Date of acceptance Grade

Instructor

Power Aware Real Time Scheduling in Constrained Devices

Teemu Kemppainen

Helsinki August 30, 2007

Master's Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta/Osasto — Fakultet/Sektion — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Teemu Kemppainen

Työn nimi — Arbetets titel — Title
Power Aware Real Time Scheduling in Constrained Devices

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's Thesis	August 30, 2007	59 + 1 pages

Tiivistelmä — Referat — Abstract

Real-time scheduling algorithms, such as Rate Monotonic and Earliest Deadline First, guarantee that calculations are performed within a pre-defined time. As many real-time systems operate on limited battery power, these algorithms have been enhanced with power-aware properties. In this thesis, 13 power-aware real-time scheduling algorithms for processor, device and system-level use are explored.

ACM Computing Classification System (CCS):
D.4.1 [Operating systems],
J.7 [Computers in other systems]

Avainsanat — Nyckelord — Keywords
scheduling, real-time systems, power-awareness

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library, serial number C-

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction

In a conventional computer system, the correctness of calculations is defined by their logical correctness. A *real-time system* has been defined as a system where the calculations need not only be correct, but also be finished within a pre-defined time [RaS94]. Real-time systems are today used in a wide variety of computing devices: in medical systems, in ABS brake system of vehicles, in Global Positioning System (GPS) devices, multimedia devices like DVD and MP3 players, in mobile phones, among others. Many of these systems are constrained devices functioning on limited battery power. Here, the usability of the device is greatly dependent upon the operational lifetime of the battery.

A *scheduler* is an operating system component responsible for sharing a resource among multiple users. A scheduler decides which process may use the processor at a particular moment. Common scheduling algorithms are for instance Round-Robin, where processes are ordered in a circular queue, and CPU time is given to each process in turn [Sta05, page 791]. Another approach is First In First Out (FIFO) scheduling, where the process that has been in the queue for the longest time will be given CPU time first.

In real-time systems, these commonplace scheduling methods cannot be used since they do not guarantee meeting the time boundaries of real-time processes. Therefore, real-time systems need special schedulers that take deadlines into account. The research in real-time scheduling seriously began in the early 1970's. In 1973, Liu and Layland published their two famous real-time scheduling algorithms, Earliest Deadline First (EDF) and Rate-Monotonic (RM) [LiL73]. EDF is based upon dynamic priorities, while in RM processes have fixed priorities. Basically all of today's real-time implementations are based upon one of these two algorithms.

In a *hard real-time system*, a task must always finish before its deadline. The most demanding area of hard real-time systems are systems where human lives are at stake. Examples include medical systems like pacemakers, military systems, and for instance nuclear power plants. Here, bulletproof evidence that the system will meet its deadlines are required. The missing of even a single deadline is unacceptable. In contrast, in a *soft real-time system* (Section 2.3) the deadline is of a somewhat more relative nature. In a multimedia system, for instance in a video decoder, it might be sufficent to guarantee that 95 percent of frames are timely decoded. Occasional out-of-sync frames are acceptable in an application of this kind.

Many processors and devices designed for portable use provide several different operational states. Besides its high power and speed state, the processor can be run at a lower speed, which provides lower throughput, but consumes less energy. At times when the processor is not needed at all, it can be put into a sleep mode which virtually consumes no energy at all. Techniques for adjusting device throughput and power consumption are for instance *Dynamic Voltage Scaling* (DVS) [VeF05, PLS01, VBH03] and *Advanced Configuration & Power Interface* (ACPI) [HIM06]. These techniques allow the operating system to change the operating frequency and voltage of the processor and other devices at run-time in order to save energy. The introduction of run-time voltage scaling has opened new possibilities even for real-time systems in constrained devices.

The most straightforward energy saving solution is to set the processor and/or disk into sleep mode after a period of user inactivity [BBC98]. Information from previous process invocations can be used to estimate the length of the sleep interval [HwA00]. Even more complex statistical methods based on use history can be used to estimate when the device will be needed next time [IGS02]. As such, none of these methods are usable in real-time systems with hard deadlines [SwC05]. The implementation of energy awareness in real-time systems is a more complex task. The waking up of the processor, disk or other device from sleep mode always introduces a certain time penalty. The device is not instantly usable but requires some time to restart. In real-time systems this wake-up delay risks missing deadlines and, therefore, needs special attention from the scheduler.

The solution is to implement energy-conserving properties into EDF and RM based real-time schedulers. Reports indicate that such techniques have provided energy savings of up to 50% [SwC03] while still guaranteeing meeting of real-time process deadlines.

This thesis describes 13 power-aware scheduling algorithms usable in constrained devices with limited battery resources. The theoretical background and terminology of real-time scheduling with RM and EDF is described in Section 2, and power-aware properties in constrained devices are discussed in Section 3. Recent energy conserving processor scheduling algorithms are presented in Section 4, and device scheduling algorithms in Section 5. The thesis is summarized in Section 7.

2 Real Time Scheduling

Real-time scheduling algorithms are responsible for sharing resources among users while guaranteeing timely execution of real-time processes. In order to present real-time scheduling algorithms, we will first introduce a system model used throughout the rest of the thesis.

2.1 System Model

A *task* is a process, a piece of independently running software code. We use the notation T_i to indicate a task, where i is the task's distinctive number. One instance of a task is called a *job*. In real-time systems, tasks typically have a *period*, a time interval between which individual jobs of the task are released for execution. We mark the period P_i. A job of task T_i in period k is marked with $J_{i,k}$. By *release time* we mean the time at which $J_{i,k}$ becomes ready for execution.

By *deadline* we mean the time when a job needs to be completed. We indicate this time D_i. A deadline relative to the current time is marked d_i. For instance, if J_i has $D_i = 20$ and the current time is 15, then $d_i = 5$.

By the *execution time*, indicated by E_i, we mean the worst case execution time of J_i: the amount of processor time needed by the job to complete. In reality, execution times of individual jobs $J_{i,k}$ vary greatly. Consider, for example, a real-time system controlling a robotic arm that is removing faulty products from a composition line. When there are no faulty products, jobs will complete extremely fast as the arm does not need moving at all. But for scheduling reasons, we must expect the worst case execution time. In the case of the robotic arm, this would mean the (hopefully rare) event when all products within the arm's range are faulty, and need to be removed from the line.

Let e_i be one instantaneous execution time of J_i, where $e_i \leq E_i$. By *slack time* we mean the time $E_i - e_i$, i.e. time allocated for process execution that is not actually needed because the job finishes earlier than budgeted. This time can be utilized for energy savings. We will return to this later.

The *utilization degree* of a task is calculated by E_i/P_i. The utilization of the entire task set is calculated using Equation 1:

$$U = \sum_{i=1}^{n} \frac{E_i}{P_i} \quad (1)$$

where n is the number of tasks. In later parts of the paper we may describe a task (P_i, E_i). For instance $(6, 3)$ means a task with period 6 and execution time 3, and $(6, 1)$ indicates a task with period 6 and execution time 1. The utilization of a task set consisting of these two tasks would be $\frac{3}{6} + \frac{1}{6} = \frac{4}{6} = \frac{2}{3}$, according to Equation 1. If no deadline is explicitly mentioned, then $d_i = P_i$, meaning that the deadline of the task equals its period. Intuitively this means, that a job of the task must be completed before the release of the next job.

2.2 Hard Real Time Scheduling

The fundamental real-time scheduling algorithms are Rate Monotonic (RM) and Earliest Deadline First (EDF) [LiL73]. Neither of these algorithms provide power-awareness, but all of the energy conscious scheduling solutions presented later in this thesis are enhancements of either RM or EDF. Therefore, an insight into RM and EDF is essential for understanding this thesis. Both RM and EDF will be presented in this section.

2.2.1 Rate Monotonic

```
input:  list of tasks
1       repeat on task set:
2          perform RM schedulability test;
3          if fail alarm OS;
4          else
5             sort jobs in ascending order according to period;
6             while (jobs left):
7                   schedule first job from list;
8                   remove finished job from list;
```

Figure 1: Pseudo code of the Rate Monotonic algorithm.

In the Rate Monotonic scheduling algorithm, the task with the shortest period P_i gets highest priority, and is scheduled first. Because periods of tasks are constant,

RM is a fixed-priority scheduler. Liu and Layland [LiL73] have shown that the schedulability condition for RM is that of Equation 2:

$$U \leq n(2^{1/n} - 1) \tag{2}$$

where n is the amount of tasks. For instance, when $n = 2$, i.e. with two tasks, RM is able to schedule the tasks if their $U \leq 0,83$. A task set consisting of 4 tasks is schedulable with RM if $U \leq 0,76$. With large numbers of n:

$$\lim_{n \to \infty} n(2^{1/n} - 1) = \ln 2 \tag{3}$$

The idea in Equation 3 is, that with large task sets, the RM schedulability condition approaches the value $\ln 2$, i.e. approximately $0,69$. The theoretical maximal utilization, which also the Earliest Deadline First algorithm accomplishes, is $U = 1$. In other words, RM as such cannot be considered very efficient.

Let us consider a sample RM schedule using a task set consisting of two tasks: τ_1=(5,2) and τ_2=(7,4). First, RM considers the schedulability of this task set. According to Equation 1, U of this task set is $\frac{2}{5} + \frac{4}{7} = \frac{34}{35}$, i.e. approximately $0,97$. According to Equation 2, the promised usage level that RM is guaranteed to be able to schedule when $n = 2$ is $U \leq 0,83$. Therefore it seems that this task set is not schedulable with RM. The scheduler might alert the operating system of this according to line 3 in the pseudo code in Figure 1. Let us, however, more closely consider the functionality of RM by simulating lines 5–8 of the RM algorithm on the before mentioned task set. The results are shown in Figure 2.

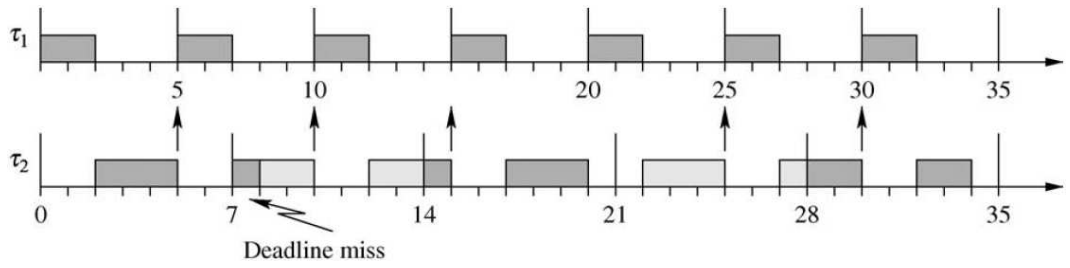


Figure 2: Tasks τ_1=(5,2) and τ_2=(7,4) scheduled using Rate Monotonic [But05].

The period of τ_1 is 5 and the period of τ_2 is 7. In RM the task with the shortest period gets highest priority. Therefore, τ_1 is scheduled first. According to the pseudo code in Figure 1, this operation is done by sorting the processes in a list according

to their periods, as seen on line 5. The *while* condition on line 6 is *true* so the algorithm advances to line 7. The first job on the list is τ_1, so it is scheduled first. Every 5 time units, τ_1 is scheduled 2 units of time. This can be seen in Figure 2. Having scheduled the highest priority task and removed it from the list (line 8), RM now proceeds to schedule the next task, since the *while* condition on line 6 is *true*. Here, τ_2 requires 4 units of CPU time every 7 units. However, in period 1 there is only 3 units of time available in the interval [0,7]. The time interval [2,5] is allocated to τ_2. At time 5 a context switch occurs, and the higher priority process τ_1 gets the CPU. This is indicated by an up-arrow in Figure 2. Because τ_1 has the processor during [5,7], τ_2 doesn't get a chance to finish its one remaining execution time unit, and $J_{2,1}$ misses its deadline at time 7. This simulation hence verifies the failed RM schedulability condition: this task set is not schedulable using RM.

2.2.2 Earliest Deadline First

```
input:  list of tasks
1       repeat:
2         perform EDF schedulability test;
3         if fail alarm OS;
4         else do while (jobs left AND no new task released):
5             put job with closest deadline first in list;
6             schedule first job;
7             remove finished job from list;
```

Figure 3: pseudo code of the Earliest Deadline First algorithm.

In the Earliest Deadline First algorithm the process with the deadline closest to the current time gets scheduled first. Because the process with the closest deadline changes as execution progresses, the EDF method leads to dynamic priorities. In EDF, the schedulability condition is:

$$U \leq 1 \tag{4}$$

This means, that EDF accomplishes full resource utilization while guaranteeing timeliness. The pseudo code of the EDF algorithm can be seen in Figure 3.

Let us consider the tasks τ_1=(5,2) and τ_2=(7,4) scheduled using Earliest Deadline First according to the pseudo code in Figure 3. On line 2, the EDF schedulability

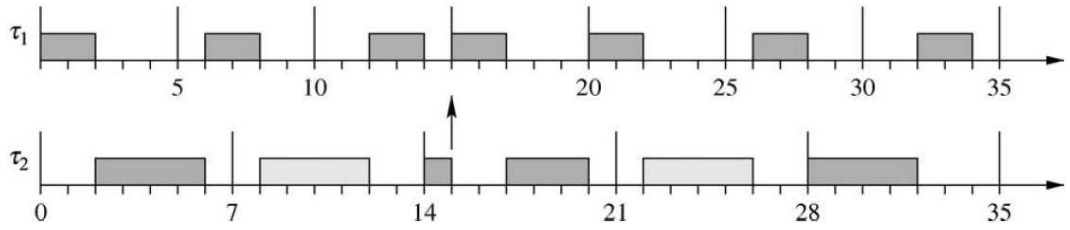


Figure 4: Tasks τ_1=(5,2) and τ_2=(7,4) scheduled using Earliest Deadline First [But05].

test is performed. According to Equation 1, $U = \frac{34}{35}$. Because the EDF schedulability condition (Equation 4) guarantees schedulability when $U \leq 1$, this task set is schedulable using EDF. The *while* condition on line 4 is true. EDF orders the tasks according to their relative deadlines. At time 0, the job with the closest deadline is τ_1, so it gets scheduled first. At its finish time at 2, τ_2 gets scheduled. Once τ_2 is finished at 6, the second job of τ_1 has been released, and is scheduled. After execution of the third job of τ_1, at time 14, τ_2 with deadline 21 get scheduled for one unit of time, but is switched out at time 15: here, the fourth job of τ_1 is released, and since its deadline is $20 \leq 21$, τ_1 gets higher priority than τ_2. Once a job is finished, it is removed from the list of jobs.

2.3 Soft Real Time Scheduling

In a *soft real-time system* the timing constraints are somewhat more relaxed than in a hard real-time system. A soft-real time application usually provides a probabilistic guarantee of p% of tasks meeting their deadlines. For instance a telephone network might be considered a soft real-time application. It will be considered usable if 95% of calls are connected within 10 seconds, and within 20 seconds for 99,95% of calls [Liu00, page 31].

The video viewing experience or enjoyability of a computer game is not spoiled if one or two frames per minute miss their deadline. Multimedia is a a very common area for soft real-time systems. Consider for instance the EScheduler [YuN06] algorithm, presented in Section 4.2.1. It calculates the actual CPU time demand of n recent jobs of task T_i. Based on this usage history, it uses as E_i (Equation 1) a value below of which p% of the considered jobs remain. Hence, it allocates enough CPU time so that p% of jobs will complete timely (assuming that the CPU demand distribution of the task is pretty stable). This is a very typical real-time guarantee that suffices for

a soft real-time application. The use of a soft real-time scheduler instead of a hard one might be motivated if, for instance, the response time of the system improve when real-time constraints are relaxed.

3 Power Awareness in Constrained Devices

By *energy*, measured in *joule*, we mean the total amount of work done during a period of time, and by *power* we mean the rate at which the work is done. Power is measured in *watts* [VeF05].

Consider a task that takes 5 seconds to finish with a CPU running at 100 MHz. Lowering the CPU speed to 50 MHz will decrease the power consumption of the processor, as lower frequencies need less power. However, the total energy needed to complete the task will not be reduced, as the task will take a longer time to finish, perhaps even twice the time. Actually, lowering only the speed of the CPU often might increase the total energy consumed by the entire system, as for instance hard disks, network adapters and other components need to be powered-up for longer periods of time. This aspect is more closely considered in Section 6.

In some cases, for instance to cool down a processor, it is desirable to lower the power consumption without considering the total need of energy [VeF05]. This kind of power reduction is, however, hardly what we wish to accomplish when using battery powered constrained devices: here, minimizing the total energy need is what matters.

Calculating and minimizing the system's total energy consumption depends on the actual system configuration. This question has been researched by for instance Zhuo and Chakrabarti [ZhC05]. In Sections 4 and 5 of this thesis, we focus on minimizing the power consumption of distinct components. The reader should note that this chosen view is a simplified one, as in reality systems are composed of multiple components.

3.1 Dynamic Voltage Scaling

Contemporary microchips are based on the *CMOS* (complementary metal-oxide-semiconductor) technology. Chips using this technology consume energy both *dynamically* and *statically* [VeF05]. The static power consumption is caused by current flowing through the transistors even when they are turned off. As this form of energy consumption cannot be altered during run-time by the scheduler, it is not of interest in this thesis.

The dynamic power consumption consists of two parts. About one tenth of a chip's power consumption is caused by instantaneous short-circuiting of transistors as they

switch states [VeF05]. Currently it is unknown how to combat this energy waste, and so we will disregard this form of dynamic power consumption. Most of the processor's dynamic power consumption can, however, be adjusted during run-time, and this is where we will focus our attention. Let P be the dynamic power consumption of a processor. The following equation indicates how it is formed [PLS01]:

$$P = C \times f \times V^2 \tag{5}$$

here, C is the capacitance of the transistors. This is a fixed value caused by the physical structure of the processor. The value f is the operating frequency of the processor. It is usually measured in megahertz or gigahertz. Adjusting the operating frequency of the processor linearly affects power consumption. The operating voltage of the chip is indicated by V. As seen in Equation 5, adjusting the voltage affects power consumption quadratically.

From Equation 5 it follows that the processor's power consumption can be regulated during run-time by adjusting its operation frequency f, voltage V, or both. Technology for accomplishing this is called *Dynamic Voltage Scaling* (DVS). The abbreviations DFS (dynamic frequency scaling) and DVFS (dynamic voltage-frequency scaling) are also used [VBH03].

Notice, however, that adjusting only f but not V linearly decreases the power consumed by the processor, but not the total energy needed to complete the task: a CPU operating at m MHz that takes s seconds to finish a task will probably take $2s$ seconds to finish the task at $m/2$ MHz.

Lowering only V might seem tempting, but a lower V generally cannot support a high f, so usually lowering the supply voltage also requires the lowering of the operational frequency. So in DVS both V and f are adjusted: the processor is made both slower and less consuming.

For an example of a real life DVS solution consider the performance states of the 1.6 GHz Pentium M processor presented in table 1. At the maximum speed, 1.6 GHz, the power consumption of the processor according to Equation 5 is $C * 1.6GHz * 1.484V$ and at the lowest speed $C * 600MHz * 0.956$. At lowest frequency and voltage the processor consumes less than one fourth of its maximum power consumption ($\frac{C*600MHz*0.956}{C*1.6GHz*1.484V} = 0.24$), while still providing 38% of the maximum computing performance ($\frac{600MHz}{1.6GHz} = 0.375$). The early Transmeta Crusoe processor provided even more impressive power savings, as seen in table 2. The Crusoe provided 29% of the

Frequency	Voltage
1.6 GHz (HFM)	1.484 V
1.4 GHz	1.420 V
1.2 GHz	1.276 V
1.0 GHz	1.164 V
800 MHz	1.036 V
600 MHz (LFM)	0.956 V

Table 1: DVS performance states of the 1.6 GHz Intel Pentium M processor [Int04].

Frequency f (MHz)	Voltage V (V)	Relative power (%)
700	1.65	100
600	1.60	80.59
500	1.50	59.03
400	1.40	41.14
300	1.25	24.60
200	1.10	12.70

Table 2: DVS states of the Transmeta TM5400 "Crusoe" processor [PLS01].

maximum throughput (200 out of 700 MHz) while consuming less than 13% of the maximum power.

For scheduling needs, DVS can be utilized basically in three different ways. These are compared in table 3. The simplest method is the *interval-based* approach [VeF05], in which the CPU frequency and voltage are adjusted downwards if the CPU utilization during the past t time units has been low, and upwards if the CPU utilization has been high. The value of t is critical. If t is too short, the CPU fequency and voltage may be adjusted back and forth causing high overhead. On the other hand, large t values may compromise efficiency as DVS adjustments are made very seldom. The interval-based method can be enhanced by considering a window of intervals. However, the interval-based method is not suitable for use in real-time systems as it does not take into consideration the deadlines of individual tasks.

The *inter-task* approach [VeF05] considers a distinct DVS value for each task and, therefore, suits well the needs of real-time applications. Voltage and frequency settings are altered at context switches and remain fixed during the execution of the

Method name	DVS occasions	Real-time suitable	Complexity
Interval-based	At threshold time intervals	No	Low
Inter-task	Context switches	Yes	Medium
Intra-task	Context switches and during task execution	Yes	High

Table 3: Comparison of fundamental DVS techniques.

entire task. The advantage of the inter-task approach over the interval-based is that each task may receive an individually suitable DVS setting. However, the execution time allocated for a task generally is much higher than the actual execution time. Using the inter-task approach, the entire task is run with the same DVS value, which in most cases can be unnecessarily high. Therefore, the power savings achieved by this method often are not optimal.

The most advanced DVS method used in real-time systems is the *intra-task* approach [VeF05]. Here DVS values may be changed even during a task execution. Algorithms utilizing this method are, for instance, Feedback DVS-EDF [DMZ02] and EScheduler [YuN06], presented in Sections 4.1.3 and 4.2.1, respectively. For instance the Feedback DVS-EDF algorithm utilizes DVS aggressively. It will divide a task's execution time E_i into two parts, C_a and C_b. During C_a the processor is run at a lowered speed, and only at the start of C_b is the CPU speed increased. Jobs finishing sooner than their budgeted execution time will never reach C_b and the system is saved from this high power execution interval. In EScheduler, the speed schedule is divided into several phases, with each having a slightly different DVS value. The task is initially executed with a low speed, and as execution time progresses, the speed is gradually increased.

3.2 Advanced Configuration and Power Interface

Processor manufacturers have different implementations for their voltage scaling technologies. AMD's technology is named *PowerNow*, Intel's *SpeedStep*, and Transmeta's *LongRun* [PLS01], or more recently, *LongRun2*. ACPI, first introduced by Intel, Microsoft and Toshiba in 1996 [Gro03], is a standardized interface between the hardware and the operating system. The general architecture of ACPI is depicted

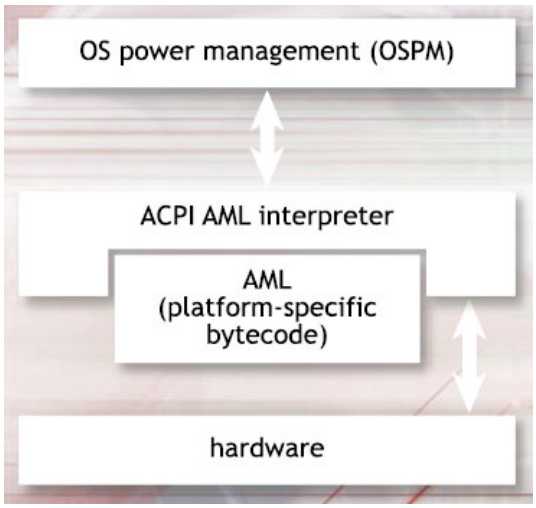


Figure 5: The ACPI provides a standard interface between the operating system and the firmware [Gro03].

in Figure 5.

The main advantage with ACPI is that both hardware and operating system (OS) components may evolve independently of each other while letting the OS fully control the system's power management. The OS may, for instance, choose to bundle disk writes to be executed in batches in order to improve system response times. This kind of functionality is not possible when power management is controlled by hardware alone.

ACPI provides standardized mechanisms for switching between different power conscious states of processors, disk drives, screens, modems, and other components that are used in todays portable computers. Both Windows and Linux platforms support ACPI for CPU frequency scaling. The ACPI design is based on *ASL* (ACPI Source Language) and *AML* (ACPI Machine Language) that reminds quite a lot the Java programming language [Gro03]. The human readable uncompiled Java source code corresponds to ASL in ACPI, whereas Java bytecode corresponds to AML, which is the compiled version of ASL. The idea here is that AML abstracts the platform-specific details from the operating system so that the OS may use standard operation names to access platform-specific features.

The current version of the ACPI specification is 3.0b. This 631 page document was released in October 2006, and is available for download at `http://www.acpi.info`.

4 Power Aware Processor Scheduling

Real-time scheduling algorithms can be divided into processor and device scheduling algorithms. This section covers power-aware real-time scheduling algorithms for CPU scheduling, while device scheduling is covered in Section 5. In this section we focus on uniprocessor systems. The scheduler is responsible for sharing this single CPU between all tasks while guaranteeing that time boundaries are met.

Energy saving is achieved by running the processor at lower speed whenever this speed is sufficent to meet the deadlines. Because the processor's power consumption cubically depends on the clock frequency and voltage (Equation 5, Section 3.1), significant energy consumption reductions can be achieved by lowering the processor's frequency and voltage at occasions when maximum throughput is not needed. Some scheduling algorithms even utilize the *sleep* state of the processor when the system is idle, if such a state is available. For instance, if the scheduler knows that the next periodic job will not be released until time t, it will set a timer to wake up the processor at time t and put the processor to sleep mode.

Lowering the processor speed to save energy works as follows. Suppose that the current job needs to finish at time t. When ran at full speed, the processor will finish the job at time $t/2$. Hence, it suffices to run the processor at half of the maximum speed in order to guarantee timely execution.

Processor scheduling algorithms can be divided into two categories, hard and soft real time scheduling algorithms. We will first study algorithms that provide hard real-time guarantees. These are the strictest type of real-time algorithms: they guarantee that all deadlines are met. All algorithms presented in Section 4.1 are enhancements of either the Rate Monotonic or Earliest Deadline First [LiL73] algorithm. In soft real-time algorithms, occasional deadline misses are allowed. Soft real-time processor scheduling algorithms are explored in Section 4.2.

4.1 Hard Real Time Scheduling

The Rate Monotonic and Earliest Deadline First algorithms as such form an excellent starting point when engineering energy aware real-time schedulers. Most contemporary hard real-time schedulers with energy conserving properties in fact are relatively small enhancements to the RM and EDF techniques. As examples of such algorithms, we will in this subsection explore a number of pseudo codes. The

LPFPS algorithm enhances the Rate Monotonic algorithm, and provides a guaranteed U of $ln2$ as indicated by Equation 3. As examples of energy conscious Earliest Deadline First based schedulers, guaranteeing $U \leq 1$, the LEDF and Extended LEDF algorithms are presented. The most ambitious algorithm that will be considered is Feedback DVS-EDF, which even utilizes a basic form of intra-task DVS, and slack-time passing between jobs. In general, EDF based schedulers are much more common in research papers than their RM based counterparts. This is due to EDF providing full utilization of the processor. RM is, however, simpler to implement in some operating system kernels that do not provide explicit support for the timeliness properties that real-time tasks require [But05].

4.1.1 The Low Power Fixed Priority Scheduling Algorithm

The Low Power Fixed Priority Scheduling (LPFPS) [ShC99] algorithm, published in 1999, is one of the earliest energy conscious scheduling algorithms. It enhances the Rate Monotonic algorithm by taking into account energy conserving properties.

For energy savings, LPFPS utilizes two different occasions. Firstly, in an RM based schedule, there usually are idle times in the schedule. Recall the RM schedulability condition $U \leq n(2^{1/n} - 1)$ of Equation 2: the maximal CPU utilization U of an RM based schedule with large task numbers n approaches the value 0.69. So with high task numbers the maximal RM utilization degree leaves the CPU idle for 30 percent of the time, and LPFPS utilizes this time for energy savings. Secondly, jobs actually often execute faster than budgeted. In other words, jobs rarely use all of the time that has been allocated to them. When a job executes faster than budgeted, the remaining time is used by LPFPS to save energy.

Both voltage and frequency scaling and the powering down of the CPU are supported by LPFPS. When the system is idle, i.e., there are no jobs ready to run, LPFPS places the CPU in a power down mode, and initiates a timer to wake up the processor so that it will be ready for use when it, according to the schedule, is needed next time. When there is only one job left ready to run, LPFPS will calculate an energy conserving voltage and frequency setting for the job, and execute it if possible at a lower CPU speed.

The LPFPS algorithm utilizes two data structures of the type *queue*. Jobs that are ready for execution and wait for processor time are placed in the *run queue*. The job with the highest RM priority (the shortest period) is at the head of the queue.

```
L1:   if current_frequency < maximum_frequency then
L2:       increase the clock frequency and the supply voltage
          to the maximum value;
L3:       exit;
L4:   end if
L5:   while delay_queue.head.release_time ≤ current_time do
L6:       move delay_queue.head to the run_queue;
L7:   end do
L8:   if run_queue.head.priority > active_task.priority then
L9:       set the active_task.executed_time;
L10:      context switch;
L11:  end if
L12:  if run_queue is empty then
L13:      if active_task is null then
L14:          set timer to (delay_queue.head.release_time − wakeup_delay);
L15:          enter power down mode;
L16:      else
L17:          speed_ratio = Compute_speed_ratio();
L18:          find a minimum allowable
              clock frequency ≥ speed_ratio · max_frequency;
L19:          adjust the clock frequency along with the supply voltage;
L20:      end if
L21:  end if
```

Figure 6: pseudo code of LPFPS the scheduling algorithm [ShC99]. Lines L5–L11 correspond to the conventional Rate Monotonic functionality.

In the *delay queue* LPFPS holds tasks whose current jobs are completed, i.e. tasks waiting for the arrival of their next jobs in the next period. The job with the closest arrival time is placed at the head of the delay queue. The job that is currently scheduled for execution is called the *active task*. Conceptually, this task is present in neither of the queues.

The LPFPS pseudo code We are now ready to consider the LPFPS pseudo code of Figure 6. Let us begin by considering lines L5–L11, where the functionality of a conventional RM scheduler is present. On line 5 it is checked whether the current time exceeds or equals the release time of job(s) at the head of the delay queue.

If so, the jobs are moved to the run queue (line 6). If the job now at the head of the run queue has a greater priority, i.e., shorter period, than the active task has (line 8), then a context switch occurs on line 10. This implies that the information belonging to the current active task in the CPU registers and operating system control structures are stored in main memory, and replaced by the information of the new active task. Prior to the context switch, on line 9, LPFPS also stores the amount of time the job has been executed. This value is later used when calculating voltage and frequency scaling parameters.

In addition to the conventional RM scheduler functionality, LPFPS provides energy saving properties. Energy savings will be sought when the run queue is empty, i.e., when there are . This condition is checked on line 12. If the run queue is empty, and there is no active task (line 13), i.e., the processor is idle, then the CPU will be put to power down mode. On line 14 a timer is set to activate the processor so it will be ready for use at the arrival of the next job. In setting the timer LPFPS takes into account the processor wakeup delay time. On line 15 the CPU is put to power down mode.

If the run queue is empty but there is one active task (line 16), LPFPS will calculate an energy saving DVS setting for it and, when possible, execute the task at lower speed and voltage. The new speed ratio is calculated by the *Compute_speed_ratio()* procedure called on line 17. The formula used by LPFPS in calculating the speed ratio is [ShC99]:

$$speed\_ratio = \frac{C_i - E_i}{t_a - t_c}$$

where C_i is the budgeted execution time, E_i the time that has already been spent executing the job, t_a is the arrival time of the next job, and t_c is the current time. In essence, the remaining execution time is divided by the time available before the arrival of the next job. Among the available CPU clock frequencies the lowest one guaranteeing timely execution is located on line 18. The processor frequency and voltage are adjusted on line 19. It should be noted that it is implicitly assumed that $D_i \geq t_a$, where D_i is the absolute deadline of the active job.

4.1.2 Low-Energy EDF and Extended Low-Energy EDF

Where LPFPS, described in the previous subsection, is based on the Rate Monotonic algorithm, we will from here on focus on Earliest Deadline First schedulers. The

pseudo code of an energy conserving EDF based processor scheduling algorithm called Low-Energy EDF (LEDF) is given in Figure 7. This algorithm was published by Swaminathan and Chakrabarty in 2000 [SwC00]. It only supports two distinct CPU speeds, low and high speed. Due to its simplicity, it is an excellent entry point into more complex schedulers.

```
1  Procedure LEDF()
2  begin
3  Repeat forever
4  {
5  Are tasks waiting to be scheduled in ready list?
6       yes
7       Sort deadlines in ascending order
8       Schedule task with earliest deadline
9       Check if deadline can be met at lower speed (voltage)
10      If deadline can be met, schedule task to execute at lower voltage (speed)
11      If deadline cannot be met, check if deadline can be met at higher speed (voltage)
12      If deadline can be met, schedule task to execute at higher voltage (speed)
13      If deadline cannot be met, task cannot be scheduled. Call exception handler!
14      no
15      do-nothing
16 }
17 end
```

Figure 7: The LEDF pseudo code [SwC00].

On line 7 of Figure 7, the jobs currently present are sorted according to their deadlines, and on line 8 the job with the closest deadline is scheduled according to the EDF principle. On line 9, LEDF checks whether or not the job would make its deadline if scheduled at a lower speed and voltage. If so, the job is scheduled at the lower speed. If the job cannot meet its deadline at the lower speed, LEDF checks on line 11 if it can make it with the higher speed, and schedules the task at the higher speed on line 12. If the deadline cannot be met even at higher speed, the exception handler (line 13) is called. It is then up to the operating system to decide what to do with this task.

Extended LEDF The authors of LEDF have improved their algorithm [SwC01]. The *Extended LEDF* (E-LEDF) algorithm given in Figure 8 considers the CPU transition delay when making scheduling decisions. A switch between the high and low speed states always introduces a certain time and energy penalty. The switch *in itself* consumes some energy and takes some time. Very short switches from high

speed state to the low speed state are not worthwhile as the state transition cost would exceed the net gain.

```
1 Procedure E-LEDF()
2 begin
3 Repeat forever
4 Remember speed at which previous task was scheduled
5 if tasks are waiting to be scheduled in ready list then
6   Sort deadlines in ascending order; Select task τi with earliest deadline
7   if this is the first task to be scheduled then
8     If t_low + t_s ≤ d_i, schedule task at lower voltage (speed); continue
9     If t_hi + t_s ≤ d_i, schedule task at higher voltage (speed); continue
10    Call exception handler!!
11  else
12      if previous speed = high speed then
13        Compute E_hi and E_low
14        if task is not schedulable then
15          Call exception handler!!
16        else
17            if t_low + t_s ≤ d_i
18              if E_low ≤ E_hi then
19                If t_low + t_s ≤ d_i, schedule task at low speed
20              else
21                  If t_high + t_s ≤ d_i, schedule at high speed
22            else
23                Schedule task at high speed
24      if previous speed = low speed then
25        Compute E_hi and E_low
26        if task is not schedulable then
27          Call exception handler!!
28        else
29            if t_hi + t_s ≤ d_i then
30              if E_hi ≤ E_low then
31                Schedule task at high speed
32              else
33                If t_low + t_s ≤ d_i, schedule task at low speed
  end
```

Figure 8: Pseudo code of the E-LEDF scheduler enhancing LEDF [SwC01]. Syntax: t_{low} and t_{hi}: execution time with low / high CPU speed, respectively; t_s state transition delay; d_i deadline; E_{low} and E_{hi} energy consumption with low / high speed, respectively.

Let us now explore the E-LEDF pseudo code. On line 6 in the pseudo code of Figure 8 tasks are sorted according to their deadlines, and the task with the closest deadline is chosen for execution according to the EDF principle. When scheduling the very first task of the session (line 7), we want to check if we can schedule the task at low speed. This is done on line 8: if the execution time with low speed t_{low} added with the processor transition delay t_s is lower or equal to the task's deadline d_i, the task is scheduled using low speed. Otherwise, it is checked if the task will meet its deadline with high speed (line 9). If the deadline cannot be met even at high speed,

the operating system exception handler is called (line 10). The operating system might, for instance, alert the application whose time constraints cannot be met.

The scheduling of the following tasks begins on line 12. If the previous task was run at high speed, then E-LEDF will compute the task's total energy consumption using both low and high speeds (E_{low} and E_{hi}) on line 13. In these calculations, the processor state transition energy costs are taken into consideration. If the task is not schedulable even at high speed (line 14) the operating exception handler is called (line 15). If the task is schedulable, E-LEDF will need to consider whether it is worthwhile to switch to low speed. If the task will meet its deadline at low speed including transition delays (line 17), and the total energy consumption at low speed E_{low} doesn't exceed energy consumption at high speed E_{hi}, then the task is scheduled at low CPU speed (line 19). Otherwise, the task is scheduled at high speed (line 21 and 23).

A similar pattern to the one described in the previous paragraph is followed if the previous task was scheduled at low speed (line 24). The total energy consumption at both speeds is calculated (line 25), and in the sum E_{hi} also the transition cost is included. The transition to the higher CPU speed is made only if the total energy consumption at high speed would be smaller than using the low speed. This condition is checked on line 30.

We believe the E-LEDF code contains redundancies and at least one error. Notice, that the **if** statement on line 19 is redundant: the condition $t_{low}+t_s \leq d_i$ has already been checked on line 17. In fact, also the **if** on line 21, and the entire lines 22 and 23, are redundant. The error we believe we have found is also quite obvious. Consider a situation where the previous task has been run at low speed, and $t_{hi}+t_s \leq d_i$, but $E_{hi} \geq E_{low}$. This would bring us to line 33 in the pseudo code. Now assume that $t_{low}+t_s \geq d_i$. This could very well be possible, since the task is schedulable at high speed ($t_{hi}+t_s \leq d_i$), and the schedulability test on line 26 would hence have been passed. In this situation, the **if** condition on line 33 would be false, and the task would never be scheduled. The pseudo code would hence need some rewriting to support tasks that would require to be run at high speed, even though they wouldn't spend less energy at that speed. The required modification is quite trivial. It suffices to add to line 33 the following: `else` *`schedule at high speed`*.

A more fundamental problem with E-LEDF is that the algorithm does not explicitly handle situations when the CPU is idle. If the previous task has left the CPU in its high speed state when the job queue becomes empty, E-LEDF will still keep the

CPU running at full speed and hence waste energy although the processor is not needed. Currently, E-LEDF supports only two distinct CPU speeds, and no power-off state. Frequency and voltage scaling decisions are made only at the beginning of each task, which limits achieved energy savings.

4.1.3 Feedback DVS-EDF

One of the more ambitious power-aware hard real-time CPU scheduling algorithms is also based on EDF and is called *Feedback DVS-EDF*. It was published by Dudani, Mueller and Zhu in 2002 [DMZ02]. The interesting parts of the pseudo code are presented in Figure 9. The code for initializing variables, pre-emption handling and setting of clock frequency are excluded, since they are of little interest to the topic of this thesis. The interested reader may, however, view the entire algorithm in appendix 1.

The idea in Feedback DVS-EDF is to utilize DVS aggressively. The algorithm is based upon the assumption that most actual task instances (jobs) will need less CPU time than scheduled to them. Therefore, Feedback DVS-EDF begins the execution of a job with a very slow CPU speed. Only if the job isn't finished after a certain time, is the CPU speed increased. In real-life situations, jobs rarely use all of the CPU time allocated to them. Therefore, for most jobs, the CPU will never need to run at its highest speed, and energy is saved.

In order to be able to calculate a statistically optimal initial speed, the Feedback DVS-EDF algorithm maintains statistical information on the execution times of a task's previous instances. Tasks are also able to pass unused slack time on to the next job. Say, for instance, that a job J_i has executed 2 time units faster than budgeted and finishes at t. Further assume, that the next job J_{i+1} has been released before t. In this case, using Feedback DVS-EDF, J_i will pass the two unused slack time units on to J_{i+1}. Now, J_{i+1} will have in its execution time budget two more time units more than usually. This extra time may be used to further slow down the processor in order to conserve energy. Information on unused slack time is stored in the variable *slack*, and by reading this variable the scheduler will know of these two superfluous time units when it goes on to schedule J_{i+1}. This increased time budget is, of course, usable only if it won't jeopardize finishing J_{i+1} within its time boundaries.

These are the main energy conserving properties of Feedback DVS-EDF. Let us

Procedure TaskActivation($\mathbf{T_{ij}}$)

1 **if** processor was idle for d **then**
2 $\quad slack \leftarrow slack - d$
3 **if** T_{ij} was preempted/interrupted **then**
4 $\quad slack \leftarrow slack + slack_{ij} - left_{ij}$
5 **forall** T_{ab} idle task jobs in $d_{pk}..d_{ij}$ **do**
6 $\quad slack \leftarrow slack + C_a$
7 $\alpha\prime \leftarrow \min\{\frac{f_1}{f_m}, \ldots, \frac{f_m}{f_m} | \frac{f_i}{f_m} \geq \frac{C_{avg\_i}}{C_{avg\_i}+slack}\}$
8 **if** $(\alpha\prime = 1)$ **then**
9 $\quad C_A \leftarrow 0$
10 **else**
11 $\quad C_A \leftarrow slack \times \alpha\prime/(1-\alpha\prime)$
12 SetInterrupt$(T_i, C_A/\alpha\prime)$
13 SetFrequency$(\alpha\prime)$

Procedure TaskCompletion($\mathbf{T_{ij}}$)

14 **if** T_{ij} was preempted **then**
15 $\quad$ **if** $c_{ij} > C_i$ **then** (late finish)
16 $\quad\quad slack \leftarrow slack - c_{ij} + C_i$
17 **else** (early finish)
18 $\quad slack \leftarrow slack + C_i - c_{ij}$
19 **forall** T_{ab} idle task jobs in $r_{ij}..d_{nl}$ **do**
20 $\quad slack \leftarrow slack - C_a$
21 $C_{avg\_i} \leftarrow (C_{avg\_i} \times (j-1) + c_{ij} \times \alpha\prime)/j$
22 $left_{i(j+1)} = C_i$

Figure 9: The central parts of the Feedback DVS-EDF algorithm [DMZ02].

now study the pseudo code of 9 in closer detail. In order to do this, a number of notations need to be explained. By T_{ij} we mean an instance, i.e. a job, j of task T_i, and with d_i we mean its deadline. The variable *slack* stores information on unused slack time, and $left_{ij}$ holds the remaining execution time of job T_{ij}. By T_{ab} we denote the set of idle tasks (tasks that currently have no jobs waiting for processor time), and by *pk* the previous, by *nj* the next and by *ij* the current job. The letter α' denotes the ratio of the processor's maximal speed, and f_i is the clock frequency of the processor. By r_{ij} we denote the release time of job T_{ij}, i.e, the time when the job is ready to be scheduled, and starts waiting for processor time. The job's actual execution time is denoted by c_{ij}, and C_i is the budgeted worst-case execution time. The execution time C_i of a job is divided into two parts, C_A and C_B, where C_A is the time interval that the job is executed at a slower and less consuming CPU

speed, and C_B denotes the time interval when the job is executed at high speed. Therefore, $C_i = C_A + C_B$. The variable C_{avg_i} notates the average execution time of T_i. This implies:

$$\frac{C_A}{\alpha'} + C_B = C_i + slack$$

By α' we mean the ratio of the maximal clock frequency, and this value in turn is calculated using the formula

$$\alpha' = \frac{C_A}{C_A + slack}$$

where *slack* denotes the unused "slack" time that emerges when a job is executed faster than budgeted.

This functionality is presented in the algorithm of Figure 9 beginning on line 7, in the procedure *TaskActivation*. (Notice that Feedback DVS-EDF uses the term "task" in the procedure names when referring both to a task, and an instance of a task. Elsewhere in this thesis, the term "job" is used for the latter.) On line 7 the value α', i.e., the ratio of the lowered CPU speed from the highest speed, is calculated. In order to find the optimal value for α' Feedback DVS-EDF utilizes statistics from previous instances of the task. It is from here that the word "Feedback" in the algorithms name is originated. Statistics is maintained in the variable C_{avg}, which indicates the average execution time of this task's previous jobs. When a job is finished, its C_{avg} is updated on line 21 in the procedure *TaskCompletion*. Here we take into consideration the execution time of the current instance c_{ij} and calculate a weighted average between c_{ij} and the previous value of C_{avg}. The value C_{avg} is then utilized when calculating an optimal value for α' at job activation. The variable *slack* is calculated and utilized at similar occasions. The value is calculated when a job finishes, in the procedure *TaskCompletion* on line 20, and is later utilized when calculating α' in *TaskActivation* on line 7. Information on "unused" time and CPU utilization statistics of previous jobs is thus passed between jobs using these two variables.

On line 9 and 11 the variable C_A is calculated and set. This variable indicates the length of the time period from the beginning of a job that the job is to be executed with the lower speed. This speed is indicated as the ratio from the maximum speed by α'. If α' is calculated to equal 1 (line 8), then the task must be executed at highest clock frequency, and the length of the lower speed interval C_A is set to 0 (line 9). If the value of α' is not equal to 1, then C_A is calculated on line 11. On

line 12, a timer interrupt is set to activate the scheduler after C_A units of time has passed. This is done by the procedure *SetInterrupt*. On line 13, the processor is adjusted to the new clock frequency. If the job isn't finished within C_A units of time, the scheduler is reactivated by the timer. The reactivated scheduler will adjust the CPU to run at full speed, and the rest of the job will be executed at highest clock frequency. This will guarantee timely finishing of the job.

4.1.4 Cycle-Conserving DVS for EDF Schedulers

Feedback DVS-EDF presented in the previous subsection utilizes DVS aggressively. For the sake of comparison, let's consider the *Cycle-Conserving DVS for EDF schedulers* (ccEDF) algorithm [PiS01] presented in Figure 10. This illustrative algorithm utilizes DVS conservatively: jobs are initially run at a higher CPU speed, and whenever jobs finish before spending their entire time budget, the processor is slowed down.

```
1  select_frequency():
2        use lowest freq. f_i ∈ {f_1, ..., f_m | f_1 < ··· < f_m}
3        such that U_1 + ··· + U_n ≤ f_i/f_m

4  upon task_release(T_i):
5        set U_i to C_i/P_i;
6        select_frequency();

7  upon task_completion(T_i):
8        set U_i to cc_i/P_i;
9              /* cc_i is the actual cycles used this invocation */
10       select_frequency();
```

Figure 10: The Cycle-conserving DVS for EDF Schedulers (ccEDF) algorithm [PiS01]. C_i **budgeted CPU cycles to task** T_i**;** cc_i **actual spent cycles;** f_i **processor frequency;** f_m **maximal processor frequency;** U_i **utilization degree.**

Now consider the pseudo code in Figure 10. Upon task completion, on line 8, the utilization degree U_i of T_i is set to $\frac{cc_i}{P_i}$, i.e., to reflect the eventual time left unused by the task. Then, on line 10, the procedure `select_frequency()` is called. Here, ccEDF chooses from among all discrete CPU speeds $\{f_i, \ldots, f_m\}$ the lowest one that will guarantee schedulability of the tasks with the newly calculated U_i. The schedulability criteria, $\sum U \leq 1$, is based on the EDF schedulability condition (Equation 4) [LiL73], but on the right side of the inequality we now have $\frac{f_i}{f_m}$ instead

of 1 to represent the lowered CPU speed. When new tasks are released, ccEDF will in `task_release(`T_i`)` on line 5 calculate the utilization for the new task, and then on line 6 call `select_frequency()`, which now may want to raise the CPU speed to reflect the increased workload. No explicit transition delay considerations, nor explicit schedulability failure handling, is present in ccEDF. Its purpose here is solely to illustrate the functionality of conservative DVS as opposed to the aggressive technique implemented in Feedback DVS-EDF. The authors of ccEDF have also presented ccRM, an energy conserving Rate Monotonic based algorithm with conservative DVS support, and *laEDF* (Look Ahead EDF), an EDF based power-aware scheduler with aggressive DVS support [PiS01].

4.1.5 Comparing the Presented Algorithms

We now have considered five different algorithms for power-aware processor scheduling. The one based on the Rate Monotonic method is called LPFPS. This algorithm is pre-emptive and seeks energy savings in two different ways: if only one job remains left to be scheduled, it is run on a lower clock frequency. If no jobs are left waiting for processor time, then the processor is put to sleep, and is later awoken with a timer. Because the Rate Monotonic method guarantees an utilization degree of approximately 0.69, in an RM schedule there most often is plenty of idle time. The LPFPS algorithm also considers the processor wakeup delay when making power down decisions.

The other four algorithms are based on the Earliest Deadline First method. The first one presented is called LEDF and supports only two different CPU speeds. At the beginning of each job the scheduler calculates whether the job will meet its deadline if scheduled at the lower speed. The higher speed is used only when needed. This simple algorithm has later been enhanced by the same authors with E-LEDF. Here also CPU state transition costs in time delays and energy waste are considered. A state transition is made only if it is worthwhile. Very short transitions not always are. Even E-LEDF supports only two different speeds.

Out of the presented algorithms the most versatile is Feedback DVS-EDF. This algorithm aggressively seeks energy savings by starting the execution of each job with a low speed. Only when needed to guarantee timely execution does the scheduler run the job at high speed. The idea here is the finding that most real-time jobs execute significantly faster than their budgeted worst-case execution times. In order to find an optimal starting speed, Feedback DVS-EDF uses statistical information

from previous instances of the task. Jobs may pass unused execution time on to the next job.

Even though Feedback DVS-EDF is advanced even it could be further improved. For instance the algorithm divides the task's execution times into two pieces, C_A and C_B, where the time C_A is spent running at the lower speed, and C_B with highest speed. By further dividing the execution time into smaller fragments, where each fragment is executed slightly faster than the previous one, even greater energy savings could be found. This would, however, add to the algorithm's complexity. The usefulness would depend on the amount of DVS states the used processor platform supports.

We ended our review of energy saving hard real-time scheduling algorithms by presenting ccEDF, a simple algorithm that utilizes DVS conservatively. Where Feedback DVS-EDF begins execution of tasks with low speed, ccEDF initially runs tasks at high speed, and once slack time is accrued, forthcoming tasks are run at slower speeds, if possible. This algorithm makes voltage and frequency scaling decisions only at the end of and upon release of tasks, but is significantly less complex than Feedback DVS-EDF.

4.2 Soft Real Time Scheduling

We will in this subsection explore two soft-real time CPU scheduling algorithms. Soft real-time schedulers provide a statistical performance guarantee. A certain percentage, say p, of the scheduled jobs will finish within a certain time period. Occasional misses of jobs are allowed. Therefore one might believe that the soft real-time schedulers would be more simple than their hard real-time counterparts. That is, however, not the case. As will be revealed, these algorithms are far more complex than their hard real-time counterparts. Their system model concepts and patterns of design are original, whereas the hard real-time schedulers evidently were offspring of the original EDF and RM algorithms published by Liu and Layland in 1973 [LiL73].

Presently, the most common implementation environment for soft real-time schedulers are multimedia systems. For instance MPEG video or audio compression decoders are considered fully usable even when they occasionally do miss a frame or sound sample. Because such a relaxation to the strict hard real-time schedulers might provide significantly better system throughput or response times to interactive systems, soft real-time schedulers are increasingly popular.

4.2.1 The EScheduler Algorithm

The EScheduler [YuN06] is based upon work done in the GRACE project [YuN03]. The algorithm gives a statistical probability guarantee that scheduled tasks (EScheduler uses the term "process") meet their deadlines. This is usually sufficient for multimedia applications, where it suffices to know that p % (where p might be for instance 95) of video frames are timely decoded. EScheduler conserves energy by utilizing DVS aggressively. It is based on the EDF algorithm.

EScheduler has two main tasks to perform: firstly, task scheduling, i.e. to schedule instances of tasks guaranteeing that they meet their deadline with probability p %, and secondly, speed scaling, i.e. to run these scheduled processes conserving as much battery power as possible. These functions will be described next.

Scheduling tasks The fundamental assumption in the design of EScheduler is, that while the actual CPU demand of a task's individual jobs varies greatly, the cycle demand distribution of the task is pretty stable. EScheduler maintains statistics of the actual CPU cycles needed by the last n jobs of a task.

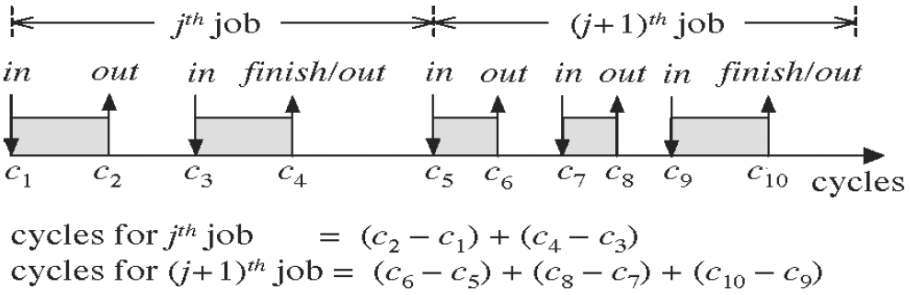


Figure 11: EScheduler counts the cycle demand of tasks [YuN06].

EScheduler calculates the cycle demand of a job as depicted in Figure 11. The counter is implemented as an extra field in the *Process Control Block* (PCB) of the operating system. Each time the task is switched out the CPU cycle counter of the job is updated, and when the job finishes, its entire cycle count is added up to the statistics. Based upon this statistics, accurate estimations of forthcoming CPU cycle demand can be made, and the task can be scheduled an appropriate amount of CPU time. Scheduling too little CPU time will result in low quality of service as for instance video frames aren't decoded timely, while scheduling too much time will waste CPU resources and consume energy superfluously.

The graph in Figure 12 depicts the cumulative cycle demand of one task's (T_i)

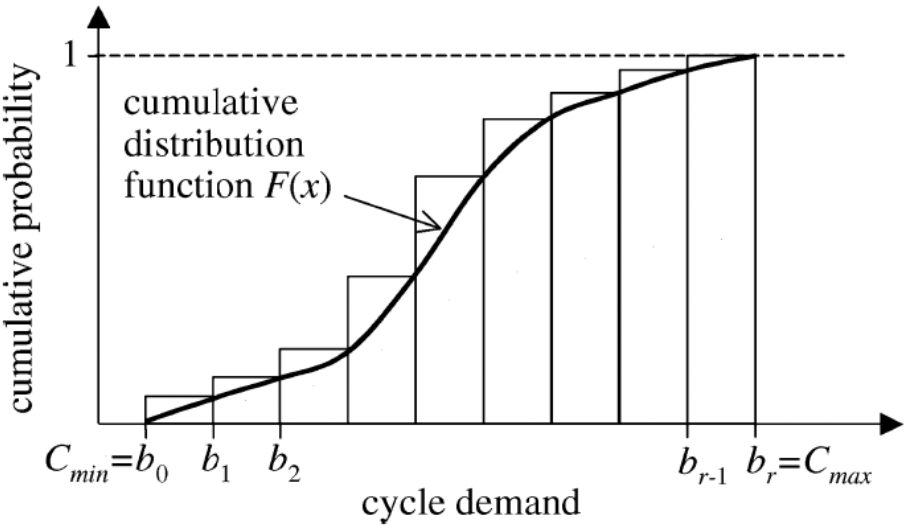


Figure 12: The cumulative cycle demand distribution in EScheduler [YuN06].

elapsed jobs (J_i). The cumulative distribution function is based on Equation 6.

$$F(x) = P[X \leq x] \tag{6}$$

Firstly remember, that jobs are instances of one task. Now let's consider this equation. It indicates the probability of the cumulative CPU cycle demand of jobs of one particular task (X), of being equal or less than x. In Figure 12, C_{min} is the smallest cycle demand among the task's considered jobs, and C_{max} the largest. The interval $[C_{min}, C_{max}]$ is divided into r sections. Each section forms an area in the histogram. The height of a section area indicates the probability that the job needs at most b_k cycles, where b_k is the upper boundary of the section. From this histogram, it is possible to extract the cycle boundary b_k below of which p percent of the jobs of the task remain.

In soft real-time applications, it suffices to provide a statistical guarantee that p percent of jobs meet their deadline. Before the task is accepted into the set of schedulable tasks, a schedulability test needs to be performed. The task is schedulable if the condition in Equation 7 is fulfilled.

$$\sum_{i=1}^{n} \frac{C_i/S_K}{P_i} \leq 1 \tag{7}$$

In this equation, C_i is the estimated cycle demand below of which the cycle demand of p percent of job instances of task i remain; S_K is the maximum number of cycles

the CPU can achieve at full speed, and P_i is the period of task i. The condition ≤ 1 originates from the EDF schedulability condition (Equation 4) [LiL73].

Adjusting the CPU speed for a task After jobs are scheduled, it is up to EScheduler to execute them at optimal CPU speed to minimize power consumption. Here, its function resembles that of Feedback DVS-EDF (see Section 4.1.3). EScheduler utilizes DVS aggressively. It starts job execution at a low CPU speed and increases speed as needed. EScheduler is, however, a little more complex in its speed scaling technique than was Feedback DVS-EDF.

EScheduler begins by calculating an aggregate CPU speed requirement for the current task set. This speed is calculated with the equation $\sum_{i=1}^{n} \frac{C_i}{P_i}$ where the unit is cycles per second (or hertz). As an example, consider a task set of two tasks, where the first one is allocated $12 * 10^6$ cycles every 40 ms and the other 10^6 cycles every 20 ms. The aggregate CPU speed would then be $\frac{12*10^6}{40} + \frac{10^6}{20} = 350MHz$ [YuN06].

The straightforward solution would be to run the tasks at this aggregate speed. This would, however, waste energy. The estimated cycle demand C_i is the value below of which the cycle demand of p percent of tasks remain. If p is for instance 95, then 95 percent of the tasks require less than C_i cycles. The cycle demand of individual tasks vary greatly. Jobs are initially ran at a low speed, and as the job cycle count increases, CPU speed is gradually increased according to a *speed schedule* that EScheduler calculates for every task.

The speed schedule of a task consists of coordinates (x, y) in an ordered list. At x or more spent cycles the CPU is accelerated to speed y. An example of a speed schedule might be: $(0, 100MHz), (1 * 10^6, 120MHz), (2 * 10^6, 180MHz)$. Here, the task would be started at CPU speed 100 MHz, and after $1 * 10^6$ cycles, the processor would be accelerated to 120 MHz. After $2 * 10^6$ cycles, if the job would still not be completed, the processor speed would be increased to 180 MHz.

With high p values most jobs consume less than C_i CPU cycles. They will hence complete before ever reaching the highest CPU speed points, and therefore avoid these most energy consuming phases. Notice that every task in the set has its own speed schedule. Therefore, processor speed changes occur, besides at scaling points, also at context switches. The EScheduler algorithm [YuN06] does not explicitly consider processor state transition delays when calculating a speed schedule.

Calculating the speed schedule The approach taken by EScheduler in calculating a speed schedule is based upon the cycle demand histogram (see Figure 12). Each area in the histogram, starting with a cycle demand of b_i, is issued a specific CPU speed. The speed schedule of any task will consist of m coordinates, $(b_i, s(b_i))$, where the CPU speed, $s(b_i)$, is calculated using Equation 8 [YuN06].

$$s(b_i) = \frac{\sum_{j=1}^{m} \mathrm{g_j} \sqrt[3]{1 - F(b_j)}}{T \sqrt[3]{1 - F(b_i)}}, i = 1, \ldots, m. \tag{8}$$

where $\mathrm{g_j}$ is the size of the j:th cycle group (the width of the area in the histogram), and T represents the *time budget* of a task. This variable represents the available time distributed among tasks according to their cycle demand. It is calculated using the following formula:

$$T = \frac{C_i}{\sum_{i=1}^{n} \frac{C_i}{P_i}}$$

This calculation of optimal processor speeds is based on the theoretical alternative, where CPU speed can be adjusted linearly. Real-world processors provide only discrete speed alternatives. For instance, the StrongArm SA-110 provide 11 different CPU speed alternatives [YuN06]. A straightforward approach to deal with this real-world limitation is to calculate the optimal speed using formula 8, and then round $s(b_i)$ to the nearest upper discrete speed. This is, however, not energy optimal, since the provided speed might execute the job unnecessarily fast and waste energy. On the other hand, rounding $s(b_i)$ downwards might jeopardize timely execution. Therefore, EScheduler explicitly considers all available processor speeds, and chooses from among them the most efficient combination for the speed schedule. Here, it even takes into consideration the processor's transition delay from active to sleep state.

The problem of choosing the optimal CPU speed schedule is NP hard [YuN06]. EScheduler uses an approximation algorithm for selecting the best speed combination. It should also be noted that these speed options are processor specific. Therefore, in order to be efficient, EScheduler needs to be rewritten for each particular hardware platform it is implemented on.

Implementing EScheduler EScheduler has been implemented into the Linux 2.6.5 kernel with 2605 lines of C code. In order to implement the cycle demand counter, the Linux Process Control Block is modified according to Figure 13. The

```
struct process_struct {
    ...
#ifdef CONFIG_UIUC_GRACE
    /* for profiling */
    unsigned long long last_sample_cycles;
    unsigned long job_cycles;
    /* for intra-job DVS */
    struct dvsPnt_struct *speed_schedule;
    unsigned short dvsPnt_count, current_dvsPnt;
#endif /*CONFIG_UIUC_GRACE*/
};
```

Figure 13: The modified Linux Process Control Block [YuN06].

long integer `job_cycles` records the number of cycles used by the job; `*speed_schedule` is a pointer to the speed schedule list of the task, and `current_dvsPnt` points to the presently used speed setting.

The Linux scheduler has been revised to (1) update the PCB fields at scheduling occasions and (2) scale the processor frequency using DVS according to the process' speed schedule. A higher resolution timer has been hooked to the standard Linux scheduler [YuN06] to allow invoking of the EScheduler every 500 μs, which enables periodic scheduling decisions to be made at a rate sufficient for soft real-time applications.

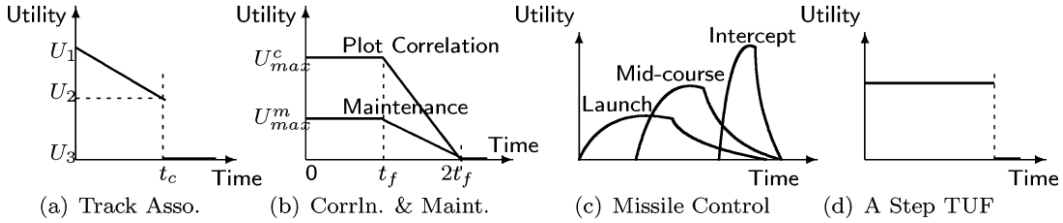
The EScheduler provides statistical real-time guarantees for multimedia applications. Tasks are scheduled CPU time according to their historical CPU demand. While executing tasks, EScheduler saves energy by adjusting the CPU speed according to a speed schedule it has calculated. Tasks are initially run at slow CPU speeds, and the speed is accelerated as execution progresses.

4.2.2 The ReUA Algorithm

This subsection presents ReUA (*Resource-constrained energy-efficient utility accrual algorithm*) [WRJ06]. It is an ambitious processor scheduling algorithm that considers system-wide energy savings, and replaces deadlines by a concept that provides higher fidelity.

The Time Utility Function replaces deadlines The classical concept of deadlines can be argued to be artificial. Consider, for instance, a missile control system.

In the traditional deadline-based approach, the missile must hit its target no later than at time D. However, in a real world situation, the hit might be considered to be *useful* even when missing D by a hair, although a perfect miss is preferred. This kind of argumentation has lead to the development of a concept of *Time Utility Function* (TUF), which replaces deadlines.



(a) Track Asso. (b) Corrln. & Maint. (c) Missile Control (d) A Step TUF

Figure 14: Example Time Utility Functions (TUF) [WRJ06].

Some example TUFs can be seen in Figure 14. The *utility* of finishing a job is depicted as a function of the completion time. In Figure 14 (a) and (b) non-increasing TUFs can be seen. Here, the *utility* of completing the task decreases or stays the same as time goes by. In (c) a TUF of a missile application is depicted. Here, the utility increases as the missile approaches its target, and then quickly decreases. A traditional deadline as a TUF is shown in Figure (d). The utility of the completion of the task stays the same until the task's deadline, after which the utility drops to zero. A scheduling algorithm that tries to maximize the sum of TUFs in the system is called *Utility Accrual*.

The TUF of task T_i is denoted by U_i, and the TUF of job J_k is denoted by U_{J_k}. The utility when J_k is completed at time t is denoted $U_{J_k}(t)$. When scheduling tasks, the aim of ReUA is to maximize the utility while minimizing energy consumption [WRJ06]. In order to achieve this, ReUA uses a unit called UER (*Utility-Energy Ratio*). The system's UER is defined as follows:

$$UER = \frac{\sum_{i=1}^{n} U_i}{\sum_{i=1}^{n} E_i}$$

where U_i denotes the TUF of task T_i, and E_i the energy (described hereafter) consumed by task T_i. Hence, UER is an indicator of system-wide energy efficiency: utility achieved per energy unit.

System wide energy considerations Reducing the CPU power requirement will lead to longer task execution times. If hardware components such as displays, hard drives or memory chips need to be powered up during this time, reducing CPU

speed might in the worst case even increase the system-wide energy requirement, as other components need to be powered up longer. When making scheduling decisions, ReUA considers the system-wide power consumption instead of only the CPU power consumption. While the CPU power consumption is calculated using the formula $P = C \times f \times V^2$ (Equation 5), The equation for the system-wide energy consumption is estimated using Equation 9 [WRJ06]:

$$P = S_3 \times f^3 + S_2 \times f^2 + S_1 \times f + S_0 \tag{9}$$

where f is the operating frequency; S_3 is the CPU power requirement; S_2 is caused by CMOS power leakage; S_1 presents the power requirement caused by components such as memory chips operating at a fixed voltage independent of frequency, and S_0 is a constant representing components such as displays, whose power requirement is independent of both operation frequency and voltage [WRJ06]. From Equation 9, the following equation for the energy consumed *per processor cycle* can be derived:

$$E(f) = S_3 \times f^2 + S_2 \times f + S_1 + S_0/f \tag{10}$$

Calculating processor cycle demand When calculating the processor cycle demand to be allocated to a task ReUA, like EScheduler (Section 4.2.1), uses statistical information. But unlike EScheduler, ReUA does not explicitly present a mechanism for collecting and processing statistical information: the CPU cycle demand mean and variance are assumed to be given. To calculate a task T_i's cycle demand C_i, ReUA uses Equation 11 which provides a statistical performance guarantee:

$$C_i = E(Y_i) + \sqrt{[p_i \times Var(Y_i)]/(1 - p_i)} \tag{11}$$

where Y_i is the cycle demand distribution; $E(Y_i)$ is the expected cycle demand, and $Var(Y_i)$ is the statistical variance of cycle demand distribution. The variable p_i is a probability. In ReUA, a pair $\{v_i, p_i\}$ is used to indicate that v_i of the maximal utility (TUF) should be achieved with probability p_i.

This statistical performance guarantee can be presented as $Pr(U(s_{i,j}) \geq v_i \times U_i^{max}) \geq p_i$ [WRJ06], where $s_{i,j}$ is the sojourn time of $J_{i,j}$. To calculate the upper bound for T_i's sojourn time, ReUA uses a variable D_i and calls it "critical time". To ensure that v_i of the maximal utility is achieved with probability p_i, ReUA needs to guarantee that

$$D_i = U_i^{-1}(v_i \times U_i^{max}) \tag{12}$$

where U_i^{-1} is TUF's inverse function. The values C_i and D_i are calculated in ReUA's `offlineComputing()` procedure that can be seen in Figure 15. Equation 12 is used on line 3 to calculate D_i. On line 4 Equation 11 is used to calculate the amount of CPU cycles to be allocated to T_i, and this number is placed in the variable C_i. This procedure also calculates $f_{T_i}^o$, the optimal speed (frequency) at which to execute T_i.

1: **input :** Task set **T**;
2: **output :** $D_i, C_i, f_{T_i}^o$;
3: $D_i = U_i^{-1}(v_i \times U_i^{max})$;
4: $C_i = E(Y_i) + \sqrt{\frac{\rho_i \times Var(Y_i)}{1-\rho_i}}$;
5: Decide $f_{T_i}^o$, such that $U_i(\frac{C_i}{f_{T_i}^o}) \Big/ \left(C_i \times E(f_{T_i}^o)\right) = \max\left(U_i(\frac{C_i}{f_j}) / \left(C_i \times E(f_j)\right)\right)$, $\forall j \in \{1, 2, \ldots, m\}$;

Figure 15: The `offlineComputing()` procedure of the ReUA algorithm [WRJ06].

The ReUA main pseudo code The algorithm for ReUA can be seen in Figure 16. As input ReUA receives the current task set $T = \{T_1, \ldots, T_n\}$ and the current unscheduled job set $\mathcal{J}_r$. From these, ReUA will calculate its output, i.e. the job to be executed J_{exe}, and its execution speed, f_{exe}.

On line 3 the `OfflineComputing(T)` procedure is called, and C_i, D_i and the optimal frequency $f_{T_i}^o$ of each task are calculated. (On line 4, the current time t_{cur} is placed in t.) The switch-statement on lines 5–8 manages the variable C_i^r which holds the remaining CPU cycles allocated to the current job. Upon task release (line 6), the entire allocated cycle amount is placed in this variable; upon task completion (line 7) the variable is set to zero, and on other scheduling occasions (line 8) C_i^r is updated to reflect the number of remaining cycles.

In the **for** loop starting on line 9, a feasibility check is performed on all unscheduled jobs. The expected calculation time of any job may not exceed its termination time at highest CPU speed. If a job is not feasible, it is aborted (lines 10–11). Otherwise, on line 13, ReUA calculates the resource dependencies of the job using the procedure `buildDep()`.

The **for** loop on lines 14–15 calculates the UER (*Utility-Energy Ratio*) for each unscheduled job. This Figure implies "how much utility would be achieved if this job

```
1:  input : T = {T_1, ..., T_n}, J_r = {J_1, ..., J_n'};
2:  output : selected job J_exe and frequency f_exe;
3:  offlineComputing(T);
4:  Initialization: t := t_cur, σ := ∅;
5:  switch triggering event do
6:      case task_release(T_i)          C_i^r = C_i;
7:      case task_completion(T_i)       C_i^r = 0;
8:      otherwise                       Update C_i^r;
9:  for ∀J_k ∈ J_r do
10:     if feasible(J_k) do = false then
11:         abort(J_k);
12:     else
13:         J_k.Dep := buildDep(J_k);
14:  for ∀J_k ∈ J_r do
15:     J_k.UER := calculateUER(J_k, t);
16:  σ_tmp := sortByUER(J_r);
17:  for ∀J_k ∈ σ_tmp from head to tail do
18:      if J_k.UER > 0 then
19:          σ := insertByECF(σ, J_k);
20:      else
21:          break;
22:  J_exe := headOf(σ);
23:  f_exe := decideFreq(T, J_exe, t);
24:  return J_exe and f_exe;
```

Figure 16: The ReUA main pseudo code [WRJ06]. Symbols: $\mathcal{J}_r$ denotes the current unscheduled job set; C_i CPU cycles allocated to J_i; C_i^r remaining cycles of current job.

were to be executed starting at this moment". The `calculateUER()` procedure even considers job dependencies calculated by `buildDep()`: if J_i is dependent of tasks $J_{i.Dep} = \{J_{Dep_1}, \ldots, J_{Dep_n}\}$, then the jobs in $J_{i.dep}$ are included when calculating the UER for J_i.

On line 16, the jobs are sorted in non-increasing order according to their UER. In the **for** loop starting on line 17, the jobs which are meaningful to run, i.e. the ones whose UER is larger than zero (line 18), are inserted into a list σ in order of their *critical times*. This is done by the procedure `insertByECF()` (line 19). Critical times are moments when, at the latest, the job needs to be finished in order to guarantee the desired performance level defined by $\{v_i, p_i\}$. The ECF value of a job J_i is not necessarily the critical time of J_i alone: if another job is dependent on J_i, the actual ECF of J_i might be earlier than its tentative ECF. The EDF principle is followed by `insertByECF()`. In essence, on lines 16–21, the jobs are first sorted

according to their UERs, and then according to their ECFs. The resulting ordered list is placed in σ.

On line 22, the job at the head of σ is chosen for execution. On line 23 in procedure `decideFreq()`, ReUA calculates the optimal execution speed for the job considering available DVS parameters. On line 24 the algorithm returns the job to be scheduled J_{exe}, and its execution frequency f_{exe}.

4.2.3 Comparing the Presented Algorithms

Presented in this section were EScheduler [YuN06] and ReUA [WRJ06], two recent algorithms for CPU scheduling in a soft real-time environment. Both algorithms provide a statistical guarantee that jobs meet the desired level of performance with probability p. In EScheduler, the process of collecting and analyzing the accumulated CPU cycle demand statistics is explicit; in ReUA, the mean and variance of CPU cycle demand is considered to be given.

EScheduler is a traditional energy conserving CPU scheduling algorithm: it only considers the power requirements and savings of the CPU (Equation 5) and ignores the power properties of the rest of the system. The approach chosen in ReUA is more realistic, as it estimates *system-wide* energy savings (Equation 9). How superior as the latter approach may seem, one should note that, in essence, the difference is just whether we choose to consider the CMOS power consumption equation $P = C \times f \times V^2$ or the system-wide equation $P = S_3 \times f^3 + S_2 \times f^2 + S_1 \times f + S_0$ when estimating task power requirements.

Where ReUA stands out in comparison to EScheduler is in its consideration of resource dependencies, and its introduction of the TUF concept that has been argued to provide higher fidelity than deadlines. Neither of the algorithms explicitly takes into consideration transition delays when making DVS frequency and voltage adjustment decisions.

5 Power Aware Device Scheduling

The main problem with device scheduling is the same as with processor scheduling. We have one resource with multiple users, and wish to share the resource between these multiple users in a purposeful way. In real-time systems especially deadlines must be met. The major difference between processor and device scheduling is that the device scheduler needs to calculate a distinct schedule for each device. Systems may contain multiple devices, and each task may use several or none of them. The situation is hence not the same as with processor schedulers, which we considered in Section 4: the processor schedulers were all aimed at uniprocessor systems, and every task naturally utilize this single processor.

Devices considered in this section have at least two power states: a sleep state and an operating or awake state. In the sleep state, the device is not able to provide its service, like disk or network I/O, but in this state the device consumes less energy than in its operating state. Some devices may have several power states, where each state ps_{i+1} consumes less energy than state ps_i, but takes a longer time to wake up from. The transition between states is controlled by the operating system. A transition between states always includes a certain penalty in terms of time and energy cost. A transition takes a certain amount of time, and requires a certain amount of energy. A proper power-aware real-time scheduler needs to consider these time and energy costs when making scheduling decisions in order to guarantee meeting of deadlines.

5.1 Hard Real Time Scheduling

The problem of power-aware real-time device scheduling has in recent research been tackled in at least two different ways. The aim in for instance the EEDS algorithm [ChG06] is to enhance the system's EDF based task scheduler with an energy aware device scheduler. One can also entirely separate the device scheduler from the processor scheduler, as has been done in MUSCLES and LEDES [SwC03]. A completely different approach is chosen in the EDS [SwC05] algorithm, which due to its CPU time and memory requirement operates offline. In the next section we will explore each of these algorithm

```
Procedure LEDES(k_j, τ_i, τ_{i+1})
curr: current scheduling instant;
1. if scheduling instant is the start of τ_i then
2.    if k_j is powered-up then
3.       if k_j ∉ L_i and L_{i+1} then
4.          shutdown k_j
5.       end if
6.       if k_j ∈ L_{i+1} then
7.          if s_{i+1} − (s_i + c_i) ≥ t_{0,j} then
8.             shutdown k_j
9.          end if
10.      endif
11.   else
12.      if k_j ∈ L_{i+1} and s_{i+1} − (s_i + c_i) < t_{0,j} then
13.         wakeup k_j
14.      end if
15.   else end
16. if scheduling instant is completion of task τ_i then
17.   if k_j is powered-up then
18.      if k_j ∉ L_{i+1} and s_{i+1} − curr ≥ t_{0,j} then
19.      shutdown k_j
20.     endif
21.   else
22.      wakeup k_j
23.   else end.
```

Figure 17: The pseudo code of the LEDES scheduler [SwC03]. Notations: k_j a device, τ_i task, L_i the set of devices needed by τ_i, s_i start time of task i, c_i execution time of task i, $t_{0,j}$ transition time of device j.

5.1.1 Low Energy Device Scheduler

The basic assumption in Low Energy Device Scheduler (LEDES) [SwC03], Figure 17, is that the *transition time*, the time needed for the device to switch from sleep state to the powered-up state (or vice versa) is shorter than the execution time of any task instance. If we accept this assumption, then it suffices to schedule only one task into the future at a time. This is enough to guarantee that no deadlines will be missed. In other words, if the current task instance is τ_i we need only consider the device schedule up to and including τ_{i+1}. This will be enough for us to wake up all devices so that they will be ready for use when needed. This assumption implies that no matter how many tasks there are in the system, LEDES need only to consider two – the current and the next one – in its device schedule calculations. This is why

the workload LEDES adds to the system is acceptable. LEDES supports, however, devices with only two different states – sleep and powered-up.

As input parameters LEDES (figure 17) receives a pointer to a device k_j, the scheduling information of the current and next tasks, T_i and T_{i+1}. LEDES is activated at either the start (line 1) or the end (line 16) of a task. If the device k_j is switched on (line 2) while not being needed by the current or the next task (line 3) the device is switched off (line 4). If k_j is needed by the next task (line 6), but will make it back online if we power it down for the remainder of the execution of the current task, and power it up when finishing the current task (line 7), we power k_j down (line 8). On line 7, LEDES also considers device state transition time $t_{0,j}$. If k_j is needed by the next job, but k_j wouldn't make it back online on time if we would initiate its wakeup as late as at the end of the current task (line 12), then k_j is immediately woken up (line 13). These considered cases include all possible cases we need to take into account at the beginning of a task.

The other scheduling instance of LEDES is at the end of tasks (line 16). If the device k_j is powered up while not being needed by the next task (line 18) it can be powered off (line 19). In addition, we must on line 18 check that the powering down of the device will be finished by the start time of the next task, as the device can be needed at that occasion. (In LEDES, the powering up and powering down state transition times are assumed to equal each other, and both are notated by $t_{0,j}$.) In other cases, the device is powered up (line 22).

We believe that one **if** sentence is missing from the LEDES pseudo code. On line 22, before waking up k_j, we would want to check that k_j actually is needed by T_{i+1}. It is, of course, unnecessary to wake up the device if it isn't needed by the next task. Because LEDES makes scheduling decisions only in the beginning and at the end of tasks, its implementation into the operating system's processor scheduler should be pretty straightforward: we just call the LEDES procedure at the end and beginning of tasks. The computational complexity of LEDES is $O(n)$, where n is the size of the set of devices attached to the system.

With LEDES, implemented into a Rate Monotonic based scheduler, device energy savings of up to 40 percent have been reported [SwC03]. As the algorithm shows, LEDES supports only two distinct power states.

Procedure MUSCLES($\mathcal{S}, \mathcal{PS}, k_i$)
curr: current scheduling instant;
At s_m:
1. Find first task τ_L that uses device k_i;
2. Compute number of valid scheduling instants X between s_m and τ_L;
3. **if** $X \geq j+1$ switch down k_i from $ps_{i,j}$ to $ps_{i,j+1}$;
4. **else if** $X = j$ wake k_i up from $ps_{i,j}$ to $ps_{i,j-1}$;

At $s_m + c_m$:
5. Find first task τ_L that uses device k_i;
6. Compute number of valid scheduling instants X between s_m and τ_L;
7. **if** $X \geq j+1$ switch down k_i from $ps_{i,j}$ to $ps_{i,j+1}$;
8. **else if** $X = j$ and **curr** is a valid scheduling instant
9. Wake k_i up from $ps_{i,j}$ to $ps_{i,j-1}$;
10. **else** leave k_i in $ps_{i,j}$.

Figure 18: The MUSCLES scheduler [SwC03]. Notations: S the task schedule; PS set of power states; k_i device; s_m start time of task m; c_m execution time of task m; $ps_{i,j}$ power state j of device i; $ps_{i,0}$ the powered up state.

5.1.2 The Multi-State Constrained Low-Energy Scheduler

Several contemporary devices and peripherals, like flash memories, hard drives and network adapters, support multiple power states for energy conservation. For these purposes, the authors of LEDES have presented an algorithm called MUSCLES (multi-state constrained low-energy scheduler) [SwC03]. In MUSCLES, devices are moved between states one step at a time. Let k_i be a device, and $ps_{i,j}$ an arbitrary power state of this device. From this state, it is possible to switch to state $ps_{i,j+1}$ or $ps_{i,j-1}$ in one step. In MUSCLES, the state $ps_{i,0}$ is the operating state of the device; the other states are power saving states, where the device doesn't provide operational functionality. State $ps_{i,j+1}$ requires less power than state $ps_{i,j}$, but takes longer to wake up from.

If we accept these assumptions, we can no longer build upon the idea of LEDES, where the wakeup transition time never exceeds the execution time of the task. MUSCLES still relies on the assumption that a transition from state $ps_{i,j}$ to $ps_{i,j+1}$ or $ps_{i,j-1}$ never exceeds the execution time c_i of any task. However, if we are in state $ps_{i,j}$, the wakeup – i.e., the transition to state $ps_{i,0}$ – may endure up to $j \times c_i$ time units. When $j \geq 2$, the wakeup time may exceed the assumption we built upon in LEDES. Therefore, in order to reliably schedule devices in MUSCLES, we need

to calculate the schedule further into the future. Whereas the time requirement of LEDES is $O(n)$, where n is the amount of devices in the system, the time requirement of MUSCLES is $O(np)$, where p is the size of the task set [SwC03].

Let us now study the pseudo code of MUSCLES, presented in Figure 18. First it is worth noticing that MUSCLES is activated at either the start time of a job – indicated by s_m in the pseudo code – or at the end of the job, indicated by $s_m + c_m$, where c_m is the job's execution time. As input parameters, MUSCLES receives S, the task schedule of the system; P, a list of devices each task uses, and a device pointer k_i. The job of MUSCLES is to calculate whether to switch k_i to a less power-consuming state, to switch the device closer to the wakeup state, or to leave the device in its current state.

On line 1, we find the first task τ_L that will need device k_i, and on line 2 we calculate the amount of scheduling instances before τ_L and denote this Figure with X. Let the current power saving state be $ps_{i,j}$. If $X \geq j + 1$, k_i may safely be switched to a lower power state (line 3), and there will still remain a sufficient number of scheduling occasions to put k_i back online on time. If there are as many scheduling occasions as there are power states between the current one and the operating state, i.e. $X = j$, then k_i is switched one state towards the wakeup state, i.e., from $ps_{i,j}$ to $ps_{i,j-1}$ (line 4). This will guarantee that the device will be woken up in time when it is needed.

The other scheduling instance is at the end of the job, at time $s_m + c_m$. Here, we proceed in the same way as at the beginning of the job. It is resolved which task first needs device k_i (line 5). Then we decide how many scheduling occasions there are before the start of this task (line 6). If there are more scheduling occasions than there are power states between the current state and the wakeup state, the device is put into a lower power state (line 7). Otherwise, if the amount of states equals the number of scheduling occasions, the device is switched one state towards the wakeup state (line 8 and 9). On other occasions, the device is left in its current state.

5.1.3 The Energy-Efficient Device Scheduling Algorithm

A state transition, as such, always requires a certain amount of energy and time. Therefore very short transitions into the sleep state and back actually do not add up to net energy savings. We will now discuss an algorithm called Energy-efficient Device Scheduling or EEDS [ChG06]. The pseudo code for the algorithm can be

```
1   Preprocessing:
2     Compute break-even time BE(λ_k) (1 ≤ k ≤ m) for each device.
3   Schedule jobs at time t when a job is put in the ready queue or is
    completed
4   // A job can join the ready queue when all needed devices are active.
5     J_run ← the job with the highest EDF priority in the ready queue.
6     Dispatch J_run;
7   Perform device state transitions at time t when a job is released,
    completed or the timer to reactivate a device is reached.
8     If (t: ∃λ_k, λ_k ∉ Dev(J_run) && λ_k = active
      && DS(λ_k, t) > BE(λ_k))
9       λ_k ← sleep;
10      // Up(λ_k) is the timer set to reactivate λ_k.
11      Up(λ_k) ← t + DS(λ_k, t) − t_wu(λ_k);
12    End If
13    // Device slack may increase; update Up(λ_k) for sleeping devices
      // in this case.
14    If (t: ∃λ_k, λ_k = sleep && t + DS(λ_k, t) − t_wu(λ_k) > Up(λ_k))
15      Up(λ_k) ← t + DS(λ_k, t) − t_wu(λ_k);
16    End If
17    // Reactivate λ_k when the timer is reached.
18    If (t: ∃λ_k, λ_k = sleep && Up(λ_k) = t)
19      λ_k ← active;
20    End If
21  End
```

Figure 19: The EEDS scheduler [ChG06]. Notations: λ_k indicates device k; BE is the breakeven time; J_{run} the job currently being executed; $Dev(J_{run})$ the set of devices J_{run} needs; $DS(\lambda_k, t)$ the device slack time of λ_k at time t; $Up(\lambda_k)$ the wakeup time of λ_k; $t_{wu}(\lambda_k)$ the transition delay time of λ_k.

seen in Figure 19. The algorithm supports devices with two power states, sleep and active. EEDS calculates the *breakeven time* for each device. This is the length of the time period it is worthwhile to put the device in sleep mode. For shorter periods than this, the state transition costs will exceed the net gain. On line 2 in the pseudo code, EEDS calculates the breakeven time BE of each device. The length of the device's breakeven time depends on the properties of the device: how long a time the transition from active to sleep state (and vice-versa) takes, how much energy the transition(s) require, and how much energy the device spends in active vs. sleep state.

EEDS utilizes a data structure of the type *queue* where the active jobs are ordered according to the EDF principle – the one with the closest deadline at the head of the queue. This job is scheduled (line 6). We call *device slack time* the length of the

time period until device λ_k is needed next time. On line 8 we resolve whether there is a woken up device λ_k whose device slack DS (the length of the time period when the device is not needed) is greater than its breakeven time BE. Such devices may be put to sleep, which is done on line 9. In order to wakeup these devices so that they will be ready when needed next time, EEDS sets a timer on line 11.

As the timer value we put the current time t added with the device's slack time $DS(\lambda_k, t)$ subtracted with the wakeup time $t_{wu}(\lambda_k)$. Due to the dynamic properties of jobs, the device slack time may increase even during the sleep time. Therefore the timer of the device may be updated on lines 14 and 15. On line 18 we check whether the timer of a device has expired, and if so, wakeup the device on line 19.

5.1.4 The Energy-Optimal Device Scheduler

The schedulers described earlier are all online schedulers. Swaminathan and Chakrabarty [SwC05] have in 2005 published a real-time device scheduler aimed at offline use. It differs from all previous algorithms described in this thesis also in the sense that it completely rejects both EDF and RM and implements a scheduling mechanism of its own. This algorithm is called *Energy-optimal device scheduler* (EDS). In order to find an energy optimal device schedule this algorithm builds a decision tree using an iterative algorithm. To limit memory space requirements, EDS prunes branches from the tree when possible.

	j_1	j_2	j_3	j_4	j_5	j_6	j_7
a_i	0	0	3	4	6	8	9
c_i	1	2	1	2	1	2	1
d_i	3	4	6	8	9	12	12

Table 4: The EDS example job set [SwC05], where a_i indicates the arrival time; c_i the execution time, and d_i the deadline of a job. The odd-numbered jobs belong to task τ_1 and use device k_1, and the even-numbered jobs belong to task τ_2 and use device k_2.

Let us start our study of the EDS algorithm by considering an example. In Table 4 we have a set of jobs from two tasks, $\tau 1$ (the odd-numbered jobs) and $\tau 2$ (the even-numbered jobs). $\tau 1$ uses the device k_1 and $\tau 2$ the device k_2. The mission of EDS is to find such start times for all of these jobs, that device energy use is minimized while deadlines are met. EDS solves this problem by building a schedule

tree. The beginning of the schedule tree built using the task set of Table 4 can be seen in Figure 20.

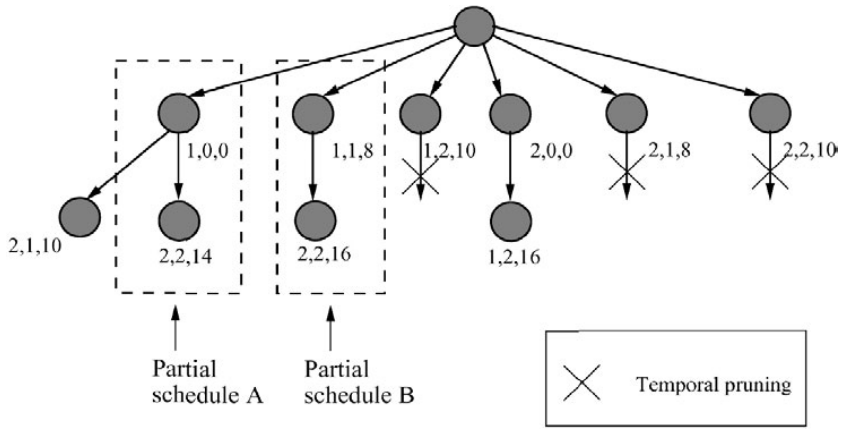


Figure 20: The EDS scheduling tree after jobs j_1 and j_2 have been scheduled [SwC05]. Syntax: $(j_i, time, E_i)$, where j_i is the job number, $time$ the start time of j_i and E_i the device energy consumption up to $time$.

The schedule tree consists of vertices, where each vertex is represented as a 3-tuple $(j_i, time, E_i)$. In this tuple j_i indicates the job number (from Table 4), $time$ is a valid start time for j_i according to this schedule, and E_i indicates the amount of *energy* spent by the device i according to this schedule up to $time$. Vertices (x_1, x_2, x_3) and (y_1, y_2, y_3) are connected by an edge if y_1 can be scheduled at y_2 when x_1 has been scheduled at x_2 [SwC05].

Calculating the energy consumption Assume that each device has two states, a low power sleep state $ps_{l,i}$ and a high power working state $ps_{h,i}$. Let $t_{0,i}$ be the transition time between these states, and $P_{0,i}$ be the transition power requirement. Let $P_{s,i}$ and $P_{w,i}$ indicate the power spent when in sleep and working states, respectively. The energy requirement is calculated using the formula

$$E_i = P_{w,i}t_{w,i} + P_{s,i}t_{s,i} + mP_{0,i}t_{0,i} \tag{13}$$

where m is the amount of state transitions; $t_{s,i}$ is the time spent in sleep state, and $t_{w,i}$ is the time spent in working state [SwC05].

Building the schedule tree The building of the schedule tree is started with a dummy vertex $(0, 0, 0)$. According to Table 4, jobs j_1 and j_2 have been released at time 0, and will hence be added to the tree. Let's begin with j_1. The completion

(execution) time of j_1 is 1 and its deadline is 3 (Table 4). Therefore, j_1 may be scheduled at time 0, 1 and 2. We therefore add three vertices, $(1, 0, e_1)$, $(1, 1, e_2)$ and $(1, 2, e_3)$ to the tree, and connect these with an edge to the root vertex. The energy consumption value e_i for each vertex is calculated using Equation 13, and the correct values for e_1, e_2 and e_3 are 0, 8 and 10, respectively (we will here exclude the details of energy consumption calculation). We add these vertices to the tree, as can be seen in Figure 20. In a similar fashion, we add to the tree the vertices of j_2 connecting it to the root vertex, because even j_2 was released at time 0. According to Table 4, the completion time of j_2 is 2 and its deadline is 4. Therefore, it can be scheduled at times 0, 1 and 2. The corresponding values for e_i (calculated using Equation 13) are 0, 8 and 10, respectively. Hence, we add the vertices $(2, 0, 0)$, $(2, 1, 8)$ and $(2, 2, 10)$ to the tree, as can be seen in Figure 20.

Pruning the schedule tree EDS performs both *temporal* and *energy* pruning. This way it will reduce the size of the schedule tree in order to ease memory space and processor time requirements. Continuing with our example, as the next step, EDS performs temporal pruning. Consider the vertex $(1, 2, 10)$ in Figure 20. If j_1 is scheduled at time 2, it will finish at time 3, because its completion time is 1 (Table 4). However, finishing j_1 at time 3 would mean that the execution of j_2 would start no earlier than at 3, and because the completion time of j_2 is 2, j_2 would miss its deadline at 3. Therefore, this schedule is unfeasible, and the branch of the tree starting with node $(1, 2, 10)$ can be pruned. This is indicated by the cross in Figure 20. By similar reasoning, we will also be able to prune the branches starting with vertices $(2, 1, 8)$ and $(2, 2, 10)$. Let us first consider $(2, 1, 8)$. If the first scheduled job is j_2 at 1, it will finish at 3 but then j_1 would certainly miss its deadline at 3, and hence this schedule is unfeasible, and this branch can be pruned. Similarily, considering vertex $(2, 2, 10)$, if j_2 at 2 is the first scheduled, it will finish at 4, but then j_1 would have missed its deadline at 3, so also this branch can be pruned.

The second form of pruning utilized by EDS is energy pruning. In Figure 21, which displays the entire scheduling tree, consider the vertices $(2, 2, 14)$ and $(2, 2, 16)$ located two edges away from the root vertex. These vertices indicate two schedules of the same job, 2, at exactly the same point in time, also 2. Also, in both branches, exactly the same job have been previously scheduled. However, the latter of the schedules consume 16 units of energy in comparison to 14 of the first one. Because our aim is to minimize energy consumption we may here utilize energy pruning, and discard the rest of the branch with the higher energy consumption. Energy

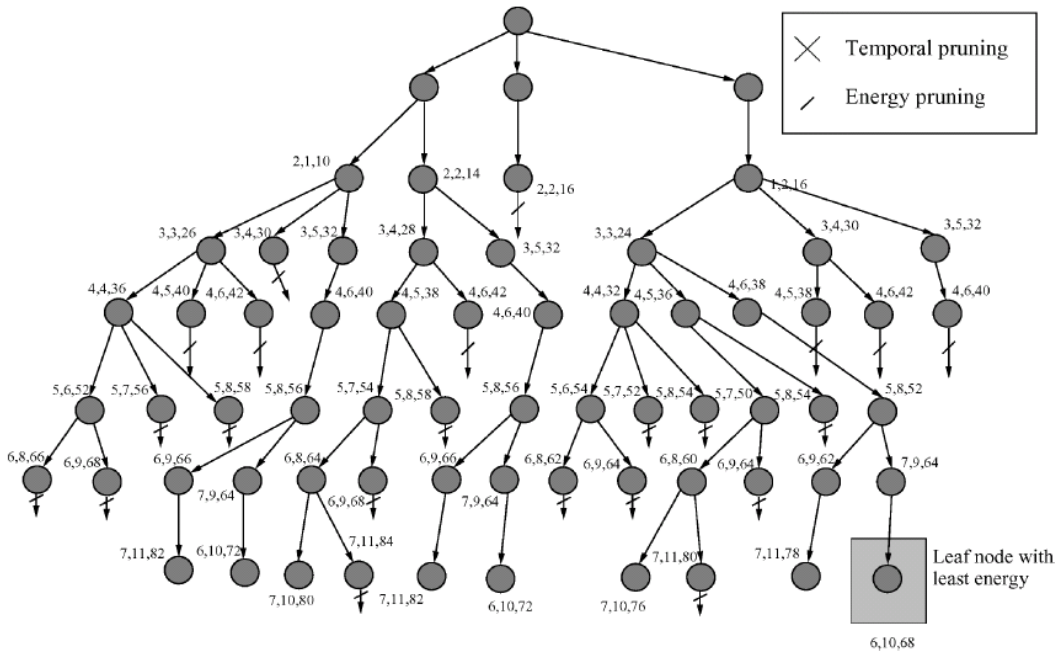


Figure 21: The complete EDS scheduling tree [SwC05]. The least energy consuming schedule of the 7 jobs has been found.

pruning can always be made when two jobs are scheduled at the same time, and the order of the previously scheduled jobs among both branches are identical [SwC05]. Once we have finished the final scheduling tree, i.e. included all the leaf vertices, we choose from among the leaf vertices the node consuming the least energy (68) by eliminating higher-energy vertices. The path from the dummy vertex $(0, 0, 0)$ to this lowest-energy leaf vertex $(6, 10, 68)$ indicates an energy-optimal schedule of the job set of Table 4.

The EDS pseudo code The pseudo code of the iterative EDS algorithm can be seen in Figure 22. As initialization, on line 2, the dummy vertex $(0, 0)$ is put into the `openList`. In the **for** loop starting on line 3 all vertices in the `openList` are processed. On line 5, a set τ' is generated out of the jobs that have been released up to the time stamp of the current vertex. Out of these jobs we generate new vertices, and prune those that would be unfeasible. On lines 15–22 we compare all pairs of vertices on the current height of the tree, and if two with identical scheduling occasions are found, we prune the one with the higher energy requirement. The EDS algorithm is finished on lines 25–27 when all jobs have been scheduled, i.e., when the height of the tree equals the number of jobs.

```
Procedure EDS(J, l)
J: Job set.
l: Number of jobs.
openList: List of unexpanded vertices.
currentList: List of vertices at the current depth.
t: time counter.

1. Set t = 0; Set d = 0;
2. Add vertex (0,0) to openList;
3. for each vertex v = (j_i, time) in openList {
4.     Set t = time + c_i;
5.     Find set of all jobs J' released up to time t;
6.     for each job j ∈ J' {
7.         if j has been previously scheduled
8.           continue;
9.         else {
10.          Find all possible scheduling instants for j; /* Temporal pruning*/
11.          Compute energy for each generated vertex;
12.          Add generated vertices to currentList;
13.        }
14.    }
15.    for each pair of vertices v_1, v_2 in currentList {
16.        if j_1 = j_2 and
             partial schedule(v_1) = partial schedule(v_2) {
17.          if E_v1 > E_v2
18.            Prune v_1;
19.          else
20.            Prune v_2;
21.        }
22.    }
23.    Add unpruned vertices in currentList to openList;
24.    Clear currentList;
25.    Increment d;
26.        If d = l
27.          Terminate.
28. }
```

Figure 22: The EDS pseudo code [SwC05].

Despite its pruning technique, its memory and computation time requirement of EDS may be excessive [SwC05]. EDS is aimed at offline use, meaning that the schedule is computed before run-time. Also, the schedule calculated by EDS is non pre-emptive. Jobs are executed from start to finish without context switches. Therefore, jobs may have to wait for long times while large jobs are being processed.

5.1.5 Comparing the Presented Algorithms

We now have presented four algorithms for power-aware device scheduling. Out of these schedulers, LEDES and MUSCLES are add-ons to the system's task scheduler.

They also have shortcomings. For instance the basic assumption in LEDES is that the transition time of a device may never exceed the execution time of any job. As a process in a real-time system may consist of just a few lines of machine code that, for instance, reads a sensor measurement figure, and for instance a hard-drive may take several seconds to wake up from sleep state, we always cannot build upon this assumption.

The bigger brother to LEDES is MUSCLES which supports several sleep states. However it does not support several operational states. Recall from our discussion of processors, that many contemporary CPU's provide several operational states, where lesser throughput is provided for less energy cost. MUSCLES does not support any similar functionality on devices.

Neither LEDES nor MUSCLES calculate the net gain of state transitions. This is, however, done by EEDS which, essentially, is an enhanced EDF scheduler. Devices that are currently not needed and which in spite of transition costs are beneficial to be slept down, are put to sleep and awoken with a timer.

Our final algorithm, EDS, calculates an energy-optimal schedule using a decision tree. Due to its complexity, this algorithm is intended for offline use. The authors of EDS also have published a heuristic algorithm, *Maximum Device Overlap* (MDO), which seeks an approximate solution to the same problem and operates in polynomial time [SwC05].

6 System-Level Power Aware Scheduling

By using Dynamic Voltage Scaling, the processor's operating frequency and voltage may be regulated during run-time. Since the processor's energy consumption cubically depends on frequency and voltage, impressive CPU energy reductions may be achieved using this technique. There is, however, a downside complicating the matter. Besides the processor, computer systems consists of other components, such as memory and cache memory chips, graphic adapters, network cards, bus controllers, graphics processors, modems, wireless network adapters, and so forth. Performing a calculation takes a longer time when the processor speed has been lowered. When considering the CPU energy consumption in isolation, a frequency and voltage reduction using DVS indeed results in energy savings. However, as the processing time increases, all the other components need to be longer in the standby state. Components such as memory chips generally require a fixed power supply regardless of the DVS setting of the CPU. Hence, when system-level energy reductions are the aim, considering the CPU power requirement in isolation is not sufficient. Most early DVS based CPU scheduling algorithms have chosen to overlook this fact in their basic assumptions [FEL04]. This is also the case with the algorithms described in Section 4.

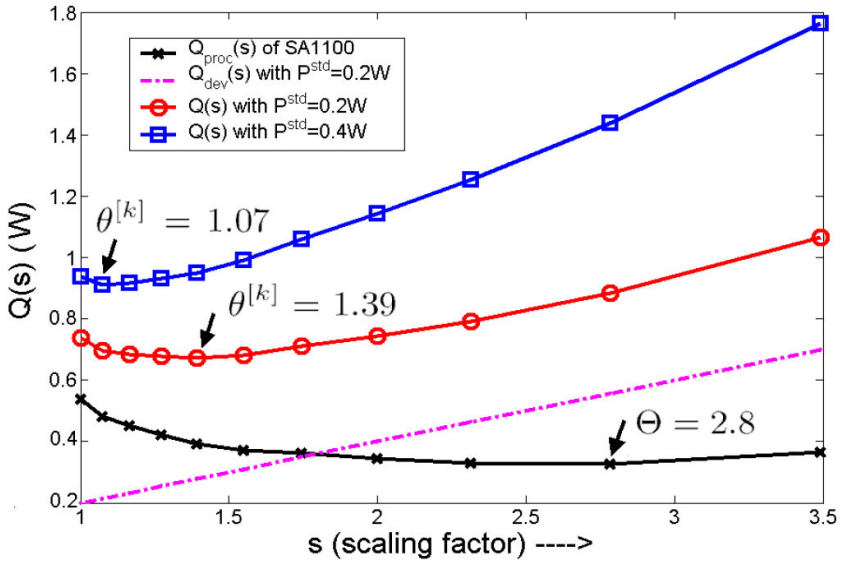


Figure 23: The effect of the processor scaling factor s on system-level energy consumption [ZhC05].

Consider Figure 23. The X axis indicates a StrongArm SA 1100 processor scaling factor s defined as $s = \frac{max\_frequency}{current\_frequency}$, and the Y axis indicates power consump-

tion of a task, in watts. In the graph, the crossed line $Q_{proc}(s)$ depicts the power consumption of the SA 1100 processor alone. The possible s values are the discrete scaling factors provided by this processor. The energy optimal s value $\Theta = 2.8$ is marked in the graph. Next, consider the dotted line $Q_{dev}(s)$. This monotonically rising line indicates the power consumption of the device set needed by the task, excluding the processor. As the scaling factor s increases, and hence the CPU speed decreases and processing times increase, the aggregate power consumption of the device set increases. The line with circles shows the combined processor and device power requirement when static device power requirement is considered to be $0.2W$, which often is the case with for instance *Synchronous Dynamic Random Access Memory* (SDRAM) memory chips [ZhC05]. The optimal scaling factor, when considering both the power consumption of the processor and the device set, is 1.39 and this value is marked in the graph with $\theta^{[k]}$. The line with squares shows the combined energy optimal voltage scaling factor when the device set static power requirement is considered to be $0.4W$. This is the case with many flash drives. With this power requirement, the energy-optimal scaling factor $\theta^{[k]} = 1.07$. Comparing this value to the CPU energy optimal value of 2.8 and the $0.2W$ optimal value 1.39 clearly illustrates how the net gain of aggressive DVS values decrease as processor independent energy consumption increases. It has actually been shown [ZhC05] that when device energy consumption is considerably large compared to CPU energy consumption, DVS implementations actually can spend more energy than non-DVS approaches.

As the processor takes a longer time to perform calculations, the standby energy requirement of the device set rises. An energy-efficient scheduling algorithm, therefore, needs to consider system-wide energy consumption when calculating an optimal scaling factor for the processor. In the next subsections, two recent algorithms will be explored.

6.1 duSYS: A System-Level EDF Algorithm

Zhuo and Chakrabarti [ZhC05] have published an EDF based system-level power-aware real-time scheduling algorithm called *duSYS*. Its high-level pseudo code is given in Figure 24. What makes this algorithm different from processor scheduling algorithms explored in Section 4 is the calculation of the energy-optimal DVS scaling factor. The idea behind duSYS is that the system-level energy consumption can be written as a function of the processor's scaling factor s.

Let P_{proc} be the processor operating power consumption, and $Pd^{[i]}$ be the standby power consumption of the device set needed by task i. Now, the energy consumption of task i can be written as $Q(s) = Q_{proc}(s) + Q_{dev}(s)$. Here, $Q_{proc}(s) = s \times P_{proc}$ and $Q_{dev}(s) = s \times Pd^{[i]}$ [ZhC05]. Because processors typically only have a handful of available speed scaling modes (values for s), for instance the SA 1100 has 11, it is possible for every task to numerically evaluate each of them [ZhC05] and choose the one that will yield the lowest aggregate power consumption. This optimal value is denoted by θ_i in duSYS. The mission of duSYS is to find for the scheduled active job J_{act} an optimal scaling factor s_{act}. The duSYS algorithm calculates the scaling factor using Equation 14 [ZhC05]:

$$s_{act} = min(\frac{D_{act} - t}{E_{act}}, \theta_{act}, du(t)) \tag{14}$$

where D_{act} is the active job's absolute deadline, t is the current time, E_{act} is the jobs worst-case execution time (the execution time that has been budgeted to the task), and θ_{act} is the optimal voltage scaling factor for the task based on the task's static execution parameters. In duSYS, θ_{act} is computed offline. Due to the dynamic nature of jobs, real execution times vary greatly, and are generally shorter than the budgeted static ones. In order to utilize emerging slack times for energy savings, duSYS also calculates and considers the dynamic utilization, $du(t)$, when selecting the appropriate scaling factor. The value $du(t)$ is calculated using Equation 15 [ZhC05].

$$du(t) = \frac{H - t - U^{-1} \times (W - E_{act})}{E_{act}}, (0 \leq t \leq H) \tag{15}$$

where H is the hyper period, i.e., the least common multiple (LCM) of the periods of the scheduled tasks, W is the estimated remaining workload and U is the utilization degree of the system. Using the value $du(t)$ for processor frequency scaling, all slack available at time t may safely be granted to the active job, while timely execution of the rest of the jobs is also being guaranteed. The term $\frac{D_{act}-t}{E_{act}}$ in Equation 14 ensures that deadlines are not violated [ZhC05].

To summarize, when selecting the optimal scaling factor s_{act} for the active job, duSYS chooses from among three different candidates the smallest one according to Equation 14. Out of these three candidates, θ_{act} is calculated offline and is based on static information (period P_i, worst-case execution time E_i) about the task, whereas the purpose of $du(t)$ is to utilize slack emerging when jobs execute faster than their budgeted worst-case execution times.

```
1    W = hyperperiod × U
2    while time() < hyperperiod do
3       determine s_act and execute J_act using s_act;
4       if J_act is not finished then
5          ExecutedPart = current_duration/s_act;
6          W = W − ExecutedPart;
7          E_act = E_act − ExecutedPart;
8          ActualExecutionTime_act = ActualExecutionTime_act − ExecutedPart;
9       else
10         W = W − E_act;
11      end if
12   end while
```

Figure 24: The high-level pseudo code of the duSYS algorithm [ZhC05]. W denotes the estimated remaining workload, E_{act} the budgeted execution time, and U the system utilization degree.

The pseudo code of duSYS can be seen in Figure 24. Released jobs are considered to be sorted in a queue with the job with the highest EDF priority at the head of the queue. On line 1, the estimated workload of the system is calculated. On line 3 the highest priority job is scheduled using the scaling factor s_{act} which has been calculated using Equation 14. During the execution of J_{act}, dynamic runtime information is maintained on lines 5–8. This information is used when calculating $du(t)$, which seeks to utilize slack times for power savings. When choosing the optimal scaling factor, duSYS considers the combined processor and device power consumption in order to minimize system-wide power requirements.

6.2 The Critical Speed DVS Algorithm

Next we will consider an earlier EDF based power-aware system-wide real-time scheduling algorithm [JeG04]. We call this algorithm *Critical Speed DVS* (CS-DVS). Like duSYS, CS-DVS considers both CPU and device energy consumption when calculating an energy-optimal DVS setting. In CS-DVS, the energy consumption E_i of a task τ_i is given by Equation 16 [JeG04]:

$$E_i(\eta) = \frac{C_i}{\eta} P(CPU, \eta) + \sum_{j=1}^{n} \frac{C_i^{R_j}}{\eta} P(R_j) \tag{16}$$

where $\eta \in [0, 1]$ represents the processor *slowdown factor* [JeG04]. This value indicates the fraction of the maximum CPU speed at which the processor is being run ($\eta = 1$ meaning the maximum speed), and corresponds to the scaling factor s used in duSYS. In Equation 16, C_i indicates the number of processor cycles budgeted to the task τ_i, and $C_i^{R_j}$ the number of cycles that device R_j spends in the standby state during the execution of the task τ_i. The notation $P(CPU, \eta)$ represents the power consumption of the CPU at slowdown factor η, and $P(R_j)$ indicates the power consumption of the device R_j. In essence, the first term in Equation 16 represents the CPU power usage at slowdown factor η, and the second term represent the sum of the standby energies consumed by the set of devices R_j that task τ_i uses at slowdown factor η. Naturally, even components such as system memory may be modeled as a device.

What CS-DVS needs to do is to minimize the energy consumption given by Equation 16. It needs to find the η that yields the lowest total energy consumption for the task. Possible η values are the discrete speed settings provided by the underlying processor architecture. CS-DVS finds the η giving the lowest total energy by calculating Equation 16 for each available η value [JeG04], and then choosing the optimal η. As visualized by Figure 23, this value need not be the one that minimizes the CPU power usage. The η value that yields the lowest total energy consumption is called the *critical speed* of the task. Because each task may have different execution times and use a different set of devices, their critical speeds need not be the same.

The pseudo code of the CS-DVS Algorithm is given in Figure 25. On line 1, the critical speed for each task is calculated, and on line 2 each task τ_i is initialized its individual critical speed η_i. Energy-optimal scaling factors might cause the task set to become unfeasible, i.e. EDF timeliness guarantees would be violated. Hence, CS-DVS might need to increase the scaling factor of some task(s). This is done in the **while**-loop on lines 3–8. A possible candidate task τ_m for speed increase fulfills two conditions (line 4). Firstly, the task's current scaling factor η_m is not the maximum speed (line 5). The second condition (line 6) is more complicated. We wish to choose the task for which a speed increase from the current factor η_i to the next one η_{i+1} causes as small an energy consumption increase per time unit as possible. Here, ΔE_m represents the energy consumption increase between η_i and η_{i+1}, and Δt_m the time gained by the speed-up [JeG04]. From among the candidates the task with the lowest $\Delta E_m / \Delta t_m$ value is chosen, and this task's η is increased. This process is repeated (line 3) until the task set becomes feasible according to the EDF principle.

```
1   Compute the critical speed for each task;
2   Initialize η_i to critical speed of τ_i;
3   while (not feasible) do
4       Let τ_m be task satisfying:
5       (a) η_m is not the maximum speed;
6       (b) ΔE_m/Δt_m is minimum;
7       Increase speed of task τ_m;
8   end while
9   return slowdown factors η_i;
```

Figure 25: The Critical Speed DVS (CS-DVS) Algorithm in pseudo code [JeG04].

6.3 Comparing the presented algorithms

In this section we explored two power-aware real-time scheduling algorithms that consider system-wide energy consumption when choosing the optimal DVS setting for the processor. Both algorithms model a real-time task's energy consumption as the sum of CPU and device set energy consumptions. The slower the processor is run, the more standby energy the devices require. A power-aware real-time scheduler needs to consider this when making DVS setting decisions.

The considered algorithms were duSYS [ZhC05] and CS-DVS [JeG04]. Both algorithms are based on the EDF principle and provide a hard real-time timeliness guarantee. The main difference between the algorithms is that duSYS is able to utilize dynamically emerging job slack, whereas CS-DVS operates on static pre-runtime task information only. It is well known that real-time jobs hardly ever consume all the processor time that has been allocated to them, but execute faster than budgeted. Hence, duSYS is potentially more energy-optimal than CS-DVS.

7 Summary

In a real-time system, calculations need not only be correct, but also be finished within a pre-defined deadline. The first serious real-time scheduling algorithms, presented in Section 2, were Rate Monotonic and Earliest Deadline First [LiL73]. In a hard real-time system, for instance in a pacemaker, the meeting of every single deadline is crucial. In a soft real-time system, for instance a video player, occasional deadline misses are tolerated.

Many contemporary real-time systems operate on constrained devices with limited battery power. Power awareness in constrained devices is discussed in Section 3. Extensive energy savings can be achieved by utilizing Dynamic Voltage Scaling (DVS) [Gro03, VeF05] to change the operating frequency and voltage of the processor during run-time. Using the Advanced Configuration and Power Interface (ACPI) [HIM06], the operating system may shut down devices, such as disk drives, for time periods when the devices are not needed.

Using low-power techniques, the challenge for the real-time scheduler is to maximize energy savings while guaranteeing that jobs meet their real-time deadlines. Due to device wakeup delay times, the scheduler needs to initiate the wakeup procedure of a slept-down device before the device is actually needed. If the device isn't awoken early enough, the job needing it might risk missing a deadline.

Advanced scheduling algorithms such as Feedback DVS-EDF [DMZ02] and duSYS [ZhC05] are also able to dynamically utilize emerging slack times for energy savings. Once one job finishes earlier than budgeted, the next job may have at its proposal extra execution time. The scheduler may use this slack time to conserve processor energy by executing the job slower.

Considerable research has been done in the field of power-aware real-time scheduling. The Rate Monotonic and Earliest Deadline First algorithms have been enhanced with power-aware properties. Power aware real-time algorithms for uniprocessor, device, and system-level scheduling are explored in Sections 4, 5 and 6, respectively.

References

BBC98 Benini, L., Bogliolo, A., Cavallucci, S. and Riccó, B., Monitoring system activity for OS-directed dynamic power management. *ISLPED '98: Proceedings of the 1998 international symposium on Low power electronics and design*, New York, NY, USA, 1998, ACM Press, pages 185–190.

But05 Buttazzo, G. C., Rate monotonic vs. EDF: judgment day. *Real-Time Systems*, 29,1(2005), pages 5–26.

ChG06 Cheng, H. and Goddard, S., Online energy-aware I/O device scheduling for hard real-time systems. *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, 3001 Leuven, Belgium, Belgium, 2006, European Design and Automation Association, pages 1055–1060.

DMZ02 Dudani, A., Mueller, F. and Zhu, Y., Energy-conserving feedback EDF scheduling for embedded systems with real-time constraints. *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, New York, NY, USA, 2002, ACM Press, pages 213–222.

FEL04 Fan, X., Ellis, C. S. and Lebeck, A. R., The synergy between power-aware memory systems and processor voltage scaling. *Power-Aware Computer Systems*, 3164(2004), pages 164–179.

Gro03 Grover, A., Modern system power management. *ACM Queue*, 1,7(2003), pages 66–72.

HIM06 Hewlett-Packard, Intel, Microsoft, Phoenix and Toshiba, Advanced configuration and power interface specification, revision 3.0b, 2006. `http://www.acpi.info/DOWNLOADS/ACPIspec30b.pdf`. [3.4.2007]

HwA00 Hwang, C.-H. and Wu, A. C.-H., A predictive system shutdown method for energy saving of event-driven computation. *ACM Transactions on Design Automation of Electronic Systems*, 5,2(2000), pages 226–241.

IGS02 Irani, S., Gupta, R. and Shukla, S., Competitive analysis of dynamic power management strategies for systems with multiple power savings

states. *DATE '02: Proceedings of the conference on Design, automation and test in Europe*, Washington, DC, USA, 2002, IEEE Computer Society, pages 117–123.

Int04 Intel Corporation, Enhanced Intel SpeedStep Technology for the Intel Pentium M Processor, 2004. `http://www.intel.com/design/intarch/papers/30117401.pdf`. [3.4.2007]

JeG04 Jejurikar, R. and Gupta, R., Dynamic voltage scaling for systemwide energy minimization in real-time embedded systems. *ISLPED '04: Proceedings of the 2004 international symposium on Low power electronics and design*, New York, NY, USA, 2004, ACM Press, pages 78–81.

Liu00 Liu, J. W. S., *Real-Time Systems.* Prentice Hall, New Jersey, USA, 2000.

LiL73 Liu, C. L. and Layland, J. W., Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20,1(1973), pages 46–61.

PLS01 Pouwelse, J., Langendoen, K. and Sips, H., Dynamic voltage scaling on a low-power microprocessor. *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, New York, NY, USA, 2001, ACM Press, pages 251–259.

PiS01 Pillai, P. and Shin, K. G., Real-time dynamic voltage scaling for low-power embedded operating systems. *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2001, ACM Press, pages 89–102.

RaS94 Ramamritham, K. and Stankovic, J., Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, 82,1(1994), pages 55–67.

ShC99 Shin, Y. and Choi, K., Power conscious fixed priority scheduling for hard real-time systems. *DAC '99: Proceedings of the 36th ACM/IEEE conference on Design automation*, New York, NY, USA, 1999, ACM Press, pages 134–139.

SwC00 Swaminathan, V. and Chakrabarty, K., Real-time task scheduling for energy-aware embedded systems, 2000. `http://www.ee.duke.edu/~krish/wip.pdf`. [27.6.2007]

SwC01 Swaminathan, V. and Chakrabarty, K., Investigating the effect of voltage-switching on low-energy task scheduling in hard real-time systems. *ASP-DAC '01: Proceedings of the 2001 conference on Asia South Pacific design automation*, New York, NY, USA, 2001, ACM Press, pages 251–254.

SwC03 Swaminathan, V. and Chakrabarty, K., Energy-conscious, deterministic I/O device scheduling in hard real-time systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22,7(2003), pages 847–858.

SwC05 Swaminathan, V. and Chakrabarty, K., Pruning-based, energy-optimal, deterministic I/O device scheduling for hard real-time systems. *ACM Transactions on Embedded Computing Systems*, 4,1(2005), pages 141–167.

Sta05 Stallings, W., *Operating Systems: Internals and Design Principles*. Pearson Prentice Hall, Upper Saddle River, NJ, USA, 2005.

VBH03 Viredaz, M. A., Brakmo, L. S. and Hamburgen, W. R., Energy management on handheld devices, 2003. `http://www.acmqueue.org/modules.php?name=Content&pa=showpage&pid=79`. [27.3.2007]

VeF05 Venkatachalam, V. and Franz, M., Power reduction techniques for microprocessor systems. *ACM Computing Surveys*, 37,3(2005), pages 195–237.

WRJ06 Wu, H., Ravindran, B., Jensen, E. D. and Li, P., Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems. *ACM Transactions on Embedded Computing Systems*, 5,3(2006), pages 513–542.

YuN03 Yuan, W. and Nahrstedt, K., Energy-efficient soft real-time CPU scheduling for mobile multimedia systems. *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, New York, NY, USA, 2003, ACM Press, pages 149–163.

YuN06 Yuan, W. and Nahrstedt, K., Energy-efficient CPU scheduling for multimedia applications. *ACM Transactions on Computer Systems*, 24,3(2006), pages 292–331.

ZhC05 Zhuo, J. and Chakrabarti, C., System-level energy-efficient dynamic task scheduling. *DAC '05: Proceedings of the 42nd annual conference on Design automation*, New York, NY, USA, 2005, ACM Press, pages 628–631.

Appendix 1. The entire Feedback DVS-EDF algorithm

This is the entire Feedback DVS-EDF algorithm [DMZ02] presented in Section 4.1.3.

Procedure Initialization
  for each $T_k \in \{T_1, T_2, \ldots, T_n\}$ **do**
    $C_{avg\_k} \leftarrow C_k/2$
    $left_{k0} = C_k$
  $U \leftarrow \frac{C_1}{P_1} + \frac{C_2}{P_2} + \ldots + \frac{C_n}{P_n}$
  $P_{n+1} \leftarrow P_1, C_{n+1} \leftarrow P_1 \times (1-U), c_{n+1} \leftarrow 0\}$ (0)
  let $slack \leftarrow 0$

Procedure TaskActivation($\mathbf{T_{ij}}$)
  if processor was idle for d **then**
    $slack \leftarrow slack - d$
  if T_{ij} was preempted/interrupted **then**
    $slack \leftarrow slack + slack_{ij} - left_{ij}$
  forall T_{ab} idle task jobs in $d_{pk}..d_{ij}$ **do**
    $slack \leftarrow slack + C_a$ (1)
  $\alpha\prime \leftarrow \min\{\frac{f_1}{f_m}, \ldots, \frac{f_m}{f_m} | \frac{f_i}{f_m} \geq \frac{C_{avg\_i}}{C_{avg\_i}+slack}\}$
  if $(\alpha\prime = 1)$ **then**
    $C_A \leftarrow 0$
  else
    $C_A \leftarrow slack \times \alpha\prime/(1-\alpha\prime)$
  SetInterrupt($T_i, C_A/\alpha\prime$)
  SetFrequency($\alpha\prime$)

Procedure TaskPreemption($\mathbf{T_{ij}}$)
  $slack_{ij} \leftarrow c_{ij} + left_{ij} - C_i$
  $slack \leftarrow slack - slack_{ij}$
  let $s \leftarrow slack_{ij}$
  forall T_{ab} idle task jobs
    in $d_{ij}..d_{pk}$ and in $r_{ij}..t$ **while** $s > 0$ **do** (2)
      $slack \leftarrow slack - C_a$
      $s \leftarrow slack - C_a$
      reserve C_a for T_{ij}

Procedure TaskCompletion($\mathbf{T_{ij}}$)
  if T_{ij} was preempted **then**
    if $c_{ij} > C_i$ **then** (late finish)
      $slack \leftarrow slack - c_{ij} + C_i$
  else (early finish)
    $slack \leftarrow slack + C_i - c_{ij}$
  forall T_{ab} idle task jobs in $r_{ij}..d_{nl}$ **do**
    $slack \leftarrow slack - C_a$ (3)
  $C_{avg\_i} \leftarrow (C_{avg\_i} \times (j-1) + c_{ij} \times \alpha\prime)/j$
  $left_{i(j+1)} = C_i$

Procedure SetInterrupt($\mathbf{T_{ij}}, \mathbf{C_A}$)
  Set timer interrupt for T_{ij},
  triggered C_A time units later

Procedure InterruptHandler($\mathbf{T_{ij}}$)
  if T_{ij} *not completed* **then**
    $slack \leftarrow slack - (c_{ij} + left_{ij} - C_i)$
    SetFrequency(1)

Procedure SetFrequency($\alpha\prime$)
  $f \leftarrow \alpha\prime \times f_m$