International Journal of Computer Applications (0975 8887)
Volume 81 - No. 12, November 2013

Real-time Operating System for Wireless Sensors Powered by Renewable Energy Source

Hussein El Ghor
Lebanese University - IUT Saida
SNCS Research Center, UT, Saudi Arabia
B.P. 813 Saida, Lebanon

El-Hadi M Aggoune
Electrical Engineering Department
Sensor Networks and Cellular Systems (SNCS) Research Center
University of Tabuk, 71491 Tabuk, Saudi Arabia

ABSTRACT

Energy management is a central problem in real-time systems design, in particular for embedded wireless devices such as sensor devices. In our work, we aim at the improvement of real-time operating systems that are powered by renewable energy source (solar energy, for example). The objective of this work is to develop software components for the design of real-time operating systems. We provide an on-line scheduling scheme, named Earliest Deadline with energy guarantee (EDeg), in order to address the limitations in energy harvesting systems. We also integrate EDeg scheduling algorithm into CLEOPATRE open-source component library, a patch to Linux/RTAI and evaluate the scheduling overheads of EDeg observed under Linux/RTAI.

Keywords:

Energy Management, Real-time, Operating Systems, Energy Harvesting, RTAI.

1. INTRODUCTION

Real-time computing systems play a critical role in our society. This is due to the fact that many complex systems rely, in part or completely, on computer control. Examples of applications that require real-time computing include: sensor and cellular networks, plant and process control, detection and tracking, etc.
A robust guarantee of the performance of a real-time system under all possible operating conditions can be achieved only by using more sophisticated design methodologies. Such methodologies are combined with a static analysis of the source code and specific operating systems mechanisms, purposely designed to support computation under time constraints. Moreover, in critical applications, the control system must be capable of handling all anticipated scenarios, including peak load situations, and its design must be driven by pessimistic assumptions on the events generated by the environment.
Typically, a real-time system is implemented as a set of concurrent tasks that are managed by a software called Real-Time Operating System (RTOS). Each task performs a computational activity according to a set of constraints. Thus, the objective of the RTOS is to manage and control the assignment of system resources (including the microprocessor) to the tasks in order to meet such constraints.
Nowadays, energy management becomes the central problem in real-time systems design, in particular for autonomous communicating devices (sensor networks). Such devices, mainly characterized by restrictions in terms of energy capacity, memory size and computing power, impose new constraints to real-time operating systems (RTOS). Today, the challenge, raised by the emergence of embedded systems powered by renewable energy, is to ensure sustainable autonomy if not perpetual that involves new guidelines for RTOS in terms of scheduling and power management.
In this paper, we address the problem of scheduling hard real-time tasks under energy constraints. The scope of the paper is to develop software components for the design of real-time operating systems for wireless sensors that are powered by renewable energy sources such as solar panel and relies on a battery with limited capacity.
The remainder of the paper is organized in the following manner. In the next section, we summarize the state of art relative to energy management in both real-time systems and real-time operating systems. Section 3 describes the CLEOPATRE project including scheduling and fault tolerance mechanisms. In section 4, we present the energy guarantee scheduler. Section 5 introduces the integration of EDeg into Linux based systems. Performance evaluation is studied in section 6. Section 7 concludes the paper and gives some new directions of work.

2. STATE OF ART

2.1 Energy Management in Real-Time Systems

Energy-aware real-time scheduling has been the subject of intensive research. Most of the works focus on either minimizing the energy consumption or maximizing the system performance such as the lifetime achieved under energy constraints [1]. However, the rechargeability of the energy storage unit is disregarded. Other works use the techniques of Dynamic Voltage and Frequency Scaling (DVFS) and Dynamic Power Management (DPM) [2]. But solely applying these techniques has limitations in energy harvesting systems because they minimize CPU power, rather than dynamically manage power according to the profiles of both available energy and processor workload.
In the last decade, researchers started to address power and scheduling issues but most of them do not consider both rechargeability of the batteries and real-time constraints. In the work by Allavena et al. in [3], power scavenged by the energy source is constant and all tasks consume energy at a constant rate. Later in [4], Moser et al. propose LSA (Lazy scheduling Scheduling Algorithm) to optimally schedule tasks with deadlines, periodic or not. In that work, the total energy consumption of every task is directly connected

to its execution time through the constant power of the processing device. But in a real application, instantaneous power consumed by tasks may vary along time depending on circuitry and devices required by the tasks.

2.2 Real-time Operating Systems

In recent years, wireless sensor networks (WSN) have received tremendous attention in the research community, with applications ranging from home automation to industrial and environmental monitoring. This attention imposes additional challenges on the design of special purpose operating systems deviated from the traditional ones. However, most of these OSs are non-real-time and cannot support real-time applications. Moreover, various problems will arise when attempting to convert a non-real-time OS to a real-time version. Some RTOS are available, which include ThreadX from Xpresslogic, VxWorks by Wind River, QNX Neutrino by QNX Software Systems, and Real-time Linux (by several vendors).
Real-time operating systems are classified into two categories: systems that use light and standard versions of optimized operating system and those who seek the best determinism by liberating any operating system. The main difficulty with standard systems is to find a balance between providing services of classical systems (such as dynamic allocation of memory pages that is non-deterministic) and the predictability of these services. This predictability is a key factor for application development with strict real-time constraints. A lot of commercial operating systems based on UNIX are derived from this approach such as QNX53, OS-954 and LynxOS55. Moreover, research projects have been emerged such as Mach Real-Time system that is an extension of the Mach project [11].
Systems from the second category are derived from research projects such as Spring [12], Maruti [14] and YARTOS [15]. Sometimes real-time operating systems of this category are based on standard systems such as Linux. But, unlike previous systems, they are modified to give priority to real-time applications that use only deterministic services. The standard operating system is relegated to the status of the task background of the real-time system that controls scheduling. These systems can be found as patches to Linux which render real-time: RTLinux [6] and RTAI [5].
Regrettably, all these systems consider time as the only limiting factor, leaving energy efficiency as a hopeful consequence of empiric decisions.
Linux is now the host of most RTOS, especially for wireless sensor networks that operate with an ambient energy source and cannot rely on a power outlet on the wall. Thus, a power management design is needed to make the best use of the available power and to adapt the performance of the environmental energy harvesting to the available energy profile.
The most well-known versions of real-time Linux are: Real Time Application Interface (RTAI) [5] and RTLinux [6]. RTAI is an open source real-time operating system based on the standard Linux kernel. The RTAI project was originated by the *Dipartimento di Ingegneria Aerospaziale del Politecnico di Milano (DIAPM)* about sixteen years ago [7]. RTAI is a plug-in which permits Linux to fulfill some real-time constraints with no event loss. On the other hand, RTLinux is a hard real-time operating system that runs Linux as its lowest priority thread. RTLinux, which was originally developed at the New Mexico Institute of Technology, is an open-source product.
Thus, we aim to enhance the power management capabilities of existing versions of Linux, such as RTLinux or Linux/RTAI, so as to ensure sustainable autonomy if not perpetual that involves new guidelines of RTOS for autonomous communicating devices (sensor networks).

3. CLEOPATRE PROJECT

3.1 Generalities

A project named CLEOPATRE (Composants Logiciels sur Etagères Ouverts Pour les Applications Temps Réel Embarquée) was labeled in April 2001 and notified in January 2002 by the French Ministry of Education and Research. The objective of this project was to provide solutions to the development of embedded real-time applications by providing free and open source software components based on Linux [7]. CLEOPATRE aims to enhance the real-time services of existing versions of Linux, such as RTLinux [6] or Linux/RTAI [5].
In addition, many components, dedicated to real-time systems, like dynamic scheduling, aperiodic task service, resource control access, fault-tolerance and QoS scheduling are offered by the CLEOPATRE library (figure 1).
RTAI permits Linux to fulfill some real-time constraints in a few milliseconds deadline and with no event loss. It is based on a Hardware Abstraction Layer (HAL) [7]. The HAL defines a clear interface which exports some Linux data and functions related to the hardware.
RTAI has a microkernel, which is very close to the proposed architecture for CLEOPATRE. In fact, CLEOPATRE components can be directly treated as microkernel system servers since they have the same goal. We therefore choose RTAI to base our development and to contribute its evolution by providing significant improvements. Moreover, the choice of RTAI enables us to develop a complete operating system where we can benefit from the rich Linux environment (drivers, compilers, interfaces, etc.).

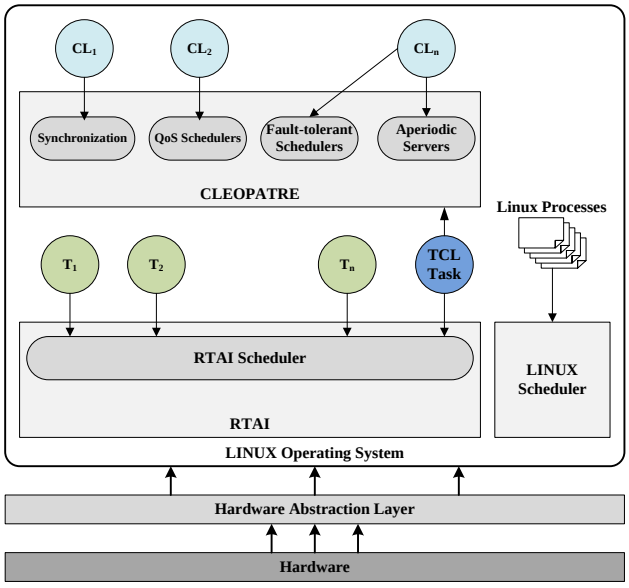

Fig. 1. CLEOPATRE Framework

The Task Control Layer (TCL) is a specific Linux module that interfaces CLEOPATRE with RTAI. TCL provides an internal interface for the CLEOPATRE services [7]. The main role of TCL is to make

the other components independent from the operating system. Consequently, we should only modify TCL to adapt the CLEOPATRE to another real-time operating system.

3.2 Fault Tolerance Mechanisms

In any real-time operating system, many interactions are found between the control system and the various actuators [16]. Consequently, many human operator functions in CLEOPATRE need to be verified especially with respect to their temporal constraints.
Fault-tolerance is achieved via time. In real-time environments, a fault-tolerance policy should be selected and implemented to recover from errors within a certain time limit [16]. In CLEOPATRE, the two major strategies are: software redundancy and time redundancy. With time redundancy, the task schedule has some slack in it. This is because, if some tasks need to be rerun, possibly with less precision, critical deadlines can be still met.
In CLEOPATRE, a deadline mechanism was implemented where each task is composed of two versions, a primary and a backup version [17].
What is missed in CLEOPATRE is the strategy related to energy redundancy. With energy redundancy, each task schedule must have some slack energy, so that some tasks can be rerun without depleting the energy reservoir and while respecting deadlines.
As a summary, fault tolerance policy must be implemented to recover from errors within an acceptable time and energy consumption limit.

3.3 Scheduling Mechanisms

In real-time systems theory, task scheduling policies are principally performed by priorities. A strong work has been done during the last decade in static and dynamic scheduling. Whereas static scheduling requires complete predictability of the real-time environment in which it is deployed. Instead, dynamic scheduling entails higher run-time costs; however, it can adapt to changes in the environment. Dynamic scheduling has been relegated to systems with dynamic nature such as robotics systems, multimedia systems, and complex control systems.
CLEOPATRE supports both static (Deadline Monotonic algorithm) and dynamic (Earliest Deadline First) scheduling policies. Deadline monotonic algorithm (DM) [18] is optimal in the sense that if a set of tasks can be scheduled by any algorithm, then it can be scheduled by DM [18]. The only problem in DM scheduling algorithm is that it cannot dynamically change periods. On the other hand, Earliest Deadline First (EDF) schedules at each instant of time t, the ready task (i.e. the task that may be processed and is not yet completed) whose deadline is closest to t. EDF is also an optimal scheduling algorithm. DM and EDF are commonly implemented in commercialized and free real-time operating systems including Linux/ RTAI [19].
However, such algorithms consider time as the only limiting constraint leaving energy efficiency as a hopeful consequence of empiric decisions. Hence, the critical issue is how to design an energy harvesting system that is able to find both power management and scheduling mechanisms that can adapt dynamically the activity of the wireless sensors according to the available energy.

4. ENERGY GUARANTEE SCHEDULER

We present a scheduling framework called EDeg [8] resulting from the extension of the earliest deadline as late as possible (EDL) server [9]. We modify earliest deadline first (EDF) scheduler so as to account for the properties of the energy source, capacity of the energy storage as well as energy consumption of the tasks. We propose a slack-based method for delaying tasks and making the processor inactive during recharging phases of the energy storage unit. On-line computing by how long the tasks should be delayed is possible thanks to EDL properties. EDeg is also based on a new concept, the slack energy, in order to execute tasks only whenever this cannot provoke energy starvation. The behavior of the Energy Guarantee scheduler is illustrated in figure 2.

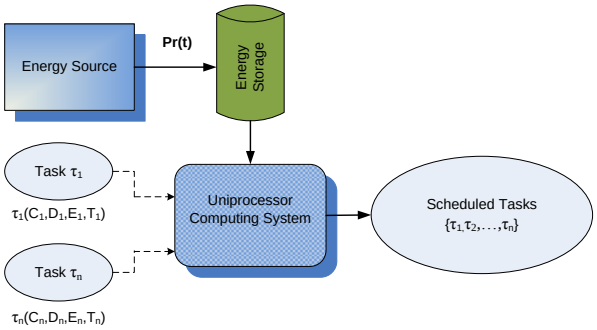

Fig. 2. A Real-Time Energy Harvesting System model

So, in order to formally present the EDeg scheduler, we need to describe precisely the two following data:

DEFINITION 1. *The slack time of the system at current time t, is the length of the longest interval starting at t during which the processor may be idle continuously while still satisfying all the timing constraints of the tasks.*

Slack time analysis has been extensively investigated in real-time applications where aperiodic (or sporadic) tasks are jointly scheduled with periodic tasks.
Slack time is computed as follows: before the system begins to operate, we compute the static EDL schedule for the given task set. More precisely, we estimate the localization and the duration of the idle times within the EDL schedule produced at time $t = 0$ till the end of the hyperperiod (T_{LCM}). The EDL schedule for the interval $[0, T_{LCM}]$ can be described by means of two vectors respectively called static deadline vector and static idle time vector, denoted by $\mathcal{K}$ and $\mathcal{D}$ [13].
Determining the slack time at any current time t then requires to dynamically build the EDL schedule at time t. The dynamic EDL schedule can be memorized thanks to two vectors denoted by $\mathcal{K}(t)$ and $\mathcal{D}(t)$ [13]. The first component of $\mathcal{K}(t)$ corresponds to current time t and the following components are the deadlines of the instances from time t until the end of the current hyperperiod. Consequently, with such a representation, the $(i+1)^{th}$ element of $\mathcal{D}(t)$ gives the length of the idle time that follows the i^{th} deadline in the dynamic EDL schedule produced at t. And the first element of $\mathcal{D}(t)$ enables us to get the slack time of the system.

Whereas slack time is a well known concept used in the conventional literature about real-time scheduling, slack energy is new and will be explained in what follows.

DEFINITION 2. *The slack energy of the system at current time t, is the maximum amount of energy that can be consumed from t continuously while still satisfying all the timing constraints of the tasks.*

So, the slack energy of the system gives the maximum energy that could be consumed by tasks which do not belong to the task set without ensuring the feasibility of this task set.
The computation of the slack energy for instance with deadline D_j is performed as follows: First, the slack energy of task instance τ_j at time t will be given by:

$$Slack\_energy(\tau_j, t) = E(t) + \int_t^{D_j} P_r(k)dk - A_j \quad (1)$$

Where where $P_r(k)$ is the recharging power that varies with time k and $E(t)$ is the energy storage capacity at time t.
A_j is considered as the energy demand within $[t, D_j)$ required by the periodic instances ready to be processed between t and D_j.

$$A_j = \sum_{D_k \leq D_j} E_k \quad (2)$$

Thus, the slack energy of the system at time t is determined by:

$$Slack\_energy(t) = min(Slack\_energy(\tau_j, t)) \quad (3)$$

The major components of the EDeg algorithm are: $E(t)$, $Slack.energy(t)$ and $Slack.time(t)$. $E(t)$ is the energy currently stored in the energy storage unit and $Slack.time(t)$ is the slack time of the system at current time t. PENDING is a boolean which equals true whenever there is at least one job in the ready list queue. We use function $wait()$ to put the processor in sleep mode and function $execute()$ to put the processor in active mode. Task instances are ordered according to the EDF rule.
The pseudo-code of EDeg algorithm is as follows.

```
1:  while (1) do
2:    while PENDING=true do
3:      while (E(t) > E_min and Slack.energy(t) > 0) do
4:        execute()
5:      end while
6:      while (E(t) < E_max and Slack.time(t) > 0) do
7:        wait()
8:      end while
9:    end while
10:   while PENDING=false do
11:     wait()
12:   end while
13: end while
```

EDeg is designed to schedule any set of time critical tasks, periodic or not, given any energy source profile with constant power production or not, and given an energy storage unit with limited capacity. It is model-free with respect to the energy source: it can be implemented in any energy harvesting system without the need for a priori information about the source which may be uncontrollable and time-varying.
We demonstrated the benefits of the Energy Guarantee scheduler by means of simulations [10]. We showed that EDeg is the most convenient ones to be extended in real world in order to bring to light how such algorithms can be implemented in practice in order to provide a better efficiency and performance.

4.1 Illustrative Example

Let us consider an application that is composed of 3 periodic tasks τ_i such that $\tau_i = (C_i, D_i, T_i, E_i)$. C_i is the worst case execution time (WCET), D_i is the critical delay, T_i is the period and E_i is the energy consumption. We consider that τ_i will be executed on a real-time operating system that relies on a battery of capacity $E = 6$ energy units. We use a solar panel with rechargeable power $P_r = 2$ to recharge the battery.
Tasks τ_i have the following characteristics: We used this application with two components combinations. First, we tested the application with EDF and the with EDeg.

Table 1. Task characteristics

Task	WCET	Critical Delay	Period	Consumed Energy
1	3 ms	6 ms	9 ms	8
2	3 ms	8 ms	12 ms	8
3	3 ms	12 ms	18 ms	8

The EDF schedule produced during the first hyperperiod is depicted in figure 3.

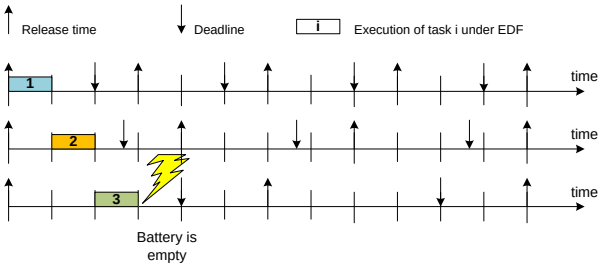

Fig. 3. EDF Scheduling

Applying a classical greedy scheduler such as EDF reveals impossible because the system fails as soon as energy shortage occurs. In this example, task instances τ_1, τ_2 and τ_3 are scheduled from $t = 0$ till $t = 9$ where the battery becomes empty. Hence, the sensor executing the application will fail without fully executing the whole application.
In contrast, singularity of EDeg lies in inserting processor idle times for recharging the storage unit only whenever necessary. And the recharging time is computed from the current slack time of the task set in order to still guarantee all the deadlines while avoiding energy overflow. Then, the sensor is let inactive as long as the energy storage has not filled completely and the latest start time of the next periodic task has not been attained. Clearly, this amounts to make slack stealing controlled by the residual capacity of the energy storage. The EDeg schedule is described in figure 4.

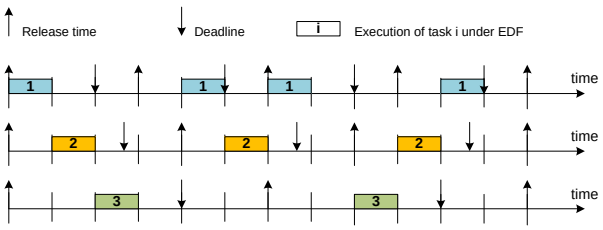

Fig. 4. EDeg Scheduling

In contrast to EDF, EDeg feasibly schedules the task set, with the same characteristics of the storage unit and the power source profile.

5. INTEGRATING EDEG INTO A LINUX BASED SYSTEM

The CLEOPATRE library offers a set of scheduling algorithms: Deadline Monotonic (DM), Earliest Deadline First (EDF) and Earliest Deadline as Late as possible (EDL). In addition, five resource management protocols are available: FIFO (First In First Out), Priority, SPP (Super Priority Protocol), PIP (Priority Inheritance Protocol) and PCP (Priority Ceiling Protocol).
As described in [16], the Task Control Layer (TCL) plays the role of managing the *TaskType* through a set of functions (TCL_CREATE, TCL_DESTROY, TCL_KILL, TCL_READY, TCL_BLOCK and TCL_SCHEDULE).
Dynamic scheduling of periodic tasks with energy constraints, namely EDeg algorithm, is put in additional shelf called *Energy Harvesting Shelf (EHS)*.

5.1 Basic Structures

The main structure of our EHS scheduler is the *task parameters*, which is defined in a header file named, $\$RTAI\_DIR/include/EHS.h$, as **struct** *TaskParameters*. This structure contains the parameters for every task as described in the following structure:

```
typedef struct TaskParameters TaskType;
struct TaskParameters{
  void (*fct) (TaskType *);          /*pointer to the task function*/
  TaskType TCL_task;                 /*task parameters*/
  TimeType delay;                    /*critical delay*/
  TimeType period;                   /*period of activation*/
  TimeType release_time;             /*time at which the task is released*/
  EnergyType consumed_energy;        /*energy needed to execute the task*/
  unsigned int slack_time;           /*slack time of the system*/
  unsigned int slack_energy;         /*slack energy of the system*/
};
```

At initialization time, the user has to set all the task parameters in the task set (period T_i, critical delay D_i, energy consumption E_i ...). In addition, the user has several functions to use in the EDeg scheduler as given below:

— *EHS_create*: create a new task.
— *EHS_resume*: resume a task.
— *EHS_wait*: wait till next period.
— *EHS_delete*: delete a task.
— *EHS_execute*: execute a task.
— *EHS_compute_E(t)*: compute the capacity of the storage unit at time t.
— *EHS_compute_slack_energy(t)*: compute the slack energy of the system at time t.
— *EHS_compute_slack_time(t)*: compute the slack time of the system at time t.

5.2 Scheduling Algorithms

The scheduling of EHS tasks is coded in the *EHS_schedule()* function where two modules are added: *EDF module* and *EDeg module*. We aim at providing scheduling solutions for weekly-hard real-time systems where energy constraints are taken into consideration. At this level, will enrich the CLEOPATRE library with enhanced scheduling components based on EDeg scheduling algorithm. Scheduling of EDeg occurs on timer handler activation (each 8254 interrupt) as used in [16]. The list of waiting tasks *(waiting_list)* is sorted in increasing order of deadline. Preemption is enable in the sense that a job can be preempted and later resumed at any time and we have to compute the time loss associated with such preemption.

5.2.1 EDF Algorithm Implementation. In EDF module, the *EHS_schedule()* routine attempts to release tasks from the waiting list. The task is executed as soon as possible with no inserted idle time. Such implementation is known as earliest deadline as soon as possible (EDS) [9]. For a given periodic task set, the EDS schedule can be pre-computed and memorized in order to reduce scheduling overheads at run time.
The algorithmic description of EDF is given below:

```
EHS_schedule (t : current time)
begin
/*Check the waiting list (waiting_list) in order to release tasks*/

while (task=next(waiting_list)=not(∅)) do
  if (task→release_time <t) then
    break
  else if (EHS_compute_E(t)<0) then
    break
  else
    EHS_execute
    Erase task from waiting_list
  end if
end while
end
```

5.2.2 EDeg Algorithm Implementation. In EDeg module, the *EHS_schedule()* routine attempts to release tasks from the waiting list. The task is executed as soon as possible as long as there is sufficient energy in the storage unit. When this condition is not verified, the processor has to sleep so that the storage unit recharges as much as possible and as long as all the deadlines can still be met despite execution postponement.

```
EHS_schedule (t : current time)
begin
/*Check the waiting list (waiting_list) in order to release tasks*/

while (task=next(waiting_list)=not(∅)) do
  while (task→release_time <t) do
    EHS_wait
  end while
  while (EHS_compute_E(t)>0 &&
  EHS_compute_slack_energy(t)>0) do
    EHS_execute
  end while
  while (EHS_compute_E(t)<E &&
  EHS_compute_slack_time(t)>0) do
    EHS_wait
  end while
  Erase task from waiting_list
end while
end
```

6. PERFORMANCE EVALUATION

The performance of our real-time operating system was studied on a 1.8GHz Core 2 Duo. During initialization and termination, we can create and destroy a task in $1.4\mu s$ and $0.22\mu s$ respectively.

6.1 Varying Number of Tasks

One of the most important experiments is the overhead introduced by the EHS schedulers. By definition, the overhead of an operating system represents the time lost in handling all kernel mechanisms, such as context-switching overhead, task scheduling management overhead and so on.
In this experiment, we take interest in the overhead caused by context switching and by computing the slack time and slack energy. Experiments done in this category assess overhead by varying the number of tasks with periods of 10 milliseconds each one. Tasks are generated with a hyperperiod of 3360 ticks. Measurements were performed over a period of 100 seconds on a computer system with a 1.8GHz Core 2 Duo processor with 2 GB RAM.

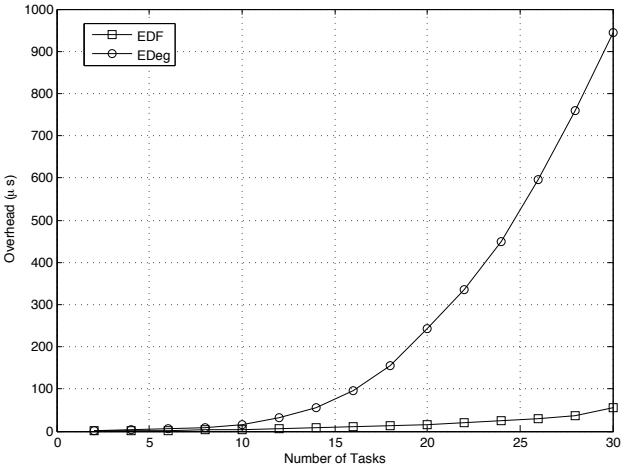

Fig. 5. Overhead of EDF and EDeg schedulers by varying the number of tasks

The overhead we show for the EHS scheduling components indicates that overhead of EDF and EDeg scales with the number of installed tasks. The overhead in EDF is directly linked to the time needed to preempt a task, save its context, load the context of another task, and resume that task. As the number of tasks increase, the number of preemptions increase and consequently the time overhead increases. This time is approximately negligible when compared to the task period (max. 0.6%).
Figure 5 shows that when the number of tasks increases, the time overhead coming from one computation of slack time and slack energy increases and consequently the time overhead increases. However, this time is very low when compared to the whole measurement period (max. 10%).

6.2 Varying Processor Utilization

In this experiment, we compute the time overhead by varying the processor utilization ($U_p = \sum_{i=1}^{n} \frac{C_i}{T_i}$). Under EDeg, as U_p varies, overhead of slack energy is approximately constant. This is because we compute slack energy each time we start the execution of a task while higher priority tasks will be released in the future. In addition, overhead for slack time is approximately constant since we compute the slack time each time the battery is empty.
As for EDF, as the processor utilization increases, the number of preemptions increases and consequently the time overhead will increase. However, the overhead time under full processor utilization

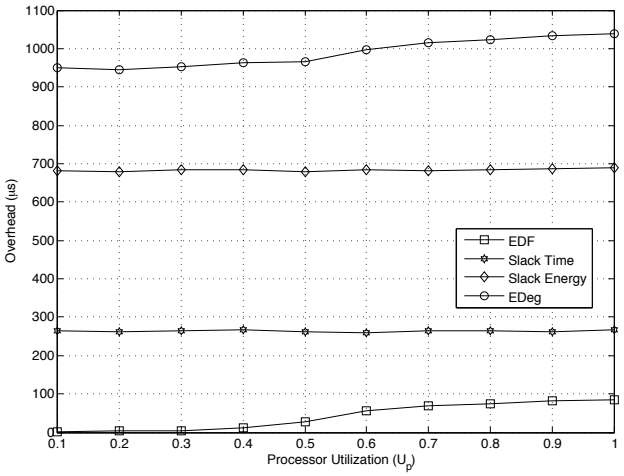

Fig. 6. Overhead of EDF and EDeg schedulers by varying U_p

is about $100\mu s$ which is 1% of the task period. This time is approximately negligible.
The overhead time for EDeg is computed by adding the overhead time due to slack time, slack energy and preemptions. The maximum overhead time is obtained when the processor utilization is equal to one. This time is about $1050\mu s$ which is 10.5% of the task period. Thus, the overhead time is still in the acceptable margin and does not affect the gain in performance of EDeg.

7. CONCLUSION

Real-time operating systems has been a very active research area in the last fifteen years. Many RTOS have been built with extremely different ideas. Real-time research has also influenced many standards such as real-time variants of Linux. Real-Time Linux provides a way to answer the needs of the new wireless sensor applications such as overload conditions and quality of service guarantees. However, new challenges continue to be posed to RTOSs. Recently, significant attention has been also paid to power management RTOSs where the operating system attempts to operate in a so-called energy neutral mode by consuming only as much energy as harvested. In addition, new generation of wireless sensors are more energy consuming and in most of them, recharging or replacing batteries is not practical or permitted. Consequently alternative power sources which are present in the environment should be employed.
Our work is then to develop a software model to evaluate the feasibility of integrating real-time scheduling algorithms under energy constraints in the form of software components on a real-time operating system. Starting from application specifications (tasks, profile of the energy source and the characteristics of the energy storage unit), the operating system must decide the feasibility of the application and build an appropriate solution regarding scheduling and dynamic power management.
Simulation study evaluate the impact of overheads on the relative performance of EDeg and EDF. From the beginning, we demonstrated that there the time overhead under EDF scheduling is approximately negligible since it does not exceed 1% of the task period in the worst strategies. In addition, we proved that the overhead time caused by EDeg scheduling is not greater than 11% of the task

period in the above experiments. Such value is still in the acceptable margin and does not affect the gain in performance of EDeg.
In practice, EDeg algorithm offered an efficient scheduling of periodic tasks in terms of the low time overhead. Hence, it will be interesting to be used in applications involving even more than 30 tasks.
Several interesting issues need further attention. To expand the applicability of our operating system, it is important to incorporate additional power management techniques including voltage/frequency scaling and dynamic power management to support more effective power-aware designs.

Acknowledgment

The Authors gratefully acknowledge the support for this study from SNCS Research Center, the University of Tabuk, and the Ministry of Higher Education in Saudi Arabia.

8. REFERENCES

[1] A. Sinha, A. Chandrasan, *Dynamic power management in wireless sensor networks*, IEEE Design and Test of Computers 18(2), pp. 62-74, 2001.

[2] M.T. Scmitz, B.M. Al-Hashimi and P. Eles. *System-Level Techniques for Energy Efficient Embedded Systems*. Kluwer Academic Publishers, 194 pages, 2004.

[3] A. Allavena and D. Mosse. *Scheduling of frame-based embedded systems with rechargeable batteries*. In Workshop on Power Management for Real-time and Embedded systems (in conjunction with RTAS 2001), 2001.

[4] C. Moser, D. Brunelli, L. Thiele, L. Benini. *Real-time scheduling for energy harvesting sensor nodes*. Real-Time Systems, Volume 37, Issue 3, Pages: 233 - 260, December 2007.

[5] P. Mantegazza. *DIAPM RTAI for Linux : Why's, what's and how's*. Real Time Linux Workshop, University de Technology of Vienna, 1999.

[6] C.M. Krishna and K.G. Shin. Real-Time Systems. McGraw-Hill Series in Computer Science, 448 pages, 1997.

[7] Maryline Silly-Chetto, Thibault Garcia-Fernandez and Audrey Marchand. *CLEOPATRE: Open-source Operating System Facilities for Real-time Embedded Applications*. Journal of Computing and Information Technology - CIT 15, 2007.

[8] Hussein El Ghor, Maryline Chetto, and Rafic Hage Chehade, *A Real-Time Scheduling Framework for Embedded Systems with environmental energy harvesting*. International Journal of Computers & Electrical Engineering, pp. 498-510, 2011.

[9] H. Chetto, and M. Chetto. *Some results of the earliest deadline scheduling algorithm*. IEEE Transactions on Software Engineering, 15(10): 1261-1269, 1989.

[10] Maryline Chetto, Hussein EL Ghor and Rafic Hage Chehade. *Real-Time Scheduling for Energy Harvesting Sensors*. The 6th International Conference for Internet Technology and Secured Transactions, Abu Dhabi, UAE, December 11-14, pp. 396 - 402, 2011.

[11] Tokuda H., Nakajima T., Rao P. Real time *MACH: Towards a predictable real-time system*. Proceedings of the Usenix MARCH Workshop, vol. 1, 1990.

[12] Stankovic J.A., Ramamritham K. *The Spring kernel: A new paradigm for real-time systems*. IEEE Software, 1991.

[13] M. Silly-Chetto, *The EDL Server for scheduling periodic and soft aperiodic tasks with resource constraints*, Real-Time Systems, 17(1), pp.1-25, 1999.

[14] Saksena M., Da Silva J. Agrawala A. *Principles of real-time systems*. chapter design and Implementation of Maruti II Prentice Hall, 1994.

[15] Jeffay K., Stone D.L., Poirier D.E. *Kernel support for efficient, predictable real-time systems*. Proceedings of Joint IEEE Workshop on real-time operating systems and software, pages 8-13, 1991.

[16] T. Garcia, A. Marchand and M. Silly-Chetto. *CLEOPATRE: A R & D project for providing new real-time functionalities to RTAI Linux*. 5th Real Time Linux Workshop, Valence (Esp.) , 9-11 Nov 2003.

[17] H. CHETTO, M. SILLY-CHETTO. *An adaptive scheduling algorithm for a fault-tolerant real time system*. Software Engineering Journal, 6(3), pp. 93?100, May 1991.

[18] Leung J-Y-T, Whitehead J. *On the complexity of fixed-priority scheduling of periodic real-time tasks*. Performance Evaluation Journal. 2(4):237?50, 1982.

[19] J.-W.-S. Liu. Real-Time Systems. Prentice-Hall, 2000.