Modeling and Analyzing the Energy Consumption of Fork-Join-based Task Parallel Programs

Thomas Rauber
Computer Science Department,
Universität Bayreuth,
95440 Bayreuth, Germany

Gudula Rünger
Computer Science Department,
Technische Universität Chemnitz,
09107 Chemnitz, Germany.

to appear in: Concurrency and Computation: Practice and Experience, 2015, vol. 27, no. 1: pp. 211-236. John Wiley & Sons

Abstract

Due to environmental and monetary concerns, it is increasingly important to reduce the energy consumption in all areas, including parallel and high performance computing. In this article, we propose an approach to reduce the energy consumption needed for the execution of a set of tasks computed in parallel in a fork-join fashion. The approach consists of an analytical model for the energy consumption of a parallel computation in fork-join form on DVFS processors, a theoretical specification of an energy-optimal frequency-scaled state, and the energy minimization by computing optimal scaling factors. For larger numbers of tasks, the approach is extended by scheduling algorithms which exploit the analytical result and aim at a reduction of the energy. Energy measurements of a complex numerical method and the SPEC CPU2006 benchmarks as well as simulations for a large number of randomly generated tasks illustrate and validate the energy modeling, the minimization and the scheduling results.

1 Introduction

In parallel computing for scientific applications, the performance and a fast computation has long been the major concern. A large variety of programming and optimization techniques such as scheduling algorithms have been developed to achieve a low or even minimal parallel execution time for parallel programs. Today, also the energy efficiency is taken into account when designing parallel application software [49]. This leads to a bi-critical design goal for the software, considering both a smaller execution time and a lower energy consumption. There are several possibilities to influence the energy consumption of a program, including software programming techniques as well as new hardware capabilities of recent processors, such as dynamic voltage scaling or dynamic frequency scaling. In this article, we exploit the technique of dynamic voltage frequency scaling (DVFS) for energy savings of task-based programs.

The DVFS technique enables processors to dynamically adjust the voltage and the frequency of the processor aiming at a reduction of the power consumption. When reducing the frequency, however, the execution time usually increases, which has to be taken into account for the actual energy consumption. In order to exploit frequency scaling for calculated energy saving, the influence of the frequency used for the execution of an application program on the power consumption and the execution time is studied. More precisely, we pursue an approach to model the power and the energy consumption at the level of tasks and to minimize the energy consumption with an analytical approach, which is then extended to a scheduling approximation algorithm for assigning tasks to processors.

An effective way to structure parallel programs is the use of tasks [41, 53]. A task-based programming approach enables a structuring of a parallel program according to the needs of an application algorithm and allows the use of load balancing and scheduling techniques for an efficient exploitation of the sequential or parallel hardware platform. It has been shown that a task-based programming model can also be advantageous for restructuring programs in the context of reducing the energy consumption [37, 57]. We especially consider tasks arranged in

a fork-join pattern starting their execution at the same time and finishing at a joint barrier synchronization. This pattern can be considered to be similar to the bag of tasks problems of a set of independent tasks to be scheduled to a fixed number of cores or processors. The fork-join pattern can also be extended to larger structures consisting of nested fork-join structures, which may be used to build larger task graphs with dependencies. In this article, we concentrate on a single fork-join pattern of tasks and distinguish the following two cases: (i) the number of tasks coincides with the number of execution units, i.e. processors or cores, or (ii) the number of tasks exceeds the number of execution units.

In this article, we describe the power as well as the energy consumption of a task by a function depending on the frequency used for its execution on a single execution unit. The tasks are assumed to be non-preemptive and use the same frequency for their entire execution, thus the execution time of a task is modeled adequately without any time penalty for a change of the frequency. For the case that the number of tasks equals the number of execution units, frequency scaling factors leading to a minimal energy consumption are calculated in an analytical way. The resulting execution scheme represents an optimal solution concerning the energy consumption. Naturally, the execution time is increased with decreasing frequencies, however, there is an upper bound of the execution time given by the longest running task and the interval of the frequencies possible, so that a compromise between a good energy consumption and a good execution time can be chosen for the bi-objective optimization problem. The approach assumes tasks of similar nature being compute-intensive and less memory-intensive so that the same energy model applies to all tasks considered.

Based on the analytically optimal solution of frequency scaling for the case that the number of tasks coincides with the number of cores in the fork-join pattern, scheduling algorithms are proposed which extend the approach to numbers of tasks being larger than the number of cores. Two algorithms are presented: the first algorithm is a greedy algorithm with the makespan as objective function and a subsequent step improving the energy consumption with only a small penalty for the makespan bounded by the frequency interval. A second algorithm with the minimization of the energy consumption as objective function and without time constraint is proven to lead to the same schedule with an inherent upper bound for the time given by the dominant task with respect to the unscaled execution time. The advantage of the scheduling algorithms presented is their combination with the analytically optimal solution for the frequency to be used, which simplifies the bi-critical optimization problem for the makespan and the energy consumption to mono-criterial problems for one of the criteria and captures the other criteria with analytical methods.

In the evaluation section, we present simulation results illustrating the influence of the frequency scaling factors on the energy consumption and on the execution time for the fully parallel execution of tasks with one task per execution unit as well as for higher numbers of tasks exceeding the number of execution units, thus requiring scheduling. We also present energy measurements of a complex application for solving a system of ordinary differential equations. The measurements give rise to an actual validation of the energy consumption model on recent Intel processors providing hardware counters for measuring the energy consumption. For the validation, standard benchmarks from the SPEC CPU2006 benchmark suite are also considered and it is shown that the frequencies computed indeed lead to the smallest energy consumption. The contribution of this article is to combine analytical as well as scheduling methods in a two-step method to minimize the energy consumption by exploiting the method of frequency scaling and to accompany the theoretical results with simulations as well as experimental results.

The rest of the article is organized as follows: Section 2 describes the task-based programming model. Section 3 summarizes the energy model used for capturing the energy consumption of individual tasks based on frequency scaling factors. Section 4 derives energy consumption functions for the parallel execution of tasks as well as corresponding optimal frequency scaling factors. Section 5 proposes scheduling algorithms that also take the energy consumption into consideration for the task mapping. Section 6 presents an experimental evaluation with simulations and measurements. Section 7 discusses related work and Section 8 concludes the article.

2 Task-based Programming Model

In parallel programming, task-based programming models are popular programming abstractions that assume applications to be composable of tasks, which represent well-defined portions of calculations and are often expressed as functions or procedures. Depending on the specific programming environment, tasks may have different form

and granularity, such as statements, functions, or entire sub-programs, and they may be independent or may exchange information. This captures general task models as they are used by many modern libraries and languages, such as OpenMP [6, 7, 42], X10 [12], Fortress [3], and Chapel [11], see also [16, 18, 23] for an overview. The advantage of task-based programming models it to provide a suitable representation of the internal structure of an application which allows a flexible way to execute the application on a parallel machine by assigning the tasks to processors or cores. Programming goals, such as low execution time or high throughput, can then be achieved by load balancing or scheduling techniques.

In this article, we consider a task-based programming model in which the parallel program to be executed consists of a set of tasks that can be executed by any of the processors or cores provided by the parallel execution platform. A homogeneous parallel platform with p identical processing units (processors or cores) is considered. Several tasks can be executed in parallel, each one on a separate processor or core, and a processor can fetch the next task for execution as soon as it becomes idle. The fork-join pattern is often used to include sections of independent tasks, in which new independent tasks are created (fork) and the creating task waits until all these tasks are terminated (join). The fork-join pattern ends with an implicit barrier operation involving all tasks of a fork-join construct, so that subsequent tasks can only be started after all tasks of the preceeding fork-join pattern have been terminated. The fork-join pattern can be used to build larger coordination structures for tasks such as nested or hierarchical fork-join structures or more general task graphs.

The overall execution time of the program is calculated from the execution times of the single tasks. Assuming that the program consists of a set $\mathcal{T}$ of tasks, the overall execution time depends on the execution time of the individual tasks $T \in \mathcal{T}$ and the coordination and waiting times of the tasks. The execution time of a task $T \in \mathcal{T}$ is given as a cost function C_T depending on the task T. The cost function C_T can express different kinds of costs, such as the actual execution time on a specific hardware platform measured in seconds, a predicted time with a suitable analytical prediction technique, see [35], or relative values expressing the execution time in relation to other task costs. Usually, the execution time also depends on other parameters, such as a problem size or parameters of the parallel execution platform. When considering a specific problem instance, these other parameters can assumed to be fixed.

The execution time for the entire parallel program consisting of task set $\mathcal{T}$ is built up from the functions $C_T, T \in \mathcal{T}$, according to the structure of the tasks and the processor assignment used. Accordingly, the execution time of a single fork-join pattern $T_{\text{fork-join}}$ is described by the formula

$$C_{T_{\text{fork-join}}} = \max_{i=1,\ldots,p} C_{T_i},$$

assuming that p tasks $T_1, \ldots, T_p$ are created (fork) and that a join requires a barrier synchronization, waiting for all tasks $T_1, \ldots, T_p$ to be completed before continuing. Thus, the execution time is dominated by the execution time of the task T_m that has the longest execution time, i.e., $C_{T_m} = \max_{i=1,\ldots,p} C_{T_i}$. Processors may have waiting times, since the processors executing other tasks must wait for the completion of the dominating task T_m. For the parallel execution time, these waiting times are insignificant. However, the situation might be different when investigating the energy consumption as it is done in this article. This issue is investigated in the following section and starts with an energy model for task-based executions.

3 Frequency-scaling energy model for sequential executions

Practical as well as theoretical investigations of the energy behavior of computer systems or applications are often based on power and energy models which capture the power or energy consumption as a function of suitable parameters. In this article, we are mainly concerned with the influence of the frequency on the energy consumption and we exploit a well-accepted energy model that uses power and energy functions depending on the frequency for the sequential case. This model has already been used for embedded systems [57], for heterogeneous computing systems [37], or for shared-memory architectures [33]. We summarize and introduce the basic functions of the model in Sect. 3.1 to 3.3 and formulate a first lemma about the solution of the time-constraint optimization of the energy consumption for a sequential execution in Sect. 3.4.

3.1 Modeling power consumption

The energy model from [57] distinguishes between the dynamic power consumption and the static power consumption. For the dynamic power consumption, the formula $P_{dyn} = \alpha \cdot C_L \cdot V^2 \cdot f$ is used where α is the switching probability, C_L is the load capacitance, V is the supply voltage, and f is the operational frequency. The dynamic power consumption P_{dyn} ignores the energy consumption of memory accesses or I/O, so that this model is especially suited for non-memory intensive programs.

For DVFS processors, the frequency f can be scaled down within a predefined interval $[f_{min}, f_{max}]$ where f_{max} is the normal operational frequency that can be scaled down. The scaling can be expressed by a dimensionless scaling factor $s \geq 1$ which describes a smaller frequency $\tilde{f}$ as $\tilde{f} = f/s$. Since the operational frequency f depends linearly on the supply voltage V, i.e., $V = \beta \cdot f$ with some constant β, the dynamic power consumption decreases by a factor s^{-3} when reducing the frequency by a scaling factor $s \geq 1$. In the following, the scaling factor s is used as a parameter of the dynamic power consumption, denoted as $P_{dyn}(s)$, and $P_{dyn}(s) = s^{-3} \cdot P_{dyn}(1)$ holds where $P_{dyn}(1)$ is the power consumption for frequency f_{max}, i.e., the dynamic power consumption decreases cubically with increasing s.

For many years, the dynamic power consumption has been the predominant factor in CMOS power consumption, but for recent technologies, leakage power has an increasing impact and represents roughly 20 % of power dissipation in current processor designs [30]. Leakage power consists of several components, including sub-threshold leakage, reverse-biased-junction leakage, gate-induced-drain leakage, gate-oxide leakage, gate-current leakage, and punch-through leakage. According to [57], we make the assumption that P_{static} is constant and is independent of the scaling factor s. which is justified by the close match between the data sheet curves of DVFS processors and the analytical curves obtained by using this assumption, see [57]. Our own experiments from Sect. 6 support these results.

3.2 Energy consumption for a sequential execution of tasks

When the frequency f is scaled down using a factor s, i.e., frequency $s^{-1} \cdot f$ is used, the sequential execution time C_T of a task $T \in \mathcal{T}$ increases linearly with the scaling factor s, resulting in a larger execution time $C_T \cdot s$. Thus, the dynamic energy consumption of a task T executed on a single processor can be modeled as a function of s by:

$$E_{dyn}^T(s) \quad = P_{dyn}(s) \cdot (C_T \cdot s) = s^{-2} \cdot E_{dyn}^T(1) \tag{1}$$

with $E_{dyn}^T(1) = P_{dyn}(1) \cdot C_T$. Analogously, the static energy consumption can be modeled as:

$$E_{static}^T(s) \quad = P_{static} \cdot (C_T \cdot s) = s \cdot E_{static}(1) \tag{2}$$

with $E_{static}(1) = P_{static} \cdot C_T$. The resulting total energy consumption for the sequential execution of task T is:

$$E^T(s) \quad = E_{dyn}^T(s) + E_{static}^T(s) = (s^{-2} \cdot P_{dyn}(1) + s \cdot P_{static}) \cdot C_T \; . \tag{3}$$

3.3 Optimal scaling factor for a single task

The optimal scaling factor minimizing the energy consumption $E^T(s)$ of a task T executed sequentially is obtained by computing the minimum of the convex function

$$Q(s) := s^{-2} \cdot P_{dyn}(1) + s \cdot P_{static}, \tag{4}$$

which is

$$s_{opt} = \left(\frac{2 \cdot P_{dyn}(1)}{P_{static}} \right)^{1/3} . \tag{5}$$

Figure 1 shows $Q(s)$ for typical values of $P_{dyn}(1)$ and P_{static}, see [25] for a detailed description of power management issues. The figure shows that a larger power consumption results for (a) $s < s_{opt}$ because the dynamic

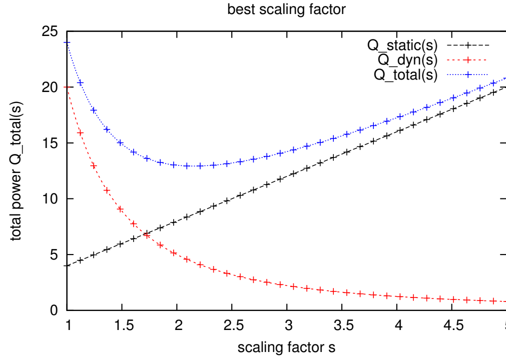

Figure 1: Optimal scaling factor for a processor with $P_{static} = 4W$ and $P_{dyn}(1) = 20W$. According to Equ. (5), $s_{opt} = 2.15$ results.

part $Q_{dyn}(s) = s^{-2} \cdot P_{dyn}(1)$ dominates and for (b) $s > s_{opt}$ because the static part $Q_{static}(s) = s \cdot P_{static}$ dominates.

In the following, we assume that all scaling factors including s_{opt} are greater than or equal to 1. According to Equ. (5), this assumption is fulfilled for s_{opt} if $P_{static} \leq 2 \cdot P_{dyn}(1)$, which is the case for current processors. Since DVFS processors only provide a fixed set of predefined frequencies, only a finite number of frequency scaling factors is available in practice, and s_{opt} has to be rounded to the neighboring scaling factor available, for which $Q(s)$ is at a minimum.

3.4 Time-constraint energy optimization

Computing the optimal scaling factor can be considered in the context of solving the bi-critical problem of minimizing the energy consumption and the execution time in the form of an energy optimization problem with time constraint. The analytical solution in Equ. (5) actually minimizes the energy consumption without assuming any constraints for the execution time. However, there is an implicit upper bound for the execution time induced by the frequency scaling interval supported by the processor. Because of the scaling factors being in the interval $[1, s_m]$ with scaling factor s_m corresponding to the smallest frequency f_{min}, the execution time of a specific task T lies in the interval $[C_T, s_m C_T]$, providing the inherent upper bound $s_m C_T$ for the execution time. If the bi-critical optimization problem has to be solved with a time constraint C_T^{ctr}, then one of the following four cases applies: (i) if $C_T^{ctr} \geq s_m C_T$ then the constraint is inherently fulfilled; (ii) if $s_{opt} C_T \leq C_T^{ctr} \leq s_m C_T$ then the time constraint is fulfilled for the optimal scaling factor; (iii) if $C_T \leq C_T^{ctr} \leq s_{opt} C_T$, then a suboptimal energy solution with $\bar{s} \leq s_{opt}$ and $C_T^{ctr} = \bar{s} \cdot C_T$ can be chosen to get the minimal solution of the energy consumption with time constraint; (iv) the last case of a time constraint $C_T^{ctr} \leq C_T$ would not define a feasible problem. This gives rise to the following result:

Lemma 1 (Time-constraint minimization of the energy consumption) *The energy consumption minimization problem under time constraint C_T^{ctr} for a task T with execution time C_T in the unscaled case executed sequentially on a processor providing frequency scaling is solved by $\tilde{s} = \min(\bar{s}, s_{opt})$ with $\bar{s} \cdot C_T = C_T^{ctr}$ and $\bar{s} \geq 1$.*

4 Frequency scaling for concurrent tasks

In this section, we investigate the energy consumption in the case that separate scaling factors are used for the concurrent execution of tasks. Section 4.1 shows that in order to obtain a minimum energy consumption, the scaling factors for the different processors should be selected such that no idle times occur. Section 4.2 computes the resulting energy consumption for two concurrently executed tasks, and Section 4.3 generalizes the result to an arbitrary number of concurrent tasks.

4.1 Necessary condition for energy optimal scaling factors

We consider the situation that $n \geq 2$ tasks $T_1, \ldots, T_n$ are executed in parallel on n different processors $P_1, \ldots, P_n$, each processor executing one task. The processors may use different frequency scaling factors $s_1, \ldots, s_n$ for the execution of their corresponding tasks. According to the fork-join semantics, all processors are busy for the same amount of time before they can start another computation. The execution time for processor P_i consists of the execution time for the task T_i and the idle time I_i elapsed before all other processors finish their execution. Thus, the time for processor P_i is

$$C_{P_i}(s_i) = C_{T_i} \cdot s_i + I_i \tag{6}$$

The energy optimal execution is achieved by a set of frequency scaling factors ${s_1}^*, \ldots, {s_n}^*$ for which the overall energy consumption

$$E^{T_1\|\ldots\|T_n}({s_1}^*, \ldots, {s_n}^*) = \sum_{i=1}^{n}(C_{T_i} \cdot Q(s_i) + I_i \cdot P_{static}) \tag{7}$$

is minimized. It is now shown that such an optimal solution has the property $I_i = 0$ for $i = 1, \ldots, n$.

Theorem 1 (Necessary condition for energy-optimal frequency scaling) *For an energy-optimal solution for n tasks $T_1, \ldots, T_n$ the scaling factors ${s_1}^*, \ldots, {s_n}^*$ with ${s_i}^* \geq 1$ for $i = 1, \ldots, n$ must be selected such that $C_{T_i} \cdot {s_i}^* = C_{T_j} \cdot {s_j}^*$ holds for all pairs of tasks $T_i, T_j, i, j \in \{1, \ldots, n\}$, where C_{T_i} denotes the unscaled sequential execution time of task T_i, $i = 1, \ldots, n$.*

Proof: The tasks $T_i, i = 1, \ldots, n$, are assumed to be ordered in decreasing order of their execution time, i.e., $C_{T_1} \geq \cdots \geq C_{T_n}$. Let ${s_1}^*, \ldots, {s_n}^*$ be the scaling factors of an energy-optimal execution of the tasks $T_1, \ldots, T_n$ on n processors. Let C_{opt} be the execution time of the energy-optimal execution and E_{opt} the corresponding energy consumption. Then, there exists at least one task $T_j, j \in \{1, \ldots, n\}$ with $C_{T_j} \cdot {s_j}^* = C_{opt}$. (Otherwise, all tasks would have an idle time with a corresponding static power consumption, which could be avoided, and thus E_{opt} would not be optimal.) If all other processors P_j also fulfill $C_{T_j} \cdot {s_j}^* = C_{opt}$ for their corresponding task T_j, the result is given.

Otherwise, we assume that there exists at least one task T_i with $C_{opt} > C_{T_i} \cdot {s_i}^*$ and idle time ${I_i}^* > 0$. The corresponding processor P_i uses static power during its idle time ${I_i}^* = C_{opt} - C_{T_i} \cdot {s_i}^* > 0$, leading to an idle power consumption of $(C_{opt} - C_{T_i} \cdot {s_i}^*) \cdot P_{static}$. Now consider the scaling factor $\tilde{s}_i \neq {s_i}^*$ with $C_{opt} - C_{T_i} \cdot \tilde{s}_i = 0$. For this scaling factor $\tilde{s}_i = C_{opt}/C_{T_i}$, the energy consumption of processor P_i would be

$$E_{P_i}(\tilde{s}_i) = (\tilde{s}_i^{-2} \cdot P_{dyn}(1) + \tilde{s}_i \cdot P_{static}) \cdot C_{T_i}$$

and corresponds to $E^{T_i}(\tilde{s}_i)$, since $I_i = 0$. It is $\tilde{s}_i > {s_i}^*$ because of $C_{T_i} \cdot {s_i}^* < C_{opt} = C_{T_i} \cdot \tilde{s}_i$. For the resulting energy consumption of processor P_i with scaling factor $\tilde{s}_i$ we get the following estimation:

$$\begin{aligned}
&E_{P_i}(\tilde{s}_i) \\
&= \tilde{s}_i^{-2} \cdot P_{dyn}(1) \cdot C_{T_i} + \tilde{s}_i \cdot P_{static} \cdot C_{T_i} \\
&< {s_i}^{*-2} \cdot P_{dyn}(1) \cdot C_{T_i} + \tilde{s}_i \cdot P_{static} \cdot C_{T_i}, \text{ since } \tilde{s}_i > {s_i}^* \\
&= {s_i}^{*-2} \cdot P_{dyn}(1) \cdot C_{T_i} + ({s_i}^* + (\tilde{s}_i - {s_i}^*)) \cdot P_{static} \cdot C_{T_i} \\
&= E^{T_i}({s_i}^*) + (\tilde{s}_i - {s_i}^*) \cdot P_{static} \cdot C_{T_i} \\
&= E_{P_i}({s_i}^*)
\end{aligned}$$

since ${I_i}^* = (\tilde{s}_i - {s_i}^*) \cdot C_{T_i}$. This shows that using scaling factor $\tilde{s}_i$ for P_i would lead to a smaller energy consumption of P_i than using ${s_i}^*$. This contradicts the assumption that $I_i > 0$ for an energy-optimal ${s_i}^*$ is possible. Thus, $C_{T_j} \cdot {s_j}^* = C_{opt} = C_{T_i} \cdot {s_i}^*$. $\square$

Theorem 1 shows that the energy consumption of a fork-join construct can be reduced by avoiding waiting times at the barrier using an appropriate scaling of the waiting processors. Theorem 1 states a necessary condition that must be fulfilled by a solution to be optimal. In the following, this necessary condition is exploited to derive an optimal solution. This is first done for two concurrent tasks in Sect. 4.2 and then for an arbitrary number of concurrent tasks in Sect. 4.3.

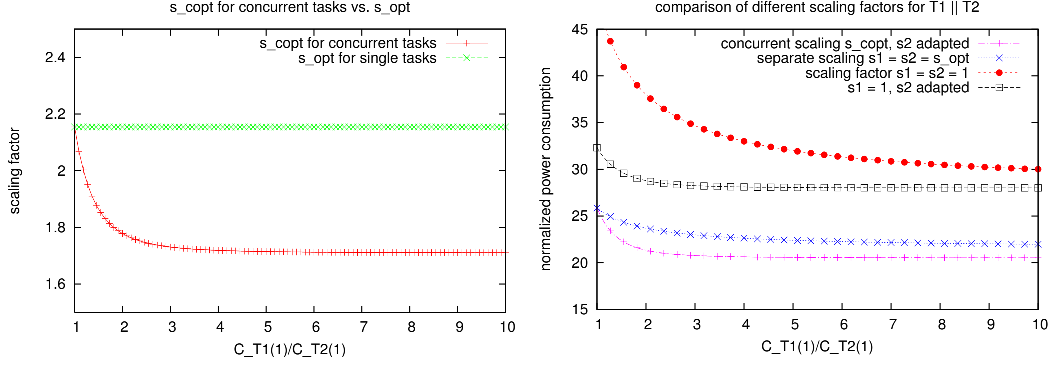

Figure 2: Left: Comparison of the optimal scaling factor s_{opt} from Equ. (5) and the scaling factor s_{copt} from Equ. (11) in dependence of C_{T_1}/C_{T_2} for an example processor with $P_{static} = 4W$ and $P_{dyn}(1) = 20W$. Right: Comparison of the normalized energy consumption $E^{T_1\|T_2}(s_1, s_2)/C_{T_1}$ of two concurrent tasks using different combinations of scaling factors for the same example processor.

4.2 Optimal scaling factor for two concurrently executed tasks

For two concurrent tasks T_1 and T_2, i.e., $n = 2$, the scaling factors s_1 and s_2 have to fulfill $s_1 \cdot C_{T_1} - s_2 \cdot C_{T_2} = 0$ according to Theorem 1. Thus,

$$s_2 = s_1 \cdot \frac{C_{T_1}}{C_{T_2}}. \tag{8}$$

Using Equ. (8) in the energy consumption equation for $T_1||T_2$

$$E^{T_1\|T_2}(s_1, s_2) = (s_1^{-3} \cdot P_{dyn}(1) + P_{static}) \cdot s_1 \cdot C_{T_1} + (s_2^{-3} \cdot P_{dyn}(1) + P_{static}) \cdot s_2 \cdot C_{T_2} \tag{9}$$

leads to

$$E^{T_1\|T_2}(s_1, s_1 \cdot \frac{C_{T_1}}{C_{T_2}}) = s_1^{-2} \cdot P_{dyn}(1) \left(C_{T_1} + \frac{C_{T_2}^3}{C_{T_1}^2} \right) + 2s_1 \cdot P_{static} \cdot C_{T_1}. \tag{10}$$

The energy function (10) is differentiable and has a minimum at

$$s_{copt} = \sqrt[3]{\frac{P_{dyn}(1)}{P_{static}} \left(1 + \frac{C_{T_2}^3}{C_{T_1}^3} \right)} \tag{11}$$

which is computed by setting the derivative

$$\frac{d}{ds_1} E^{T_1\|T_2} \left(s_1, s_1 \cdot \frac{C_{T_1}}{C_{T_2}} \right) = -2s_1^{-3} P_{dyn}(1) \left(C_{T_1} + \frac{C_{T_2}^3}{C_{T_1}^2} \right) + 2P_{static} \cdot C_{T_1}$$

to zero. Figure 2 (left) shows scaling factor s_{copt} from Equ. (11) for varying values of C_{T_1}/C_{T_2} and compares it with s_{opt} from Equ. (5). Equation (11) is a generalization of s_{opt} in Equ. (5), and for $C_{T_1} = C_{T_2}$, Equ. (11) simplifies to Equ. (5).

Figure 2 (right) shows the normalized energy consumption for $T_1||T_2$ according to Equ. (9) for different scaling factors s_1 and s_2. Again, no specific values are assumed for C_{T_1} and C_{T_2}, and the x-axis depicts different relative sizes of C_{T_1} compared to C_{T_2}. The energy consumption shown is normalized with respect to C_{T_1}, i.e., the values $E^{T_1\|T_2}(s_1, s_2)/C_{T_1}$ are depicted. Four different combinations of scaling factors are compared: (i) concurrent scaling with s_{copt} according to Equ. (11) and s_2 chosen according to Equ. (8); (ii) scaling with $s_1 = s_2 = s_{opt}$ according to Equ. (5); (iii) no scaling, i.e., $s_1 = s_2 = 1$, and (iv) no scaling for s_1 and s_2 adapted according to Equ. (8). The figure illustrates that the smallest amount of energy is consumed for the scaling factors according to Equ. (11) and (8). The resulting energy consumption is smaller than the energy consumption resulting when using the optimal scaling factor s_{opt} for both tasks. The reason is that s_{opt} ignores the influence of the waiting time. For $C_{T_1} = C_{T_2}$, the cases (i) and (ii) result in the same energy consumption, since $s_{copt} = s_{opt}$ and $s_2 = s_{opt}$ holds in that case. The other two cases lead to a much larger energy consumptions.

4.3 Arbitrary numbers of tasks

The result obtained for choosing the optimal scaling factors for two concurrently executed sequential tasks is now generalized to an arbitrary number n of tasks. Let $T_1, \ldots, T_n$ be a set of independent tasks that have been generated by a fork statement and that are executed concurrently on n processors. We assume that the tasks are ordered in decreasing order of their sequential unscaled execution time C_{T_i}, $i = 1, \ldots, n$, i.e., T_1 is the task with the largest execution time. According to Equ. (8), the scaling factor for each task $T_i \in \{T_2, \ldots, T_n\}$ is set to

$$s_i = s_1 \cdot \frac{C_{T_1}}{C_{T_i}} \tag{12}$$

to get an optimal energy result. Using Equ. (12) for s_i results in the following total energy consumption:

$$E^{T_1 \| \cdots \| T_n}(s_1, \ldots, s_n) = s_1^{-2} \cdot P_{dyn}(1) \left(C_{T_1} + \sum_{i=2}^{n} \frac{C_{T_i}^3}{C_{T_1}^2} \right) \; + \; n \cdot s_1 \cdot P_{static} \cdot C_{T_1} \; . \tag{13}$$

To compute the minimum, the derivative

$$\frac{d}{ds_1} E^{T_1 \| \cdots \| T_n} (s_1, \ldots, s_n) = -2 s_1^{-3} P_{dyn}(1) \left(C_{T_1} + \sum_{i=2}^{n} \frac{C_{T_i}^3}{C_{T_1}^2} \right) + n \cdot P_{static} \cdot C_{T_1}$$

is considered and set to zero. This yields that $E^{T_1 \| \cdots \| T_n}$ is minimized, if the scaling factor

$$s_{copt}(n) = \sqrt[3]{\frac{2}{n} \frac{P_{dyn}(1)}{P_{static}} \left(1 + \sum_{i=2}^{n} \frac{C_{T_i}^3}{C_{T_1}^3} \right)} \tag{14}$$

is used for s_1. The scaling factors s_i, $i = 2, \ldots, n$, are then determined according to Equ. (12). If all tasks $T_1, \ldots, T_n$ have the same execution time, $s_{copt}(n)$ from Equ. (14) simplifies to s_{opt} from Equ. (5). Depending on the distribution of the task execution times and the values of $P_{dyn}(1)$ and P_{static}, a value smaller than 1 may result for $s_{copt}(n)$ from Equ. (14). In that case, the value has to be rounded up to 1 in order to fulfill the constraints for the scaling factors.

The choice of the scaling factor $s_{copt}(n)$ according to Equ. (14) for the task with the largest execution time in the unscaled state and with the scaling factors according to Equ. (12) for the other tasks minimizes the energy consumption for the case $n = p$, i.e. the number n of tasks is exactly the same as the number p of processors. This means that the largest task determines the overall execution time $s_{copt}(n) \cdot C_{T_1}$, which also represents an implicit time bound. Due to the choice of the scaling factors for the other $n - 1$ tasks, their scaled execution time is just the same as $s_{copt}(n) \cdot C_{T_1}$. If a time constraint C_T^{ctr} is explicitly given, the same procedure can be applied as described at the end of Sect. 3, however now using $s_{copt}(n)$ instead of s_{opt}, i.e., $\min(\bar{s}, s_{copt})$ with $C_T^{ctr} = \bar{s} C_{T_1}$ is chosen as scaling factor.

For the case that the number n of tasks is larger than the number of processors p, i.e. $n > p$, the situation might change, since some of the processors now execute more than one task. In the next section, we propose a scheduling procedure for the case $n \geq p$, which extends the solution of the minimization of the energy consumption that we have presented so far.

5 Energy-based Scheduling Algorithms for Tasks

This section addresses the energy-aware scheduling of n independent tasks $T_1, \ldots, T_n$ that are created by a fork-join construct and are to be computed on p processors $\mathcal{P} = \{P_1, \ldots, P_p\}$ with $n \geq p$. Two scenarios are considered. In the first scenario, a conventional scheduling algorithm based on the execution time as objective function is used and it is shown how the scaling factor s_{copt} from Equ. (14) can be applied to derive a schedule with a smaller energy consumption. In the second scenario, a new scheduling algorithm is proposed that uses the energy consumption as objective function.

ALGORITHM 1: Time-based scheduling algorithm.

```
1 begin
2   Sort tasks {T_1, ..., T_n} such that C_{T_1} ≥ C_{T_2} ≥ ... ≥ C_{T_n};
3   for (j = 1, ..., n) do
4     Assign T_j to processor P_i with T_{[1:j-1]}(P_i) minimal;
5   Compute M(1) = max_{1≤i≤p} T_{[1:n]}(P_i);
6   Let P_e be the processor with T_{[1:n]}(P_e) = M(1);
7   Compute s_copt(P_e) according to Equ. (15);
8   Compute scaling factors for P_i ≠ P_e according to Equ. (16);
```

5.1 Time-based scheduling with energy improvement

The tasks $T_1, \ldots, T_n$ are assigned to the processors with a greedy scheduling algorithm [45, 48] that uses the minimization of the execution time as objective function. The tasks are ordered according to their unscaled execution time in decreasing order, i.e., we assume $C_{T_1} \geq C_{T_2} \geq \ldots \geq C_{T_n}$. The assignment of a task T_k to a processor $P_i \in \mathcal{P}$ is denoted by $A(k) = P_i$. The greedy scheduling is based on the accumulated execution times of the processors $P \in \mathcal{P}$: when the tasks $T_1, \ldots, T_j$ have already been assigned to processors, the accumulated execution time $T_{[1:j]}(P)$ of a processor $P \in \mathcal{P}$ is defined as

$$T_{[1:j]}(P) = \sum_{\substack{1 \leq k \leq j \\ A(k) = P}} C_{T_k} \,.$$

The greedy scheduling algorithm assigns the tasks in decreasing order of their execution time one after another, see Algorithm 1. The next task T_k is assigned to the processor $P_i \in \mathcal{P}$ for which the accumulated time $T_{[1:k-1]}(P_i)$ is smaller than all other accumulated times $T_{[1:k-1]}(P_j)$ for $j \neq i$. The resulting overall execution time without scaling, i.e., for scaling factor $s = 1$, is denoted as

$$M(1) = \max_{1 \leq i \leq p} T_{[1:n]}(P_i).$$

The time $M(1)$ is determined by the processor P_e that finishes its work last. An experimental comparison of this greedy scheduling algorithm with other scheduling algorithms using the overall execution time (makespan) as objective function is included in [17, 36].

From an energy-consumption perspective, the resulting schedule can be improved by applying the results from Sect. 4:

1. For processor P_e, the scaling factor

$$s_{copt}(p) = \sqrt[3]{\frac{2}{p} \frac{P_{dyn}(1)}{P_{static}} \left(1 + \sum_{i=1, i \neq l}^{n} \frac{T_{[1:n]}(P_i)}{T_{[1:n]}(P_e)}\right)}, \tag{15}$$

   is chosen, which is a straightforward generalization of Equ. (14). This leads to an overall execution time $M(s_{copt}) = s_{copt} \cdot M(1)$.

2. For all other processors $P_i \neq P_e$, the idle time $T_{[1:n]}(P_e) - T_{[1:n]}(P_i)$ can be avoided by using a scaling factor

$$s_i = s_{copt} \frac{T_{[1:n]}(P_e)}{T_{[1:n]}(P_i)}, \tag{16}$$

   which is a generalization of Equ. (12). By using the scaling factors (16), the resulting energy consumption is reduced without further increasing the overall execution time $M(s_{copt})$.

ALGORITHM 2: Energy-based scheduling algorithm.

```
1 begin
2     Sort tasks {T_1, ..., T_n} such that E^{T_1}(s_opt) ≥ ... ≥ E^{T_n}(s_opt);
3     for (k = 1, ..., n) do
4         Assign T_k to processor P_i with E_[1:k-1](P_i) minimal;
5     Compute E = Σ_{i=1}^{p} E_[1:n](P_i);
6     Compute M(s_opt) = max_{1≤i≤p} T_[1:n](P_i)
7     let P_e be the processor with T_[1:n](P_e) = M(s_opt);
8     Compute s_copt(P_e) according to Equ. (15);
9     Compute scaling factors for P_i ≠ P_e according to Equ. (16);
```

The greedy scheduling Algorithm 1 uses the makespan as objective function [48] and has a worst-case suboptimality bound of $4/3 - 1/(3p)$ [20]. This result holds for Algorithm 1 in the unscaled case. Using scaling factor s_{copt} for the longest-running processor P_e increases the execution time of P_e by a factor of s_{copt}. Using scaling factor s_i according to Equ. (16) for the rest of the processors $P_i \neq P_e$ does not increase the execution time. Thus, Algorithm 1 leads to a suboptimality bound of $(3/4-1/(3p)) \cdot s_{copt}$ for the makespan in the scaled version. Again, if a time constraint C^{ctr} is given for the resulting schedule, the scaling factor $\min(\bar{s}, s_{copt})$ with $C^{ctr} = \bar{s} \cdot M(1)$ is used as scaling factor for P_e, see Subsect. 4.3.

If more than one task is assigned to a processor, Algorithm 1 executes all its tasks assigned with the same frequency. In [15, 39] it has been shown that this indeed leads to the smallest dynamic energy consumption, i.e., the energy consumption of a single processor cannot be reduced by using different frequencies for the different tasks of this processor.

5.2 Scheduling based on power consumption

The approach in the last subsection modifies an existing schedule that has been generated by a scheduling algorithm using the execution time as objective function. An alternative approach is to use the minimization of the energy consumption as objective function, which is considered in this subsection. The energy consumption of a task according to Equ. (3) with s_{opt} according to Equ. (5) is used to define an accumulated energy consumption of a processor $P_i \in \mathcal{P}$ which executes several tasks one after another:

$$E_{[1:j]}(P_i) = \sum_{\substack{1 \leq k \leq j \\ A(k) = P_i}} E^{T_k}(s_{opt}).$$

The minimization of the accumulated energy consumption is now used as objective function in a modified greedy scheduling algorithm. This algorithm assigns the tasks to processors in decreasing order of their energy consumption: Task T_k is assigned to the processor $P_i \in \mathcal{P}$, denoted as $A(k) = P_i$, for which $E_{[1:k-1]}(P_i)$ is smaller than all $E_{[1:k-1]}(P_j)$, $i \neq j$. The algorithm is given in Algorithm 2.

The resulting overall energy consumption is the accumulated energy consumption over all processors, i.e.,

$$E = \sum_{i=1}^{p} E_{[1:n]}(P_i).$$

The resulting processor assignment $A(k)$ for $k = 1, \ldots, n$ leads to an accumulated execution time

$$\tilde{T}_{[1:n]}(P) = \sum_{\substack{1 \leq k \leq n \\ A(k) = P}} C_{T_k}(s_{opt})$$

for each processor $P \in \mathcal{P}$, yielding an overall execution time

$$M(s_{opt}) = \max_{1 \leq i \leq p} \tilde{T}_{[1:n]}(P)$$

Theorem 2 (schedules produced by Algorithms 1 and 2) *The greedy energy-based scheduling Algorithm 2 yields the same processor assignment as the greedy scheduling Algorithm 1 based on the execution time. Moreover, the same scaling factors are computed for the individual processors.*

Proof: When assigning the tasks, the same scaling factor s_{opt} is used for each task. Also, s_{opt} is independent of the execution times of the individual tasks. Thus, for each task T, $E^T(s_{opt})$ is a multiple of the unscaled execution time C_T of task T, and the same multiplication factor s_{opt} is used for each task. In consequence, in each step of the processor assignment algorithm the accumulated energy consumption values are multiples of the accumulated execution times. Therefore, in each step Algorithm 2 chooses the same processor assignment as Algorithm 1. The scaling factors computed after the processor assignment are therefore also the same in both algorithms. □

The execution time of the scheduling Algorithms 1 and 2 is dominated by the time to sort the tasks in the order of decreasing runtime or energy consumption, which is $O(n \cdot \log n)$. After the sorting, the tasks are assigned in turn to the processors. When keeping the processors sorted according to their accumulated execution time or energy consumption, determining the new processor with the currently smallest execution time or energy consumption can be done in time $O(\log p)$ in each of the n steps. Hence, the overall execution time of both scheduling algorithms is $O(n \cdot (\log n + \log p))$.

5.3 Integration into a dynamic execution environment

The scheduling Algorithms 1 and 2 are based on the assumption that the tasks and their execution times are known before starting the scheduling. Although this situation is not given in the general case, many applications from scientific computing are amenable for this kind of scheduling because of their regular structure of computations. For these regular applications, two different methodologies for providing task execution times can be identified.

First, for many applications from scientific computing, the number of computations to be performed by each task can be estimated a priori because of their regular computation structure. Based on such an estimation, the corresponding execution time of the tasks can be predicted using a suitable prediction model [31, 34, 54]. If such a prediction model is not available, the tasks can be ordered according to the amount of computations performed. Interpreting these computations as a virtual time unit, the scheduling algorithms can be based on these virtual time units, and for each task a corresponding virtual energy consumption can be computed. This approach is justified, since the scheduling algorithms require only a relative ordering of the tasks.

A second methodology for providing task execution times can be suitable for time-stepping methods. Time-stepping methods iteratively compute an approximation by applying the same computations to more and more precise intermediate results. An example is the extrapolation method for solving ordinary differential equations that is used in our experimental evaluation, see Sect. 6.4. For such applications, the task execution times can be measured during the first time steps and these measured execution times can then be used to perform the scheduling of the tasks for the remaining time steps.

6 Experimental evaluation

The experimental evaluation of the effects of frequency scaling includes simulations as well as actual energy measurements. In Subsect. 6.1, simulations based on randomly generated task sets illustrate the quantitative effects of the analytically computed optimal scaling factors. Subsection 6.2 studies the effect of the scheduling algorithms from Section 5 for different task scenarios. Subsection 6.3 describes the experimental setup for the energy measurements to be used in the subsequent subsection for a solver of ordinary differential equations, see Subsect. 6.4, and the SPEC CPU2006 benchmarks, see Subsect. 6.5.

6.1 Scaling factor variations

For the scaling factor experiments, the task sets are generated using different distribution functions (uniform distribution, Gaussian tail distribution, Rayleigh distribution, and Beta distribution), which select the task execution

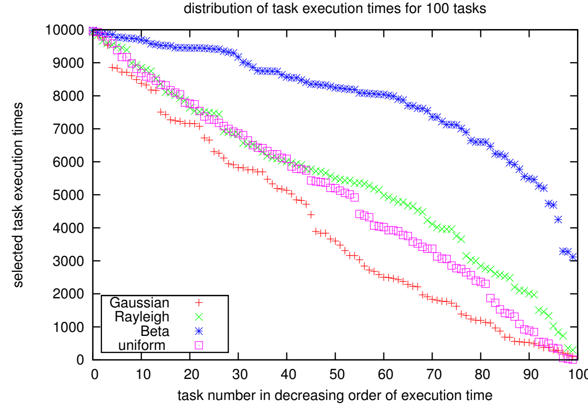

Figure 3: Distribution of the task execution times for a task set with 100 tasks, selecting the task execution times between 1s and 10000s using different distribution functions.

times randomly between 1 sec and 10000 sec. To generate the distributions the GNU Scientific Library has been used. As example, Fig. 3 shows the randomly selected task execution times for a task set of 100 tasks using the different distributions. It can be seen that the Beta distribution tends to select larger task execution times, whereas the Gaussian tail distribution favors smaller task execution times. The Gaussian tail distribution results by cutting off the Gaussian distribution at $x = 0$ and considering the resulting right part of the distribution function. For the Gaussian and Rayleigh distribution, the versions with $\sigma = 1$ are used. For the Beta distribution, the version $a = 4$ and $b = 1$ is used.

Figure 4 (left) compares the percentage energy consumption of scaled systems with respect to the unscaled system. The experiment has been performed for the following numbers of processors: $p = 10$, $p = 100$, $p = 1000$, and $p = 10000$. Each processor is assumed to execute one task. The tasks are generated with randomly selected execution times between 1 and 10000 seconds, leading to a uniform distribution of the task execution times. Each experiment has been repeated 50 times to balance extreme situations caused by the random task creation. The energy consumption has been computed according to Equ. (13) for an example processor with $P_{static} = 4W$ and $P_{dyn}(1) = 20W$. The following six choices of scaling factors for the different processors are compared (from left to right in the diagram): (a) scaling factor $s = 1$ is used for all processors; (b) scaling factor $s = s_{opt}$ according to Equ. (5) is used for all processors; (c) scaling factor $s = s_{copt}$ according to Equ. (14) is used for all processors; (d) $s = 1$ is used for the processor with the largest execution time and Equ. (12) is used to adapt the remaining scaling factors; (e) s_{opt} according to Equ. (5) is used for the processor with the largest execution time and Equ. (12) is used to adapt the remaining scaling factors; (f) s_{copt} according to Equ. (14) is used for the processor with the largest execution time and Equ. (12) is used to adapt the remaining scaling factors.

Figure 4 shows that the energy consumption can be considerably reduced by using the frequency scaling factors s_{opt} or s_{copt} instead of $s = 1$. In particular, the scaling factor adaptation can be applied to reduce the total energy consumption significantly. Using s_{copt} for the processor with the largest execution time and adapted scaling factors for the remaining processors leads to the smallest total energy consumption for all numbers of processors. The resulting energy consumption lies below 60% of the energy consumption resulting for the unscaled case. Figure 4 (right) compares the choices of frequency scaling factors for a Gaussian tail distribution of task execution times, which has a higher percentage of smaller tasks. For this task distribution, the scaling factor s_{copt} is close to 1, and therefore the adaptive scaling with 1 and s_{copt} lead to similar energy consumptions. Figure 5 (left) uses a Rayleigh distribution for the task execution times. In this case, the non-adapted scaling versions for s_{opt} and s_{copt} already lead to significant energy savings. Figure 5 (right) uses a Beta distribution having a higher percentage of tasks with a larger execution time. In this case, the energy consumption for s_{opt} and s_{copt} in the adapted and non-adapted case are quite similar. Using $s = 1$, the energy consumption is much larger also in the adapted case.

Figure 6 compares the execution times of task sets and their resulting energy consumption for different numbers of processors ($p = 10$, $p = 100$, $p = 1000$, $p = 10000$). Again, 50 sets of randomly generated tasks have been used for the experiments, and the figure shows the average information. Each task in each task set has a

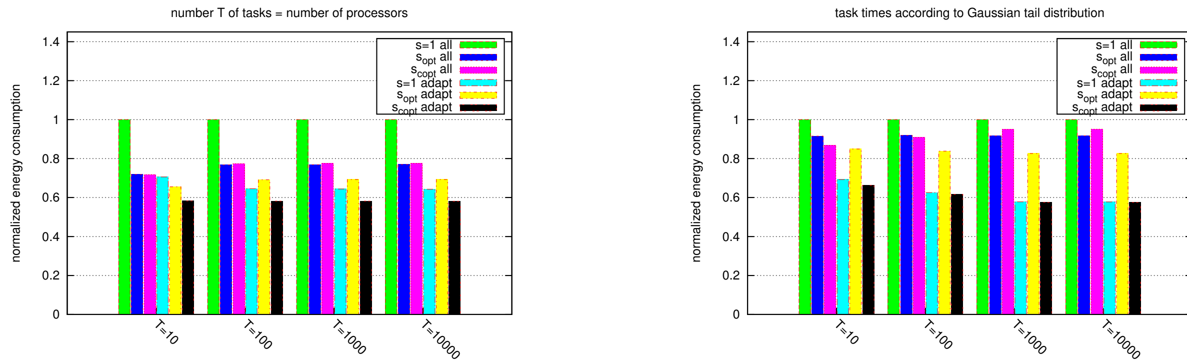

Figure 4: Normalized energy consumption with task execution time created according to a uniform distribution (left) and a Gaussian tail distribution (right). Percentage energy consumption of scaled systems with respect to the unscaled system.

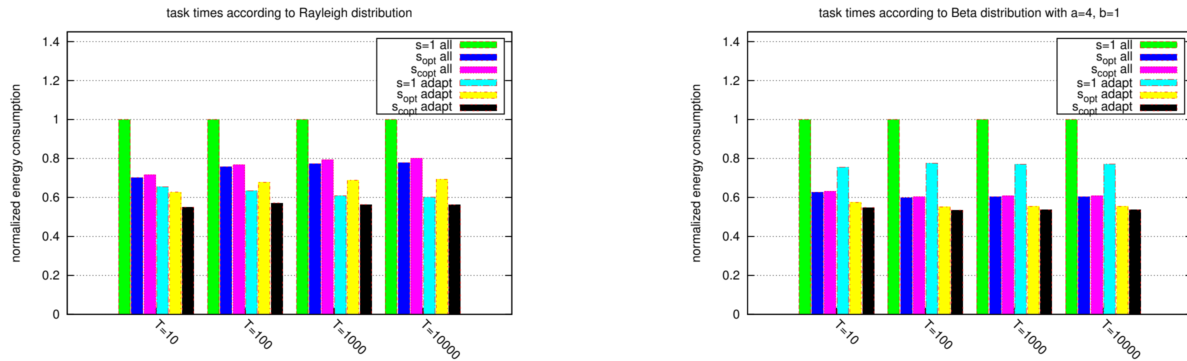

Figure 5: Normalized energy consumption with task execution time created according to a Rayleigh distribution (left) and a Beta distribution (right). Percentage energy consumption.

randomly generated execution time between 1 and 10000 seconds using a uniform distribution. For each task set, different scaling factors, the resulting execution times and energy consumptions are computed. The same scaling versions as in Fig. 4 are compared. From the figures the following observations can be made: As expected, using scaling factor 1 always results in the smallest execution time. However, adapting the scaling factors of the other processors can significantly reduce the energy consumption without negatively affecting the overall execution time. Using $s = s_{copt}$ for the processor with the largest execution time leads to a smaller energy consumption than using $s = 1$, but it also increases the resulting execution time accordingly.

Figure 7 depicts the resulting execution times and energy consumptions for different distributions of the task execution times and different choices of frequency scaling factors for each of these distributions. For $p = 1000$ processors, random task creations according to a uniform, a Gaussian, a Rayleigh, and a Beta distribution are compared. The figure shows that the increase in execution time for the scaled versions compared to the unscaled version strongly depends on the distribution of the task execution time. This increase is small for a Gaussian distribution, tolerable for a uniform and Rayleigh distribution, and large for a Beta distribution. Using the adapted scaling versions always leads to a considerable reduction in energy consumption. The reduction is largest when using $s = 1$ or $s = s_{copt}$ for the largest task and adapting the remaining scaling factors accordingly. The experiments have shown that the effect of choosing frequency scaling factors strongly depends on the distribution of the execution times of the tasks to be executed.

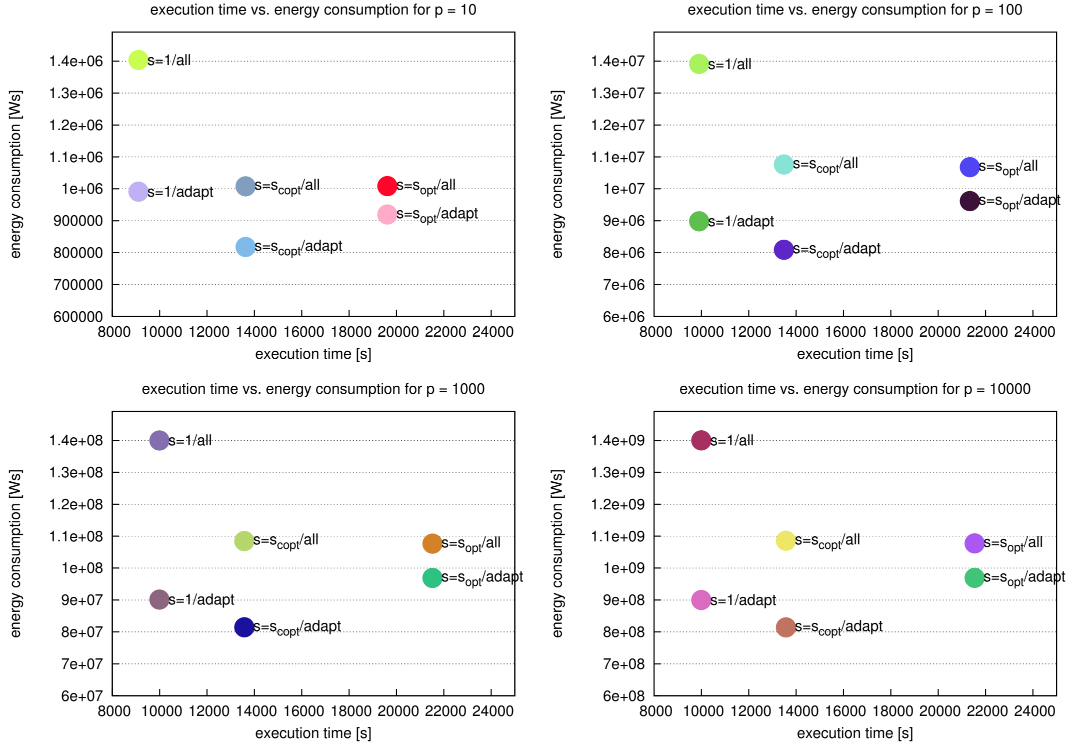

Figure 6: Execution time vs energy consumption for different numbers of processors using a uniform distribution of the task execution times.

6.2 Scheduling experiments

If there are more tasks available in a fork-join construct than there are processors, the scheduling Algorithms 1 or 2 can be applied. As stated by Theorem 2, both algorithms produce the same schedule. In the following, we describe scheduling experiments for randomly generated task sets and consider the resulting energy consumption. For a comparison with other scheduling algorithms using the makespan as objective function, we refer to [17, 36].

Figure 8 illustrates the result of the scheduling algorithms for the scheduling of 20 (left), 50 (middle), and 100 (right) tasks executed on 10 processors. The task execution times have been randomly selected between 0 ms and 10 ms using a Gaussian tail distribution with $\sigma = 1$. Other distributions lead to similar figures. From the figure it can be seen that a large number of tasks compared to the number of processors leads to a quite even distribution of the work between the processors such that the processors finish their execution at about the same time, see the schedules for 50 and 100 tasks. In this case, the waiting times of the processors in the fork-join construct are small and there is only a small potential for saving energy by avoiding the waiting times. However, the potential of saving energy by using an optimal scaling factor for the longest-running processor remains. Larger deviations in the finishing times of the processors can be observed, if the number of tasks is small compared to the number of processors (Fig. 8 (left)) or if there are large differences in the task execution times. In the latter case, it can happen that one processor has a long-running task and the execution of this task takes longer than the execution of the tasks assigned to the other processors, resulting in a large difference of the finishing times. In this case, avoiding waiting times by scaling factor adaptations can lead to significant energy savings.

Fig. 9 shows the energy consumption resulting when scheduling 100 tasks on 10 processors using different scaling factors. The task execution times are randomly selected between 0 ms and 10 ms using different distributions (uniform distribution, Gaussian tail distribution, Rayleigh distribution, Beta distribution) with the same parameters as described above. Each scheduling experiment has been repeated 10 times. Since there is only a small difference in the resulting finishing times of the processors, an adaptation of the remaining scaling factors only leads to a marginal additional energy saving. This would be different if there were larger variations in the task execution

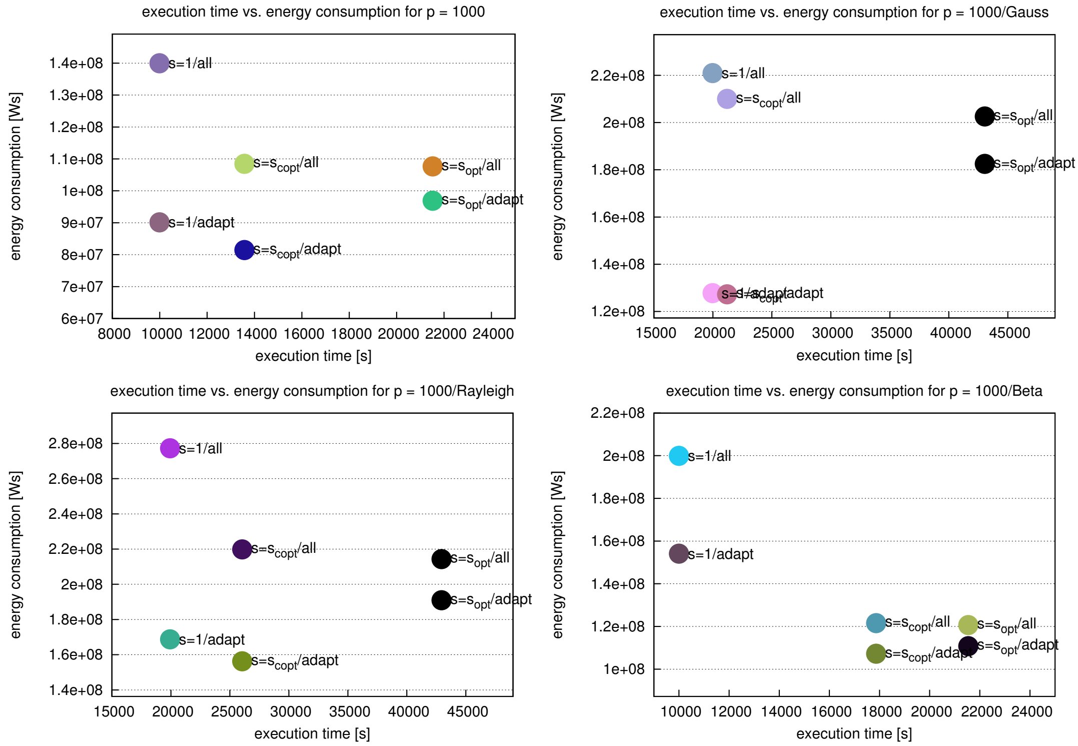

Figure 7: Execution time vs energy consumption for a fixed number of processors ($p = 1000$) using different distributions of the task execution times: uniform (top left), Gaussian tail (top right), Rayleigh (bottom left), and Beta (bottom right).

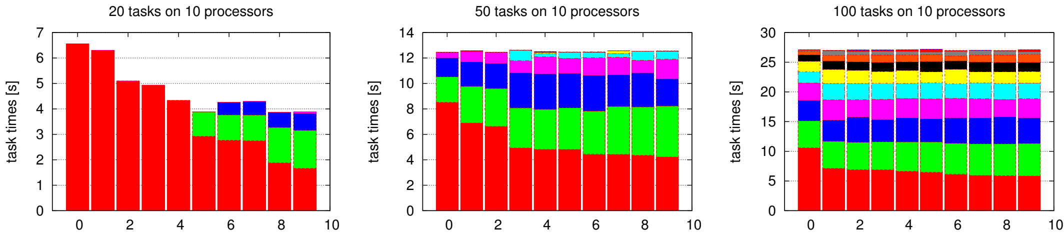

Figure 8: Example schedules produced by the scheduling algorithms for randomly generated task sets with 20 (left), 50 (middle), and 100 (right) tasks executed on 10 processors with task execution times randomly selected between 0 ms and 10 ms using a Gaussian distribution.

times. The difference in the energy consumption of the different distribution function comes from a different distribution of the task execution times in the predefined execution time interval. For example, a Gaussian tail distribution favors small tasks whereas the Beta distribution usually produces more longer-running tasks.

6.3 Setup for energy measurements

Execution time and energy experiments have been performed on three Intel Core i7 processors: (i) An Intel Core i7-2620M mobile (Sandy Bridge) processor with a maximum frequency of 2.7 GHz with two physical cores and hyper-threading, leading to four logical cores. The memory hierarchy includes a 4 MB shared L3 cache as well as a 256 KB L2 cache and a 32 KB L1 cache per core. The main memory size is 8 GB. The specified thermal design power (TDP) is 35W. (ii) An Intel Core i7-2600 desktop (Sandy Bridge) processor with a maximum frequency of

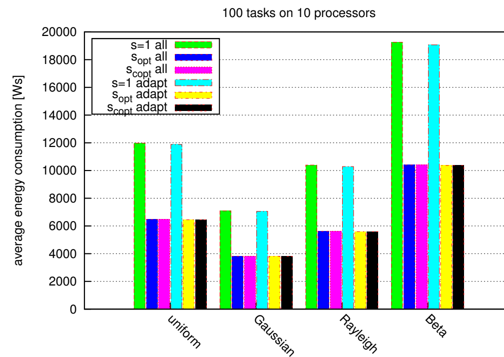

Figure 9: Result of scheduling experiments for executing 100 tasks on ten processors with task execution times randomly selected using different distributions.

3.4 GHz with four physical cores and hyper-threading, leading to eight logical cores. The L1 and L2 cache sizes and the main memory size are the same as for the Core i7-2620M, the L3 cache is 8 MB. The TDP is 95 W. (iii) An Intel Core i7-4770 with the Haswell architecture, maximum frequency of 3.4 GHz, four physical cores with hyper-threading, leading to eight logical cores. The cache and main memory sizes are the same as for (ii). The specified thermal design power (TDP) is 84 W.

The Core i7 architecture incorporates a power management technology which supports four power management states: performance states (P-states), throttle states (T-states), idle states (C-states) and sleep states (S-states) [26]. P-states are predefined sets of frequency and voltage combinations at which an active core can operate; the various P-states are implemented by using a combination of dynamic frequency scaling (DFS) and dynamic voltage scaling (DVS). A C-state is an idle state in which parts of the processor are powered down to save energy. Various C-states are supported by Intel processors, and a higher numbered C-state indicates more power savings. The Core i7 processor has two power planes on chip: PP0 contains the processor cores and all caches, whereas PP1, also referred to as Uncore, contains additional devices, such as graphics devices or the power control unit (PCU).

For measuring the energy consumption, we have used the RAPL (Running Average Power Limit) interface of the i7 architecture, which provides mechanisms to measure and control the power consumption [26]. An alternative hardware-based technique for obtaining energy values is the measurement with power-meters connected to the pins used for the supply voltage of the CPU and other components of a computer system. Experiments have shown that the energy measurement with power-meters and the energy measurement with RAPL show a good match [47], meaning that the RAPL interface (to be used in our experiments) provides reliable energy information. The RAPL interface allows the access to Model Specific Registers (MSRs), which provide information about the energy status of the PP0 and PP1 power planes via the MSR_PP0_ENERGY_STATUS and MSR_PP1_ENERGY_STATUS MSRs. The corresponding energy status unit is obtained via the MSR_RAPL_POWER_UNIT MSR; the default value is 15.3 micro-Joules. The MSR can be accessed by the *rdmsr* and *wrmsr* instructions (privilege level 0). For accessing the MSRs in our experiments, we have used the likwid tool-set (Version 2.2) [52], which we have modified slightly to obtain the energy consumption of the PP0 and PP1 power planes for our example application.

To set the frequencies of the cores to a fixed value, we have used the *cpufreq_set* tool, see. e.g., `wiki.archlinux.org`; the minimum and the maximum core frequencies have been set to the same value. During program execution, the frequency of each core has been observed with the *i7z* tool, see, e.g., `code.google.com/p/i7z/`. The runtime experiments have been performed with no other application program running on the machine, i.e., interferences can only come from jobs of the operating system. To reduce the effect of such interferences, the runtime experiments have been performed five times.

6.4 Energy measurements for the extrapolation method

As example application, we consider the extrapolation method [21, 32, 44], which is a solution method for initial value problems (IVPs) of ordinary differential equations (ODEs). In each time step, the extrapolation method

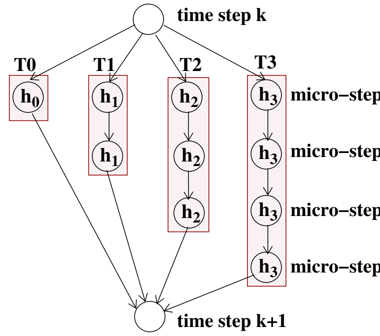

Figure 10: Task structure of one time step of the extrapolation method for $r = 4$.

computes several approximations with different step-sizes using a base method, such as the Euler method. Using r different step-sizes, r different numerical approximations are obtained, which are combined by an extrapolation table to get the approximation for that time step. The main computational work lies in the execution of the micro-steps, which includes the evaluation of the right-hand side function $\mathbf{f}$ of the ODE system to be solved. The task structure of one time step is illustrated in Fig. 10.

For a multi-threaded implementation with Pthreads, we use r worker threads $w_0, \ldots, w_{r-1}$, where thread w_i performs the micro-steps for step-size h_i. Worker thread w_0 also computes the extrapolation table. A barrier synchronization is used between the time steps. The threads are bound to a fixed logical core using *pthread_setaffinity_np()* to ensure that the tasks assigned to a worker-thread are executed by the same (logical) core. The experiments are performed using Linux Kernel 2.6.37. The multi-threaded versions are compared with on a purely sequential version without thread generation.

Figure 11 shows results of the energy measurement for an extrapolation method with $r = 4$ different step-sizes executed on an Intel Core i7 2620M processor with two physical cores (four logical cores). The extrapolation method is used to solve a Brusselator ODE system, which results from spatial discretization of a two-dimensional partial differential equation describing the reaction with diffusion of two chemical substances [21]. The parameter N determines the number of discretization points used in each dimension. The following experiments are based on measurements with a fixed number of 10 time steps solving a system of size $N = 2000$. The following three versions are compared: (i) a sequential execution of the four tasks executed on a single (logical) core in each time step, i.e., the tasks are executed one after another, (ii) a concurrent execution of the four tasks on four (logical) cores such that core i executes a different task T_i with step-size h_i; this causes different load on different (physical) cores, since T_i performs i micro-steps, (iii) a concurrent execution with a modified task assignment to physical cores such that each (physical) core performs the same number of micro-steps. Using *cpufreq-set*, the frequency of the cores is set to a fixed frequency (0.8 GHz, 1.4 GHz, 2.0 GHz, 2.7 GHz) corresponding to different frequency scaling factors, or is allowed to vary between 0.8 GHz and 3 GHz. The diagrams in Fig. 11 show the resulting execution times (top left), energy consumption (top right), power consumption (bottom left), and energy-delay product (bottom right). The energy-delay product EDP is defined as the energy consumed by an application program multiplied by its execution time [46]. It is used as a single metric to combine the effects of execution time and energy consumption for different configurations of an application and captures the translation of energy into useful work. Smaller EDP values indicate a better efficiency, i.e., a larger performance per energy unit [46].

It can be observed that the execution time of each of the versions decreases with increasing frequency. Correspondingly, the energy consumption increases with the frequency. The parallel versions always lead to a faster execution time than the sequential version, and also the energy consumption is reduced. Therefore, the parallel versions have much smaller EDP values as the sequential version, showing a better energy efficiency. The large energy consumption of the sequential version is caused by the fact that only one core transforms energy into useful work, while the other cores still consume energy due of the static power consumption.

Comparing the two parallel versions, the smallest amount of energy is consumed by the extended task assignment scheme, as this scheme assigns the same number of micro-steps to each physical core, thus avoiding idle times. This conforms to our calculations in Sect. 4. A similar effect could be achieved by using different frequency

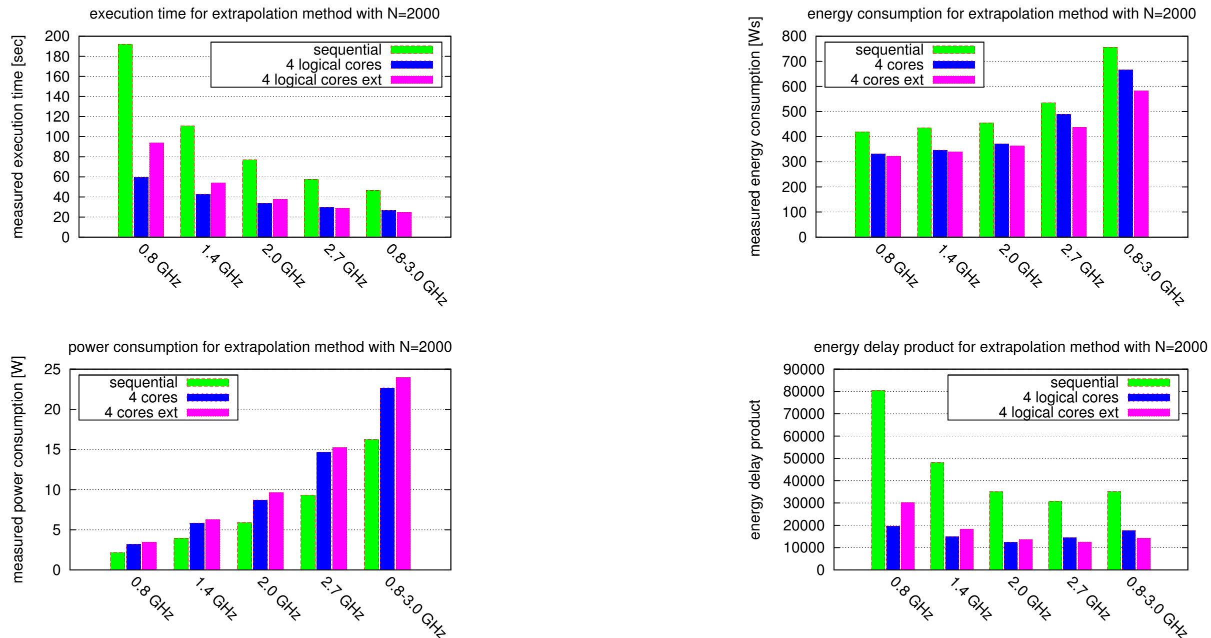

Figure 11: Execution time (top left), energy consumption (top right), power consumption (bottom left), and energy-delay product (bottom right) for an extrapolation method with four different stepsizes on Intel Core i7 2620M processor.

values for different cores, which was impossible for our experimental setup due to hardware restrictions. The diagrams in Fig. 11 show that due to differences in the power consumption, a small energy consumption does not necessarily correspond to a small execution time: for small frequency values, the extended task assignment scheme is outperformed by the regular task assignment scheme. The bottom left diagram shows that the power consumption increases with the frequency, and a sequential execution leads to the smallest power consumption. Similar observations can be made for the Core i7-2600 processor, see Fig. 12 for an extrapolation method with $r = 8$ different stepsizes, corresponding to eight tasks in each time step. As the Core i7-2600 is a desktop processor, higher frequency values can be used. The maximum over-clocking frequency is set to 3.7 GHz in the BIOS. For this processor, the extended task assignment scheme leads to the smallest EDP values for all frequencies due to the avoidance of waiting times.

The measured power consumption shown in Fig. 11 and 12 (bottom left) can be exploited to determine the power consumption values $P_{dyn}(1)$ and P_{static} based on Equ. (3) or Equ. (4). For the Intel Core i7 2620M, using the two largest frequencies f_1 = 2.7 GHz and f_2 = 2.0 GHz (corresponding to scaling factor $s_1 = 1$ and $s_2 = 1.35$, respectively) the values $P_{dyn}(1)$ = 13.65 W and P_{static} = 1.56 W result. For the Intel Core i7 2600 processor, the same method yields $P_{dyn}(1)$ = 45 W and P_{static} = 5.94 W.

The values for $P_{dyn}(1)$ and P_{static} can be used to predict the energy consumption and the power consumption for other frequencies according to Equ. (3) or Equ. (4). The comparison of these predicted values with the measured values in Fig. 11 and 12 provides a good match for most frequencies, and the deviation lies below 10 % in most cases. The deviation gets larger for smaller frequencies and only for the smallest frequencies, the predicted power consumption values are significantly larger than the measured values, which might be caused by additional features of the PCU to save even more power at low frequencies.

Using the values for $P_{dyn}(1)$ and P_{static} in Equ. (5) yields the optimal scaling factors $s_{opt} = 2.6$ for the Core i7 2620M, which corresponds to a frequency of 1.0 GHz, and $s_{opt} = 2.47$ for the Core i7 2600, which corresponds to a frequency of 1.5 GHz.

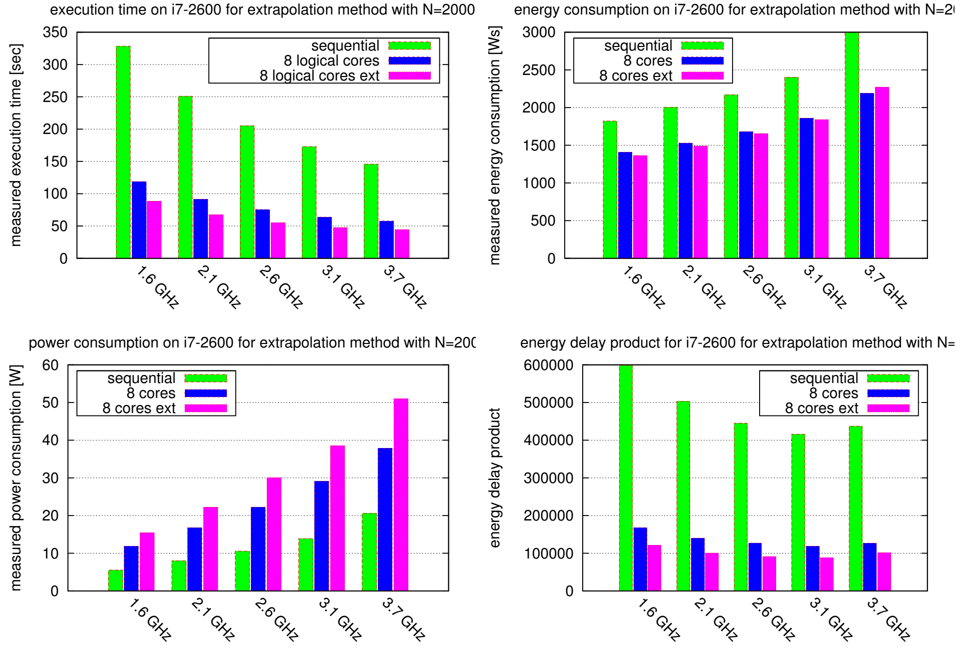

Figure 12: Execution time (top left), energy consumption (top right), power consumption (bottom left), and energy-delay product (bottom right) for an extrapolation method with eight different stepsizes on Intel Core i7 2600 processor.

6.5 Energy consumption of the SPEC benchmarks

For a further validation of the energy model from Sect. 2, we also have performed energy measurements for the SPEC CPU2006 benchmarks on an Intel Core i7-4770 Haswell processor with four cores and hyperthreading. For this processor, the frequency can be set between 0.8 GHz and 3.4 GHz. The frequency stepsize is 200 MHz with two exceptions (1.4/1.5 GHz and 2.7/2.8 GHz). The SPEC CPU2006 benchmark suite consists of integer and floating-point benchmarks, which are real programs covering different application areas. The benchmarks are sequential C, C++ and Fortran programs, see, e.g., [22] and `www.spec.org` for more details. For the energy measurements discussed in the following, the benchmarks have been compiled with the gcc 4.7.2 compiler using the compiler option -ftree-parallelize-loops=4, enabling an automatic parallelization at loop level.

Figure 13 shows the energy values for the SPEC CPU2006 integer (top) and floating-point (bottom) benchmarks executed on the Core i7 Haswell architecture using different frequencies. Again, the energy consumption values have been obtained with the likwid toolset [52]. The figures shows that for each benchmark program there is a slight variation of the energy consumption with the frequency. The energy consumption is large for small frequencies and decreases with increasing frequencies. Starting with a frequency of about 2.5 GHz, the energy consumption increases slightly with the frequency for most of the benchmarks. The minimum energy consumption is often obtained at frequency values between 2.0 GHz and 2.5 GHz.

For each of the SPEC benchmarks, the measured energy values shown in Fig. 13 and the corresponding measured execution times for each frequency value can be used to determine $P_{dyn}(1)$ and P_{static} based on Equ. (3) with the least squares method. The resulting values for $P_{dyn}(1)$ lie between 9.73 W and 13.50 W for the different benchmarks; the average value is 11.10 W. The values for P_{static} lie between 6.59 W and 8.52 W with an average value of 7.20 W. Using the average values for computing the optimal scaling factor s_{opt} results in $s_{opt} = 1.45$, which corresponds to a frequency of 2.34 GHz. For most of the benchmarks, this is quite near the frequency where the minimum energy consumption can be observed, i.e., the energy model used is indeed able to determine the operational frequency that leads to the smallest energy consumption.

The SPEC benchmarks have also been investigated for the the Sandy Bridge architecture (Core i7-2600). The following values for $P_{dyn}(1)$ and P_{static} can be determined from the execution times and energy measurements of

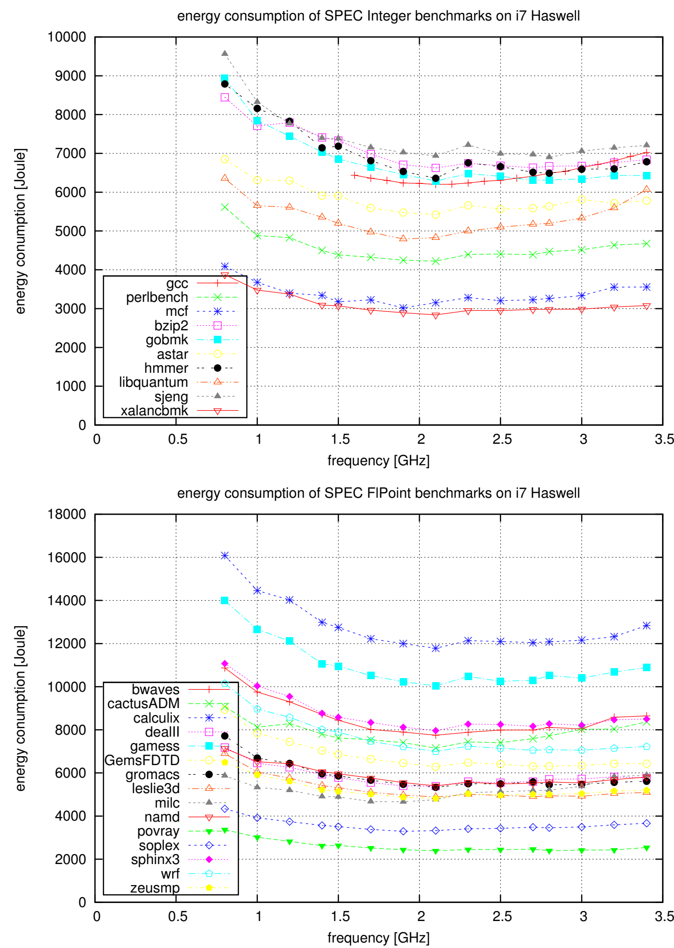

Figure 13: Energy consumption of SPEC integer (top) and floating point (bottom) benchmarks on an Intel Core i7 Haswell processor using different frequencies.

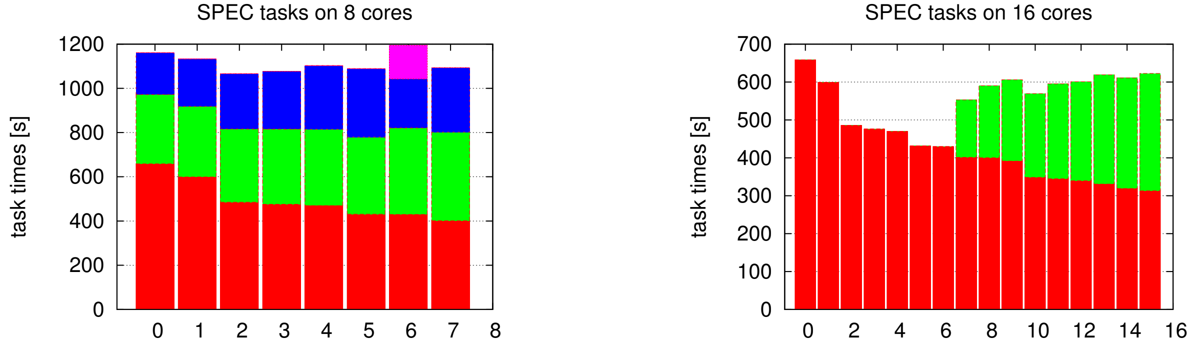

Figure 14: Schedule for the SPEC tasks executed on eight cores (left) and 16 cores (right).

the SPEC benchmarks (not shown in a figure): $P_{dyn}(1)$ lies between 10.0 W and 14.19 W with an average value of 11.26 W. P_{static} lies between 9.89 W and 11.87 W with an average value of $P_{static} = 10.35$. The average values lead to an optimal scaling factor $s_{opt} = 1.29$, which corresponds to a frequency of 2.6 GHz. Compared to the power values $P_{dyn}(1)$ and P_{static} determined for the extrapolation method, the values for $P_{dyn}(1)$ determined for the SPEC benchmarks are quite small. This can be explained by the fact that most SPEC programs produce a sequential workflow for most of their execution time, i.e., only one of the four cores is working and the other cores are set into sleep state, whereas the extrapolation method produces a parallel workflow with all four cores performing computations. A sequential execution of the extrapolation method leads to similar power consumption values as the SPEC benchmarks, see Fig. 12 (bottom left).

The SPEC benchmarks have been used for another scheduling experiment in which the execution of each individual program of the SPEC benchmarks on a single core is considered as one task. This yields 25 tasks (with different execution times), which can be executed in parallel on the cores of a multicore architecture. As task execution times, the execution times of the SPEC benchmarks on a single un-scaled core of the Haswell architecture are taken. Figure 14 shows the schedules for 8 and 16 cores resulting from the scheduling algorithms from Sect. 5. Since the number of tasks is small, a significant idle time arises especially for 16 cores. Thus, applying frequency scaling is beneficial for reducing the energy consumption. For the average values $P_{dyn}(1) = 11.10W$ and $P_{static} = 7.20$ of the Intel Core i7-4770 Haswell processor, the optimal scaling factors s_{copt} for the longest-running core are: (i) $s_{copt} = 1.39$ for $p = 4$ cores, (ii) $s_{copt} = 1.36$ for $p = 8$ cores, and (iii) $s_{copt} = 1.25$ for $p = 16$ cores. The optimal scaling factor for a single task is $s_{opt} = 1.45$ for all three cases. From the energy model from Sect. 3, the energy consumptions (in Joules) given in the following table result for the different choices of the scaling factors:

number of cores	$s = 1$ all	$s = s_{opt}$ all	$s = s_{copt}$ all	$s = 1$ adapt	$s = s_{opt}$ adapt	$s = s_{copt}$ adapt
$p = 4$	165982	144237	144320	158090	140512	140269
$p = 8$	167749	146809	147026	155429	140994	140374
$p = 16$	174867	157170	158179	150150	145502	142383

The table shows that the execution on four cores leads to the smallest energy consumption in the un-scaled case, which is caused by the smaller idle times compared to an execution with 8 or 16 cores, see the first three columns of the table. The adaptation of the scaling factors reduces the energy consumption by avoiding the idle times, so that lower energy consumption values result for all cases, see the last three columns of the table. According to the theoretical results, the last column shows the smallest energy values with the smallest value for $p = 4$ cores. The corresponding parallel execution times, however, are smallest for 16 cores, so that there is a tradeoff between execution time and energy consumption and the application programmer can select a suitable setting.

7 Related Work

Energy consumption is emerging to be a dominant performance factor and there is an enormous amount of research dealing with a reduction of the energy consumption on different levels, including technological power management as well as application-oriented bi-critical optimization or scheduling. Power-management features are integrated in computer systems of almost every size and class, from handheld devices to large servers [49]. An important feature is the DVFS technique that trades off performance for power consumption by lowering the operating voltage and frequency if this is possible [57]. The approach to determine the voltage scaling factor that minimizes the total CPU energy consumption by taking both the dynamic power and the leakage power into consideration has been discussed in [27, 28, 57] for sequential tasks.

The energy consumption of parallel algorithms for shared memory architectures based on the parallel external memory (PEM) model [4] has been discussed in [33]. The interaction between the parallel execution and energy consumption is considered in [13] by partitioning a parallel algorithm into sequential and parallel regions and computing optimal frequencies for these regions. No task structuring of the parallel algorithms is considered. Approaches for an energy complexity metric are discussed in [9]. [51] proposes a system-level iso-energy-efficiency model to analyze, evaluate and predict energy-performance of data intensive parallel applications. The energy consumption of interconnection networks of chip multiprocessors (CMP) is addressed in [10, 19]. An energy-oriented evaluation of communication optimization for networks is given in [29] with a focus on sensor networks which have different characteristics as networks in high-performance computing.

The main paradigms in energy-aware scheduling are speed scheduling algorithms and power-down scheduling algorithms [14]. Our approach belongs to the first paradigm. The survey of [50] discusses energy- and performance-aware scheduling algorithms of the three possible ways to solve the bi-critical problem, i.e. performance-constrained energy optimization, energy-constrained performance optimization, and dual energy and performance optimization. Our approach can be considered to be in the first category, however, our specific approach is different from the methods discussed, since we first compute an analytic solution for an optimal frequency scaling and then use the results in a scheduling algorithm.

One of the main differences in energy-aware scheduling approaches is the energy model used. Most earlier articles consider the dynamic power consumption but ignore the static power consumption [50]. The work in [39] seems mostly related to our approach at first glance, since it combines analytical methods with scheduling methods. However, the difference of both approaches originates from the power model considered, which consists only of the dynamic power in [39] and a combination of dynamic power and static power in our case. This gives rise to different methods and results. An analytically proven necessary condition of equal power for all processors on a multiprocessor is used to formulate and solve a scheduling problem equivalent to the sum-of-bag problem in [39]. In contrast, our necessary condition of an optimal solution requires to eliminate idle times by different frequency scales and an optimal frequency scale can be computed analytically and can then be exploited for scheduling. Also, [39] provides only simulations of the results proven. In contrast, we were able to perform hardware measurements for recent processor architectures of the energy which supports the energy model that we have used.

The scheduling of independent unit-size tasks on a heterogeneous master-worker platform with communication costs using different energy models concerning switching overhead and memory constraints is considered in [43]. The goal is to maximize the throughput while minimizing the energy consumed. This is a different problem than the problem we consider, since we investigate tasks with different execution times; moreover the energy model in [43] assumes that the static energy consumption can be ignored. The scheduling of task graphs with different dependency structures (tree, series-parallel, general DAG) is investigated in [5] for energy models with different switching modes. Again the static energy consumption is not taken into account.

Algorithmic research on speed scaling processors and related scheduling algorithms using the total energy consumption as objective function has been initiated by the article [55]. Many different scenarios and algorithms have been investigated in the literature, see [1, 8, 14] for a good overview. For example, theoretical foundations of scheduling algorithms in a setting with dynamic speed scaling processors are investigated in [2], considering the scheduling of n jobs on m identical variable speed processors working in parallel, where each job is specified by a release date, a deadline, and a processing volume. Different scenarios concerning the job size, release dates and deadlines are considered and approximation algorithms for the resulting NP-hard scheduling problems are presented. In most of the articles in this research line, the emphasis lies on a theoretical investigation of the approximation algorithm derived and no simulations or measurements on real hardware systems are provided.

In the domain of real-time scheduling, many DVFS-based techniques have been considered for utilizing waiting times, see, e.g., [24, 40, 56]. These approaches are usually based on heuristics and are not based on an analytical model as presented in this work. The effects of dynamic concurrency throttling (DCT) and DVFS in the context of a hybrid MPI/OpenMP programming model are considered in [38]. In particular, frequency selection is formulated as a variant of the 0-1 knapsack problem and dynamic programming is used to compute an approximation. In contrast, we propose an analytical solution using scaling factors to derive an optimal solution.

8 Conclusions

In this article, it has been demonstrated how an analytical energy model capturing frequency scaling can be used to model and analyze the power consumption of applications which are formulated using a task-based programming model with fork-join parallelism. The approach assumes that the task execution times are known, which can be provided by a performance prediction or measurements of task execution times. For iterative methods, such measurements can be performed by using the measured task runtimes of one iteration as estimation for the task execution times in later iterations. Using the estimations for the task execution times, frequency scaling factors are computed for each task in a fork-join construct such that waiting times are avoided at join points. We have shown that this minimizes the energy consumption for a given assignment of tasks to processors.

The scaling factor technique derived in this article aims at a system software support and can be used in two ways: For a given schedule with assignments of tasks to processors, the scaling factors can be used to reduce the energy consumption without affecting the execution time (unscaled case for the longest-running processor) or to minimize the energy consumption of the given schedule (using the optimal scaling factor for the longest-running processor). Both techniques aim at a post-processing of given schedules to reduce the energy consumption, and the given schedules can be computed with the makespan as objective function. Additionally, the energy model can be used to employ the energy consumption directly as objective function in the scheduling algorithm.

Simulations with randomly generated task sets have been performed to illustrate the quantitative effects of different scaling factors on the resulting execution time and power consumption. Energy measurements have been performed for three Intel Core i7 processors executing a complex application program from numerical analysis and the SPEC CPU2006 benchmarks to evaluate the power consumption for different frequencies and to validate the energy model used.

References

[1] S. Albers. Energy-efficient algorithms. *Commun. ACM*, 53(5):86–96, May 2010.

[2] S. Albers, F. Müller, and S. Schmelzer. Speed scaling on parallel processors. In *Proc. of the 19th Annual ACM Symp. on Parallel Algorithms and Architectures*, SPAA '07, pages 289–298, New York, NY, USA, 2007. ACM.

[3] E. Allen, D. Chase, V. Luchangco, J.-W. Maessen, S. Ryu, G.L. Steele Jr., and S. Tobin-Hochstadt. The Fortress Language Specification. Technical report, Sun Microsystems, 2006.

[4] L. Arge, M.T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *SPAA '08: Proc. of the 20th Ann. Symp. on Parallelism in Algorithms and Architectures*, pages 197–206. ACM, 2008.

[5] G. Aupy, A. Benoit, F. Dufossé, and Y. Robert. Reclaiming the energy of a schedule: models and algorithms. *Concurrency and Computation: Practice and Experience*, 25(11):1505–1523, 2013.

[6] E. Ayguadé, R. M. Badia, P. Bellens, D. Cabrera, A. Duran, R. Ferrer, M. González, F. D. Igual, D. Jiménez-González, and J. Labarta. Extending OpenMP to Survive the Heterogeneous Multi-Core Era. *Int. J. of Parallel Programming*, 38(5-6):440–459, 2010.

[7] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang. The Design of OpenMP Tasks. *IEEE Trans. Parallel Distrib. Syst.*, 20(3):404–418, 2009.

[8] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. Speed scaling to manage energy and temperature. *J. ACM*, 54(1):3:1–3:39, March 2007.

[9] B.D. Bingham and M.R. Greenstreet. Computation with Energy-Time Trade-Offs: Models, Algorithms and Lower-Bounds. In *ISPA '08: Proc. of the 2008 IEEE Int. Symp. on Parallel and Distributed Processing with Applications*, pages 143–152. IEEE Computer Society, 2008.

[10] J. Cebrian, J. Aragon, and S. Kaxiras. Power-token balancing: Adapting CMPs tp power constraints for parallel multithreaded workloads. In *Proc. of the 25th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS 11)*. IEEE, 2011.

[11] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, 21:291–312, August 2007.

[12] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: An object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538, New York, NY, USA, 2005. ACM.

[13] S. Cho and R. Melhem. Corollaries to Amdahl's Law for Energy. *IEEE Comput. Archit. Lett.*, 7(1):25–28, 2008.

[14] M. Chrobak. Algorithmic Aspects of Energy-Efficient Computing. In I. Ahmad and S. Ranka, editors, *Handbook of Energy-Aware and Green Computing*, pages 311–329. CRC Press, 2012.

[15] P. Cichowski, J. Keller, and C. Kessler. Energy-efficient Mapping of Task Collections onto Manycore Processors. In *Proc. MULTIPROG'13 workshop at HiPEAC'13*, 2013.

[16] J. Dinan, S. Krishnamoorthy, D.B. Larkins, J. Nieplocha, and P. Sadayappan. Scioto: A Framework for Global-View Task Parallelism. In *Proc. of the 37th Int. Conf. on Parallel Processing*, ICPP '08, pages 586–593. IEEE Computer Society, 2008.

[17] J. Dümmler, R. Kunis, and G. Rünger. SEParAT: Scheduling Support Environment for Parallel Application Task Graphs. *Cluster Computing*, 15(3):223–238, 2012.

[18] K. Faxen. Efficient Work Stealing for Fine Grained Parallelism. In *Proc. of the 39th Int. Conf. on Parallel Processing*, ICPP '10, pages 313–322. IEEE Computer Society, 2010.

[19] A. Flores, J. L. Aragon, and M. E. Acacio. Heterogeneous Interconnects for Energy-Efficient Message Management in CMPs. *IEEE Transactions on Computers*, 59:16–28, 2010.

[20] R. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Appl. Math.*, 1969.

[21] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, Berlin, 2nd edition, 2000.

[22] J.L. Hennessy and D.A. Patterson. *Computer Architecture - A Quantitative Approach (5. ed.)*. Morgan Kaufmann, 2012.

[23] R. Hoffmann and T. Rauber. Adaptive Task Pools: Efficiently Balancing Large Number of Tasks on Shared-address Spaces. *International Journal of Parallel Programming*, 39(5):553–581, 2011.

[24] T. Horvath, T. Abdelzaher, K. Skadron, and X. Liu. Dynamic Voltage Scaling in Multitier Web Servers with End-to-End Delay Control. *IEEE Trans. Comput.*, 56(4):444–458, 2007.

[25] Intel. *Quad-Core Intel Xeon Processor 5400 Series Datasheet*, 2008.

[26] Intel. *Intel 64 and IA-32 Architecture Software Developer's Manual, System Programming Guide*, 2011.

[27] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. *ACM Trans. Algorithms*, 3(4):41, 2007.

[28] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 275–280. ACM, 2004.

[29] I. Kadayif, M. Kandemir, A. Choudhary, and M. Karakoy. An energy-oriented Evaulation of Communication Optimizations for Microsensor Networks. In *Proc. of the EuroPar 2003 conference*, pages 279–286. Springer LNCS 2790, 2003.

[30] S. Kaxiras and M. Martonosi. *Computer Architecture Techniques for Power-Efficiency*. Morgan & Claypool Publishers, 2008.

[31] D.J. Kerbyson and A. Hoisie. A practical approach to performance analysis and modeling of large-scale systems. In *Proc. ACM/IEEE SC2006 Conference*, page 206, 2006.

[32] M. Korch, T. Rauber, and C. Scholtes. Scalability and locality of extrapolation methods on large parallel systems. *Concurrency and Computation: Practice and Experience*, 23(15):1789 – 1815, 2011.

[33] V.A. Korthikanti and G. Agha. Towards optimizing energy costs of algorithms for shared memory architectures. In *SPAA '10: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, pages 157–165, New York, NY, USA, 2010. ACM.

[34] M. Kühnemann, T. Rauber, and G. Rünger. A Source Code Analyzer for Performance Prediction. In *Proc. of the IPDPS-Workshop on Massively Parallel Processing (CDROM)*. IEEE, 2004.

[35] M. Kühnemann, T. Rauber, and G. Rünger. Performance Modelling for Task-Parallel Programs. In M. Gerndt, V. Getov, A. Hoisie, A. Malony, and B. Miller, editors, *Performance Analysis and Grid Computing*, pages 77–91. Kluwer, 2004.

[36] R. Kunis and G. Rünger. Optimizing layer-based scheduling algorithms for parallel tasks with dependencies. *Concurrency and Computation: Practice and Experience*, 23(8):827–849, 2011.

[37] Y.C. Lee and A.Y. Zomaya. Minimizing Energy Consumption for Precedence-Constrained Applications Using Dynamic Voltage Scaling. In *CCGRID '09: Proc. of the 2009 9th IEEE/ACM Int. Symp. on Cluster Computing and the Grid*, pages 92–99. IEEE Computer Society, 2009.

[38] D. Li, B.R. de Supinski, M. Schulz, D. Nikolopoulos, and K.W. Cameron. Strategies for Energy Efficient Resource Management of Hybrid Programming Models. *IEEE Transaction on Parallel and Distributed Systems*, 2012.

[39] Keqin Li. Performance analysis of power-aware task scheduling algorithms on multiprocessor computers with dynamic voltage and speed. *IEEE Trans. Parallel Distrib. Syst.*, 19(11):1484–1497, 2008.

[40] R. Mishra, N. Rastogi, D. Zhu, D. Mossé, and R. Melhem. Energy Aware Scheduling for Distributed Real-Time Systems. In *IPDPS '03: Proc. of the 17th Int. Symp. on Parallel and Distributed Processing*, page 21.2. IEEE Computer Society, 2003.

[41] T. N'takpé, F. Suter, and H. Casanova. A Comparison of Scheduling Approaches for Mixed-Parallel Applications on Heterogeneous Platforms. In *Proc. of the 6th Int. Symp. on Par. and Distrib. Comp.* IEEE, 2007.

[42] OpenMP Architecture Review Board. *OpenMP Application Program Interface, Version 3.0*, May 2008.

[43] J.-F. Pineau, Y. Robert, and F. Vivien. Energy-aware scheduling of bag-of-tasks applications on master-worker platforms. *Concurrency and Computation: Practice and Experience*, 23(2):145–157, 2011.

[44] T. Rauber and G. Rünger. Load Balancing Schemes for Extrapolation Methods. *Concurrency: Practice and Experience*, 9(3):181–202, 1997.

[45] T. Rauber and G. Rünger. Compiler Support for Task Scheduling in Hierarchical Execution Models. *Journal of Systems Architecture*, 45:483–503, 1998.

[46] G. Rong, F. Xizhou, S. Shuaiwen, C. Hung-Ching, L. Dong, and K.W. Cameron. PowerPack: Energy Profiling and Analysis of High-Performance Systems and Applications. *IEEE Transactions on Parallel and Distributed Systems*, 21(5):658 –671, may 2010.

[47] E. Rotem, A. Naveh, A. Ananthakrishnan, D. Rajwan, and E. Weissmann. Power-Management Architecture of the Intel Microarchitecture Code-Named Sandy Bridge. *IEEE Micro*, 32(2):20–27, 2012.

[48] S. K. Sahni. Algorithms for scheduling independent tasks. *Journal of the ACM*, 23(1):116–127, 1976.

[49] E. Saxe. Power-efficient software. *Commun. ACM*, 53(2):44–48, 2010.

[50] H. F. Sheikh, H. Tan, I. Ahmad, S. Ranka, and P. Bv. Energy- and Performance-Aware Scheduling of Tasks on Parallel and Distributed Systems. *ACM Journal on Eemerging Technologies in Computing Systems*, 8(4):1–37, Article 32, 2012.

[51] S. Song, C.-Y. Su, R. Ge, A. Vishnu, and K.W. Cameron. Iso-energy-efficiency: An approach to power-constrained parallel computation. In *Proc. of the 25th IEEE Int. Parallel and Distributed Processing Symp. (IPDPS 11)*. IEEE, 2011.

[52] J. Treibig, G. Hager, and G. Wellein. LIKWID: A Lightweight Performance-Oriented Tool Suite for x86 Multicore Environments. In *39th International Conference on Parallel Processing Workshops*, ICPP '10, pages 207–216. IEEE Computer Society, 2010.

[53] N. Vydyanathan, S. Krishnamoorthy, G.M. Sabin, U.V. Catalyurek, T. Kurc, P. Sadayappan, and J.H. Saltz. An Integrated Approach to Locality-Conscious Processor Allocation and Scheduling of Mixed-Parallel Applications. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1158–1172, 2009.

[54] A. Wong, D. Rexachs, and E. Luque. Extraction of Parallel Application Signatures for Performance Prediction. In *Proc. of the 2010 IEEE 12th Int. Conf. on High Performance Computing and Communications*, HPCC '10, pages 223–230. IEEE Computer Society, 2010.

[55] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proc. of the 36th Annual Symp. on Foundations of Computer Science*, FOCS '95, pages 374–, Washington, DC, USA, 1995. IEEE Computer Society.

[56] D. Zhu, R. Melhem, and D. Mossé. Energy efficient redundant configurations for real-time parallel reliable servers. *Real-Time Syst.*, 41(3):195–221, 2009.

[57] J. Zhuo and C. Chakrabarti. Energy-efficient dynamic task scheduling algorithms for DVS systems. *ACM Trans. Embed. Comput. Syst.*, 7(2):1–25, 2008.