Energy-Efficient Static Scheduling of Streaming Task Collections with Malleable Tasks

Christoph Kessler[1], Patrick Eitschberger[2], Jörg Keller[2]

[1]Dept. of Computer and Information Science (IDA)
Linköpings Universitet
58183 Linköping, Sweden
christoph.kessler@liu.se
[2]Faculty of Mathematics and Computer Science
FernUniversität in Hagen
58084 Hagen, Germany
⟨firstname⟩.⟨lastname⟩@fernuni-hagen.de

Abstract: We investigate the energy-efficiency of streaming task collections with parallelizable or malleable tasks on a manycore processor with 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 on user level. A stream of data flows through the tasks and intermediate results are forwarded to other tasks like in a pipelined task graph. We first show the equivalence of task mapping for streaming task collections and normal task collections in the case of continuous frequency scaling, under reasonable assumptions for the user-level scheduler, if a makespan, i.e. a throughput requirement of the streaming application, is given and the energy consumed is to be minimized. We then show that in the case of discrete frequency scaling, it might be necessary for processors to switch frequencies, and that idle times still can occur, in contrast to continuous frequency scaling. We formulate the mapping of (streaming) task collections on a manycore processor with discrete frequency levels as an integer linear program. Finally, we propose two heuristics to reduce energy consumption compared to the previous results by improved load balancing through the parallel execution of a parallelizable task. We evaluate the effects of the heuristics analytically and experimentally on the Intel SCC.

1 Introduction

In a streaming task application, all 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). As an example one might consider an application where a digitized image is input, preprocessed by the first task to enhance contrast, forwarded to two further tasks that extract horizontal and vertical edges, respectively, and forward their results to a fourth task, that computes the final result, e.g. the decision whether a certain pattern is present in the image or not. The fact such an application works on a sequence of (blocks of) images that "stream" through the tasks, gives the name to this type of application. Streaming task applications are wide-spread in embedded systems, e.g. as applications for multiprocessor systems on a chip (MPSoC).

A streaming task application can be modeled by a static streaming task graph, where the nodes represent the tasks, which are not annotated with runtimes but with computational rates. An edge between tasks u and v does not indicate a precedence constraint but denotes a communication of outputs from u to become inputs of v, and is annotated with a bandwidth. As the communication typically is done with 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, and as there are normally more tasks than cores on the underlying machine, several tasks might run simultaneously 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 , so that the required throughput can be achieved. In the following we will only consider computational loads and neglect the influence of communication, hence we talk of streaming task collections.

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

We are interested in mapping the tasks to processors in such a way that a required throughput is achieved and that, after a suitable frequency scaling of each core, power consumption is minimized. As the load for each core does not change over time, the initial frequency scaling will not have to be altered, so that this power consumption also results in a minimized energy consumption over the runtime of the streaming application.

In this respect, we make three contributions: First, we show that under continuous frequency scaling, the problem of energy-optimal mapping of streaming task collections is identical to the energy-optimal mapping of task collections without precedence constraints, so that existing approximations for the latter problem [PvSU08] can be applied.

Second, we show that under discrete frequency scaling, the problem of energy-optimal mapping of streaming task collections can be expressed as a variant of bin packing with fixed costs. We give a formulation as an integer linear program (ILP). For larger problem instances, existing heuristics for the bin packing problem [CPRT10, HS09] can be used. We also show that in contrast to continuous frequency scaling, an energy-optimal solution with discrete frequency scaling might require processors to switch their operating frequency, and might even comprise processor idle times.

Third, we show that if the tasks are parallelizable (so-called malleable tasks), then this can be exploited to improve the balancing of loads between the processors, and thus reduce power consumption, although the parallelization typically reduces efficiency, i.e. increases the total load. We give two heuristics to select which tasks should be parallelized.

We test our findings with several example applications, and use power measurements from the Intel Single Chip Cloud (SCC) manycore processor [How11] for validation.

Streaming task applications have been investigated previously mainly by the MPSoC community (cf. e.g. [ACP06]), but mostly in an adhoc or application-specific manner. Hulten et al. [KKH12] 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. [CKK12a]. Energy-efficient frequency scaling of task collections without precedence constraints has been considered by Pruhs et al. [PvSU08], although only as a side issue, and for the case of continuous frequency scaling.

Static scheduling of task collections with parallelizable tasks for makespan optimization has been investigated by Eriksson et al. and Kessler and Löwe [EKC06, KL08]. Dynamic scheduling of task sets with parallelizable tasks for flow time optimization has been investigated by Pruhs et al. [GIK$^+$10]. In both settings, frequency scaling was not considered.

Energy-efficiency for static scheduling of task collections with parallelizable tasks onto multiprocessors with frequency scaling has been considered by Li [Li12]. However, the number of processors allocated to each task (called task size by Li) is fixed and given as part of the problem scenario in that work, and continuous frequency scaling is used. Sanders and Speck [SS12] present an optimal solution for concave energy functions and non-integer processor allocation, yet still with continuous frequency scaling.

The remainder of this article is organized as follows. In Section 2, we treat energy-efficient mapping of streaming task collections with sequential tasks under continuous and discrete frequency scaling. In Section 3, we treat energy-efficient mapping of streaming task collections with parallelizable tasks, and give preliminary experimental results for example applications. In Section 4, we give an outlook onto future work.

2 Energy-efficient Mapping of Sequential Tasks

We first consider the equivalence between task collections and streaming task collections, which leads to an algorithm for the case of continuous frequency scaling, and afterwards investigate the case of discrete frequency scaling.

Continuous Frequency Scaling Pruhs et al. [PvSU08] consider the static scheduling of a set of tasks with given load, i.e. number of instructions to be executed, onto a set of processors. The tasks assigned to a processor are executed one by one. The power model is quite simple: the computational power, i.e. number of instructions executed per time unit by a processor is proportional to the operating frequency f, which can be continuously scaled, and the power consumption is given by f^α, where α typically is between 2 and 3.

They note that for an energy-optimal schedule, a processor should always run at the same frequency, that the frequencies should be adapted such that the processors all finish with their respective workload at the same time, and that there are no idle time gaps between execution of tasks. The frequencies are thus determined by the distribution of the workloads onto the processors. If a processor i has total workload w_i and the makespan is M, then the processor must run at frequency $f_i = w_i/M$. The power consumption is $P = \sum_i f_i^\alpha = \sum_i (w_i/M)^\alpha$, and the energy consumption is

$$E = P \cdot M = M^{1-\alpha} \cdot \sum_i w_i^\alpha \ . \tag{1}$$

Thus, the minimization of the energy for a given makespan M is equivalent to minimizing the l_α-norm of the processor workloads by distributing the tasks over the processors in a suitable manner. They refer to an existing polynomial-time approximation scheme (PTAS) for this problem. Note that Pruhs et al. use a variant of (1) to minimize the makespan for a given energy budget, but that both problems are equivalent in their setting.

Now consider our streaming task collection. If we consider the tasks' computational rates to represent the workload to consume one input packet and produce an output packet, then one round of execution of all tasks assigned to each processor (according to the round-robin scheduling) exactly looks like the problem considered by Pruhs et al. Also here, the processors will run at fixed speeds, have no idle times, and finish their rounds at the same time. The makespan M here represents the throughput requirement T of the application in the form of the time for one round of the round-robin scheduler: if the application repeatedly produces one output packet of size c after time M, then the throughput is c/M. To fulfill the throughput requirement, M may be at most c/T.

It follows, that minimizing the power consumption of a streaming task collection (and thus the energy consumption if a certain operating timespan of the application is considered) under continuous frequency scaling is achieved by distributing the tasks over the processors in such a way that the l_α-norm of the processor workloads (defined as the sum of the computational rates of the tasks assigned to each processor) is minimized.

Discrete Frequency Scaling Despite the insight derived from the above analytical solution, the reality looks slightly different: processors can only assume a small number s of discrete frequency levels, and the dependency of the power consumption on the frequency level is more complex than a simple polynomial [CKK12b].

For this case, one can model the task assignment as a bin packing problem: The p processors are represented by s sets of p bins, the bins in each set S_j having a capacity equivalent to the frequency f_j at level $j = 1, \ldots, s$. The cost Pw_j of a bin in set S_j is equivalent to the power consumption of a processor at frequency f_j, where there is no need to derive an analytical power model, but it is sufficient to choose the cost according to power measurements of the target machine [CKK12b]. The only restriction is that the cost is increasing with size. When the tasks have been distributed over the bins, let S'_j denote the subset of non-empty bins from S_j. The problem to be solved is to distribute the tasks in such a way that the workload of any bin i in set S'_j never surpasses the capacity f_j, that never more than p of the $p \cdot s$ bins are in use (because ultimately there are only p processors), and that the power consumption

$$\sum_{j=1}^{s} |S'_j| \cdot Pw_j$$

is minimized.

Note however that in the case of discrete frequency scaling, some properties of energy-optimal schedules from continuous frequency scaling do not hold anymore. These are that each processor always runs at the same frequency and that all processors finish at the same time given by the makespan, so that there are no idle times. This can be shown by the following counter-example. Assume that we have $2p + 1$ unit-time single-threaded tasks to be scheduled onto p processors. The most balanced schedule has makespan $M = 3$; w.l.o.g. assume that processor P_1 gets 3 tasks and P_2 to P_p get 2 tasks each and one unit of idle time on a machine running all cores at maximum frequency. Assume further that the processors each have discrete frequency downscaling factors 1.0, 1.33, 1.67, 2.0 and a cubic power function.

If we use downscaling factor 1.33 on processors P_2 to P_p, then they finish their tasks after time 2.66 and thus exhibit an idle time of 0.34. The energy consumption in this case is $3 \cdot 1^3 + (p - 1) \cdot 2.66 \cdot 1.33^{-3} \approx 1.13p + 1.87$, and even higher if we consider the power consumption during the idle times. Using downscaling factor 1.67 on any of P_2 to P_p is not possible as it would result in a runtime of 3.34 and violate the makespan $M = 3$.

An alternative here is to have each of the processors P_2 to P_p run its first task with downscaling factor 1.33, and the second task with downscaling factor 1.67. Then all processors finish after $M = 3$, i.e. there are no idle times anymore, and the energy consumption is $3 \cdot 1^3 + (p - 1) \cdot (1.33 \cdot 1.33^{-3} + 1.67 \cdot 1.67^{-3}) \approx 3 + (p - 1) \cdot 0.924 = 0.924p + 2.076$. This is lower than the above energy, even if an energy penalty for frequency switching would be taken into account. Note that we do not consider time or energy overhead from frequency scaling during runs.

Note that while using scaling factors of 1.0 and 2.0 for the tasks on processors P_2 to P_p would also be possible to achieve makespan 3, the energy is higher due to the greater disparity of the frequencies. Note also that there are scenarios where idle times on processors remain even if task-wise frequency scaling is applied, e.g. if the last task on P_1 in the above scenario would have runtime 0.98 instead of 1.

The task-wise frequency scaling scenario can still be formulated as a linear program: we only have to remove the requirement that for each processor, only one bin (i.e. only one frequency) is used. The target function also has to be adapted to minimize energy, by taking into account the time that each processor spends in each frequency level.

The linear program uses $p \cdot s \cdot n$ binary variables where $x_{i,j,t} = 1$ denotes that task t has been assigned to bin i of set S_j, i.e. to processor i on frequency level j. Then we minimize the target energy function

$$\sum_{i=1}^{p} \sum_{j=1}^{s} Pw_j \cdot \sum_{t=1}^{n} x_{i,j,t} \cdot \frac{w_t}{f_j}$$

where w_t is the work of, i.e. number of instructions executed by task t and f_j represents the frequency (number of instructions executed per time) at level j. Thus w_t/f_j (both values are constants in the linear program and thus may be divided) is the time that task t needs if it is mapped onto a processor running at frequency level j, and by multiplication with the power consumption Pw_j we achieve the energy consumed for task t. Note that the energy for idle time is not modelled, as we assume that idle times are minimized by task-wise frequency scaling compared to processor-wise scaling. Note also, that the energy penalty for frequency switching is not modelled. At most s switches are needed for each round of scaling, as all tasks running at a particular frequency level on a processor can be scheduled successively. If some frequency levels are not used, the number of frequency switches is reduced accordingly.

The following constraints apply on the binary variables $x_{i,j,t}$.

Each task is mapped exactly to one bin:

$$\forall t = 1, \ldots, n : \sum_{i=1}^{p} \sum_{j=1}^{s} x_{i,j,t} = 1 \ .$$

No processor is overloaded:

$$\forall i = 1, \ldots, p : \sum_{j=1}^{s} \sum_{t=1}^{n} x_{i,j,t} \cdot \frac{w_t}{f_j} \leq M \ .$$

One sees that $p \cdot s \cdot n$ variables and $p + n$ constraints are needed to map n tasks onto p cores with s frequency levels. For problem sizes where it is not possible anymore to employ an ILP solver such as gurobi or CPLEX, there exist heuristics for the bin packing problem with fixed costs [CPRT10, HS09].

Note that while we have modelled the frequency levels to be identical for all processors, this is not a restriction and can be adapted by replacing f_j and Pw_j by processor-specific constants $f_{i,j}$ and $Pw_{i,j}$.

3 Energy-Efficient Mapping of Parallelizable Tasks

We first devise two strategies to choose which tasks shall be parallelized to reduce energy consumption, and then evaluate both with several benchmark task collections.

Strategies for Parallelization We have seen in the previous section that energy minimization for streaming task collections requires balancing the workloads over the processors. However, if there are large tasks with a workload notably above the average workload of a processor, or if the number of tasks is not a multiple of the processor count p, then a balance cannot be achieved, if tasks cannot be divided into smaller parts. A possibility to divide a task into smaller pieces is parallelizing it into a set of subtasks.

Assume for example that the task collection consists of $n = 101 = p + 1$ tasks with workloads $w_t = 1$, to be mapped onto $p = 100$ processors. The best solution is to assign 2 tasks to processor 1 and run it at frequency $f_1 = 2$, and assign one task to each remaining processor, running those at frequencies $f_i = 1$, where $i \geq 2$. For a cubic power function, the power consumption would be $P = 2^3 + 99 \cdot 1^3 = 107$. If one of the tasks (the second assigned to processor 1) could be parallelized into p subtasks, each having a workload $\tilde{w} \geq 1/p$, all processors would have an even workload of $1 + \tilde{w}$ and could run at an appropriate frequency. Even if the parallel efficiency is only 50%, i.e. $\tilde{w} = 0.02$ instead of the optimal $1/p = 0.01$, then the power consumption would be $P = p \cdot (1 + \tilde{w})^3 \approx 106$, and thus lower as before.

One of the strategies presented in [EKC06] to reduce makespan was called *serialization*, where each task was parallelized for p processors. Obviously, in this case a perfect load balance can be achieved, as each task t will put a load of $(1/e(p)) \cdot w_t/p$ onto each processor, where $e(p)$ is the parallel efficiency. Compared to the power consumption in the optimal case (perfect load balance without any overhead), the power consumption is higher by a factor $e(p)^{-\alpha}$. Whether this strategy saves energy depends on the load imbalance in the case where all tasks are sequential. Hence, knowing the task collection, the result of the ILP computation, and the parallel efficiency that can be achieved, one can check in advance whether this strategy is successful. Please note that the parallel efficiency includes overhead for barrier synchronization after execution of a parallelized task in the case that the subtasks' runtimes are not completely identical, which might occur due to data dependencies.

We propose a second, greedy strategy that we call *bigblock*. Here, one parallelizes the largest task onto p processors, and then maps the other tasks with the ILP from the previous section. This reduces overhead from parallelization and synchronization, because the majority of tasks remain sequential, and still improves balance by balancing a large part of the load. The parallelized task is run at the average of the frequencies chosen by the ILP from the previous section, where all tasks are sequential. Then the makespan for the ILP mapping of the remaining tasks can be shortened according to the runtime of the parallel task. The energy consumption is the sum of the energy for running the parallelized task and the energy from the ILP mapping of the remaining tasks. If the bigblock strategy leads to a lower energy consumption than the ILP from the previous section, then it can be extended to an iterative strategy: parallelize also the next largest task as long as energy consumption is improved. Note that the split between parallel and sequential tasks requires a single frequency switch during execution of the task collection.

Evaluation We evaluate both strategies with two synthetic examples, summarized in Tab. 1. As target platform, we use the Intel single chip cloud (SCC) processor [How11], an 48-core multiprocessor where core frequencies can be set by an application program, however only for groups of 8 cores (power domains), and where the application program repeatedly records the chip's power consumption via a library call. If the time interval between successive measurements m_i is δ, then the energy consumed in a run is computed as $\sum_i \delta \cdot m_i$. The available discrete frequency levels are $1600/j$ MHz, where $j = 2, \ldots, 16$. The operating voltage is set to the lowest possible value when frequency changes.

	benchmark 1	benchmark 2
profile	56 tasks	48 tasks + 8 heavy tasks
sequential tasks	26.6 J	27.88 J
serialization	25.9 J if $e(p) \geq 0.94$	27.64 J if $e(p) \geq 0.9$
bigblock	25.9 J if $e(p) \geq 0.7$	27.64 J if $e(p) \geq 0.7$

Table 1: Benchmark details and energy consumption on Intel SCC. A task executes in $0.925s$ (bench. 1) and $0.75s$ (bench. 2) on one core at 200 MHz, a heavy task needs 2.67 times as long.

As our first benchmark, we use a variant of the $n = p + 1$ tasks problem presented above. We have $n = 56$ tasks with equal runtime $w_t = 1.85 \cdot 10^8$, so that a core with one task can run at 200 MHz, and a core with 2 tasks runs at 400 MHz operating frequency to achieve a makespan of $M = 1\,s$. We choose $n = p + 8$ tasks because frequency scaling only occurs for groups of 8 processors. The ILP model delivers the obvious solution: 8 cores receive two tasks each, 40 cores receive one task each. The power consumption in this case is 26.6 Watt over the complete runtime, for synthetic tasks that do nothing than adding up numbers. It follows that the energy consumption is 26.6 Joule for a runtime of $1\,s$. We used the gurobi ILP solver to compute optimal solutions of the ILP models. In all cases, the gurobi runtime was less than 2 seconds on a notebook.

The serialization strategy, where each of the tasks is parallelized onto 48 cores, leads to a workload of $\tilde{w}_t = 1.85/48 \cdot 10^8/e(48)$ per subtask, with 56 subtasks per core. For an optimal parallel efficiency of 1, this leads to a workload of about $2.15 \cdot 10^8$ per core, which could be handled by an operating frequency of 228 MHz. Even a parallel efficiency of 94% can be tolerated. The power consumption of all cores running at 228 MHz is 25.9 Watt over the complete runtime, giving an energy consumption of 25.9 Joule. Hence, the the energy consumption in this case is slightly lower.

The bigblock strategy, which we adapt to include 8 tasks because of the power domains, leads to a similar result: each core carries 8 subtasks (one from each parallelized task), and one sequential task. Here, in the case of optimal parallel efficiency, also a frequency of 228 MHz is sufficient, and in this case even a parallel efficiency of 70% can be tolerated, because of the reduced number of parallelized subtasks per core. The power consumption and thus the energy consumption is identical to the serialization strategy. Thus, for this benchmark, both parallelization strategies lead to an advantage over strictly sequential tasks. The bigblock strategy might be advantageous in that case because it maintains the same frequency level and energy consumption for lower parallel efficiency.

The reason why the gain might seem smaller than expected is that the processor chip's total power consumption also comprises power for the on-chip network and the memory controllers, which basically remains the same for different load distributions. Thus, the difference in power consumption between different frequency levels is smaller than for a simple cubic power function: if a core consumes 0.4 Watt at 100 MHz (48 cores then would consume about 20 Watt) and a cubic power function is used, the sequential tasks would consume $8 \cdot (4 \cdot 0.4)^3 + 40 \cdot (2 \cdot 0.4)^3 \approx 53\,J$ in one second, while both parallelization strategies would lead to an energy consumption of $48 \cdot (2.28 \cdot 0.4)^3 \approx 36\,J$.

Our second benchmark also is a benchmark with 56 tasks, however 8 "heavy" tasks have a load of $4 \cdot 10^8$ instructions, and the remaining 48 have a load of $1.5 \cdot 10^8$ instructions, respectively. The makespan is $M = 1\,s$. To simplify the ILP model, we only use the operating frequencies of 100, 200, and 400 MHz, and model each task only once per power domain. For sequential tasks, the 8 cores of the first power domain each receive one heavy task, and the 8 cores of the second power domain each receive two normal tasks; the cores in these power domains run at 400 MHz. The remaining cores each receive one normal task and run at 200 MHz. The energy consumption is 27.88 Joule.

With the serialization strategy, each core would receive 8 subtasks with load $0.0833 \cdot 10^8$ each, and 48 subtasks with load $0.03125 \cdot 10^8$ each. Assuming an optimal parallel efficiency for the first subtasks, each core would run at frequency 400 MHz for $1/6$ seconds, and run at 200 MHz for the rest of the time (0.83 seconds)[1] , allowing for a parallel efficiency of 90% for the other subtasks. The power consumption in the two parts of the computation are 33.8 Watt and 26.28 Watt, respectively, leading to an energy consumption of 27.64 Joule, which is lower than with sequential tasks.

The bigblock strategy would parallelize the heavy tasks as in the serialization strategy, and leave the normal tasks sequential. The frequencies and energy are as with the serialization strategy, however a parallel efficiency of 70% is tolerable for the parallelized tasks.

4 Conclusions and Outlook

We have investigated the similarities and differences between static scheduling of task collections and streaming task collections onto manycore processors with continuous and discrete frequency scaling, respectively. We have presented a linear program to schedule (streaming) task collections onto manycore processors with discrete frequency levels, that includes the use of frequency switching to achieve energy-optimal schedules. Furthermore, we have investigated how using the possibility to execute tasks in parallel can be used to improve the energy-efficiency of those schedules, and presented several heuristics which tasks to execute in parallel. We have evaluated our proposals with a small number of application scenarios analytically and experimentally.

Future work will comprise varying the degree of parallelism for malleable tasks, as parallel efficiency often is higher for a smaller processor count, and thus outweighs the increased imbalance, and the use of task-wise instead of processor-wise frequency scaling (cf. Sect. 2 and [KMEK13]). If frequency switches during tasks are allowed, even imitation of continuous frequency scaling becomes possible. We also plan a larger experimental evaluation, and to extend our scheduling algorithms to consider the influence of communication between tasks on energy consumption and the restrictions from SCC power domains (frequencies can only bet set for groups of 8 processors).

[1]We here allow dynamic frequency changes of cores to realize an average frequency of 233 MHz. As all cores switch frequency simultaneously, this does not introduce new load imbalances even within parallel tasks.

Acknowledgements 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.

References

[ACP06] Giuseppe Ascia, Vincenzo Catania, and Maurizio Palesi. A Multi-objective Genetic Approach to Mapping Problem on Network-on-Chip. *Journal of Universal Computer Science*, 12(4):370–394, 2006.

[CKK12a] Patrick Cichowski, Jörg Keller, and Christoph Kessler. Energy-efficient Mapping of Task Collections onto Manycore Processors. In *Proc. 5th Swedish Workshop on Mulitcore Computing (MCC 2012)*, November 2012.

[CKK12b] Patrick Cichowski, Jörg Keller, and Christoph Kessler. Modelling Power Consumption of the Intel SCC. In *Proc. 6th MARC Symposium*, pages 46–51. ONERA, July 2012.

[CPRT10] Teodor Gabriel Crainic, Guido Perboli, Walter Rei, and Roberto Tadei. Efficient Heuristics for the Variable Size Bin Packing Problem with Fixed Costs. Technical Report 2010-18, CIRRELT, 2010.

[EKC06] M. Eriksson, C. Kessler, and M. Chalabine. Load Balancing of Irregular Parallel Divide-and-Conquer Algorithms in Group-SPMD Programming Environments. In *Proc. 8th Workshop on Parallel Systems and Algorithms (PASA)*, pages 313–322, 2006.

[GIK$^+$10] Anupam Gupta, Sungjin Im, Ravishankar Krishnaswamy, Benjamin Moseley, and Kirk Pruhs. Scheduling jobs with varying parallelizability to reduce variance. In *Proc. 22nd ACM Symp. Parallelism in Algorithms and Architectures*, pages 11–20, 2010.

[How11] J. Howard et al. A 48-Core IA-32 message-passing processor in 45nm CMOS using on-die message passing and DVFS for performance and power scaling. *IEEE J. of Solid-State Circuits*, 46(1):173–183, January 2011.

[HS09] M. Haouari and M. Serairi. Heuristics for the variable sized bin-packing problem. *Computers & Operations Research*, 36:2877–2884, 2009.

[KKH12] Jörg Keller, Christoph Kessler, and Rikard 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.

[KL08] C. Kessler and W. Löwe. A Framework for Performance-Aware Composition of Explicitly Parallel Components. In *Proc. ParCo-2007 conference*, pages 227–234, 2008.

[KMEK13] C.W. 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. 23rd International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS 2013)*, 2013.

[Li12] Keqin Li. Energy efficient scheduling of parallel tasks on multiprocessor computers. *Journal of Supercomputing*, 60(2):223–247, 2012.

[PvSU08] Kirk Pruhs, Rob van Stee, and Patchrawat Uthaisombut. Speed Scaling of Tasks with Precedence Constraints. *Theory of Computing Systems*, 43(1):67–80, July 2008.

[SS12] Peter Sanders and Jochen Speck. Energy efficient frequency scaling and scheduling for malleable tasks. In *Proc. Euro-Par 2012*, pages 167–178, 2012.