J Sched (2016) 19:3–19
DOI 10.1007/s10951-015-0463-8

A survey of offline algorithms for energy minimization under deadline constraints

Marco E. T. Gerards[1] · **Johann L. Hurink**[1] · **Philip K. F. Hölzenspies**[2]

Published online: 8 January 2016

Abstract Modern computers allow software to adjust power management settings like speed and sleep modes to decrease the power consumption, possibly at the price of a decreased performance. The impact of these techniques mainly depends on the schedule of the tasks. In this article, a survey on underlying theoretical results on power management, as well as offline scheduling algorithms that aim at minimizing the energy consumption under real-time constraints, is given.

Keywords Scheduling · Algorithmic power management · Speed scaling · Sleep modes · Energy minimization

1 Introduction

Energy consumption is nowadays an important design constraint for computing systems (Zhuravlev et al. 2013). On the one hand, computing power of embedded systems increases rapidly, whereas the battery capacity does not grow with the same pace. On the other hand, like for datacenters, the energy consumption is an important cost factor.

To decrease the energy consumption of computing devices while still meeting performance constraints power management techniques are often deployed (e.g., Irani and Pruhs 2005; Albers 2010). Herein, software is used to influence the energy consumption of computers. This software allows control of hardware parameters like the speed (speed scaling), or decides to transition devices to a low-power sleep mode when they are not used. Combined with such power management techniques, scheduling algorithms play a crucial role, since the underlying schedules have a critical impact on the efficiency of power management techniques. The collection of all of these techniques is often referred to with the generic term *Algorithmic power management* (see, e.g., Pruhs 2011).

In this article, we discuss many algorithmic power management results. More precisely, we survey theoretical results on power management as well as *offline* algorithms for *energy minimization under deadline constraints* (called the "server problem"; Bunde 2006). Furthermore, we discuss both speed scaling and low-power sleep modes. Speed scaling is used to adapt the speed of a system, so that its power consumption is reduced. It can be hard to determine the optimal speeds, because these have to be chosen globally—all tasks have to be taken into account—instead of chosen locally on a task-by-task basis.

An idle device can be put in a low-power sleep mode to reduce the energy consumption; however, energy is required to wake it up again. This poses a trade-off between sleeping or remaining idle. Sleep modes can significantly reduce the energy consumption when the system has long idle periods. Because of this, scheduling algorithms are deployed to create schedules with many and sufficiently long idle periods. Note, that speed scaling and sleep modes are not mutually exclusive: sometimes it is better to use both in combination.

Note, that in the last years, also peak power minimization became an important topic of research (e.g., Lee et al. 2014; Manoj et al. 2013). We argue that many of the speed scaling algorithms that we survey also minimize the peak power of a system.

✉ Marco E. T. Gerards
m.e.t.gerards@utwente.nl

Johann L. Hurink
j.l.hurink@utwente.nl

Philip K. F. Hölzenspies
drphil@fb.com

[1] University of Twente, Enschede, The Netherlands

[2] Facebook, London, UK

This article is organized as follows. In the next section, we briefly discuss some related surveys. After that, in Sect. 3, we provide introductions into modeling of speed scaling and sleep modes, and we introduce the notations that are used throughout this survey. The latter is important, as different authors use different notations when describing their power management problems. Many are loosely based on the notation by Graham et al. (1977).

In Sect. 4, we present several (orthogonal) theoretical power management results, which form the foundation of many power management algorithms, and show how these results interact.

Section 5 surveys algorithms that minimize the energy consumption of single-processor systems with deadline constraints. The relation between similar problems is discussed and it is shown how the theoretical power management results from Sect. 4 are applied. This discussion is followed by a survey of multiprocessor power management problems, and algorithms for these problems in Sect. 6. In Sect. 7 several open problems are discussed, and Sect. 8 concludes this article with a discussion.

2 Related surveys

The recent article by Zhuravlev et al. (2013) surveys many energy-aware scheduling techniques. Many of the papers they survey are on thermal-aware scheduling and scheduling for asymmetric systems. In their survey, there is no emphasis on algorithms and their properties, which is the focus of this article.

Benini et al. (2000) give an important survey on sleep modes (DPM) that is mainly application oriented. They present a lot of background, and discuss implementation details, including a discussion on the Advanced Configuration and Power Interface (ACPI). The algorithms they discuss are intended for general operating systems, and depend on predictive schemes and stochastics. In contrast, this article focuses on (clairvoyant) offline algorithms for real-time systems. Moreover, this article discusses speed scaling and scheduling.

There are several articles that survey results from algorithmic power management. The very broad overview by Chen and Kuo (2007) discusses power-related scheduling techniques, but does not focus on algorithms. Irani and Pruhs (2005) and Albers (2010) present surveys that do focus on algorithms. The first survey (Irani and Pruhs 2005) contains a relatively small set of algorithms, while the second and more recent survey article (Albers 2010) discusses more algorithms. Although both surveys treat results from the entire spectrum of algorithmic power management, only a few offline algorithms for energy minimization under deadline constraints are discussed.

None of these surveys discussed in this section focuses on offline energy minimization under deadline constraints, nor treated many papers on this subject. Furthermore, to the best of our knowledge, the survey in this article is the first that links the many different theoretical concepts of algorithmic power management.

3 Modeling and notation

Many algorithmic power management papers have different modeling assumptions and there is no unique notation to describe both speed scaling and sleep mode problems. In this section, we structure the modeling assumptions and present a unifying notation for power management problems. Section 3.1 discusses the notation and models for tasks. Some practical aspects for speed scaling on a computer processor and a notation for these aspects are discussed in Sect. 3.2, while modeling of sleep modes is discussed in Sect. 3.3. Finally, a notation for algorithmic power management problems is presented in Sect. 3.4.

3.1 Task models

In general, a finite number (N) of tasks is considered, which we denote by $T_1, \ldots, T_N$. These tasks are scheduled on M processors, where in many cases $M = 1$. Each task T_n has a workload w_n. For speed scaling, a speed s_n at which task T_n is executed must be determined, which leads to an execution time $e_n = \frac{w_n}{s_n}$. In some cases, the speed may be changed during a task, which leads to an adaption of the used notation. Then the speed function $s : \mathbb{R}_0^+ \to \mathbb{R}_0^+$, which gives the speed as a function of the time, is used.

The available speeds are given by a set $\mathcal{S}$, which is either an interval ($\mathcal{S} = [s^{\min}, s^{\max}]$) or a finite discrete set with K speeds ($\mathcal{S} = \{\bar{s}_1, \ldots, \bar{s}_K\}$, where we assume w.l.o.g. that $\bar{s}_1 \leq \cdots \leq \bar{s}_K$). When a speed must be chosen from a continuous (discrete) set, we call this speed a continuous (discrete) speed, and refer to a problem with such restriction as a continuous (discrete) speed scaling problem.

Besides its workload, each task has an arrival time a_n and a deadline d_n. The tasks have to be scheduled to meet these constraints, implying that the begin time b_n and completion time c_n have to be chosen so that $a_n \leq b_n \leq c_n \leq d_n$. If the tasks are scheduled without interruption, we furthermore have $c_n = b_n + e_n$.

3.2 Processor models for speed scaling

An important objective used in the majority of papers that we survey is energy minimization of microprocessors. Hence, in the following we concentrate on speed scaling of microprocessors. Furthermore, we discuss some modeling

assumptions that are not studied in the current algorithmic power management literature.

Microprocessors have a clock frequency, which represents the speed of the processor. For many systems the speed of the computer memory (and other peripherals) does not scale with the clock frequency of the processor because it is a separate device that does not necessarily use the same clock frequency. In other words, in most practical settings the speed of the overall system (and of tasks) does *not* scale linearly with the clock frequency of the microprocessor (Devadas and Aydin 2012). However, all algorithms that we survey assume that the speed *does* scale linearly with the clock frequency, and hence we also assume this throughout this article. Note, that this assumption leads to an underestimation of the execution times of the tasks in case the clock frequency is decreased with respect to some reference clock frequency, which means that tasks finish earlier than is predicted using the models. Note, that for a multicore processor with only local memories (e.g., scratchpad memory) the speed *does* scale linearly with the processor clock frequency.

As a consequence of the above assumption, clock frequency and speed are synonyms, and therefore s_n and $s(t)$ are used to denote the clock frequency. In this article, we mostly use the terms speed and *speed scaling*, instead of clock frequency and Dynamic Voltage and Frequency Scaling (DVFS), in line with the majority of papers on algorithmic power management.

For multicore processors, there are two main flavors of speed scaling, namely *local speed scaling* and *global speed scaling* . While local speed scaling changes the speed per individual core, global speed scaling makes these changes for the entire chip. For this reason, the optimal solutions to the local and global speed scaling problems are not interchangeable. Global speed scaling is the most commonly applied of these techniques, since it is cheaper to implement (March et al. 2011; Chaparro et al. 2007). Examples of modern processors and systems that use global speed scaling are the Intel Itanium, the PandaBoard (dual-core ARM Cortex A9), IBM Power7, and the NVIDIA Tegra 2 (Kalla et al. 2010; March et al. 2011; Kandhalu et al. 2011; Zhang et al. 2012).

Nowadays, most modern microprocessors are built using CMOS transistors. When the clock frequency of a CMOS processor is decreased, the voltage may be decreased as well. Dynamic voltage and frequency scaling (DVFS) (Weiser et al. 1996) is a power management technique that allows the clock frequency and voltage to be changed at run-time. Both the clock frequency and the voltage influence the power consumption of a processor and the energy consumption is obtained by integrating this power consumption over time.

In general, there are two major sources of power consumption, namely dynamic power consumption and static power consumption. *Dynamic power* is consumed due to activities of the processor, i.e., due to transitions of logic gates. A CMOS transistor charges and discharges (parasitic) capacitances when it switches between logical zero and logical one. The dynamic power can be calculated by $ACV_{\mathrm{dd}}^2 s$, where V_{dd} is the supply voltage, s is the clock frequency (i.e., speed), C is the switched capacitance, and A is the activity factor, the average number of transitions per second (Ishihara and Yasuura 1998). For a given clock frequency, the minimal supply voltage is bounded and many papers (implicitly) assume that this minimal voltage is used, i.e., they used the simplified relation $V_{\mathrm{dd}} = \beta s$ for some constant $\beta > 0$ (e.g., Yao et al. 1995; Huang and Wang 2009). This gives the dynamic power model

$$p^{\mathrm{dyn}}(s) = \gamma_1 s^{\alpha}, \tag{1}$$

where α is a system-dependent constant (usually, $\alpha \approx 3$) and $\gamma_1 = AC\beta^{\alpha-1}$ contains both the average activity factor and switched capacitance. Most papers assume that γ_1 is constant for the entire application. Some papers use a separate constant $\gamma_1(n)$ for each task (referred to as *nonuniform loads* by Kwon and Kim (2005), or as *nonuniform power*), because the activity may deviate for different types of tasks. This makes the power function (to some extent) nonuniform, but throughout this article we assume γ_1 is constant. On the one hand this is done to keep the notation simple, and on the other hand we assume that when the power function is nonuniform, the theory that we present in Sect. 4.7 can be applied.

Static power is the power that is consumed *independently* of the activity of the transistors, and, thereby, it is independent of the clock frequency. However, there are two different definitions of static power that are used in the literature. The first definition of static power, popular in algorithmic papers (e.g., Cho and Melhem 2010), takes static power as a constant function (i.e., independent of the clock frequency), and is given by

$$p^{\mathrm{static}}(s) = \gamma_2,$$

where γ_2 is a system dependent constant. The second definition—often used in computer architecture papers—uses the voltage to express the static power. Although it is physically modeled using an exponential equation, the following linear approximation with system dependent constants γ_2 and γ_3 is popular (Park et al. 2013):

$$p^{\mathrm{static}}(V_{\mathrm{dd}}) = \gamma_2 + \frac{\gamma_3}{\beta} V_{\mathrm{dd}},$$

and the relation between the voltage and the clock frequency ($V_{\mathrm{dd}} = \beta s$) gives

$$p^{\mathrm{static}}(s) = \gamma_2 + \gamma_3 s.$$

Note, that this relation makes the static power—which is independent of the clock frequency—indirectly dependent on the clock frequency. The resulting static energy for w work executed at speed s is $\gamma_2 \frac{w}{s} + \gamma_3 w$, when it is assumed that static power is consumed until all work is completed (see the discussion in Sect. 4.3). As a consequence, the constant γ_3 does not influence the choice of the optimal clock frequency in the case of energy minimization, which is the focus of this article. Thus, we can assume without loss of generality that $\gamma_3 = 0$ and use $p^{\text{static}}(s) = \gamma_2$ to model the static power. Since both static power models lead to the same optimal solution, it is not relevant for optimization, which of the two static power models is used.

Generally, we define the total power consumption (both static and dynamic) as a *power function* $p : \mathbb{R}_0^+ \to \mathbb{R}_0^+$, which maps speed to power.

For microprocessors, the power function does not fully describe all energy that is used, since changing the clock frequency also has an energy and time overhead. The recent article by Park et al. (2013) shows that the time and energy overheads of DVFS are in the same order of magnitude as the overhead of context switching. For example, the transition delay overhead is at most 62.68 μs on an Intel Core2 Duo E6850 (Park et al. 2013). Furthermore, most algorithms avoid changing the clock frequency often because of the convexity of the power function (see Sect. 4.1), hence the number of speed changes is relatively low. Because of these two reasons, we assume that the energy overhead of changing the clock frequency is negligible in case of DVFS.

Note, that speed scaling is not restricted to microprocessors, but can also be used for flash memory (Lee and Kim 2010), hard disks (Liu et al. 2004), and may even be relevant to applications outside of computer science.

3.3 Sleep modes

As already mentioned in the previous subsection, devices also consume power when they are idle. Several devices like microprocessors, hard disks, communication devices (e.g., network interfaces) can switch to a sleep mode by powering (parts of) the device down to decrease the power when idle. For example, when a processor is transitioned to a sleep mode, the current state is stored, and the state is recovered when the processor is awakened. Another example is a hard-disk drive, which spins down when put to sleep mode, while it spins up when it is awakened. These devices have in common that a cost in both latency and energy is associated with switching to a sleep mode and waking up. The energy consumption determines the *break-even time*, which is the minimum length of an idle period which makes it worthwhile to transition to a sleep mode. It is commonly assumed that the break-even time for a sleep mode is longer than the latency associated of switching to and from this sleep mode. It was shown empirically that algorithms that use this assumption still work well when the latency is taken into account (Irani et al. 2007).

Devices can even have multiple sleep modes, with different break-even times, or there can be multiple devices within a system with different break-even times. The energy consumption during an idle period is generally modeled as a piecewise concave function $E^{\text{SL}} : \mathbb{R}_0^+ \to \mathbb{R}_0^+$ of the length of the idle period (Augustine et al. 2008; Gerards and Kuper 2013).

3.4 Problem notation and qualification

To classify a wide variety of algorithmic power management problems, in this section a compact notation (based on the three-field notation for scheduling problems that was introduced by Graham et al. 1977) to describe a wide variety of algorithmic power management problems is introduced. The notation is similar to what is used in the algorithmic power management literature (e.g., Bampis et al. 2015), but avoids several ambiguities, by making explicit what kind of power management techniques are used.

We specify a general power management problem by three fields $\mathfrak{a}|\mathfrak{b}|\mathfrak{c}$, where $\mathfrak{a}$ denotes the system properties, $\mathfrak{b}$ describes the tasks and their constraints, and $\mathfrak{c}$ is the objective for optimization. The fields with their possible entries and their meaning are given in Table 1. A brief discussion of this notation follows below.

- $\mathfrak{a}$: The system field describes the architecture of the system. This includes the number of processors (or devices), whether speed scaling (*ss*) and/or sleep modes (*sl*) are used, and properties of the system with respect to speed scaling and/or sleep modes (see Table 1). The entries *nonunif*, *disc*, and *global* all imply speed scaling (*ss*) to keep the notation concise.
- $\mathfrak{b}$: The second field contains the task characteristics like arrival time, deadline, restrictions on the ordering of timing constraints of tasks (*agree*, *prec*, *lami*), and scheduling properties (*migr*, *pmtn*, *prio*, *sched*). E.g., when a_n occurs in this field, it means that tasks have arrival times, otherwise a_n=0 (for all n) is implied.
  As we focus on energy minimization under deadline constraints, d_n always occurs in $\mathfrak{b}$ and implies that deadlines must be met.
- $\mathfrak{c}$: The third field contains the scheduling objective. In the context of this article, the field $\mathfrak{c}$ only contains "E" to denote that the energy should be minimized, but we maintain this field to preserve compliance with Graham's notation.

Table 1 Notation for algorithmic power management problems

Field	Entry	Meaning
α	1	Single processor
	P_M	M parallel processors
	ss	Speed scaling is supported
	nonunif	A nonuniform power function is used (ss implied)
	disc	Discrete speed scaling is used (ss implied)
	global	Global speed scaling is used (ss implied)
	sl	Sleep modes supported
β	a_n	Arrival time
	$a_n{=}a$	Same arrival time a for all tasks
	d_n	Deadline constraint
	$d_n{=}d$	Same deadline constraint d for all tasks
	$w_n{=}w$	All tasks have workload w
	agree	Agreeable deadlines ($a_n \leq a_m \Leftrightarrow d_n \leq d_m$)
	lami	Laminar instances ($[a_i, d_i] \subset [a_j, d_j] \vee [a_j, d_j] \subset [a_i, d_i] \vee [a_i, d_i] \cap [a_j, d_j] = \emptyset$)
	prec	Tasks have precedence constraints
	pmtn	Preemptions are allowed
	prio	Tasks have a fixed priority
	migr	Task migration is allowed
	sched	A schedule is given
γ	E	Minimize the energy consumption

4 Fundamental results

Over the years, many fundamental results on algorithmic power management have been obtained, which form the basis of many algorithms, or relate problems to each other, so that the solution to one problem can be used to find a solution to another problem. This section introduces these fundamental results and concepts in the area of algorithmic power management. One of the most important results is that for the single-processor case it is optimal to use a constant speed between begin and completion time of tasks due to the convexity of the power function (Sect. 4.1). Although this result only holds for convex power functions, using the idea presented in Sect. 4.2, it can also be used for the nonconvex situation as all power functions can be "made" convex. Convexity is not the only requirement for optimization, one has to be careful that the chosen speed for a task is not too low because then static power may dominate (Sect. 4.3).

Whereas the above results are often presented in a continuous speed scaling context, in practice, discrete speed scaling is more often used. Many speed scaling problems (with a given schedule) can be formulated as a linear program (Sect. 4.4). Moreover, in the single-processor case it is furthermore straightforward to derive the solution to this discrete problem from the solution to the continuous case (Sect. 4.5).

For multiprocessor problems, it can be shown that in the optimal solution of several problems the power consumption remains constant over time. This fact is referred to as the *power equality* (Sect. 4.6). The problem wherein every task has a different power function (Sect. 4.7) is related to this multiprocessor problem. We present a simple transformation that transforms this problem with multiple power functions to the problem wherein all tasks have the same power function.

Finally, we briefly discuss that speed scaling problems wherein preemptions are not allowed can sometimes be written as a flow problem (Sect. 4.8), and that when scheduling for sleep modes, it is often best to unbalance the length of idle periods (Sect. 4.9).

4.1 Constant speed

Whenever a single processor executes a single task using varying speeds, the energy consumption can be decreased by running it at the average speed. This even holds when the task is executed with interruptions (i.e., on times given by any set $\mathcal{T}$). This result holds for all convex power functions, where this property does not form a restriction as is discussed in Sect. 4.2. We formalize this result, which is a direct consequence of Jensen's inequality (Irani et al. 2007), in the following theorem.

Theorem 1 *Given a task with w work, which is executed at the times given by the set $\mathcal{T}$ (i.e., $w = \int_{\mathcal{T}} s(\tau)d\tau$) and is executed on a processor with a convex power function. Then the following inequality holds:*

$$p\left(\frac{w}{e}\right) e \leq \int_{\mathcal{T}} p(s(\tau))\mathrm{d}\tau.$$

Proof The infinite version of Jensen's inequality states:

$$p\left(\frac{1}{\int_{\mathcal{T}} 1d\tau} \int_{\mathcal{T}} s(\tau)\mathrm{d}\tau\right) \leq \frac{1}{\int_{\mathcal{T}} 1d\tau} \int_{\mathcal{T}} p(s(\tau))\mathrm{d}\tau.$$

Multiplying this equation by $\int_{\mathcal{T}} 1d\tau$ directly leads to the result of the theorem. □

Theorem 1 shows that for continuous speed scaling, there always exists a constant speed that is optimal for a single task on a single processor. Many papers (e.g., Huang and Wang 2009; Yao et al. 1995; Li et al. 2006) use the idea behind Theorem 1, and show that minimizing unnecessary speed fluctuations on a single processor is optimal also for situations with more than one task, i.e., $N > 1$. However, when there are arrival times, deadlines, etc., the optimal constant speed may change on these specific times, meaning that the optimal speed function is piecewise constant.

4.2 Nonconvex power function

The previous section (and with it, a large part of the literature) assumes that the power function is convex, but for technical reasons this is not always the case. However, it is possible to circumvent this by not using the speeds of the regions where the function is not convex, since we can show that these speeds are not efficient. This process is first explained for discrete speed scaling.

Assume three given speeds $\bar{s}_i < \bar{s}_j < \bar{s}_k$ (let $\bar{s}_j = \lambda \bar{s}_i + (1-\lambda)\bar{s}_k$ for some $\lambda \in (0,1)$) and w work, where

$$p(\bar{s}_j)w \leq p(\bar{s}_i)\lambda w + p(\bar{s}_k)(1-\lambda)w, \tag{2}$$

does not hold. This implies that executing the work at speed $\bar{s}_j$ would cost more energy than executing a part of the work at $\bar{s}_i$ and the remaining work at $\bar{s}_k$. In this case, we call $\bar{s}_j$ an *inefficient speed* as it is never beneficial to use this speed.

Based on the above, we may assume that all speeds in $\mathcal{S}$ are *efficient speeds*, thus Eq. (2) holds for all speeds (i.e., inefficient speeds are "discarded"), as is discussed by Hsu and Feng (2005). This illustrates that we can always assume without loss of generality that the power function is convex.

Bansal et al. (2013) state that a similar procedure can be followed for continuous speed scaling. Note, that the static and dynamic power models from Sect. 3.2 are already convex.

4.3 Critical speed

With the presence of static power, convexity of the power function is not the only aspect which has to be taken into account when finding an optimal solution for some speed scaling problems.

In practice, processors consume static power ($\gamma_2 > 0$), i.e., the power consumption at speed 0 is nonnegative ($p(0) > 0$). Unfortunately, most papers do not clearly define for which time period they take the static power into account. In this survey, we assume that the application begins at some given time t^B, and the power consumption of the processor is accounted for until some time t^C. Furthermore, we either assume that $t^C = c_N$ (completion time of the last task) or $t^C = d_N$ (deadline of the last task). For example, Yao et al. (1995) only assume that the power function is convex and do not mention static power. However, their result only holds when the static power cannot be influenced, i.e., when it is accounted for until the deadline of the last task and not only to the completion time of the last task. As in this case, static power cannot be influenced, the situation where $p(0) = 0$ gives the same solution as the case where $p(0) > 0$. This scenario is mentioned by Irani et al. (2007).

For the other scenario, where the static power is active until the last task has finished, not only the power function should be studied, but also the *energy-per-work function*:

$$\bar{p}(s) = \frac{p(s)}{s}.$$

This function gives the energy consumption of a unit work (instead of a unit time), has a global minimizer s^{crit} (called the *critical speed* by Jejurikar et al. 2004), and is increasing on $s \geq s^{\text{crit}}$ (Irani et al. 2007). All speeds below s^{crit} require more energy per unit work, while it takes longer to execute. Hence, the schedule length can be decreased by increasing speeds to s^{crit}, and the energy consumption is reduced.

4.4 Discrete speed scaling as a linear program

Besides static power, many processors have the restriction that only a small set of speeds is allowed (*discrete speed scaling*). Many discrete speed scaling problems with a given schedule can be formulated as a linear program, as we show in the following.

When discrete speed scaling is considered with K discrete speeds, the decision to be made is the amount of work of task T_n that is executed at speed $\bar{s}_k$. If we denote this amount by $w_{n,k}$ (i.e., $\sum_{k=1}^{K} w_{n,k} = w_n$), the total energy consumption of all tasks together is given by

$$\sum_{n=1}^{N}\sum_{k=1}^{K} p(\bar{s}_k)w_{n,k},$$

which is a linear function of the decision variables $w_{n,k}$. These variables, together with the begin time of tasks, form the decision variables of the linear program.

Constraints like arrival time, deadline, and precedence constraints can all be formulated as linear constraints. Therefore, many discrete speed scaling problems (with or without a given schedule) can be formulated as a linear program (Kwon and Kim 2005; Rountree et al. 2007) and, thus, can be solved in polynomial time.

4.5 Relation between continuous and discrete speed scaling

Formulating *discrete speed scaling* problems as a linear program and solving it with linear programming software provides few insights. Instead, a tailored algorithm for finding the optimal speeds is desirable. Such algorithms are described in many papers (e.g., Yao et al. 1995; Pruhs et al. 2008; Huang and Wang 2009) for continuous speed scaling, while in practice most processors support only discrete speed scaling. Therefore, in the following, we investigate the relation between continuous speed scaling and discrete speed scaling.

When a single task is considered, the optimal speed s resulting from the continuous case can be used to determine the optimal speeds for the discrete case. When the speed s is not one of the available discrete speeds, using only the neighboring speeds $\bar{s}_i \leq s \leq \bar{s}_{i+1}$ leads to an optimal solution. More precisely, the first part of the work is executed at speed $\bar{s}_{i+1}$ and the remaining work is executed at speed $\bar{s}_i$. These fractions of work are calculated so that the overall time remains the same. We refer to this as *simulating* continuous speed scaling.

The above-described simulating process has been proven to be optimal for the execution of a single task, and can be extended to multiple tasks. For multiple tasks, many continuous speed scaling algorithms only require that the power function is convex. Given a set of discrete speeds, we can fill the intervals between these speeds by taking the weighted average speed of a task using two neighboring speeds. This leads to a power function that gives as power for a given speed the weighted average power of the two used speeds (this function is called the *average power function*). Kwon and Kim (2005) and Hsu and Feng (2005) have proven that this average power function is a convex piecewise linear function. Hence, any continuous speed scaling algorithm that assumes only convexity can be used to find the optimal average speeds, after which the discrete assignment can be determined using simulation.

4.6 Power equality

The previous sections mainly focused on the single-processor case. In the multiprocessor case with precedence constraints, new issues arise that are best illustrated with an example.

Example 1 Consider the three tasks from Fig. 1, each with w work, which are to be executed on a local speed scaling *multiprocessor* system. Task T_1 has to be finished before tasks T_2 and T_3 can be executed, and the application as a whole has a global arrival time 0 and a global deadline d. An example of a naive speed assignment is $s_1 = s_2 = s_3 = \frac{2w}{d}$. Note that Theorem 1 cannot be used to argue that this assignment is optimal, because now multiple processors are active. In fact, this assignment is not optimal, since it can be improved by slightly increasing s_1 so that task T_1 consumes slightly more energy, while the two tasks T_2 and T_3 can decrease their energy consumption. The speed of task T_1 should not be too high (discussed below), because then its energy consumption is no longer compensated by tasks T_2 and T_3.

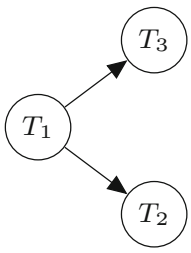


Fig. 1 Task graph

This example illustrates that the optimal speeds depend on the amount of parallelism of the scheduled tasks. Pruhs et al. (2008) introduce the *power equality* for tasks with a common arrival time and deadline: in the optimal solution, the power consumption remains constant. Thus, the power is constant, and the speeds can be calculated using this power and the number of parallel executed tasks. For the concrete situation of Fig. 1, this means that $p(s_1) = p(s_2) + p(s_3)$. This power equality generalizes Theorem 1.

Example 2 Consider again the task graph from Fig. 1 with the power function $p(s) = s^3$, and assume that all the tasks have 10 work, and the global deadline is 40. A naive speed assignment uses the constant speed $s_1 = s_2 = s_3 = \frac{1}{2}$.

As in an optimal solution, tasks T_2 and T_3 complete simultaneously, and we get $s_2 = s_3$. Due to the power equality, for the optimal solution it holds that

$$p(s_1) = p(s_2) + p(s_3) = 2p(s_2).$$

Using $p(s) = s^3$ and some elementary algebra gives $s_1 = \sqrt[3]{2}s_2$. Furthermore, the energy consumption is minimized when $\frac{w_1}{s_1} + \frac{w_2}{s_2} = 40$. Thus $s_1 = \frac{1+\sqrt[3]{2}}{4}$.

4.7 Nonuniform power

Most papers assume that uniform power is used (see Sect. 3.2), while in practice the parameter γ_1 of the power function is not constant (i.e., nonuniform) for all tasks (Kwon and Kim 2005), and a task specific factor $\gamma_1(n)$ for the dynamic power of task T_n is more appropriate. A similar situation occurs in the multicore situation with m active cores, where the dynamic power must be multiplied by m. This fact is used by several papers on multicore speed scaling (e.g., Gerards et al. 2015).

The dynamic energy consumption for N tasks with nonuniform power functions is given by (see Eq. (1) and Sect. 3.2)

$$E = \sum_{n=1}^{N} \gamma_1(n) s_n^{\alpha} \frac{w_n}{s_n}. \tag{3}$$

Fortunately, there is an elegant transformation due to Kwon and Kim (2005) that can reduce this expression to one with a constant power parameter γ_1. Using the substitution of variables $\mathring{w}_n = \sqrt[\alpha]{\gamma_1(n)}w_n$ and $\mathring{s}_n = \sqrt[\alpha]{\gamma_1(n)}s_n$, (3) becomes

$$E = \sum_{n=1}^{N} \mathring{s}_n^{\alpha} \frac{\mathring{w}_n}{\mathring{s}_n}. \tag{4}$$

Table 2 Uniprocessor algorithmic power management problems

Section	Problem	Papers
General tasks (Sect. 5.1)	1; ss$\|a_n$; d_n; pmtn$\|E$	Yao et al. (1995), Bansal et al. (2007), Li et al. (2006)
	1; disc$\|a_n$; d_n; pmtn$\|E$	Li et al. (2006), Hsu and Feng (2005)
	1; ss$\|a_n$; d_n; pmtn; prio$\|E$	Quan and Hu (2003)
	1; ss$\|a_n$; $d_n\|E$	Antoniadis and Huang (2013), Bampis et al. (2015), Huang and Ott (2014)
		Bampis et al. (2014a), Cohen-Addad et al. (2015), Bampis et al. (2014b)
	1; ss$\|a_n$; d_n; $w_n = 1\|E$	Huang and Ott (2014)
	1; ss; nonunif$\|a_n$; d_n; pmtn$\|E$	Kwon and Kim (2005)
	1; ss; nonunif; disc$\|a_n$; $d_n\|E$	Kwon and Kim (2005)
	1; sl$\|a_n$; d_n; pmtn$\|E$	Baptiste et al. (2012)
	1; ss; sl$\|a_n$; d_n; pmtn$\|E$	Irani et al. (2007), Albers and Antoniadis (2014), Antoniadis et al. (2015)
Agreeable deadlines (Sect. 5.2)	1; ss$\|a_n$; d_n; agree$\|E$	Huang and Wang (2009), Wu et al. (2011)
	1; sl$\|a_n$; d_n; agree$\|E$	Angel et al. (2014)
	1; sl; ss$\|a_n$; d_n; agree$\|E$	Bampis et al. (2012a)
Laminar instances (Sect. 5.3)	1; ss$\|a_n$; d_n; pmtn; lami$\|E$	Li et al. (2006)
	1; ss$\|a_n$; $d_n = d$; pmtn$\|E$	Li et al. (2006)
	1; ss$\|a_n = a$; d_n; pmtn$\|E$	Li et al. (2006)
	1; ss$\|a_n$; d_n; lami$\|E$	Huang and Ott (2014)

This corresponds to an instance where the execution time of task T_n becomes $\frac{\mathring{w}_n}{\mathring{s}_n}$, and $\gamma_1 = 1$ for all tasks, i.e., γ_1 disappears from the costs.

The newly obtained problem has uniform power, can be solved using classic algorithms, and the resulting solution can be transformed back to a solution to the problem with nonuniform power.

4.8 Flow problems

Several power management problems can be reduced to (convex) flow problems. However, as these formulations as flow problems depend on the concrete algorithmic power management problem, we do not discuss this technique in more detail. We refer the interested readers to three papers, namely Bampis et al. (2012b), Albers et al. (2011), and Angel et al. (2012b), which use such techniques to solve the problem P_M; $ss|a_n$; d_n; pmtn; migr$|E$. In Sect. 6.1 these papers are briefly discussed.

4.9 Sleep modes

A device can have multiple sleep modes that can be used to decrease the power consumption when the device is idle. A deeper sleep mode requires less power, but the transition costs are higher. As already mentioned, only when the idle period is longer than the *break-even time* of a sleep mode, it becomes worthwhile to use this sleep mode. Furthermore, for the case that in any idle period the best possible sleep mode is used (i.e., that with the lowest total energy consumption), we can derive an important property of the sleep mode problem. This property is based on the following two properties of the energy consumption function $E^{\mathrm{SL}}(\tau)$: the function $E^{\mathrm{SL}}(\tau)$ is an increasing and concave function and $E^{\mathrm{SL}}(0) = 0$. Because of these properties, it holds that for $0 \leq \delta \leq x \leq y$ (Gerards and Kuper 2013) we have

$$E^{\mathrm{SL}}(x-\delta) + E^{\mathrm{SL}}(y+\delta) \leq E^{\mathrm{SL}}(x) + E^{\mathrm{SL}}(y). \tag{5}$$

This means that, for any two idle periods of length x and y ($x \leq y$), the energy consumption does not increase when a certain amount δ of the smallest period gets shifted to a bigger idle period. This implies that a schedule that "unbalances" the length of idle periods reduces the energy consumption.

5 Uniprocessor problems

The previous section introduced many general concepts that can be applied to a variety of power management problems. This section surveys concrete algorithms for uniprocessor power management problems (see Table 2 for an overview), and relates these algorithms (when applicable) to the results that were presented in the previous section.

Recall that for each task T_n we have a workload w_n, an arrival time a_n, and a deadline d_n before which the task has to finish. In the case of speed scaling, a speed s_n is to be determined, leading to an execution time e_n. We use b_n and c_n to denote the begin and completion time of task T_n, respectively.

The problems in this section are grouped depending on restrictions on the ordering of the timing constraints of tasks. For *all* problems discussed in this section, the problem consists of finding a schedule together with speeds and/or sleep decisions. First, the problems without any restrictions on the timing constraints are discussed in Sect. 5.1. Several variants of this problem are solved by algorithms with a relatively high-polynomial time complexity, or are NP-hard. Second, in Sect. 5.2, the simpler case of problems with agreeable deadlines is discussed. For many variants of this problem, algorithms with a quadratic time complexity are known. Third, laminar problems are discussed in Sect. 5.3.

5.1 General tasks

In this section, we discuss general tasks, i.e., tasks that have arbitrary arrival times and deadlines. The first variant that we consider allows preemptions of tasks $(1; \text{ss}|a_n; d_n; \text{pmtn}|E)$. According to Albers et al. (2011), this is the most extensively studied speed scaling problem in the algorithm-oriented literature. Yao et al. (1995) present the well-known YDS algorithm (named after the authors) to solve this problem. This algorithm is often used as a subroutine by other algorithms, and in complexity proofs.

The considered problem involves both scheduling and speed scaling. However, if we have specified the speed to use over the complete time horizon, or if we have specified the speed of each task, we can find a corresponding feasible schedule—if it exists for this speed assignment—by planning successively always the available task with the smallest deadline (Yao et al. 1995). The basic idea of the YDS algorithm is to avoid unnecessary speed changes (see Sect. 4.1), and has the property that the speeds in the optimal solution cannot be lowered to decrease the energy consumption without violating deadlines.

More precisely, the YDS algorithm works with time intervals of the form $I_{i,j} = [a_i, d_j]$, where $a_i < d_j$. The density of such an interval is defined as

$$g(I_{i,j}) = \frac{\sum_{n \in T_{i,j}} w_n}{d_j - a_i},$$

where $T_{i,j} := \{T_n \mid [a_n, d_n] \subseteq I_{i,j}\}$ is the set of all tasks that have to be scheduled completely within the interval $I_{i,j}$. The density determines the minimal average speed that has to be used to execute the tasks from $T_{i,j}$ completely within this interval. The YDS algorithm takes a so-called *critical interval*—an interval $I_{i,j}$ with the highest density—and assigns to all tasks from $T_{i,j}$, and to the interval $I_{i,j}$ this density as speed. The algorithm creates a new subproblem by removing these tasks from the task set, and by removing the interval $I_{i,j}$ from the time axis leading to an adjustment of the arrival times and deadlines of the other tasks to take unavailability of the processor during this time interval into account. Next to leading to an optimal solution, by construction, YDS also avoids unnecessary speed fluctuations and obviously YDS also minimizes the peak power.

Example 3 (YDS algorithm) Consider the tasks from Table 3 of which the arrival times and deadlines are depicted in Fig. 2a. The YDS algorithm first determines the critical interval, which is $I_{2,2}$ in the first iteration of the algorithm (see Table 4). Since the density of this interval is $g(I_{2,2}) = 2$, task T_2 is assigned the speed $s_2 = 2$. Next, the interval $I_{2,2}$

Table 3 Tasks for Example 3

Task	Arrival time	Deadline	Workload
T_1	0	30	30
T_2	5	10	10
T_3	15	55	10
T_4	25	35	10

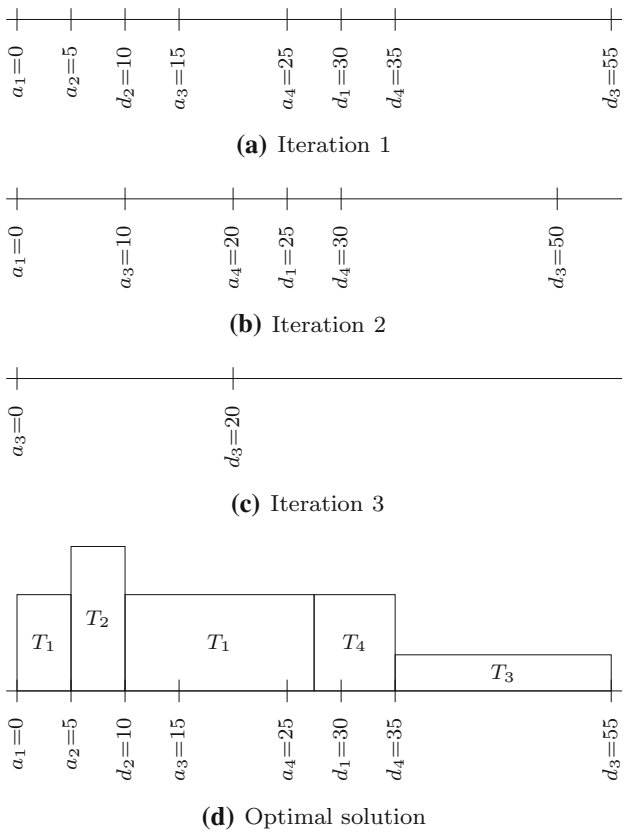


Fig. 2 Arrival times, deadlines, and optimal solution for Example 3. (a) Iteration 1. (b) Iteration 2. (c)Iteration 3. (d) Optimal solution

Table 4 Interval densities for Example 3

Interval	Iteration 1 $g(I_{i,j})$		Iteration 2 $g(I_{i,j})$		Iteration 3 $g(I_{i,j})$	
$I_{1,1}$	$\frac{40}{30}$	≈ 1.333	$\frac{30}{25}$	$= 1.2$		
$I_{1,2}$	$\frac{10}{10}$	$= 1$				
$I_{1,3}$	$\frac{50}{55}$	≈ 0.909	$\frac{50}{50}$	$= 1$		
$I_{1,4}$	$\frac{50}{35}$	≈ 1.429	$\frac{40}{30}$	≈ 1.333		
$I_{2,1}$	$\frac{10}{25}$	$= 0.4$				
$I_{2,2}$	$\frac{10}{5}$	$= 2$				
$I_{2,3}$	$\frac{30}{50}$	$= 0.6$				
$I_{2,4}$	$\frac{20}{30}$	≈ 0.667				
$I_{3,1}$	0		0			
$I_{3,2}$	0					
$I_{3,3}$	$\frac{20}{40}$	$= 0.5$	$\frac{20}{40}$	$= 0.5$	$\frac{10}{20}$	$= 0.5$
$I_{3,4}$	$\frac{10}{20}$	$= 0.5$	$\frac{10}{20}$	$= 0.5$		
$I_{4,1}$	0		0			
$I_{4,2}$	0					
$I_{4,3}$	$\frac{10}{30}$	≈ 0.333	$\frac{10}{30}$	≈ 0.333		
$I_{4,4}$	$\frac{10}{10}$	$= 1$	$\frac{10}{10}$	$= 1$		

is removed, and the arrival times and deadlines of the other tasks are adapted accordingly (see Fig. 2b).

In the second iteration, interval $I_{1,4}$ yields the critical density $g(I_{1,4}) = \frac{4}{3}$ (see Table 4), which is assigned as speed to task T_1 and T_4 (i.e., $s_1 = s_4 = \frac{4}{3}$). After removing these tasks, only task T_3 remains in the last iteration (see Fig. 2c), which is assigned the speed $s_3 = \frac{1}{2}$. A preemptive Earliest Deadline First (EDF) schedule with the aforementioned speeds ensures that the deadlines are met and the energy consumption is minimized.

In a schedule created by this YDS algorithm, the processor is active from the arrival of the first task to the deadline of the last task (unless there are no tasks in some interval). Hence, because of static power, this algorithm is only optimal when it is assumed that the processor remains active until the last deadline (Irani et al. 2007). To the best of our knowledge, there is no optimal algorithm known for the situation where no static energy is consumed after the last executed task.

The original implementation of the YDS algorithm has a time complexity of $O(N^3)$ (Li et al. 2006). As the original paper (Yao et al. 1995) does not contain a proof of optimality, several proofs of optimality have appeared in the literature afterwards. Bansal et al. (2007) use the Karush Kuhn Tucker (KKT) conditions (Boyd and Vandenberghe 2004) to prove optimality of YDS for the power function $p(s) = s^\alpha$. Li et al. (2006) give a different proof, and present an efficient implementation of YDS with time complexity $O(N^2 \log N)$. They also provide an $O(KN \log N)$ algorithm for the variant with discrete speed scaling with K speeds (1; disc$|a_n; d_n;$ pmtn$|E$). A recent technical report by Li et al. (2014) states that the continuous problem can be solved in $O(N^2)$ and the discrete problem can be solved in $O(N \log \max\{N, K\})$. An alternative method for obtaining the optimal speeds in the discrete case is by applying the YDS algorithm, and then simulating the obtained speeds as discussed in Sect. 4.5 (Kwon and Kim 2005; Hsu and Feng 2005).

The YDS algorithm schedules tasks in EDF order. This implies that when tasks must be scheduled in a predefined order (e.g., based on priorities), the YDS algorithm cannot be used (Quan and Hu 2003). Yun and Kim (2003) show that the fixed priority variant of this problem (1; ss$|a_n; d_n;$ pmtn; prio$|E$) is NP-hard, and give an FPTAS for the problem.

There exist several other variations of the problem introduced by Yao et al. (1995). The variant that does not allow preemptions of tasks (1; ss$|a_n; d_n|E$) is NP-hard (Antoniadis and Huang 2013). Bampis et al. (2015) designed an algorithm for this problem with the approximation ratio $(1 + w^{\max}/w^{\min})^\alpha$, where $w^{\max}$ and $w^{\min}$ are, respectively, the upper and lower bounds on the work of tasks. Bampis et al. (2014b) use results from several papers (Huang and Ott 2014; Bampis et al. 2014a; Cohen-Addad et al. 2015) for this problem to design an algorithm with approximation ratio $(1 + \epsilon)^\alpha \tilde{B}_\alpha$, where $\tilde{B}_\alpha = \sum_{k=0}^{\infty} \frac{k^\alpha e^{-1}}{k!}$ is a generalization of the Bell numbers that works for fractional values of α. When all tasks have the same workload (1; ss$|a_n; d_n; w_n = 1|E$), the problem can be solved in polynomial time (Huang and Ott 2014).

Kwon and Kim (2005) study another variation, where the dynamic power consumption may differ per task (1; ss; nonunif$|a_n; d_n;$ pmtn$|E$). This is, for example, due to switched capacitances. They solve this problem using a substitution of variables (see Sect. 4.7). They formulate the discrete speed scaling variant of this problem (1; ss; nonunif; disc$|a_n; d_n;$ pmtn$|E$) as a linear program (see Sect. 4.4).

The sleep mode counterpart of the YDS problem is 1; sl$|a_n; d_n;$ pmtn$|E$. Baptiste et al. (2012) present an algorithm that is commonly referred to as BCD (named after the authors), that uses dynamic programming to solve the problem in $O(N^4)$ time. Their algorithm is restricted to instances where processors have only a single sleep mode.

Other authors (Albers and Antoniadis 2014; Irani et al. 2007) study the combination of speed scaling and sleep modes, namely 1; ss; sl$|a_n; d_n;$ pmtn$|E$, which is an NP-hard problem. The heuristic by Irani et al. (2007) is a 2-approximation and is relatively easy to implement. This heuristic uses YDS to determine the speeds, and whenever YDS determines a speed $s_n < s^{\text{crit}}$, this speed is replaced by the speed s^{crit} (this is called an s^{crit}-schedule). These changes create idle time, that can be used to put the processor into a

sleep mode. As long as there are tasks available, they are consecutively executed, followed by an idle period of maximal length. This scheduling method is used to create relatively large idle periods. Albers and Antoniadis (2014) use a similar method, but with the cut-off speed s^* instead of s^{crit}, where s^* is determined by solving $\bar{p}(s^*) = \frac{4}{3}\bar{p}(s^{\text{crit}})$. Furthermore, they use BCD instead of the scheduling algorithm by Irani et al. (2007). This results in a 4/3-approximation, but has a higher time complexity ($O(N^4)$) because of the use of BCD. When the power function $p(s) = \gamma_1 s^\alpha + \gamma_2$ is used (realistic for DVFS), the approximation ratio becomes 137/117 (<1.171). Recently, Antoniadis et al. (2015) presented an FPTAS for this problem that is based on dynamic programming. In this dynamic programming approach, the time horizon is discretized by a polynomial number of intervals, where the number of intervals depends on the required approximation ratio.

5.2 Agreeable deadlines

In applications like multimedia and telecommunication, the arrival times and deadlines are usually in the same order (i.e., $a_n < a_m \Leftrightarrow d_n \leq d_m$). Such applications are said to have *agreeable deadlines*. This special structure of the timing constraints makes the development of efficient speed scaling and sleep mode algorithms possible. One main reason for this is that we can assume w.l.o.g. that the tasks are scheduled in order of their timing constraints (i.e., deadlines) and that no preemption is used (for the latter, see e.g., Bampis et al. 2015)

Speed scaling for systems with agreeable deadlines $(1; \text{ss}|a_n; d_n; \text{agree}|E)$ is studied by many authors (e.g., Huang and Wang 2009; Wu et al. 2011). Huang and Wang (2009) present an algorithm that calculates the optimal speeds in quadratic time. Their algorithm first schedules the task using the same speed for all tasks. This speed is calculated, so that all tasks are scheduled exactly within the time interval between the first arrival time and the last deadline without any idle time. Then, a task T_n with the largest violation of an arrival or a deadline in this schedule is used to divide the set of tasks into two subsets: the tasks before and the tasks after the violation. For a deadline violation, the completion time of task T_n is fixed to d_n, while for an arrival time violation the begin time of task T_n is fixed to a_n. Then the procedure is recursively repeated for both subsets.

In a variant of this problem, the maximal rate of change of the speed is bounded from above by R (i.e., $\max_t |s'(t)| \leq R$, for some $R > 0$). For this problem Wu et al. (2011) present an algorithm, which finds the optimal solution in quadratic time.

Next to agreeable deadlines with speed scaling, also the problem with sleep modes and the combination of speed scaling and sleep modes is studied in the literature. For the problem where the processor has a single sleep mode $(1; \text{sl}|a_n; d_n; \text{agree}|E)$, the algorithm by Angel et al. (2012a) (see also Angel et al. 2014) can be applied to find an energy optimal schedule. The authors observe that there always exists an optimal solution in which every task T_n starts at either (i) a_n, (ii) c_{n-1}, or (iii) $d_n - e_n$. Note, that the options for the completion time c_{n-1} depends on the begin times of tasks $T_1, \ldots, T_{n-1}$. By this, for each task T_k (tasks ordered in EDF order), there are $O(k)$ possible begin times, leading to a quadratic time complexity. This result by Angel et al. (2012a) is extended by Bampis et al. (2012a) leading to a cubic time algorithm to find the optimal combination of speed scaling and sleep modes $(1; \text{sl}; \text{ss}|a_n; d_n; \text{agree}|E)$.

5.3 Laminar instances

In this section, we study tasks with a nested structure, called *laminar instances*. A real-time system is a laminar instance whenever, for each pair of tasks, the permissible intervals ($[a_n, d_n]$ for task T_n) do not overlap, or one is completely contained within the other. In a graphical representation, a task T_i is drawn on top of task T_j when $[a_i, d_i] \subset [a_j, d_j]$, which creates layers of tasks and explains the term "laminar instances." According to Li et al. (2006) these structures occur in recursive programs. Since the tasks can be arranged in a tree structure that expresses this recursion, laminar instances are also referred to as tree-structured tasks (Li et al. 2006). Li et al. (2006) give an efficient polynomial time algorithm to find the optimal speeds for laminar instances $(1; \text{ss}|a_n; d_n; \text{pmtn}; \text{lami}|E)$. The variant of this problem that does not allow preemptions $(1; \text{ss}|a_n; d_n; \text{lami}|E)$ is NP-hard. Huang and Ott (2014) present a Quasi-Polynomial Time Approximation Scheme (QPTAS) for this problem.

Just as for the problem with agreeable deadlines, the restriction to laminar instances makes the problem easier to solve. In fact, the case where all deadlines or all arrival times are the same has both agreeable deadlines *and* is a laminar instance. For both problems, a linear time solution is available (Li et al. 2006).

6 Multiprocessor problems

This section discusses multiprocessor algorithmic power management problems (see Table 5 for an overview). The problems in this section consist of finding a multiprocessor schedule together with speeds and/or sleep decisions. General tasks (i.e., tasks without special restrictions on arrival times and deadlines) are discussed in Sect. 6.1. Algorithms for tasks with agreeable deadlines are discussed in Sect. 6.2, followed by a discussion of tasks with precedence constraints in Sect. 6.3.

Table 5 Multiprocessor algorithmic power management problems

Section	Problem	Papers
General tasks (Sect. 6.1)	P_M; ss$\|a_n = a; d_n = d\|E$	Albers et al. (2014), Pruhs et al. (2008), Chen et al. (2004)
	P_M; ss$\|a_n; d_n$; pmtn; migr$\|E$	Bingham and Greenstreet (2008), Albers et al. (2011), Angel et al. (2012b), Bampis et al. (2012b)
	P_M; ss$\|a_n; d_n$; pmtn$\|E$	Albers et al. (2014), Greiner et al. (2014)
	P_M; ss$\|a_n; d_n\|E$	Cohen-Addad et al. (2015), Bampis et al. (2015), Bampis et al. (2014a)
Agreeable deadlines (Sect. 6.2)	P_M; ss$\|a_n; d_n; w_n = 1$; agree$\|E$	Bampis et al. (2015)
Tasks with precedence constraints (Sect. 6.3)	P_M; ss$\|a_n = a; d_n = d$; prec$\|E$	Li (2012)
	P_M; global$\|d_n = d$; prec$\|E$	Gerards et al. (2015)
	P_M; global$\|a_n; d_n$; sched; prec$\|E$	Gerards et al. (2014)

6.1 General tasks

We first consider the variant of the problem, where all tasks arrive at time 0, have a shared global deadline, and local speed scaling is used to minimize the total energy consumption (P_M; ss$|a_n = a; d_n = d|E$). This problem is strongly NP-hard (Albers et al. 2014), since the 3-partition problem can be reduced to it. Pruhs et al. (2008) show that the problem of minimizing the makespan under an energy constraint can be formulated as the problem of minimizing the ℓ_α norm of the processor loads (where α is the exponent in the dynamic power function, see Sect. 3.2). For the latter problem, a PTAS exists (Alon et al. 1997). In a similar fashion, also a PTAS can be derived for energy minimization under a global deadline constraint. Such a PTAS cannot exist (unless $\mathcal{P} \neq \mathcal{NP}$) if there is a maximum speed $s^{\max}$, i.e., $s_n \leq s^{\max}$ for all n (Chen et al. 2004). Chen et al. (2004) study both the general tasks problem (P_M; ss$|a_n = 0; d_n = d|E$) and the variant with restricted speeds. For the first problem they provide an algorithm with a 1.13 approximation ratio, which also attains this ratio for the second problem under some additional restrictions. Furthermore, they presented an algorithm that can solve both problems optimally when migrations are allowed.

There are several variations of the problem with arbitrary arrival times and deadlines considered in the literature. They differ depending on whether preemptions and migrations of tasks are allowed or not. The widely studied problem P_M; ss$|a_n; d_n$; pmtn; migr$|E$ uses the combination of local speed scaling and scheduling, where preemptions and migrations of tasks are allowed. This problem was first studied by Bingham and Greenstreet (2008), wherein the authors show that the problem is convex. They present an algorithm that is polynomial in the number of tasks, but according to the authors, the complexity is too high for practical applications. However, as they also discuss properties of the optimal solution, their paper is important when studying multiprocessor speed scaling with preemptions and migrations. Albers et al. (2011) present a more efficient polynomial time algorithm for the same problem. Their algorithm uses repeated maximum flow computations to minimize the energy consumption. A closely related approach by Angel et al. (2012b) also uses maximum flow computations to find the optimal solution in polynomial time. The resulting algorithm is more efficient than that of Albers et al. (2011) for the case that a reduced accuracy is allowed. Another approach to the same problem is discussed in the paper by Bampis et al. (2012b), wherein the optimal speeds are determined by solving a convex flow problem. In this approach, execution times correspond to amounts of flow, which have to be sent through the network. The algorithm that solves this problem has a time complexity that depends on the latest deadline. Although this dependency on the deadline is a drawback, the presented approach is straightforward and its concepts are interesting for future research in this direction.

Albers et al. (2014) study the variant of the problem where migrations are not allowed (P_M; ss$|a_n; d_n$; pmtn$|E$). They show that the problem is NP-hard, even for tasks with unit workload (for which a PTAS is given). The difficult part of this problem is the assignments of tasks to processors. If such an assignment is given, determining the optimal speeds and scheduling order is straightforward, since YDS can be used for the tasks on each individual processor. The heuristic by Albers et al. (2014) sorts the tasks in order of nondecreasing deadlines, and assigns the tasks in this order to the processor with the lowest amount of work assigned to it. This heuristic has an approximation ratio of $2(2 - \frac{1}{N})^\alpha$. A more general version of this problem that considers a weighted sum of the energy consumption and flow time as objective is studied by Greiner et al. (2014).

In recent years, the problem that allows neither migration nor preemption (P_M; ss$|a_n; d_n|E$) has caught some attention

(Cohen-Addad et al. 2015; Bampis et al. 2015). Bampis et al. (2014a) use results from this previous research to develop an algorithm with the approximation ratio $\tilde{B}_\alpha\big((1+\epsilon)(1+w^{\max}/w^{\min})\big)^\alpha$.

6.2 Agreeable deadlines

Just as for the uniprocessor problem with agreeable deadlines, in the multiprocessor case a solution to the preemptive problem with no migration can be transformed to a non-preemptive solution with no migration with the same costs (Bampis et al. 2015).

Albers et al. (2014) present an optimal algorithm for the multiprocessor agreeable deadline problem where tasks have unit workload (P_M; ss|a_n; d_n; w_n=1; agree|E). This algorithm sorts the tasks in order of nondecreasing deadlines, assigns them to the processors using round robin scheduling and applies an algorithm that solves 1; ss|a_n; d_n; $w_n = 1$; agree|E (e.g., YDS) to the task sets for each individual processor. For tasks with an arbitrary workload they give an $\alpha^\alpha 2^{4\alpha}$-approximation algorithm.

6.3 Tasks with precedence constraints

According to the survey by Chen and Kuo (2007) ... *energy-efficient scheduling for jobs with precedence constraints with theoretical analysis is still missed in multiprocessor systems*. Only a few papers have studied speed scaling of tasks with precedence constraints, and to the best of our knowledge no papers studied the sleep mode variant of this problem. Since the local speed scaling problem (P_M; ss|$a_n = a$; $d_n = d|E$) from Sect. 6.1 is already NP-hard, the variant with precedence constraints (P_M; ss|$a_n = a$; $d_n = d$; prec|E) is also NP-hard.

Li (2012) studies the latter problem, and shows that under specific conditions the optimal solution to this problem becomes straightforward to approximate, namely for graphs with precedence constraints that have more parallelism than processors (called wide task graphs). Due to the amount of parallelism, the tasks are easy to schedule and using a single speed for the entire application gives near-optimal results.

The global speed scaling variant of this problem (P_M; global|$a_n = a$; $d_n = d$; prec|E) is also NP-hard, and was studied by Gerards et al. (2015). This problem consists of both scheduling and speed scaling. However, the second step is easy to solve, since the concept of power equality (see Sect. 4.6) can be applied to find the optimal speeds. Gerards et al. (2015) give a scheduling criterion that—together with optimal speeds—leads to a minimal energy consumption. Furthermore, they show how well existing scheduling algorithms perform at approximating the energy consumption.

A closely related problem that also assumes global speed scaling is P_M; global|a_n; d_n; sched; prec|E, where tasks have individual arrival times and deadlines, and a schedule of the tasks is already *given*. Gerards et al. (2014) give a method that finds the optimal speeds by combining the results on nonuniform power (Sect. 4.7) and the power equality (Sect. 4.6). The given schedule is subdivided into pieces, whereby a piece is a chunk of workload with a *constant* number of active cores, during which no tasks start or complete. Using the results on nonuniform power and the power equality, these pieces are transformed in such a way that a uniprocessor problem with agreeable deadlines 1; ss|a_n; d_n; agree|E is achieved, which can be solved in quadratic time (see Sect. 5.2). This solution can be transformed back to obtain the optimal solution of the original problem.

7 Open problems

This section discusses some open problems related to speed scaling. The first problem (Sect. 7.1) is about the relation between continuous and discrete speed scaling for a multiprocessor system. This problem was already solved for single-processor systems. The second problem is about speed scaling of tasks with precedence constraints on a local speed scaling system. Even for a given schedule, this problem may be hard.

7.1 Multiprocessor discrete speed scaling

Discrete speed scaling for a single processor is often considered a simpler problem than continuous speed scaling. There is an $O(N^2 \log N)$-time algorithm for the frequently studied problem 1; ss|a_n; d_n; pmtn|E, while there is a $O(KN \log N)$-algorithm to the discrete speed scaling variant of this problem with K speeds (in practice, $K \ll N$). Furthermore, a solution to a continuous speed scaling problem can be converted to the discrete speed scaling variant in $O(N \log K)$ time by simulating the continuous speeds (Sect. 4.5). To the best of our knowledge, there are no papers that relate optimal continuous and discrete speed scaling for multiprocessor systems, or that solve discrete multiprocessor speed scaling problems algorithmically. Only in the simple case where tasks have no precedence constraints and local speed scaling is used, the techniques from single-processor speed scaling can be applied to individual processors. More research on discrete speed scaling for multiprocessor systems, and the relation between continuous and discrete speed scaling on such systems is desirable.

7.2 Local speed scaling for tasks with precedence constraints

Local speed scaling for tasks with precedence constraints is an unsolved and important problem. Even the case where the tasks have been scheduled (i.e., task have been assigned to processors, and per processor a sequence of the assigned tasks is given) and only speeds need to be determined ($P_M|a_n = a; d_n = d$; prec; sched$|E$) is currently unsolved. The power equality (discussed in Sect. 4.6) can be used as a first step toward solving the problem.

The following example illustrates why this problem may be difficult.

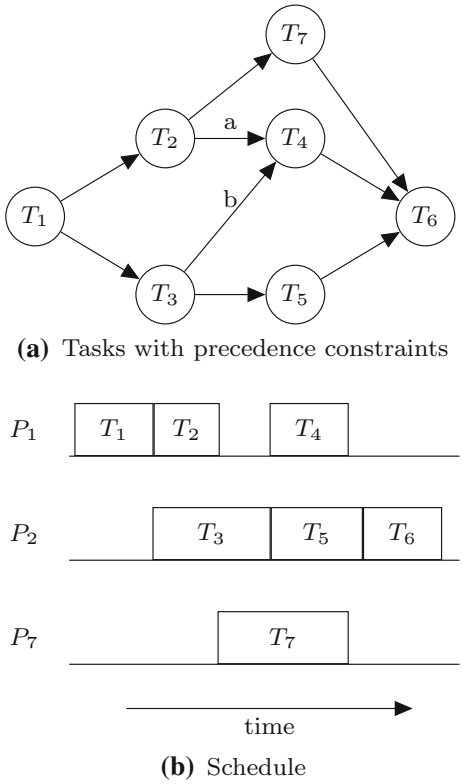
Example 4 Consider the power function $p(s) = s^3$ for a three-processor system with local speed scaling. The tasks have precedence constraints as given in Fig. 3a. All tasks share the common deadline $d = 1$.

We keep the work of the tasks variable in this example, to demonstrate the influence of the work on the solution. The schedule (with some arbitrarily chosen workload) is given in Fig. 3b. Note, that the position of the gaps in the schedule will change when the workload changes. The optimisation problem is: for a given schedule, determine the optimal speed assignment that minimizes the energy consumption, respects precedence constraints, and meets the deadline.

Due to convexity of the power function, in the optimal solution it must hold that $s_1 = s_6$. To ease the discussion, we consider two situations:

(a) Task T_2 finishes before task T_3, or at the same time.
In the discussion below, we may assume that the edge "a" between task T_2 and T_4 does not exist, as (with the given assumption) it does not influence the optimal solution. In the optimal solution, we have $e_2+e_7 = e_3+e_4 = e_3+e_5$ (same execution time for tasks, avoiding gaps in the schedule), otherwise the energy consumption can be decreased by decreasing the speed of a task that is next to a gap in the schedule. These relations can be used to determine the speeds of these tasks. Using the power equality, the relation between the speeds s_3, s_4, and s_5 can be determined. It can also be used to relate speeds s_1, s_2, and s_3. Now enough information is available to find the optimal speeds.

(b) Task T_2 finishes after task T_3.
In the discussion below, we may assume that the edge "b" between tasks T_3 and T_4 does not exist, as (with the given assumption) it does not influence the optimal solution. In the optimal solution we have that $e_2+e_7 = e_2+e_4 = e_3+e_5$. Again, using the convexity of the power function and using the power equality, the optimal speeds can be determined.



(a) Tasks with precedence constraints

(b) Schedule

Fig. 3 Precedence constraints and schedule for Example 4. *a* Tasks with precedence constraints. *b* Schedule

A possible method for finding the optimal speeds now is by calculating the energy consumption for both situations and selecting the one with the lowest costs.

This example indicates that solving the overall continuous problem depends on a number of discrete cases. These cases are specified by whether some task finishes before or after some other task. As it is unclear how many of these decision points may occur, and if there is an efficient (polynomial time) algorithm to make these decisions, the above example suggests that the local speed scaling problem with a given schedule of tasks with precedence constraints may be difficult.

8 Discussion

Algorithmic power management can be used to significantly reduce the energy consumption of computing devices. Combined with such power management techniques, scheduling algorithms play a crucial role, since the underlying schedules have a critical impact on the efficiency of power management techniques. This survey discusses a great variety of such scheduling algorithms that reduce the energy consumption of real-time systems by either decreasing the speed (speed scaling), or by turning devices off (sleep modes). We also

argued that many of these speed scaling algorithms minimize the peak power consumption, although they are designed to minimize the energy consumption. Furthermore, we pointed out that many power management algorithms rely on the same theoretical concepts. Therefore, we did not only survey algorithms, but also the fundamental ideas behind these algorithms.

As many papers on algorithmic power management do not consider several important architectural details, there is a gap between theory and practice. Therefore, in this survey we gave a short overview of some of these aspects, and how they can be modeled or treated. An example of such an aspect is nonuniform power, which is rarely mentioned in the theoretical literature.

Another important aspect missing in the theoretic literature is the interaction between global and local speed scaling ("voltage and frequency islands"). These hybrids of local and global speed scaling, and multiprocessor discrete speed scaling are—in our view—the major theoretical challenges that need to be addressed in the near future.

Acknowledgments This work is supported through NWO Project EASY.

References

Albers, S. (2010). Energy-efficient algorithms. *Communications of the ACM*, *53*(5), 86–96. doi:10.1145/1735223.1735245.

Albers, S., & Antoniadis, A. (2014). Race to idle: New algorithms for speed scaling with a sleep state. *ACM Transactions on Algorithms*, *10*(2), 9:1–9:31. doi:10.1145/2556953.

Albers, S., Antoniadis, A., & Greiner, G. (2011). On multi-processor speed scaling with migration: Extended abstract. In: *Proceedings of the 23rd ACM symposium on parallelism in algorithms and architectures, ACM, New York, NY, USA, SPAA '11* (pp. 279–288). doi:10.1145/1989493.1989539.

Albers, S., Müller, F., & Schmelzer, S. (2014). Speed scaling on parallel processors. *Algorithmica*, *68*(2), 404–425. doi:10.1007/s00453-012-9678-7.

Alon, N., Azar, Y., Woeginger, G.J., & Yadid, T. (1997). Approximation schemes for scheduling. In: *Proceedings of the 8th annual ACM-SIAM symposium on discrete algorithms, society for industrial and applied mathematics, SODA '97* (pp. 493–500). Philadelphia, PA. http://dl.acm.org/citation.cfm?id=314161.314371.

Angel, E., Bampis, E., & Chau, V. (2012a). Low complexity scheduling algorithm minimizing the energy for tasks with agreeable deadlines. In: D. Fernández-Baca (Ed.) LATIN 2012: Theoretical informatics. Lecture Notes in Computer Science, vol 7256 (pp. 13–24). Springer, Berlin. doi:10.1007/978-3-642-29344-3_2.

Angel, E., Bampis, E., Kacem, F., & Letsios, D. (2012b). Speed scaling on parallel processors with migration. In: Kaklamanis C, Papatheodorou T, & Spirakis P (eds) Euro-Par 2012 parallel processing. Lecture Notes in Computer Science, vol. 7484 (pp. 128–140). Springer, Berlin. doi:10.1007/978-3-642-32820-6_15

Angel, E., Bampis, E., & Chau, V. (2014). Low complexity scheduling algorithms minimizing the energy for tasks with agreeable deadlines. *Discrete Applied Mathematics*, *175*, 1–10. doi:10.1016/j.dam.2014.05.023.

Antoniadis, A., & Huang, C. C. (2013). Non-preemptive speed scaling. *Journal of Scheduling*, *16*(4), 385–394. doi:10.1007/s10951-013-0312-6.

Antoniadis, A., Huang, C.C., & Ott, S. (2015). A fully polynomial-time approximation scheme for speed scaling with sleep state. In: *Proceedings of the 26th annual ACM-SIAM symposium on discrete algorithms, SIAM, SODA '15* (pp. 1102–1113). http://dl.acm.org/citation.cfm?id=2722129.2722203.

Augustine, J., Irani, S., & Swamy, C. (2008). Optimal power-down strategies. *SIAM Journal on Computing*, *37*(5), 1499–1516. doi:10.1137/05063787X.

Bampis, E., Dürr, C., Kacem, F., & Milis, I. (2012). Speed scaling with power down scheduling for agreeable deadlines. *Sustainable Computing: Informatics and Systems*, *2*(4), 184–189. doi:10.1016/j.suscom.2012.10.003.

Bampis, E., Letsios, D., & Lucarelli, G. (2012b). Green scheduling, flows and matchings. In: Chao KM, Hsu TS, Lee DT (eds) Algorithms and computation. Lecture Notes in Computer Science, vol. 7676 (pp. 106–115). Springer, Berlin. doi:10.1007/978-3-642-35261-4_14.

Bampis, E., Kononov, A., Letsios, D., Lucarelli, G., & Sviridenko, M. (2014a). Energy efficient scheduling and routing via randomized rounding (pp. 1–27). arXiv:1403.4991.

Bampis, E., Letsios, D., & Lucarelli, G. (2014b). Speed-scaling with no preemptions. In: H.K. Ahn & C.S. Shin (eds) Algorithms and computation. Lecture Notes in Computer Science, vol. 8889 (pp. 259–269). Springer, Berlin. doi:10.1007/978-3-319-13075-0_21.

Bampis, E., Kononov, A., Letsios, D., Lucarelli, G., & Nemparis, I. (2015). From preemptive to non-preemptive speed-scaling scheduling. *Discrete Applied Mathematics*, *181*, 11–20. doi:10.1016/j.dam.2014.10.007.

Bansal, N., Kimbrel, T., & Pruhs, K. (2007). Speed scaling to manage energy and temperature. *Journal of ACM*, *54*(1), 3:1–3:39. doi:10.1145/1206035.1206038.

Bansal, N., Chan, H. L., & Pruhs, K. (2013). Speed scaling with an arbitrary power function. *ACM Transactions on Algorithms*, *9*(2), 18:1–18:14. doi:10.1145/2438645.2438650.

Baptiste, P., Chrobak, M., & Dürr, C. (2012). Polynomial-time algorithms for minimum energy scheduling. *ACM Transactions on Algorithms*, *8*(3), 26:1–26:29. doi:10.1145/2229163.2229170.

Benini, L., Bogliolo, A., & De Micheli, G. (2000). A survey of design techniques for system-level dynamic power management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *8*(3), 299–316. doi:10.1109/92.845896.

Bingham, B.D., & Greenstreet, M.R. (2008). Energy optimal scheduling on multiprocessors with migration. In: *International symposium on parallel and distributed processing with applications, ISPA '08* (pp. 153–161). doi:10.1109/ISPA.2008.128.

Boyd, S., & Vandenberghe, L. (2004). *Convex optimization*. New York: Cambridge University Press.

Bunde, D.P. (2006). Power-aware scheduling for makespan and flow. In: *Proceedings of the 18th annual ACM symposium on parallelism in algorithms and architectures, SPAA '06* (pp. 190–196). ACM, New York. doi:10.1145/1148109.1148140.

Chaparro, P., González, J., Magklis, G., Qiong, C., & González, A. (2007). Understanding the thermal implications of multi-core architectures. *IEEE Transactions on Parallel and Distributed Systems*, *18*(8), 1055–1065. doi:10.1109/TPDS.2007.1092.

Chen, J.J., & Kuo, C.F. (2007). Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms. In:

13th IEEE international conference on embedded and real-time computing systems and applications, RTCSA 2007 (pp. 28–38). doi:10.1109/RTCSA.2007.37.

Chen, J.J., Hsu, H.R., Chuang, K.H., Yang, C.L., Pang, A.C., & Kuo, T.W. (2004). Multiprocessor energy-efficient scheduling with task migration considerations. In: *Proceedings of 16th Euromicro conference on real-time systems, ECRTS 2004* (pp. 101–108). doi:10.1109/EMRTS.2004.1311011.

Cho, S., & Melhem, R. G. (2010). On the interplay of parallelization, program performance, and energy consumption. *IEEE Transactions on Parallel and Distributed Systems*, *21*(3), 342–353. doi:10.1109/TPDS.2009.41.

Cohen-Addad, V., Li, Z., Mathieu, C., & Milis, I. (2015). Energy-efficient algorithms for non-preemptive speed-scaling. In: E. Bampis, & O. Svensson (Eds.) Approximation and online algorithms. Lecture Notes in Computer Science, vol. 8952 (pp. 107–118). Springer, Berlin. doi:10.1007/978-3-319-18263-6_10.

Devadas, V., & Aydin, H. (2012). On the interplay of voltage/frequency scaling and device power management for frame-based real-time embedded applications. *IEEE Transactions on Computers*, *61*(1), 31–44. doi:10.1109/TC.2010.248.

Gerards, M., Hurink, J., Holzenspies, P., Kuper, J., & Smit, G. (2014). Analytic clock frequency selection for global DVFS. In: *22nd Euromicro international conference on parallel, distributed and network-based processing (PDP)* (pp. 512–519). doi:10.1109/PDP.2014.103.

Gerards, M. E. T., & Kuper, J. (2013). Optimal DPM and DVFS for frame-based real-time systems. *ACM Transactions on Architecture and Code Optimization*, *9*(4), 41:1–41:23. doi:10.1145/2400682.2400700.

Gerards, M. E. T., Hurink, J. L., & Kuper, J. (2015). On the interplay between global DVFS and scheduling tasks with precedence constraints. *IEEE Transactions on Computers*, *64*(6), 1742–1754. doi:10.1109/TC.2014.2345410.

Graham, R. L., Lawler, E. L., Lenstra, J. K., & Rinnooy Kan, A. H. G. (1977). Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics v5*, *5*, 287–326. doi:10.1016/S0167-5060(08)70356-X.

Greiner, G., Nonner, T., & Souza, A. (2014). The bell is ringing in speed-scaled multiprocessor scheduling. *Theory of Computing Systems*, *54*(1), 24–44. doi:10.1007/s00224-013-9477-9.

Hsu, C.H., & Feng, W.C. (2005). When discreteness meets continuity: Energy-optimal DVS scheduling revisited. Tech. Rep. LA-UR 05-3104, Los Alamos National Laboratory, http://sss.cs.vt.edu/pubs/tr05-3104.pdf.

Huang, C.C., & Ott. S. (2014). New results for non-preemptive speed scaling. In: E. Csuhaj-Varj, M. Dietzfelbinger, & Z. Sik (Eds.) Mathematical foundations of computer science 2014. Lecture Notes in Computer Science, vol. 8635 (pp. 360–371). Springer, Berlin. doi:10.1007/978-3-662-44465-8_31.

Huang, W., & Wang, Y. (2009). An optimal speed control scheme supported by media servers for low-power multimedia applications. *Multimedia Systems*, *15*(2), 113–124. doi:10.1007/s00530-009-0153-5.

Irani, S., & Pruhs, K. R. (2005). Algorithmic problems in power management. *SIGACT News*, *36*(2), 63–76. doi:10.1145/1067309.1067324.

Irani, S., Shukla, S., & Gupta, R. (2007). Algorithms for power savings. *ACM Transactions on Algorithms*, *3*(4), 41:1–41:23. doi:10.1145/1290672.1290678.

Ishihara, T., & Yasuura, H. (1998). Voltage scheduling problem for dynamically variable voltage processors. In: *Proceedings of the 1998 international symposium on low power electronics and design, ISLPED '98* (pp. 197–202). ACM, New York. doi:10.1145/280756.280894.

Jejurikar, R., Pereira, C., & Gupta, R. (2004). Leakage aware dynamic voltage scaling for real-time embedded systems. In: *41st Proceedings of design automation conference, DAC '04* (pp. 275–280). ACM, New York. doi:10.1145/996566.996650.

Kalla, R., Sinharoy, B., Starke, W. J., & Floyd, M. (2010). Power 7: IBM's next-generation server processor. *IEEE Micro*, *30*(2), 7–15. doi:10.1109/MM.2010.38.

Kandhalu, A., Kim, J., Lakshmanan, K., & Rajkumar, R.R. (2011). Energy-aware partitioned fixed-priority scheduling for chip multiprocessors. In: *17th international conference on embedded and real-time computing systems and applications*, vol. 1, (pp. 93–102). IEEE Computer Society, Los Alamitos. doi:10.1109/RTCSA.2011.75.

Kwon, W. C., & Kim, T. (2005). Optimal voltage allocation techniques for dynamically variable voltage processors. *ACM Transactions on Embedded Computing Systems*, *4*(1), 211–230. doi:10.1145/1053271.1053280.

Lee, J., Yun, B., & Shin, K. G. (2014). Reducing peak power consumption inmulti-core systems without violatingreal-time constraints. *IEEE Transactions on Parallel and Distributed Systems*, *25*(4), 1024–1033. doi:10.1109/TPDS.2013.131.

Lee, S., & Kim, J. (2010). Using dynamic voltage scaling for energy-efficient flash-based storage devices. In: *SoC Design Conference (ISOCC), 2010 International* (pp. 63–66). doi:10.1109/SOCDC.2010.5682971.

Li, K. (2012). Scheduling precedence constrained tasks with reduced processor energy on multiprocessor computers. *IEEE Transactions on Computers*, *61*(12), 1668–1681. doi:10.1109/TC.2012.120.

Li, M., Liu, B., & Yao, F. (2006a). Min-energy voltage allocation for tree-structured tasks. *Journal of Combinatorial Optimization*, *11*(3), 305–319. doi:10.1007/s10878-006-7910-6.

Li, M., Yao, A. C., & Yao, F. F. (2006b). Discrete and continuous min-energy schedules for variable voltage processors. *Proceedings of the National Academy of Sciences of the United States of America*, *103*(11), 3983–3987. doi:10.1073/pnas.0510886103.

Li, M., Yao, F. F., & Yuan, H. (2014). An $O(n^2)$ algorithm for computing optimal continuous voltage schedules (pp. 1–12). arXiv:1408.5995.

Liu, X., Shenoy, P., & Gong, W. (2004). A time series-based approach for power management in mobile processors and disks. In: *Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video—NOSSDAV '04* (pp. 74–79). doi:10.1145/1005847.1005864.

Manoj, P.D.S., Wang, K., & Yu, H. (2013). Peak power reduction and workload balancing by space-time multiplexing based demand-supply matching for 3d thousand-core microprocessor. In: *Proceedings of the 50th annual design automation conference, DAC '13* (pp. 175:1–175:6). ACM, New York. doi:10.1145/2463209.2488950.

March, J. L., Sahuquillo, J., Hassan, H., Petit, S., & Duato, J. (2011). A new energy-aware dynamic task set partitioning algorithm for soft and hard embedded real-time systems. *The Computer Journal*, *54*(8), 1282–1294. doi:10.1093/comjnl/bxr008.

Park, S., Park, J., Shin, D., Wang, Y., Xie, Q., Pedram, M., et al. (2013). Accurate modeling of the delay and energy overhead of dynamic voltage and frequency scaling in modern microprocessors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *32*(5), 695–708. doi:10.1109/TCAD.2012.2235126.

Pruhs, K. (2011). Green computing algorithmics. In: *IEEE 52nd annual symposium on foundations of computer science (FOCS)* (pp. 3–4). doi:10.1109/FOCS.2011.44.

Pruhs, K., van Stee, R., & Uthaisombut, P. (2008). Speed scaling of tasks with precedence constraints. *Theory of Computing Systems*, *43*(1), 67–80. doi:10.1007/s00224-007-9070-1.

Quan, G., & Hu, X. S. (2003). Minimal energy fixed-priority scheduling for variable voltage processors. *IEEE Transactions on Computer-*

Aided Design of Integrated Circuits and Systems, *22*(8), 1062–1071. doi:10.1109/TCAD.2003.814948.

Rountree, B., Lowenthal, D.K., Funk, S., Freeh, V.W., de Supinski, B.R., & Schulz, M. (2007). Bounding energy consumption in large-scale MPI programs. In: *Proceedings of the 2007 ACM/IEEE conference on supercomputing—SC '07*. ACM, New York (pp. 49:1–49:9). doi:10.1145/1362622.1362688.

Weiser, M., Welch, B., Demers, A., & Shenker, S. (1996). Scheduling for reduced CPU energy. In: T. Imielinski & H.F. Korth (Eds.) Mobile computing. The Kluwer International Series in Engineering and Computer Science, vol. 353 (pp. 449–471). Springer, New York. doi:10.1007/978-0-585-29603-6_17.

Wu, W., Li, M., & Chen, E. (2011). Min-energy scheduling for aligned jobs in accelerate model. *Theoretical Computer Science*, *412*(12–14), 1122–1139. doi:10.1016/j.tcs.2010.12.013.

Yao, F., Demers, A., & Shenker, S. (1995). A scheduling model for reduced CPU energy. In: *Proceedings of IEEE 36th annual foundations of computer science* (pp. 374–382). doi:10.1109/SFCS.1995.492493.

Yun, H. S., & Kim, J. (2003). On energy-optimal voltage scheduling for fixed-priority hard real-time systems. *ACM Transactions on Embedded Computing Systems*, *2*(3), 393–430. doi:10.1145/860176.860183.

Zhang, D., Guo, D., Chen, F., Wu, F., Wu, T., Cao, T., et al. (2012). TL-plane-based multi-core energy-efficient real-time scheduling algorithm for sporadic tasks. *ACM Transactions on Architecture and Code Optimization*, *8*(4), 47:1–47:20. doi:10.1145/2086696.2086726.

Zhuravlev, S., Saez, J. C., Blagodurov, S., Fedorova, A., & Prieto, M. (2013). Survey of energy-cognizant scheduling techniques. *IEEE Transactions on Parallel and Distributed Systems*, *24*(7), 1447–1464. doi:10.1109/TPDS.2012.20.