Improving Energy-Efficiency of Static Schedules by Core Consolidation and Switching Off Unused Cores

Nicolas MELOT [a], Christoph KESSLER [a] and Jörg KELLER [b,1]
[a] *Linköpings Universitet, 58183 Linköping, Sweden*
[b] *FernUniversität, 58084 Hagen, Germany*

Abstract. We demonstrate how static, energy-efficient schedules for independent, parallelizable tasks on parallel machines can be improved by modeling idle power if the static power consumption of a core comprises a notable fraction of the core's total power, which more and more often is the case. The improvement is achieved by optimally packing cores when deciding about core allocation, mapping and DVFS for each task so that all unused cores can be switched off and overall energy usage is minimized. We evaluate our proposal with a benchmark suite of task collections, and compare the resulting schedules with an optimal scheduler that does however not take idle power and core switch-off into account. We find that we can reduce energy consumption by 66% for mostly sequential tasks on many cores and by up to 91% for a realistic multicore processor model.

Keywords. Energy-efficiency, static scheduling, moldable tasks, core power model

Introduction

Frequency scaling of cores has for several years been the method of choice to minimize energy consumption when executing multiple tasks on a parallel machine till a fixed deadline, cf. e.g. [6]. In recent VLSI generations however, the static power consumption in a processor core slowly starts to dominate the dynamic power consumption, so that frequency scaling loses its importance. Running with higher frequency and switching off idle cores might be an alternative, but the overhead for putting a core to sleep, and waking it up again later is considerable and much higher than the overhead for changing the frequency. This hurts especially if a streaming application is processed, where the same scheduling round occurs repeatedly, i.e. many times.

This overhead might be avoided if one packs the tasks onto fewer cores and have the remaining cores switched off all the time. This requires knowledge about the hardware, operating system and the application behavior; consequently, a solution must lie in a form of a user-level, application-specific scheduler. In order to achieve this in a scheduler, the underlying energy model of the cores must reflect this and distinguish between an idle

[1]Corresponding Author: J. Keller, FernUniversität in Hagen, Faculty of Mathematics and Computer Science, Parallelism and VLSI Group, Postfach 940, 58084 Hagen, Germany; E-mail: joerg.keller@fernuni-hagen.de.

core and a core that is switched off. However, this is typically not the case, as the idle power of a core often is much smaller than the power when the core is under full load and processing a compute-intensive task.

In order to test our hypothesis, we have extended an energy-optimal scheduler for independent, parallelizable tasks with deadline [5] so that its processor model incorporates idle power, which in turn necessitates to clearly distinguish between static and dynamic power consumption in the core model. We schedule a benchmark suite of task sets with the new scheduler, and compare the resulting schedules with the corresponding schedules from the scheduler without taking idle time into account. We find that the energy consumption of the schedules is improved by 66% for mostly sequential tasks on many cores and by up to 91% for a realistic multicore processor model.

The remainder of this article is organized as follows. In Section 1, we briefly summarize basics about power consumption of processor cores. In Section 2, we briefly explain the scheduler from [5] and extend it to include idle power consumption. In Section 3, we present the results of our experiments. In Section 4, we give conclusions and an outlook onto future work.

1. Power Consumption in Multicore Processors

The power consumption of a processor core can be roughly split into dynamic power and static power. Dynamic power P_d is consumed because transistors are switching. This is influenced by the energy needed for a switch, which depends quadratically on the supply voltage (as long as this voltage is far enough from the threshold voltage), how often the switch occurs, i.e. the frequency f within set F of applicable discrete frequency levels, and how many transistors are switching, i.e. on the code executed [3]. There are more influence factors such as temperature, but for simplification we assume that minimum possible voltage for a given frequency (and vice versa) are linearly related, and that for compute intensive tasks, the instruction mix is such that we know the average number of transistors switching. Therefore, ignoring the constants, we get

$$P_d(f) = f^3 \ .$$

Static power is consumed because of leakage current due to imperfect realization of transistors and due to momentary shortcuts when switching both transistor parts of CMOS circuits. Static power is both dependent on frequency and on a device-specific constant κ [1]. For simplification, we express it as

$$P_s(f) = f + \kappa \cdot \min F \ .$$

For simplification, we assume a proportionality factor of 1 and get:

$$P_t(f) = \zeta \cdot P_d(f) + (1 - \zeta) \cdot P_s(f) \ ,$$

where $\zeta \in [0; 1]$ expresses the relative importance of dynamic and static power consumption.

The energy consumed by a processor core while processing a load can be computed as the product of power consumption and time, while the power consumption is fix,

or by summing over sub-intervals in which power consumption is fix. The total energy consumed by a multicore is the sum of the core energy consumptions.

We consider a processor set P and a task set T. If a processor core runs idle, it consumes power as well. However, as long as idle power is sufficiently lower than the power under load, it might be ignored. Moreover, consider the situation that n tasks $i \in T$ with workloads of τ_i cycles each are processed at frequencies f_i on p cores before a deadline M. Then the runtime r_i of task[2] i is $r_i = \tau_i / f_i$, and the total energy can be computed as

$$E = \sum_{i \in T} r_i \cdot P_t(f_i) + \sum_{j \in P} t_j \cdot P_{idle} \ , \tag{1}$$

where t_j is the sum of the lengths of all idle periods on core j, and P_{idle} is the idle power. We ignore the overhead in time and energy to scale frequencies between tasks, as we assume that the task workloads are much larger than the scaling time.

If the power under load is modified to

$$\tilde{P}_t(f) = P_t(f) - P_{idle} \ ,$$

then the formula for the total energy changes to

$$E = \sum_{i \in T} r_i \cdot \tilde{P}_t(f_i) + \sum_{i \in T} r_i \cdot P_{idle} + \sum_{j \in P} t_j \cdot P_{idle} \tag{2}$$

$$= \sum_{i \in T} r_i \cdot \tilde{P}_t(f_i) + M \cdot p \cdot P_{idle} \ , \tag{3}$$

because the sum of the task runtimes and the idle periods is the total runtime of all cores. The latter part is fixed for given p and M and hence not relevant when energy is to be optimized.

Please note that even if we specify the idle periods explicitly and optimize with Eq. (1), energy consumption remains the same for different placements of tasks as long as the frequencies are not changed. For example, both schedules in Fig. 1 consume the same amount of energy, as each task is run at the same frequency in both schedules. In the following, we will assume $P_{idle} = \eta P_t(f_{min})$, where f_{min} is the minimum possible operating frequency. This is a simplification, as even in idle mode, some transistors are used and thus $P_d > 0$, but it gives a good lower bound.

Another possible view on idle periods is that cores might enter a sleep-mode during such periods, and hence the energy consumption of idle phases might be neglected. However, the times to bring an idle core into a sleep-mode and to wake it up again are much larger than frequency scaling times, and hence cannot be ignored or is not even feasible in many applications. Yet, one possibility to take advantage of sleep-modes is core consolidation, where tasks are packed onto fewer cores, and unused cores are not switched on at all. E.g. in the schedules of Fig. 1, one core could remain switched off in the right schedule, while all cores would be used in the left schedule. Thus, if idle power is counted and if core consolidation is possible, the energy consumption would be

[2]The formula is for a sequential task. For a parallel task on q processors, the runtime is $\tau_i/(f_i q e(q))$, where $e(q)$ is the parallel efficiency of that task on q processors.

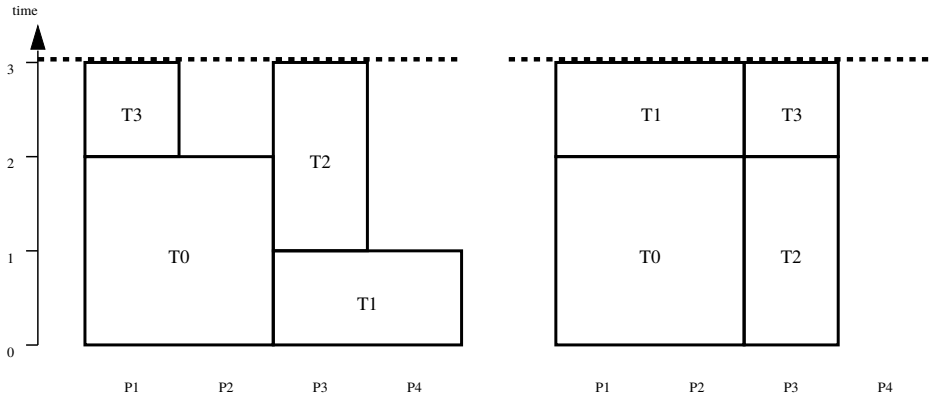


Figure 1. Different schedules for identical task set.

lower in the right case: Eq. (3) would change in the sense that p in the last term would be replaced by $p - u$, where u is the number of switched-off cores.

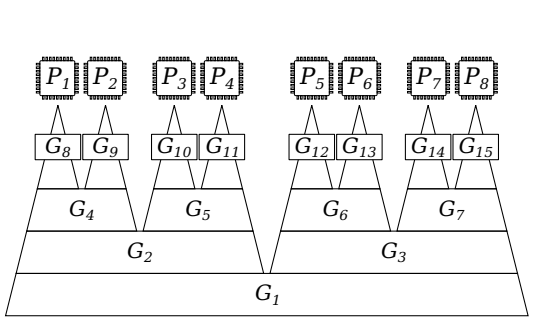
2. Energy-efficient Static Schedules of Parallelizable Tasks

We presented a new static scheduling algorithm called crown scheduling for task sets with parallelizable tasks and a deadline, where the energy consumption is to be minimized [5]. Such a task set models a streaming application with throughput requirements or real-time constraints, where all tasks are active simultaneously, and forward intermediate results directly to follow-up tasks, which avoids costly off-chip memory accesses. Multiple tasks on one core are scheduled round-robin up to a scheduling point, i.e. the production of the next intermediate result. If task communications go from one scheduling round to the next, then the tasks are independent within one round, where the deadline M of the round is derived from the throughput requirements, and the workloads τ_i of the tasks within one round are derived from the tasks' computational requirements. Thus the scheduling round is executed repeatedly, as indicated in Fig. 2(b). Tasks are assumed to be possibly parallelizable, i.e. we assume moldable tasks that are assigned widths w_i with $1 \leq w_i \leq \min\{W_i, p\}$, where W_i is the task-specific maximum width. A task uses w_i cores from beginning to end. With a task-specific parallel efficiency function $e_i(q)$, the runtime of a task running with q cores at frequency f_i can be given as $r_i = \tau_i/(f_i \cdot q \cdot e_i(q))$.

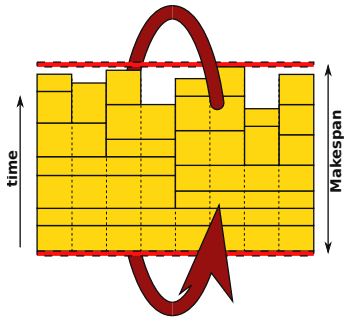
We assume a parallel machine with p cores, where each core can be independently scaled to a frequency from a finite set $F = \{F_1, \ldots, F_s\}$ of frequencies, and where the core power consumptions under load, i.e. $P_t(F_k)$, and idle power P_{idle} is known.

Then the scheduling consists of allocating w_i cores to each task, determine the operating frequency for each task, and arranging the parallel tasks such that all tasks are completed before the deadline, and that energy consumption is minimized.

In crown scheduling, we impose the following restrictions (assume p is a power of 2): Each task width must be a power of 2, and for each width w, there are p/w groups of that width. For example, for $w = p/2$, there are 2 groups, one comprising cores 1 to $p/2$, the other comprising the remaining cores. This reduces the number of possible



(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.

Figure 2. A group hierarchy and a possible corresponding unscaled crown schedule for 8 processors.

allocations from p to $\log p$ and the number of possible mappings from 2^p to $2p-1$. The groups are executed in order of decreasing width, so that a structure like in Fig. 2(a) arises, which also gives rise to the name "crown". As the rounds are executed over and over, a conceptual barrier is necessary at the beginning of a round (Fig. 2(b)), however, in practice, the parallel algorithm in the first task of group 1 (for width p) ensures an implicit synchronization of the cores.

In order to compute an optimal crown schedule, an integer linear program is used. The program uses $n \cdot p \cdot s$ binary variables $x_{i,j,k}$ with $x_{i,j,k} = 1$ if and only if task i is mapped to group j and run at frequency F_k. The width of group j is $w(j) = p/2^{\lfloor \log_2 j \rfloor}$, ie. group 1 has width p, groups 2 and 3 have width $p/2$, etc. The runtime of a task i can then be expressed as

$$r_i = \sum_{j,k} x_{i,j,k} \cdot \frac{\tau_i}{w(j) \cdot F_k \cdot e_i(w(j))}$$

and thus the target function, i.e. the energy to be minimized, is

$$E = \sum_i \sum_{j,k} x_{i,j,k} \cdot \frac{\tau_i \cdot P_t(F_k)}{w(j) \cdot F_k \cdot e_i(w(j))} \ . \tag{4}$$

In order to obtain feasible schedules, we formulate the constraint

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

i.e. each task is mapped exactly once. Furthermore, let C_m be all groups that comprise core m. Then

$$\sum_i \sum_{j \in C_m} \sum_k x_{i,j,k} \cdot \frac{\tau_i}{w(j) \cdot F_k \cdot e_i(w(j))} \leq M \ ,$$

i.e. the sum of the runtimes of all tasks mapped onto core m does not supersede the deadline. Finally, we forbid allocating more cores to a task than its maximum width allows:

$$\forall i \ : \sum_{j,w(j)>W_i} \sum_{k} x_{i,j,k} = 0 \ .$$

In the form stated above, idle power is treated as described in the previous section. To explore whether core consolidation is helpful, we additionally use p binary variables u_m where $u_m = 1$ if and only if core m is not used at all. Then the target function can be derived from Eq. (3) by adding idle power as in Eq. (4) but subtracting from p all unused cores:

$$E_{cons} = \sum_{i} \sum_{j,k} x_{i,j,k} \cdot \frac{\tau_i \cdot (P_t(F_k) - P_{idle})}{w(j) \cdot F_k \cdot e_i(w(j))} + (p - \sum_{m} u_m) \cdot P_{idle} \ . \qquad (5)$$

In order to set the u_m variables correctly, we need further constraints. If a core m is used, then u_m must be forced to 0:

$$\forall m \ : \ u_m \leq 1 - (1/n) \cdot \left(\sum_{j \in C_m, i, k} x_{i,j,k} \right) \ .$$

If any task i is mapped (at any frequency) to a processor group that uses m, then the sum is larger than zero, and hence the right-hand side is less than 1. Multiplying by $1/n$ ensures that the right-hand side can never be less than 0.

Forcing u_m to 1 if core m is not used is not strictly necessary, as an optimal solution will set as many u_m as possible to 1 to minimize energy. Yet, to also have feasible non-optimal schedules (e.g. in case of time-out), one can set

$$u_m \geq 1 - \sum_{j \in C_m, i, k} x_{i,j,k} \ .$$

3. Experiments

In this section, we compare schedules computed with target function (4) and schedules computed with target function (5), both for the same task sets. Then we evaluate corresponding schedules for their energy consumption if unused cores can be switched off.

To do this, we implement the ILP models from the previous section in AMPL and an analytical schedule energy evaluator in C++. We use the benchmark suite of task collections that already served to evaluate the crown scheduler heuristics [5].

We model the Intel i5 4690k Haswell processor with 2, 4, 8, 16 and 32 identical cores. We assume that each core's frequency can be set independently to one discrete value in the set $F = \{1.2, 1.4, 1.5, 1.6, 1.7, 1.9, 2, 2.1, 2.2, 2.3, 2.5, 2.7, 2.9, 3, 3.2, 3.3, 3.5\}$. For small and medium problems and machine sizes, several categories of synthetic task collections are 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 widths 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.

We also use task collections of real-world streaming algorithms: parallel FFT, parallel-reduction and parallel mergesort. FFT comprises $2 \cdot p - 1$ parallel tasks in 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. For these regular task collections, we use more constraining target makespan M.

Finally, we use the technique of [2] to extract tasks' workload and parallel efficiency from the Streamit benchmark suite [7]. We schedule all variants of the applications *audiobeam*, *beamformer*, *channelvocoder*, *fir*, *nokia*, *vocoder*, *BubbleSort*, *filterbank*, *perfectest* and *tconvolve* from the Streamit compile source package. We use their compiler to obtain each task's estimated workload, parallel degree and its communication rate that we use to compute the task's parallel efficiency.

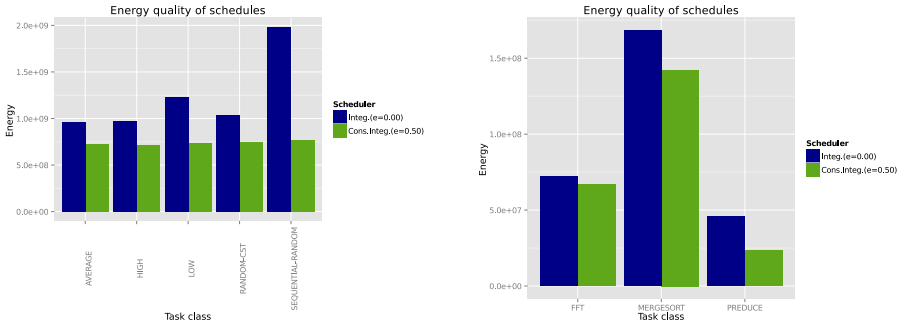
We use ψ_j to compute tasks j's parallel efficiency $e_j(q)$ (Eq. 6).

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

We measure the overall scheduling quality (energy consumption) using our integrated crown scheduler [5] and the enhanced version of this paper. We use the energy model as described by Eq. 5 with $\zeta = 0.649$, $\kappa = 52.64$ and $\eta = 0.5$. These parameters are derived from power values of an Intel Haswell processor. We only left out a constant factor, because that does not show when comparing energy values of schedules, and we weakened the term κ, because a high value might favor switching off cores too much. We run the ILP solver on a quad-core i7 Sandy Bridge processor at 3GHz and 8GB of main memory. We use Gurobi 6.0.0 and ILOG AMPL 10.100 with 5 minutes timeout to solve both ILP models, on Ubuntu 14.04.

Figures 3, 4 and 5 shows that our core consolidation technique benefits mainly to task collections whose tasks' parallel degree is low. This is not surprising as parallelization of tasks already balances load, so that re-arranging the tasks is difficult [4]. The benefits increase with the number of tasks and the number or cores. The energy consumption of classic task collections is reduced by 12% on average for 63 tasks, 16% on average with mergesort and 63% on average with 32 cores. The energy consumption of the synthetic task collection is reduced by 41% on average for 80 tasks, 53% on average with sequential or tasks of low parallel degree and 63% on average with 32 cores. Fig. 6 indicates this results also apply to our Streamit Task collections. The energy consumption of the Streamit task collection is reduced by 74% on average for 127 tasks, 61% on average with the application *FIR* and 74% on average with 32 cores.

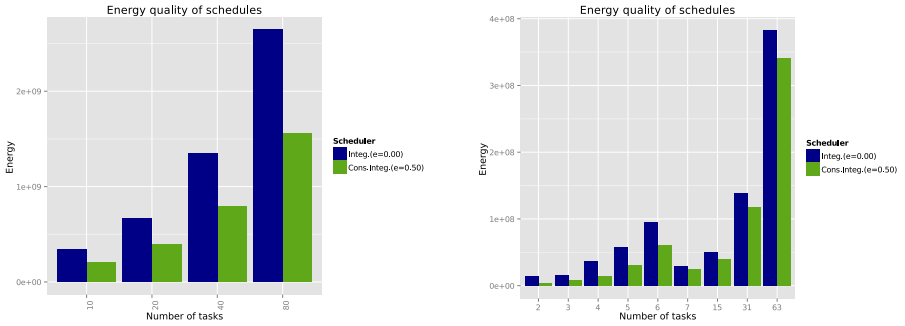
The consolidation technique can lead to energy savings only if schedules yield enough idle time to move tasks from a core to another. If there is no room to move a task to a processor, then all tasks already mapped to this processor as well as the task to be moved need to be run at a higher frequency. Because of the rapid growth of the dynamic power function (f^3), this can lead to the consumption of more energy than the



(a) Energy consumption per number of tasks for the synthetic task collection.

(b) Energy consumption per number of tasks for the class task collection.

Figure 3. Our technique performs better on sequential task collections.



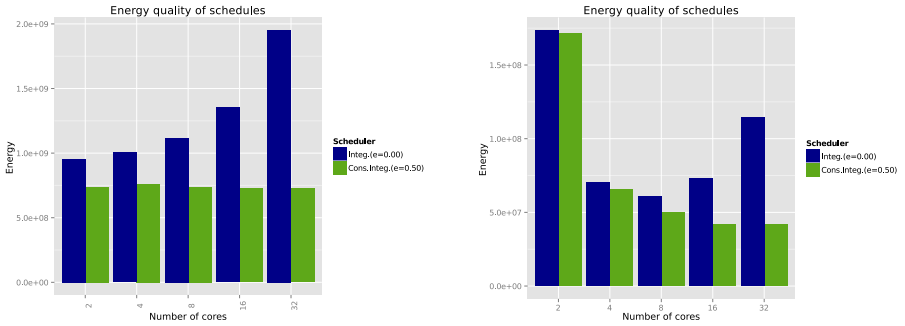
(a) Energy consumption per number of tasks for the synthetic task collection.

(b) Energy consumption per number of tasks for the classic task collection.

Figure 4. Our technique performs better on task collections with many tasks.

energy saved by switching one or several cores off. The more tasks are sequential, the more difficult it is for schedulers to eliminate idle time and the more energy saving can our technique provide. Also, the more cores are available to switch off, the more energy saving opportunities. This happens in our experimental setup and that more loose target makespan values can result in greater differences in schedules quality. In extreme cases, core consolidation reduces the energy consumption to 91% of the one by our crown scheduler without consolidation for the classic task collection, 81% for the Streamit task collection and 81% for the synthetic collection, while both schedulers could find an optimal solution withing the 5 minutes timeout.

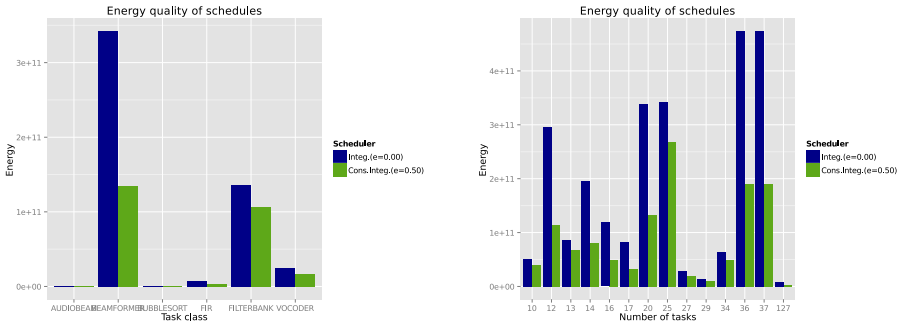
Finally, Fig. 7 shows that the core consolidation technique does not influence the optimization time of our crown scheduler.



(a) Energy consumption per number of tasks for the synthetic task collection.

(b) Energy consumption per number of tasks for the classic task collection.

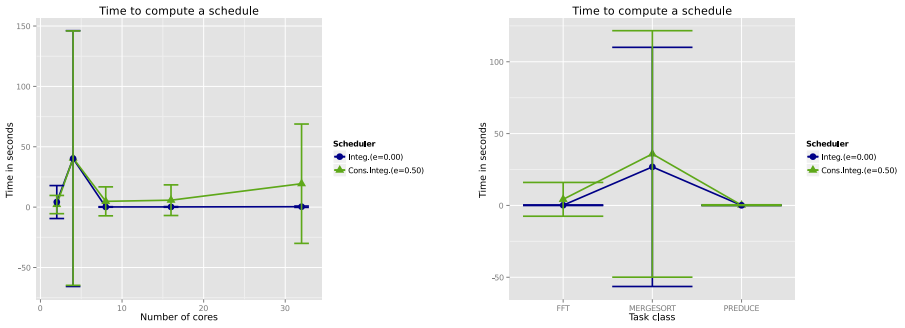
Figure 5. Our technique performs better on many cores.



(a) Energy consumption per number of tasks for the Streamit task collection.

(b) Energy consumption per number of tasks for the class task collection.

Figure 6. The Streamit task collection is mostly insensitive to our technique.



(a) Optimization time by number of cores for the classic task collection.

(b) Optimization time by number of cores for the classic task collection.

Figure 7. The additional packing constraint over our integrated crown scheduler does not affect significantly the optimization time.

4. Conclusions and Future Work

We have presented a study on how core consolidation, i.e. switching off unused cores, can be integrated in the integer linear program of a static, energy-efficient scheduler for moldable tasks. We have evaluated with a benchmark suite of synthetic and real-world task graphs that on a generic multicore architecture, about 66% of energy consumption can be saved on average and up to 91% in the most extreme case. The experiments also show that the core consolidation doesn't affect significantly the optimization of our integrated ILP crown scheduler.

In the future, we would like to integrate the overhead for frequency scaling and core switch-off/wake-up into our models, in order to explore at which level of task granularity it becomes worthwhile switch off idle cores temporarily. Also, our consolidation technique could be integrated into other schedulers not restricted by the crown constraints.

Acknowledgements

C. Kessler acknowledges partial funding by EU FP7 EXCESS and SeRC. N. Melot acknowledges partial funding by SeRC, EU FP7 EXCESS and the CUGS graduate school at Linköping University. The authors would like to thank Denis Trystram's group at INRIA Grenoble for fruitful discussions that inspired our work on this paper.

References

[1] A. P. Chandrasakaran and R. W. Brodersen. Minimizing power consumption in digital CMOS circuits. *Proc. IEEE*, 83(4):498–523, Apr. 1995.

[2] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proc. 12th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 151–162. ACM, 2006.

[3] D. Helms, E. Schmidt, and W. Nebel. Leakage in CMOS circuits — an introduction. In *Proc. PATMOS 2004*, LNCS 3254, pages 17–35. Springer, 2004.

[4] 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, 2013.

[5] N. Melot, C. Kessler, J. Keller, and P. Eitschberger. Fast Crown Scheduling Heuristics for Energy-Efficient Mapping and Scaling of Moldable Streaming Tasks on Manycore Systems. *ACM Trans. Archit. Code Optim.*, 11(4):62:1–62:24, Jan. 2015. ISSN 1544-3566.

[6] 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.

[7] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 179–196. Springer Berlin Heidelberg, 2002. doi: 10.1007/3-540-45937-5_14.