Power Management in the EPOS System

Geovani Ricardo Wiedenhoft, Lucas Francisco Wanner,
Giovani Gracioli and Antônio Augusto Fröhlich
Laboratory for Software and Hardware Integration
Federal University of Santa Catarina
P.O.Box 476, 88040900 – Florianópolis – Brazil
{grw,lucas,giovani,guto}@lisha.ufsc.br

ABSTRACT

Power management strategies for embedded systems typically rely on static, application driven deactivation of components (e.g. sleep, suspend), or on dynamic voltage and frequency scaling. However, the design and implementation of these strategies in embedded operating system often fail to deal with real-time and quality-of-service (QoS) requirements.

The EPOS system implements an infra-structure that supports both static (application-driven) and dynamic (system-driven) power management. In this work, this infrastructure is used to explore energy as a parameter for QoS in embedded systems, with the goal of guaranteeing energy consumption metrics, while preserving the deadlines of essential (hard real-time) tasks. Given a set of real-time tasks and their associated energy consumption, we provide equations to check schedulability in project-time. At runtime, a preemptive scheduler for imprecise tasks prevents the execution of optional subtasks whenever there is the possibility of deadline loss or depletion of the energy source. We show that this mechanism is effective in controlling energy consumption and ensuring "best-effort" computation without deadline loss.

Categories and Subject Descriptors

B.10.1 [**Power Management**]: Energy-aware systems; D.4.1 [**Process Management**]: Scheduling; D.4.7 [**Organization and Design**]: Real-time systems and embedded systems

General Terms

Algorithms, Design, Measurement

Keywords

Power Management, Embedded Systems, Imprecise Computation

1. INTRODUCTION

Power management techniques rely on the ability of certain components in a system to be turned on and off dynamically, enabling the system as a whole to save energy when those components are not being used. Additionally, techniques such as *Dynamic Voltage Scaling* (DVS) have been introduced to enable some components to operate at different energy levels along the time [3].

While these techniques are essential to the design of a power-aware embedded system, their use often incurs in overhead (e.g. through a dynamic power manager or monitor) or latency (e.g. the time required for a component to enter or exit an energy-saving mode). These factors cannot be ignored if the system is to respect real-time metrics. On the other hand, it is not enough to guarantee real-time metrics if, while doing so, the system exhausts its energy budget (e.g. discharges its batteries) and is unable to complete its tasks.

In this work, we use an application's energy budget, that is, its expected battery lifetime, as a QoS parameter. Our goal is not only to save energy, but to ensure that the application's energy budget and the deadlines of hard real-time tasks are met. In our system, QoS control of applications was inspired by imprecise computation [13], and divides each task into two sub-tasks, one mandatory, and one optional. By monitoring energy consumption, our scheduler is able to decrease QoS level by preventing the execution of optional subtasks in order to reduce energy consumption when the system is overusing its budget.

We provide equations to analyse the schedulability of a set of tasks in terms of deadlines and energy budget at project time. At runtime, our scheduler periodically checks energy levels to ensure that optional sub-tasks only execute when there is enough energy to meet the budget specified by the application. This control creates more idle periods in the system, in which the scheduler can use power management techniques to reduce the energy consumption of components.

The remainder of this text discusses power management strategies, their use in embedded real-time systems, their design and implementation in the EPOS [5, 14] system, and the combination of power management techniques, real-time scheduling and imprecise computation mechanisms to provide QoS in terms of energy.

2. RELATED WORK

Few systems targeting embedded computing can claim to deliver a real Power Management API. Nevertheless, most systems do deliver mechanisms that enable programmers to directly access the interface of some hardware components. This is the case of popular embedded operating systems like μCLINUX and TINYOS. These mechanisms, though not specifically designed for power management, can be used for that purpose at the price of binding the application to hardware details.

μCLINUX, like many other UNIX-like systems, does not feature a proper power management API [1]. Some device drivers provide power management functions inspired on ACPI [7]. Usually these mechanisms are intended to be used by the kernel itself, though a few device drivers export them via the **/sys** or **/proc** file systems, enabling applications to directly control the operating modes of associated devices.

TINYOS, a popular operating system in the wireless sensor network scene, allows programmers to control the operation of hardware components through a low-level, architecture-dependent API [9]. This API ensures direct access to the hardware and thus can be used for power management. OS-driven power management is implemented by the task scheduler, which makes use of the **StdControl** interface to start and stop components. When the scheduler queue is empty, the main processor is put in *sleep* mode. This way, new tasks will only be enqueued during the execution of an interrupt handler. This method yields good results for the main microcontroller, but leaves more aggressive methods, including starting and stopping peripheral components up to the application. When compared to μCLINUX, TINYOS delivers a lighter mechanism, more adequate for most embedded systems, yet suffers from the same limitations with regards to usability and portability.

There is an important connection between real-time requirements and power management strategies. On the one hand, a power management strategy may introduce overhead (e.g. through a dynamic power manager or monitor) or latency (e.g. the time required for a component to enter or exit a energy-saving mode) into a real-time system. On the other hand, information contained in scheduling algorithms may be used to guide the power management strategy dictating, for example, when to reduce processor frequency or when to turn off a given peripheral component. Recent projects by Yuan, Scordino, and Niu have explored this connection [15, 16, 21].

GRACE-OS is an energy-efficient operating system for mobile multimedia applications [21]. This system uses a cross-layer adaptation technique to guarantee quality-of-service on systems with adaptive software and hardware. It combines real-time scheduling with DVS mechanisms to allow dynamically managing energy consumption. The GRACE-OS scheduler decides each task's execution speed and time, based on a probabilistic estimation of how many cycles each task will need to complete its computation. GRACE-OS was implemented over the LINUX operating system and supports soft real-time tasks. GRUB-PA shares the same objectives of GRACE-OS, but supports both soft and hard real-time tasks [16]. In GRUB-PA, the scheduler also reduces processor frequency to save energy, but without compromising the deadlines of real-time tasks.

Niu proposed to minimize energy consumed by soft real-time systems while guaranteeing quality-of-service requirements [15]. This goal is achieved through the use of a hybrid static/dynamic scheduling algorithm that uses DVS mechanisms and partitions the set of tasks in mandatory and optional tasks. In this work, the quality-of-service requirements are qualified by *(m,k)* constraints which specify that tasks must meet at least m deadlines in any k consecutive task releases.

Other projects explore the trade-off between application's quality-of-service and energy consumption through adaptative scheduling. The ODYSSEY system periodically monitors an application's energy budget, and selects different power consumption and performance levels according to the application's needs [4]. Whenever energy consumption is too high, the system decreases QoS by selecting lower performance and power consumption levels. This allows the system to meet a specified lifetime by dinamically adapting performance to save energy.

ECOSYSTEM [22] is another operating system that supports dynamic QoS adaptation. This system is based in a "currency" that applications use to allocate ("to pay for") system resources (e.g., access to memory, network or disks), called *currentcy*. The system distributes *currentcies* periodically to tasks accordingly to an equation that defines the discharge rate that the system battery can assume to force the system to last for a defined period of time. This allows applications to adapt their execution based on their *currentcy* balance. This model unifies the calculation of energy on the various hardware devices, but requires applications to manage and act upon a specific energy budget.

Harada explores the trade-off between QoS maximization and energy consumption minimization, by allocating processor cycles and frequency with quality-of-service guarantees, and dividing tasks into mandatory and optional parts [6].

The division of mandatory and optional subtasks allows a system to satisfy timing requirements of real-time tasks preventing the execution of optional subtasks, and thus decreasing levels of quality-of-service that is the basic premise of imprecise computation [13]. Imprecise computation unites real-time computing and *best effort* techniques for, respectively, the mandatory and optional subtasks. The mandatory subtask of imprecise tasks generates imprecise results which reflect the minimum of quality-of-service to guarantee that these results are useful. These imprecise results have their quality enhanced when the optional subtask executes, generating the precise results.

While originally proposed to ensure hard real-time tasks deadlines, imprecise computation may be used to save energy, by preventing the execution of a given optional subtask according to a power management criterion. In this work, we introduce the concept of subtasks into the EPOS system to guarantee that all hard real-time deadlines are met, and that optional sub-tasks only execute when there is enough energy to meet the budget specified by the application.

3. EPOS

EPOS (Embedded Parallel Operating system) [5, 14] is a component-based framework for the generation of dedicated runtime support environments. The EPOS system framework allows programmers to develop platform-independent applications. Analysis tools allow components to be automatically selected to fulfill the requirements of these particular applications. By definition, one instance of the system aggregates all the necessary support for its dedicated application, and nothing else.

EPOS provides a wide set of operating system services through platform-independent interfaces, and supports a wide range of architectures, such as IA32, PowerPC, Sparc, MIPS, H8 and AVR [14]. EPOS provides an API for application-driven power management services that supports *power aware* operation of deeply embedded systems, without compromising application portability and without incurring excessive overhead. Furthermore, the system provides an active, oportunistic *power manager*, a system resource monitor, and a real-time scheduler. This section introduces the power management and real-time infrastructure in EPOS, which was used to support the contributions in this work.

3.1 Power-Management Infra-Structure

EPOS provides application-driven power management services, that allow *power aware* operation of deeply embedded systems, without compromising application portability and without incurring excessive overhead. The goal of this power management system is to allow applications to express when certain software components are not being used, permitting the system to migrate hardware resources associated with them to lower power levels.

The Power Management API featured in EPOS [10] supports direct application control of energy-related issues, without excluding cooperation with an automatic power manager. Furthermore, it is not restricted to the interface of hardware components, but also acts at the level of user visible components, thus promoting portability and usability. Finally, the API includes, but is not restricted to, semantic modes, enabling programmers to express power management operations easily, but avoiding the limitations of a small, fixed number of operating modes (as is the case of ACPI).

The EPOS power management API features two methods: **Power_Mode power()**, and **power(Power_Mode)**. The first method returns the current power mode of the associated object (i.e., component), while the second allows for mode changes. Aiming at enhancing usability, four power modes have been defined with semantics that must be respected for all components in the system: *off*, *stand-by*, *light* and *full*. Each component is still free to define additional power modes with other semantics, as long as the four basic modes are preserved. Enforcing universal semantics for these power modes enables application programmers to control energy consumption without having to understand the implementation details of underlying components (e.g., hardware devices). Allowing for additional modes, on the other hand, enables programmers to control the operation of special component precisely, whose operation transcend the pre-defined modes.

Mode	FULL	LIGHT	STANDBY	OFF
Energy	high	low	very low	none
Functionality	all	all	none	none
Performance	high	low	NA	NA
Wake up Delay	NA	very low	high	very high

Table 1: Semantic power modes of the proposed PM API.

Table 1 summarizes the semantics defined for the four universal operating modes. A component operating in mode *full* provides all its services with maximum performance, possibly consuming more energy than in any other mode. Contrarily, a component in mode *off* does not provide any service and also does not consume any energy. Switching a component from *off* to any other power mode is usually an expensive operation, specially for components with high initialization and/or stabilization times. The mode *stand-by* is an alternative to *off*: a component in *stand-by* is also not able to perform any task, yet, bringing it back to *full* or *light* is expected to be quicker than from mode *off*. This is usually accomplished by maintaining the state of the component "alive" and thus implies in some energy consumption. A component that does not support this mode natively must opt between remaining active or saving its state, perhaps with aid from the operating system, and going off.

The hierarquical organization of components in EPOS allows applications to act on different levels of components. The power management API features the concept of a **System** pseudo-component, which can be seen as a kind of aggregator for the actual components selected for a given system instance. The goal of the **System** component is to aid programmers to express global power management actions, such as putting the whole system in a given operating mode, perhaps after having defined specific modes for particular components. Thus, the application may use the **System** component to trigger a system-wide power mode change. Another way the application may use this interface is through subsystems (e.g., *Inter-Process Communication* (**IPC**), **Processing**, **Sensing**). In this way, messages are propagated only to the components used in the implementation of each subsystem. The application may also access the hardware directly, using the API available in the device drivers, such as *Network Interface Card* (**NIC**), **CPU**, or **Thermistor**.

A simple strategy for opportunistic power management is to make use of the scheduler in an operating system. Whenever the scheduler has no tasks to schedule, the system's main processor is put in standby mode. However, this technique is very limited, as it does not regard peripheral components, which in deeply embedded systems potentially consume more energy than the main microcontroller. In order to contemplate these devices, an active power manager is required. This manager checks component usage, and if a certain component is inactive for a given period of time, it changes the components power state to a lower level. One technique for detecting component usage makes use of hardware activity counters for each component [2]. However, embedded systems hardware typically does not feature such counters, and a software-based infra-structure for energy accounting must be used.

EPOS features energy accounting in two levels: a system-wide *energy monitor*, and individual component *access counters*. The energy monitor uses pre-determined information about the system's hardware and power source to build an expected energy consumption curve, and periodically samples the power source (e.g. battery) through a charge sensor channel to calibrate its expected discharge rate. This allows low-overhead energy accounting with reasonable accuracy [19].

Access counters are implemented through an aspect program [11] that updates a timestamp each time the component is accessed [20]. EPOS supports aspects through a technique called *Scenario Adapters* [5]. Through this technique, an aspect **Scenario** implements code that may be inserted before and/or after the targeted operations. This mechanism also permits the extension of the targeted component's interface and data structures. Through this system, event counters were added to the accounting **Scenario**, and code to use these counters was added to each component.

An opportunistic power manager, which is executed either periodically or when there are no tasks to schedule, checks the utilization timestamps of each component against the current timestamp of the system. A configurable power management heuristic then decides if and when to change a component's power mode. In its simplest form, the power manager puts all *idle* components (components that have not been accessed for a pre-set period of time) to sleep mode.

Since both access counters and the power manager are implemented through aspect programs, they may be extended, configured, or disabled without incurring in significant interference in the application. The power manager may be statically and dynamically configured through modification of the power management algorithm. This modification is made available because the management algorithm is one of the main factors in the balance between power economy and application overhead. This way, reconfigurability allows the application programmer to decide what power management algorithm his system will support [20].

3.2 Real-Time Scheduling

The real-time scheduling infrastructure in EPOS is comprised of three central components: **Thread**, **Criterion**, and **Scheduler**. The main difference between the EPOS real-time modelling and other real-time operating systems is related to the scheduler queues. These are ordered according to a Criterion object, which may be replaced according to the needs of specific applications.

Figure 1 presents the class hierarchy of EPOS Criterion components. Real-time support is provided by the **Real-Time** class, which inherits the priority and scheduling information from the task. Periodic task scheduling algorithms such as **RM** (Rate Monotonic) and **EDF** (Earliest Deadline First), inherit period information from the **Periodic** components.

The separation of scheduling policy and scheduler allows the same scheduler to be used with different policies. In cases where the policy requires specific scheduling treatment, a new scheduler may be created by extending the existing schedulers.

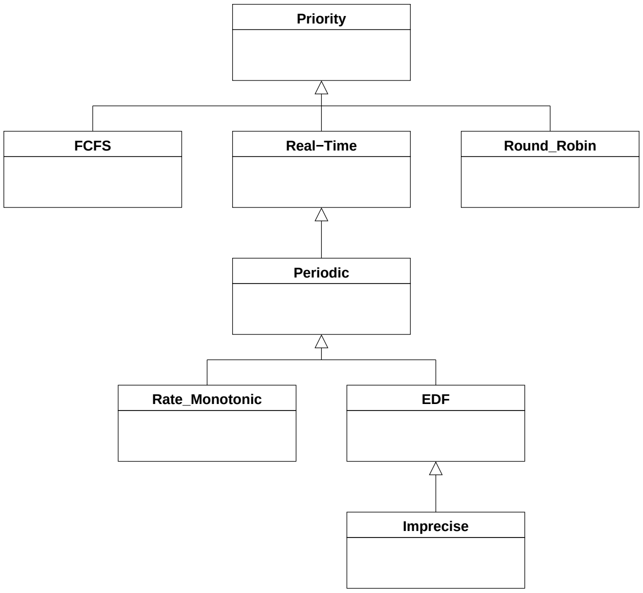

Figure 1: Class diagram of EPOS Criterion components

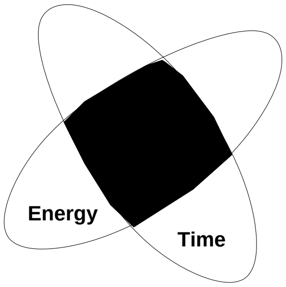

Figure 2: Intersection between energy and time

4. ENERGY-AWARE REAL-TIME SCHEDULING

In this work, we extend the real-time infra-structure in EPOS to be able to use an application's energy budget (e.g. its expected battery lifetime) as a QoS parameter. As in imprecise computation, we divide each task into two parts: one mandatory and optional. In order for an optional subtask to execute it must meet two criteria: time and energy. It cannot interfere with the deadlines of mandatory subtasks, and it may only execute when there is enough energy to complete the set of mandatory tasks.

Our scheduler, which is based on the EDF policy (*Earliest-Deadline First*) [12], guarantees the execution and meeting the deadlines of mandatory subtasks, independently of the system energy level. However, the execution of optional subtasks is not guaranteed. The optional subtasks are executed only if the mandatory subtasks deadlines and the system energy budget defined by application are met. Figure 2 represents the tasks that comply with both the energy (system lifetime) and time (mandatory subtasks deadlines) criteria. Tasks outside the intersection of the two criteria fail scheduling tests for this algorithm.

```
for each task in the READY state do
    Determine the new absolute deadline ;
    Calculate priority according to deadline;
    Insert into the queue according to calculated priority;
end
for each rescheduling do
    Select the highest priority task in the READY state;
    for each $\pi$ time units do    /* $\pi$ is configurable */
        Read the energy monitor;
        Check if there is enough energy to meet the time
        specified by the application;
    end
    if task is hard real-time then
        Execute the selected task;
    else                        /* task is best effort */
        if there is enough energy to meet the required
        system lifetime then
            Execute the selected task;
        else                /* energy is not sufficient */
            Execute the opportunistic power manager;
        end
    end
end
```

Figure 3: Scheduling Algorithm

The objective of this scheduler is not only to save energy — otherwise, the technique would simply avoid executing the optional subtasks — but to meet the energy budget (e.g. battery lifetime) specified by the application and to meet the deadlines of mandatory subtasks with the execution of the maximum possible of the optional subtasks, thus improving the application *utility.*

Figure 3 presents our scheduling algorithm, in the subtasks are treated as tasks in terms of scheduling. π is the interval between energy measurements, that can be specified by the application programmer and must take into consideration that each measurement consumes energy. This interval depends on the energy state found in the last measurement.

4.1 Schedulability Tests at Project-Time

Since our scheduler is based on the EDF algorithm, it is possible to adapt the classic EDF schedulability tests to take both energy and the division of tasks into mandatory and optional parts into acount. We assume the system has a set of n periodic and independent tasks, $\tau = \{\tau_0, \tau_1, ..., \tau_{n-1}\}$. P_i, D_i, and C_i are parameters of each τ_i, where P_i is the period in which task i is scheduled, D_i is the maximum relative deadline of the task i, and C_i is the worst case execution time of task i. In our tests, we assume that $\forall \tau_i,\ D_i \leq P_i$. The utilization U_i of task i in terms of processing is represented by $U_i = \frac{C_i}{D_i}$.

In the imprecise computation model, each τ_i is divided into mandatory and optional subtasks with worst case execution times of μ_i and θ_i, respectively. Therefore, the worst case total execution time of τ_i is $C_i = \mu_i + \theta_i$. In order to guarantee that no mandatory subtask deadlines will be lost, equation (1) must be respected.

$$\sum_{i=1}^{n} \left(\frac{\mu_i}{D_i} \right) + \sigma \leq \omega \tag{1}$$

Where $\omega = 1$ for a system with a single-processor and σ represents the run-time overhead in the worst case, which includes: time spent in the operating system, context switch, scheduler algorithm. Equation (1) must be met in order for the tasks to be schedulable in relation to mandatory subtasks deadlines, otherwise, the processor is overloaded.

With the inclusion of the optional subtask execution time in equation (1), we can determine if the tasks as a whole (both mandatory and optional subtasks) will be executed. However, it is important to note that equation (2) is not an obligatory requirement in our algorithm and only will be relevant when equation (1) is true, otherwise, the tasks are not schedulable.

$$\sum_{i=1}^{n} \left(\frac{\mu_i + \theta_i}{D_i} \right) + \sigma \leq \omega \tag{2}$$

Mandatory and optional subtasks are schedulable in relation to their deadlines when equation (2) is respected. Otherwise, a certain fraction χ of optional subtasks is discarded, as shown in equation (3).

$$\chi = \frac{\sum_{i=1}^{n} \left(\frac{\mu_i + \theta_i}{D_i} \right) + \sigma - \omega}{\sum_{i=1}^{n} \left(\frac{\theta_i}{D_i} \right)} \tag{3}$$

In order to deal with energy requirements, it is necessary to take into account the energy consumption rate of each task. The E_i worst case energy consumption of a task τ_i, is given by the sum of the worst case energy consumption of the mandatory and optional subtasks $E_{\mu i}$ and $E_{\theta i}$, respectively (i.e., $E_i = E_{\mu i} + E_{\theta i}$). As it is the case with the worst case execution times, it is necessary to assess the worst case energy consumption rate beforehand. These values can be obtained by energy profiling or another techniques. The maximum number of possible executions η_i of τ_i in the time T_t required by the application is given by the division between the time required and the execution interval of τ_i, i.e., $\eta_i = \frac{T_t}{P_i}$. T_t is determined by the application developer based on energy capacity. In order to guarantee the execution of at least the mandatory subtasks, equation (4), which indicates if the set of tasks will be schedulable with respect to energy, must be respected.

$$\sum_{i=1}^{n} \left(\frac{E_{\mu i} \times \eta_i}{E_t} \right) + \epsilon \leq 1 \tag{4}$$

Where E_t is the total energy available to the system (e.g. battery capacity), ϵ represents the worst case energy consumption of operating system services such as alarms, context switches and the scheduler itself. In this equation, the total energy is set to 1 (e.g. battery capacity is 100%). By substituting the maximum number of possible executions η_i in equation (4) we obtain equation (5).

$$\sum_{i=1}^{n}\left(\frac{E_{\mu i}\times T_t}{P_i\times E_t}\right)+\epsilon\leq 1 \tag{5}$$

The tasks are schedulable in relation to energy if equation (5) is respected. Otherwise, the system will not meet energy budget required by application for this set of tasks. The inclusion of the energy consumed by optional subtasks in equation (5) allows us to check if the tasks as a whole will be executed. As discussed previously, this is not an obligatory requirement and equation (6) only should be calculated if equation (5) is respected, i.e., the mandatory subtasks have met their energy budget.

$$\sum_{i=1}^{n}\left(\frac{(E_{\mu i}+E_{\theta i})\times T_t}{P_i\times E_t}\right)+\epsilon\leq 1 \tag{6}$$

All mandatory and optional parts of the tasks are executed in relation to system energy if equation (6) is respected. Otherwise, a certain fraction γ of optional subtasks will not be executed because the system would not meet the energy budget specified by the application. Equation (7) provides a fraction γ of optional subtasks discarded in relation to energy.

$$\gamma=\frac{\displaystyle\sum_{i=1}^{n}\left(\frac{(E_{\mu i}+E_{\theta i})\times T_t}{P_i\times E_t}\right)+\epsilon-1}{\displaystyle\sum_{i=1}^{n}\left(\frac{E_{\theta i}\times T_t}{P_i\times E_t}\right)} \tag{7}$$

In order to provide real-time and energy-aware scheduling, it is necessary to respect all the mandatory substasks deadlines and to have enough energy to execute all the mandatory substasks. Equation (8) is used to determine if a given set of tasks is schedulable in our system.

$$\left[\sum_{i=1}^{n}\left(\frac{\mu_i}{D_i}\right)+\sigma\leq\omega\right]\wedge\left[\sum_{i=1}^{n}\left(\frac{E_{\mu i}\times T_t}{P_i\times E_t}\right)+\epsilon\leq 1\right] \tag{8}$$

The mandatory subtasks have their executions guaranteed in our scheduler in relation to time and energy if equation (8) is respected. The maximum fraction λ of possible optional subtasks lost in relation to time and energy can be obtained by equation (9).

$$\lambda=\max\left(\chi,\gamma\right) \tag{9}$$

4.2 Schedulability at Execution-Time

In order to provide quality of service in terms of energy and make better use of resources through the execution of optional subtasks, it is necessary to periodically check if the system lifetime can be achieved at execution-time. Therefore, the system lifetime, $T_{t\kappa}$, and the system energy (e.g. battery charge), $E_{t\kappa}$, at instant κ are recalculated. Equation (10) can be calculated with the new values in order to check if $T_{t\kappa}$ can be met at instant κ. This equation uses the discharge rate in the last period to check wether the $T_{t\kappa}$ can be attained at the current power levels.

$$\left[\left(\frac{E_{t\kappa}}{E_{t\kappa-1}-E_{t\kappa}}\right)\times\left(T_{t\kappa-1}-T_{t\kappa}\right)\right]\geq T_{t\kappa} \tag{10}$$

If equation (10) is respected, there is enough energy to meet $T_{t\kappa}$. Thus, all mandatory subtasks are executed and optional subtasks are scheduled. Otherwise, some optional subtasks will be discarded, decreasing quality-of-service, and the scheduler invokes the oportunistic power manager using the time that the optional subtasks would execute in order to save energy. In a future instant $\kappa+1$, if $T_{t\kappa+1}$ is satisfied, the optional subtasks will resume execution, increasing quality-of-service.

5. CASE STUDY

Sensor data acquisition is an example of an application that may benefit from imprecise computing. Multiple averaged readings from an unreliable data source may improve accuracy when compared to a single reading from the same source. Multiple readings, however, may also incur in excessive computation time and energy consumption. An imprecise scheduler may choose when to perform multiple readings to improve accuracy, and when to perform a single reading to save system resources.

In this case study, we used Mica2 sensor nodes [8] to design a data acquisition application that periodically samples data from a temperature sensor and sends sensor data to the network. Figure 4 features a sequence diagram that illustrates the behaviour of this application. Following our imprecise computation model, the mandatory part of our sensing task performs a single reading. If there is enough time and energy available in the system, an optional part performs multiple averaged readings. Finally, a communication task sends the sensor data, either from the single mandatory reading or from the multiple averaged readings.

We implemented this application for the EPOS system, using our imprecise computation model. Figure 5 illustrates the system APIs used by the application. The **main** function first creates a temperature sensor and a communication link [17, 18], and proceeds to create an imprecise and a periodic thread, wich represent our sensing and communication tasks. The mandatory part of the data acquisition task performs a single reading of the sensor, and waits for its next execution period. When executed, the optional part performs multiple readings (in this case 10 readings if possible), averaging all the previous readings. When finished, the optional part also releases the processor and waits for its next sheduling. Finally, the communication task sends the

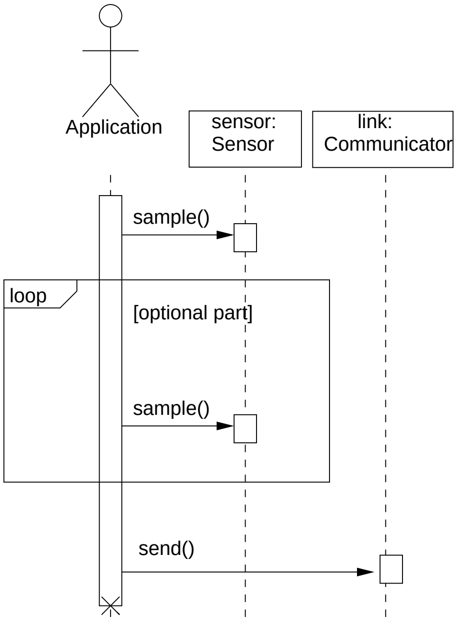

Figure 4: Sequence Diagram of the Data Acquisition Application

```
void sensor_main() {
    // ...
    sensor = new Temperature_Sensor();
    link = new Communicator(SENSOR_SINK);
    Imprecise_Thread sensing(&mandatory_sensor,
        &optional_sensor,  Criterion (150e3,170e3,INFINITE,0));
    Periodic_Thread  communication(&send_data,
        Criterion (20e3,170e3,INFINITE,150e3), SUSPENDED);
}

int mandatory_sensor() {
    while(1) {
        // ...
        temperature = sensor->sample();
        Imprecise_Thread :: wait_next ();
    }
}

int  optional_sensor () {
    while(1) {
        // ...
        for (int  i = 1; i <= MAX_SAMPLES; i++){
            sample = sensor->sample();
            temperature =
                (temperature * i + sample) / (i+1);
        }
        Optional_Thread::wait_next ();
    }
}

int send_data() {
    while(1) {
        // ...
        link ->write(temperature);
        Periodic_Thread :: wait_next ();
    }
}
```

Figure 5: System APIs used by the Data Acquisition Application

Thread	Phase (ms)	Deadline (ms)	Period (ms)	Times
Sensing	0	150	170	∞
Communication	150	20	170	∞

Table 2: Scheduling Criteria

Part	Time (ms)	Energy (J)
Mandatory	11.683	0.0004254
Optional	116.831	0.0042543
System Overhead	0.138	0.0098289

Table 3: Worst-case execution time and energy consumption.

collected data, which was last updated by the mandatory or by the optional part.

The imprecise thread receives its mandatory and optional functions as a parameter, while the periodic thread receives a single function. Both threads receive a criterion as a parameter, which specifies deadline, period, number of execution times, and phase for each thread. The deadline of each thread is set so that the order of mandatory sensing, optional sensing and communication is preserved. All threads run for an infinite number of times. Table 2 presents the scheduling criteria for both threads. Figure 6 presents the scheduling diagram for the application.

In order to check schedulability at project time, we profiled our application for time and energy with a digital osciloscope. Table 3 presents the worst case execution time and energy consumption for one execution of our tasks and total system overhead for one period. These values represent the average from 9100 samples.

In this example, our energy budget was the charge provided by two regular AA batteries with a total capacity of 58320 J (5400 mAh at 3 V). We configured the system with an expected lifetime of 11 days, or 950400000 ms. It should be noted that, in order to better illustrate energy budget exaustion, our application purposedly set the system to a full load. Most applications would tipically run at a considerably lower duty-cycle, and thus for a longer period of time. In order to check schedulability of the application with relation to time, we applied the application values to equation (1):

$$\begin{aligned} \sum_{i=1}^{n}\left(\frac{\mu_i}{D_i}\right)+\sigma &\le \omega \\ \left(\frac{11.683\text{ ms}}{150\text{ ms}}\right)+\frac{0.138\text{ ms}}{150\text{ ms}} &\le 1 \\ 0.078 &\le 1 \end{aligned} \tag{11}$$

Since (11) is true, the mandatory parts are schedulable with relation to time. The application of equation (2), presented in (12) shows that, both mandatory and optional parts are schedulable with relation to time.

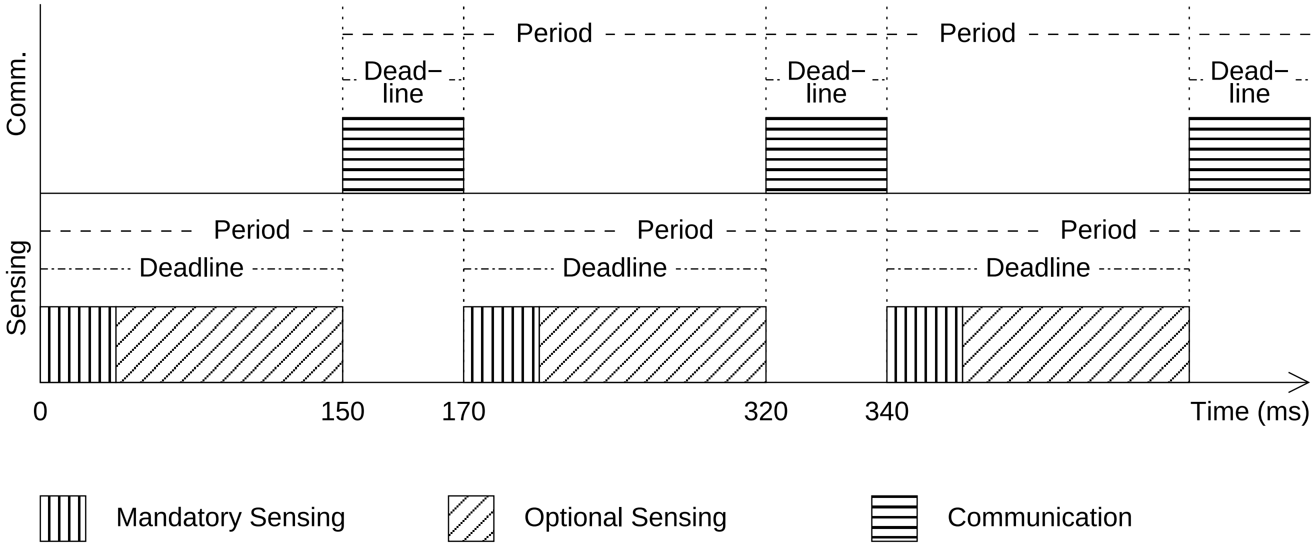

Figure 6: Scheduling Diagram

$$
\begin{aligned}
\sum_{i=1}^{n}\left(\frac{\mu_i+\theta_i}{D_i}\right)+\sigma &\le \omega \\
\left(\frac{(11.683+0.138+116.831+0.138)\text{ ms}}{150\text{ ms}}\right) &\le 1 \\
0.8586 &\le 1 \qquad (12)
\end{aligned}
$$

In order to check schedulability of the application with relation to energy, we applied the application values to equation (5):

$$
\begin{aligned}
\sum_{i=1}^{n}\left(\frac{E_{\mu i}\times T_t}{P_i\times E_t}\right)+\epsilon &\le 1 \\
\left(\frac{425.4\times 10^{-6}\text{J}\times 950.4\times 10^{6}\text{ms}}{170\text{ ms}\times 58320\text{ J}}\right)+ & \\
\left(\frac{9828.9\times 10^{-6}\text{J}\times 950.4\times 10^{6}\text{ms}}{170\text{ ms}\times 58320\text{ J}}\right) &\le 1 \\
\left(\frac{425.4\times 950.4}{170\times 58320}\right)+\left(\frac{9828.9\times 950.4}{170\times 58320}\right) &\le 1 \\
0.0407791+0.9422039 &\le 1 \\
0.9829830 &\le 1 \qquad (13)
\end{aligned}
$$

Since (13) is true, the tasks are schedulable with relation to energy. However, the application of equation (6) shows that some optional subtasks will be lost due to energy constraints, as seen in (14):

$$
\begin{aligned}
\sum_{i=1}^{n}\left(\frac{(E_{\mu i}+E_{\theta i})\times T_t}{P_i\times E_t}\right)+\epsilon &\le 1 \\
\left(\frac{(425.4+4254.3)\times 10^{-6}\text{J}\times 950.4\times 10^{6}\text{ms}}{170\text{ ms}\times 58320\text{ J}}\right)+ & \\
\left(\frac{9828.9\times 10^{-6}\text{ J}\times 950.4\times 10^{6}\text{ms}}{170\text{ ms}\times 58320\text{ J}}\right) & \\
\left(\frac{4679.7\times 950.4}{170\times 58320}\right)+\left(\frac{9828.9\times 950.4}{170\times 58320}\right) & \\
0.4485987+0.9422039 & \\
1.3908026 &\nleq 1 (14)
\end{aligned}
$$

Since the application of equation (6) fails for this test, the system will decide at run-time when to execute the optional subtasks. The system is configured with a master timer of 72Hz, and each thread has a quantum of 15ms. In each rescheduling, the scheduler checks if the next task is a real-time or mandatory task by cheking its priority. If this test is true, then the task receives the processor. Otherwise, the task is optional, and receives the processor only if there is enough energy to meet the desired system lifetime. If the task is optional, and energy is not enough, the processor and peripheral components are put in low power mode through the power manager [20].

Figure 7 presents the power rate of our test application along the time. In the curve with the highest power rate, the system is configured so that it executes every task, optional and mandatory. In the curve with the lowest power rate, the system is configured so that it only executes the mandatory parts. The midddoes notle curve represents the use of our imprecise computing model, where optional tasks only execute if the system is within its energy budget.

Figure 8 presents different QoS levels along time for the imprecise computing model. In this graphic, QoS level is defined as the number of sensor readings performed in a given

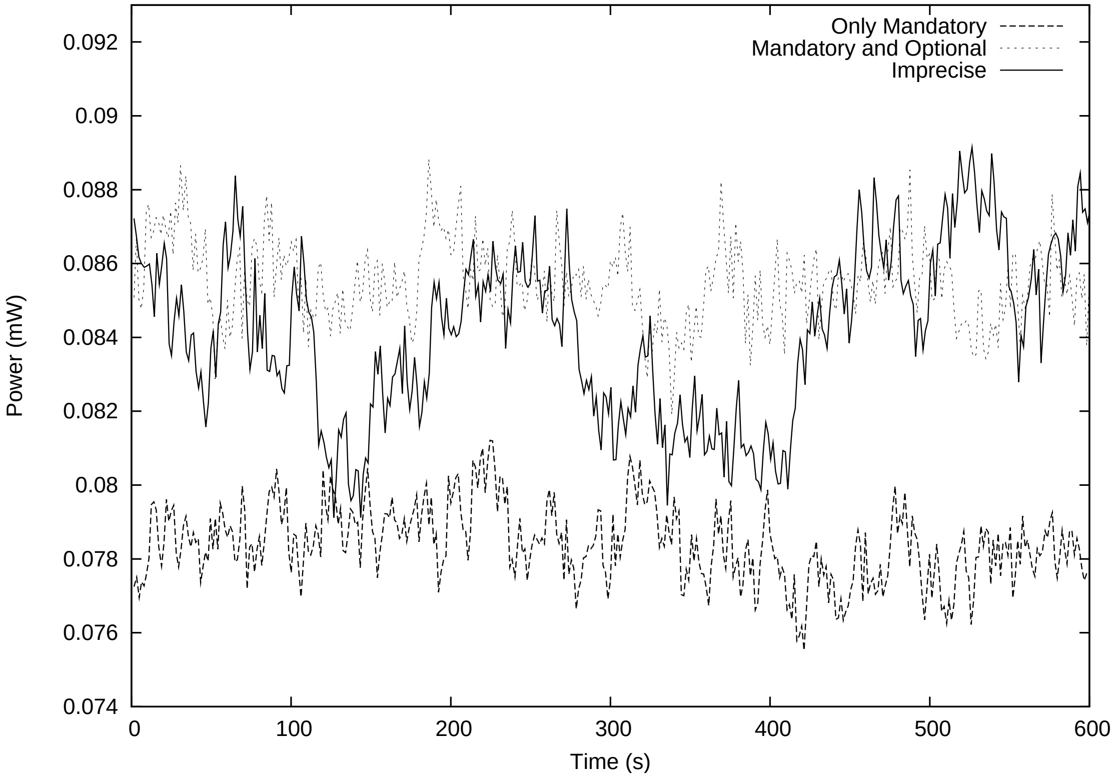

Figure 7: Power rate along time

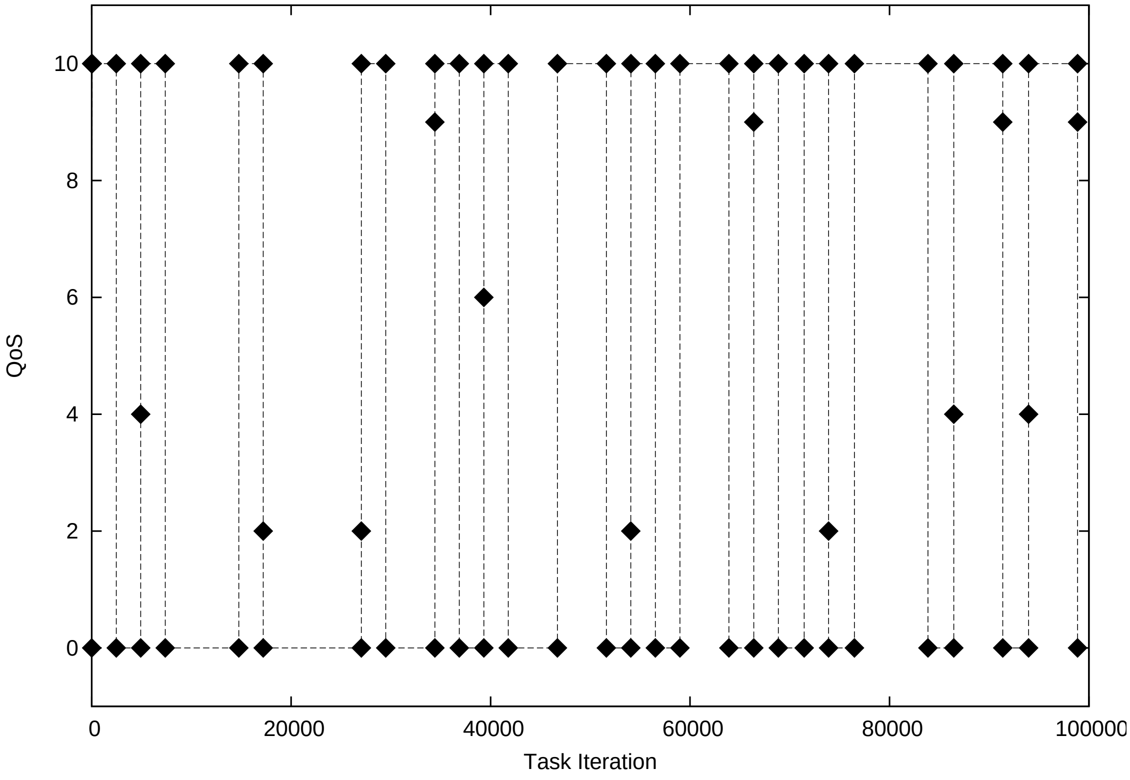

Figure 8: QoS levels along time

period. When there is enough energy in the system, optional subtasks are scheduled, increasing quality of service. When energy levels are lower than expected, the optional parts are discarded. The battery monitor periodically checks the battery level. If it detects a lower than expected energy level, it may preempt the execution of optional subtasks, resulting in intermediate levels of quality of service (e.g. when only two iterations of the optional subtask were executed).

6. CONCLUSION

In this work, we extended the real-time scheduling and the power management infrastructure in the EPOS operating system to use an application's energy budget as a QoS parameter. Through our imprecise computation model, application programmers define mandatory and optional parts of the application's tasks, an expected system lifetime, and an energy budget. Whenever there is enough energy in the system to meet the application's budget, the optional parts are executed, increasing QoS. Otherwise, only the mandatory parts run, allowing the system to save energy.

In addition to our case study of single point data acquisition, this system could also be used to manage QoS in a network of sensors. For example, in an application that uses freshness of information as the main parameter of QoS, our system could be used to dynamically update sensor information according to energy availability. Thus each node would decide, based on its available time and energy, whether to update its sensor readings or to keep an older reading. In this example, the quality of service of the sensor network application would be automatically adapted to suit the available resources. As demonstrated by our case study, this automatic, adaptative quality control allows applications to achieve their compromise between quality-of-service and energy consumption.

7. REFERENCES

[1] Embedded Linux/Microcontroller Project. http://www.uclinux.org/, 2008.

[2] F. Bellosa. The benefits of event: driven energy accounting in power-sensitive systems. In *ACM EW 9*, pages 37–42, New York, NY, 2000. ACM.

[3] L. Benini, A. Bogliolo, and G. D. Micheli. Dynamic power management of electronic systems. In *Proc. of the 1998 IEEE/ACM int. conf. on Computer-aided design*, pages 696–702, New York, NY, 1998. ACM.

[4] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *ACM SOSP'99*, pages 48–63, New York, NY, 1999. ACM.

[5] A. A. Fröhlich. *Application-Oriented Operating Systems.* GMD - Forschungszentrum Informationstechnik, Sankt Augustin, 2001.

[6] F. Harada, T. Ushio, and Y. Nakamoto. Power-aware resource allocation with fair QoS guarantee. *RTCSA'06: Proc. of the 12th IEEE int. conf. on Embedded and Real-Time Computing Systems and Applications*, pages 287–293, 2006.

[7] Hewlett-Packard Corp., Intel Corp., Microsoft Corp., Phoenix Technologies Ltd., and Toshiba Corp. *Advanced Configuration and Power Interface Specification*, Third edition, Oct. 2006.

[8] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, 2004.

[9] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proc. of the ninth int. conf. on Architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM.

[10] A. S. Hoeller Jr., L. F. Wanner, and A. A. Fröhlich. A Hierarchical Approach For Power Management on Mobile Embedded Systems. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems.*, IFIP int. Federation for Information Processing, pages 265–274. Springer, 2006.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proc. of the European conf. on Object-oriented Programming'97*, pages 220–242, Finland, June 1997. Springer.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[13] J. W. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proc. of the IEEE*, 82(1):83–94, Jan 1994.

[14] H. Marcondes, A. S. Hoeller Jr., L. F. Wanner, and A. A. Fröhlich. Operating Systems Portability: 8 bits and beyond. In *11th IEEE ETFA*, pages 124–130, Prague, Czech Republic, Sept. 2006.

[15] L. Niu and G. Quan. A hybrid static/dynamic dvs scheduling for real-time systems with (m, k)-guarantee. In *IEEE RTSS'05*, pages 356–365, Washington, DC, USA, 2005. IEEE Computer Society.

[16] C. Scordino and G. Lipari. Using resource reservation techniques for power-aware scheduling. In *Proc. of the 4th ACM int. conf. on Embedded software*, pages 16–25, New York, NY, 2004. ACM.

[17] L. F. Wanner, A. B. de Oliveira, and A. A. Fröhlich. Configurable Medium Access Control for Wireless Sensor Networks. In *Embedded System Design: Topics, Techniques and Trends*, IFIP int. Federation for Information Processing, pages 401–410. Springer, 2007.

[18] L. F. Wanner, A. S. Hoeller Junior, A. B. de Oliveira, and A. A. Fröhlich. Operating System Support for Data Acquisition in Wireless Sensor Networks. In *11th IEEE ETFA*, pages 582–585, Prague, Czech Republic, Sept. 2006.

[19] G. R. Wiedenhoft and A. A. Fröhlich. Using Imprecise Computation Techniques for Power Management in Real-Time Embedded Systems. In *6th IFIP Working conf. on Distributed and Parallel Embedded Systems*, Milano, Italy, Sept. 2008.

[20] G. R. Wiedenhoft, A. S. Hoeller Jr., and A. A. Fröhlich. A Power Manager for Deeply Embedded Systems. In *12th IEEE ETFA*, pages 748–751, Patras, Greece, Sept. 2007.

[21] W. Yuan and K. Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. *Oper. Syst. Rev.*, 37(5):149–163, 2003.

[22] H. Zeng, C. S. Ellis, A. R. Lebeck, and A. Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, 2002.