for discrete frequency levels, already the optimality criterion of Pruhs et al. [2008] for sequential tasks does not hold anymore. Keller et al. [2012] and Avdic et al. [2011] 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. [2012]. Zahedi and Lee [2014] investigate the fair sharing of a computer resources to run independent tasks. Fairness is defined after a model derived from game theory to ensure each task receives enough resources to be computed fast without impacting the performance of other tasks. This work integrates better in a resource sharing context, than optimizing energy usage of a particular task-based parallel application under makespan constraints as Crown scheduling does.

Static scheduling of task collections with parallelizable tasks for makespan optimization has been discussed e.g. by Blasewicz et al. [2004]. Dynamic scheduling of task collections with parallelizable tasks for flow time optimization has been investigated by Gupta et al. [2010]. In both settings, frequencies were fixed.

Our LTLG heuristic relates to NPTS (Non-Malleable Parallel Tasks) scheduling, where the number of cores for each task is known and fixed. Fan et al. [2012] describe many previous research, including a 2-approximation [Garey and Graham 1975] and

a fully polynomial approximation [Amoura et al. 1997]. More approaches restrict parallel tasks to be mapped to contiguous cores [Baker et al. 1980] such as the NFDH (Next-Fit Deceasing Height) heuristic [Coffman et al. 1980], FFDH (First-Fit Decreasing Height) [Coffman et al. 1980] and BFDH (Best-Fit Decreasing height) [Lodi et al. 2002]. However, they don't capture the crown structure that further restricts the mapping of tasks to all cores within exactly one predefined group of cores. The LTLG heuristic is inspired from the LPT (Longest Processing Time) algorithm [Graham 1969], which is a 4/3-approximation algorithm to optimize the makespan of a collection of sequential tasks for multiprocessors.

8. CONCLUSION AND FUTURE WORK

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 moldable, partly moldable and sequential streaming tasks onto manycore processors in order to support energy-efficient execution of on-chip pipelined task graphs. Arbitrary power profiles for cores, analytic like f^α or from measurements, can be used.

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 conceptual barrier synchronizations necessary, and thus external fragmentation of scheduled idle time, thus improving opportunities for energy savings by voltage/frequency scaling.

We presented the *Longest Task, Lowest Group* (LTLG) heuristic, a generalization of the LPT algorithm, to produce load-balanced mappings of moldable tasks, and the *Height* frequency scaling heuristic to minimize energy consumption under a throughput constraint. We demonstrated that using these heuristics lowers the overall optimization time of phase-separated crown scheduling, and showed that both LTLG and Height heuristics produce solutions of quality near to that generated by crown-optimal ILP formulations with a much faster scheduling time. We also presented a method based on binary search to compute a good initial allocation and a simulated annealing meta-heuristic to further improve solutions toward the quality by an integrated optimal crown scheduler. Our best heuristic runs in near-linear time in the number of cores and tasks with a factor γ depending on properties of the task collection. We observe that a good processor allocation improves the energy efficiency of schedules, but that a good allocation comes at a high price in optimization time.

Our technique could be used in a compiler, as we consider off-line scheduling. The inputs would be a description of the taskgraph with workload and efficiency parameters for tasks (and in future work even communication load for edges) and a description of the target architecture. When considering a different target architecture, the application needs to be re-optimized. The crown structure could be relaxed to groups having non-power-of-2 number of cores, as long as all groups in a level have the same amount of cores, and all cores of a higher level group, belong to a unique ancestor group for each lower levels. If used to schedule tasks for a multiple many-core chips system, higher communication penalty to transmit messages from chip to chip, as opposed to communications from core to core, could be reflected in tasks' efficiency function. Future work will also consider on-line crown rescaling to additionally reduce energy consumption at runtime in a round where one or several tasks are not data-ready, and communication-

aware crown scheduling to consider bandwidth requirements of on-chip network links to better adapt to many-core network-on-chip modern architectures.

Also, additional constraints on scaling might be considered, such as power domains comprising whole processor subsets as in the case of Intel SCC where possible frequencies are constrained by the current voltage, voltage scaling is slow and can only be set for groups of 8 processors, and frequency scaling is constrained to tiles of two cores. Frequency and sleeping state switching time need to be taken into account to better model architectural constraints. The energy cost of communication between tasks also needs to be taken up in the mapping problem. Finally, we plan to validate energy saving by crown scheduling for a benchmark set on concrete many-core platforms such as the Intel SCC or Kalray MPPA, and compare the performance of our crown scheduler heuristics to other crown and non-crown schedulers for moldable tasks.

References

A. Amoura, E. Bampis, C. Kenyon, and Y. Manoussakis. Scheduling independent multiprocessor tasks. In R. Burkard and G. Woeginger, editors, *Algorithms – ESA '97*, volume 1284 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1997. .

K. Avdic, N. Melot, C. Kessler, and J. Keller. Parallel sorting on Intel Single-Chip Cloud Computer. In *Proc. A4MMC workshop on applications for multi- and many-core processors at ISCA-2011*, 2011. ISBN 978-3-86644-717-2. URL http://liu.diva-portal.org/smash/record.jsf?pid=diva2%3A624512.

B. S. Baker, E. G. Coffman, Jr., and R. L. Rivest. Orthogonal packings in two dimensions. *SIAM J. on Computing*, 9(4):846–855, November 1980.

K. P. Belkhale and P. Banerjee. An approximate algorithm for the partitionable independent task scheduling problem. In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 72–75, 1990.

J. Blasewicz, M. Machowiak, J. Weglarz, M. Kovalyov, and D. Trystram. Scheduling malleable tasks on parallel processors to minimize the makespan: Models and algorithms for planning and scheduling problems. *Annals of Operations Research*, 129: 65–80, 2004.

H.-L. Chan, J. Edmonds, and K. Pruhs. Speed scaling of processes with arbitrary speedup curves on a multiprocessor. *Theory Comput. Syst.*, 49(4):817–833, 2011.

P. Cichowski, J. Keller, and C. Kessler. Modelling power consumption of the Intel SCC. In *Proceedings of the 6th MARC Symposium*, pages 46–51. ONERA, July 2012.

E. Coffman, Jr., M. Garey, D. Johnson, and R. Tarjan. Performance bounds for level-oriented two-dimensional packing algorithms. *SIAM Journal on Computing*, 9(4): 808–826, 1980.

J. Edmonds and K. Pruhs. Scalably scheduling processes with arbitrary speedup curves. *ACM Trans. Algorithms*, 8(3):28, 2012.

L. Fan, F. Zhang, G. Wang, and Z. Liu. An effective approximation algorithm for the malleable parallel task scheduling problem. *J. Parallel Distrib. Comput.*, 72(5):693–704, May 2012.

M. Garey and R. Graham. Bounds for multiprocessor scheduling with resource constraints. *SIAM Journal on Computing*, 4(2):187–200, 1975. .

M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 151–162, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0. .

R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

A. Gupta, S. Im, R. Krishnaswamy, B. Moseley, and K. Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *Proc. SPAA'10*, pages 11–20. ACM, 2010.

G. Kahn. The semantics of simple language for parallel programming. In *IFIP Congress*, pages 471–475, 1974.

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*, 18(14):1987–2023, 2012.

C. Kessler, P. Eitschberger, and J. Keller. Energy-efficient static scheduling of streaming task collections with malleable tasks. In *Proc. 25th PARS-Workshop*, number 30 in PARS-Mitteilungen, pages 37–46, 2013a.

C. Kessler, N. Melot, P. Eitschberger, and J. Keller. Crown Scheduling: Energy-Efficient Resource Allocation, Mapping and Discrete Frequency Scaling for Collections of Malleable Streaming Tasks. In *Proc. of 23rd Int. Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2013)*, 2013b.

E. A. Lee and D. G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

K. Li. Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed. *IEEE Transactions on Parallel and Distributed Systems*, 19(11):1484–1497, 2008.

K. Li. Energy efficient scheduling of parallel tasks on multiprocessor computers. *J. of Supercomputing*, 60(2):223–247, 2012.

A. Lodi, S. Martello, and M. Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2):241 – 252, 2002. .

A. Mazouz, A. Laurent, B. Pradelle, and W. Jalby. Evaluation of CPU frequency transition latency. *Computer Science - Research and Development*, 29(3-4):187–195, August 2014. .

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.*, 57(1):79–94, 2011.

K. Pruhs, R. van Stee, and P. Uthaisombut. Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, 43(1):67–80, July 2008.

P. Sanders and J. Speck. Energy efficient frequency scaling and scheduling for malleable tasks. In *Proc. of the 18^{th} Int. Conference on Parallel Processing*, Euro-Par'12, pages 167–178, 2012.

H. Xu, F. Kong, and Q. Deng. Energy minimizing for parallel real-time tasks based on level-packing. In *2012 IEEE 18th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pages 98–103, Aug 2012.

S. M. Zahedi and B. C. Lee. REF: Resource Elasticity Fairness with sharing incentives for multiprocessors. *SIGARCH Comput. Archit. News*, 42(1):145–160, Feb. 2014. ISSN 0163-5964. . URL http://doi.acm.org/10.1145/2654822.2541962.

ACKNOWLEDGMENTS

C. Kessler acknowledges partial funding by EU FP7 EXCESS, Vetenskapsrådet and SeRC. N. Melot acknowledges partial funding by the CUGS graduate school at Linköping University.