Proceedings of the 2nd IEEE International Conference on Energy Aware Computing (ICEAC 2011)

Energy Management for Tiny Real-Time Kernels

Mario Bambagini, Francesco Prosperi, Mauro Marinoni, Giorgio Buttazzo
{*m.bambagini, f.prosperi, m.marinoni, g.buttazzo*}*@sssup.it*
Scuola Superiore Sant'Anna Pisa, Italy

***Abstract*—In battery operated embedded systems, an efficient energy management is a key feature for increasing the system lifetime, as well as for controlling the application performance. In this paper, we present a power management module designed for tiny embedded operating systems and implemented in the Erika Enterprise, an OSEK-compliant kernel. The obtained results show both the effectiveness of the presented component and the impact of operating mode changes on global performance.**

I. Introduction

Nowadays, thanks to the progress of information and electronic technologies it is possible to produce very small devices characterized by considerable performance and a low cost per unit. Hi-tech products are widespread and, in the future, this trend will increase. In this scenario, the issue of energy saving becomes central, especially for portable devices and autonomous mobile robots, for which the minimization of energy consumption not only prolongs their lifetime, but allows money saving and has also a long term impact for reducing environmental pollution.

In general purpose computing systems, power management is implemented through the Advanced Configuration and Power Interface (ACPI) [1], whose specification defines a standard for devices configuration and monitoring. In particular, ACPI offers the operating system an easy and flexible interface to discover and configure the compliant devices. For instance, unused devices, including the entire system, can be switched to a low-power state, according to the current state and user preferences. The ACPI approach is suitable to all classes of computers, including personal computers, laptops, workstations, and servers, but is considerably expensive in terms of computation and memory requirements to work on tiny systems.

In the embedded systems domain, Brock and Rajamani [2] proposed a valid solution in which the system includes a set of policies and tasks are divided into groups according to their energy request or importance. The current policy is chosen by the policy manager, a component provided by the system designer. The system behavior is encoded as a grid, where each cell represents the configuration to adopt when a task of a specific group runs and a policy is set as active.

In CMOS technology, which is leading today's hardware circuits, the power consumption of a gate can be expressed as a function of the supply voltage V and the clock frequency f through Equation (1) [3]:

$$P_{gate} = C_L V^2 p_s f + V I_{short} + V I_{leak} \tag{1}$$

where C_L is the total capacitance driven by the gate, p_s is the gate activity factor (i.e., the probability of gate switching), I_{short} is the current flowing between the supply voltage and ground during gates switching, and I_{leak} is the leakage current. In particular, the three terms describe the dynamic, the short circuit, and the static power component dissipated in a gate, respectively.

Aiming at reducing the dynamic component, Dynamic Voltage and Frequency Scaling (DVFS) techniques consist of varying the voltage V and the frequency f in Equation (1), in order to slow down the processor while keeping the task set feasible, according to the actual system workload. An adverse effect of such an approach is represented by the fact that V alters the circuit delay, thus limiting the maximum usable frequency. This phenomenon is reported in Equation (2), where V_T is the *Threshold Voltage*, that is, the minimum voltage between gain and source able to create a channel from drain to source in a MOSFET transistor.

$$circuit\ delay = \frac{V}{(V - V_T)^2}. \tag{2}$$

Given such a limitation, not all the pairs (V,f) are usable.

Another well-known approach is the Dynamic Power Management (DPM). Such technique is used to switch the processor off during long idle intervals, thus postponing tasks execution as long as possible, still preserving the schedulability of the task set.

It is worth observing that both voltage/frequency changes performed by DVFS methods and operating mode switches (e.g., run to sleep) actuated in DPM approaches introduce an overhead in terms of both switching time and energy loss. In real implementations on real-time kernels, such contributions cannot be ignored.

Contribution of the paper: This paper presents a module for managing power consumption in tiny kernels for real-time embedded systems with limited resources, such as memory, CPU and power. The proposed module achieves considerable energy savings, satisfying the application's timing constraints. This paper improves the original work proposed by Marinoni et al. [4] by generalizing the device manager with a modular approach that allows the user to select a policy customized for a specific device providing a uniform interface. The proposed solution has been implemented in the Erika Enterprise kernel [5] to manage CPUs, timers, and servomotors.

Organization of the paper: Section II presents the architecture of the kernel module, its working flow and its interaction with the operating system. Section III describes the policies implemented in the module. Section IV reports the experimental results performed on the hardware, while Section V ends the paper with the concluding remarks, pointing out ideas for future improvements.

A. Related work

One of the first paper about power management was proposed by Yao et al. [6] in 1995. The authors studied three algorithms that, given a task set, compute the minimum energy schedule. The proposed analysis compared the algorithms efficiency with respect to different power models but without taking into account switching overheads.

Swaminathan et al. [7], [8] presented two algorithms, LEDF and E-LEDF, which set the lowest CPU speed that allows the earliest deadline task to finish within its deadline. The difference between the two algorithms is that E-LEDF considers the switching overheads, while LEDF does not. Such methods, however, do not provide hard guarantee for all the tasks, hence they can only be used for soft real-time systems.

In 2004 and 2007, Seong et al. proposed two algorithms. The first algorithm (OLDVS) [9] accumulates the time generated by early terminations and exploits it to decrease the CPU speed so that the current task is completed at the same time at which it would have completed in the worst case. The second algorithm (OLDVS*) [10] divides each task in two parts: the first one is executed at a slower speed, while the second one is executed at a higher speed. This approach is based on the assumption that the probability of ending the task instance in the first part is significantly higher than finishing on the second part.

Aydin et al. [11] proposed three algorithms with growing complexity. The first one computes the lowest CPU speed such that the task set is kept schedulable under the assumption that all tasks execute for their Worst-Case Execution Time (WCET). The second algorithm (DRA) keeps track of the times at which a task is going to be dispatched (computed off-line and stored in a dispatch queue). At runtime, if a task is dispatched earlier, the CPU is slowed down to prolong the execution until its original finishing time. The third algorithm (AGR) estimates the tasks completion times based on past instances and computes the lowest CPU speed to keep the task set feasible assuming that tasks execute for such estimates. However, since the estimations can be optimistic, the algorithm may speed the CPU up to recover from a task overrun.

Zhu et al. [12] proposed an algorithm similar to AGR by using a feedback controller to estimate the task execution time.

The problem of obtaining an optimal frequency from a discrete frequency range was discussed by Bini et al. [13]. The authors provided a method for computing the optimal speed off-line (that could be unavailable in the specific architecture) and introduced a speed modulation technique to achieve the required speed using two discrete values. The analysis selects the pair of frequencies that minimizes energy consumption also considering switching overheads into account. Despite of its innovative contribution, such an off-line approach does not take advantage of tasks early terminations to further reduce consumption.

The raising impact of the leakage power in modern architectures is driving the research on power management toward DPM techniques. Huang et al. [14], [15] proposed an off-line analysis that combines DPM and Real-Time Calculus to estimate tasks arrivals and compute the CPU idle intervals. Jejurikar et al. [16] proposed an approach based on task procrastination to maximize the time spent in sleep mode.

II. Architecture

The energy saving module proposed in this work (also referred to as the *Power Manager*) is part of the kernel and interacts with the scheduler, the hardware devices, and the application, as illustrated in Figure 1. While the scheduler selects the next task to execute, the Power Manager chooses an appropriate running configuration (i.e., speed and voltage).

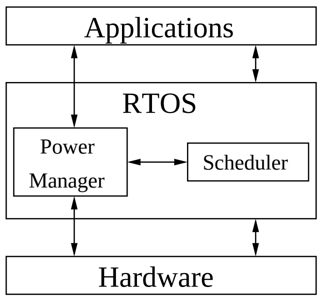

Fig. 1. Interaction of the Power Manager with the other system components.

A block diagram of the Power Manager is reported in Figure 2. It consists of three hierarchically organized modules: the Application Programming Interface (API), the CPU Manager and the Devices Manager. Such a modular implementation allows the programmer to easily remove sub-components when not needed by the application, so helping to reduce the footprint.

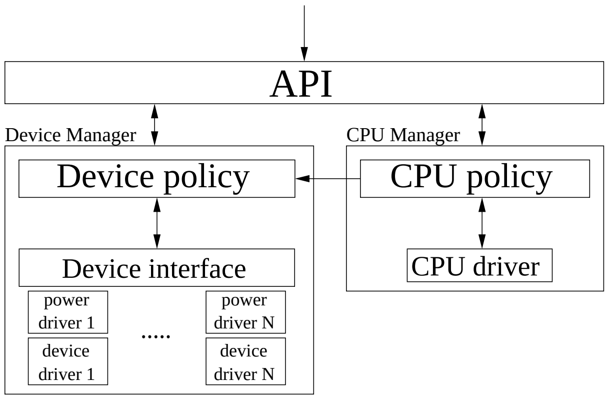

Fig. 2. Block diagram of the Power Manager.

The API module implements the interface defined for the interaction with the kernel and the applications. The CPU Manager is responsible for the power management of the CPU. Using a set of special callback functions called *hooks*, the kernel informs the module about four scheduling events: task activation, task termination, task preemption, and task dispatch.

The *CPU policy* submodule implements the energy saving policies, which typically select the best speed to meet the applications constraints, while satisfying a given set of performance requirements. The *CPU driver* is in charge of setting the CPU parameters, such as frequency and energy saving state. It is located at the lowest abstraction level as its code is hardware-dependent.

The Devices Manager handles internal and external peripherals. Inside it, the *Device policy* submodule contains all the device policies, developed according to the *Device Interface*, which offers a single access point to the devices. For each of them, two stacked components, *Power driver* and *Device driver*, abstract the device behavior using a discrete set of states, as shown in Figure 3. Each state is characterized by a specific power consumption and quality of service level.

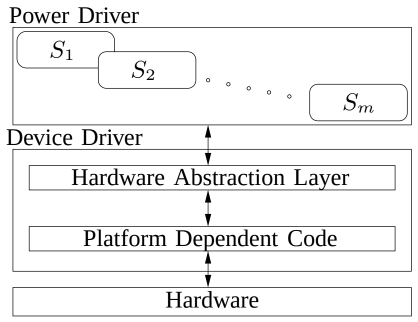

Fig. 3. Device stack.

The link between the CPU and the Device module is necessary to adjust the configurations of devices, whenever a speed scaling or mode switching event occurs. For instance, when a new speed is set, the system timers need to be automatically reconfigured to offer the same tick period.

When an internal error occurs, a user-defined callback function is invoked, demanding the user to manage the exception. A typical scenario could occur on speed scaling: if a device detects that the modified configuration is not able to guarantee the same performance of the previous state, the callback is invoked to solve the situation. For instance, if an UART transceiver with a modified system speed is not able to sustain the communication baud rate, the user has to specify how to fix this issue.

A. Sample scenarios

This section describes two examples to better explain how the modules interact with each other. The first scenario, shown in Figure 4, supposes that a new task instance becomes running. The kernel, after having managed the event, informs the CPU Manager by invoking the corresponding hook (1). Once the event is notified, the active policy selects the best frequency to execute the actual workload within the specified timing constraints. The new speed is communicated to the CPU driver, which makes it effective (2).

Once the new frequency is fully operational, the CPU manager notifies the new configuration to the Device Manager (3), which in turn informs the devices under its control (4) (5). Finally, each device sets its hardware registers to obtain the same performance with the new configuration (6). If this is not possible, the module invokes the error callback to solve the situation.

Figure 5 presents another situation in which a running task has to control a servomotor to properly hold a given load with the minimum energy, using a torque sensor. The controller reads the torque sensor, computes the load value and notifies it, through the API, to the Device Manager (1).

The active policy chooses the appropriate state able to hold the actual load with the minimum energy consumption and notifies it, through the Device interface (2), to the corresponding

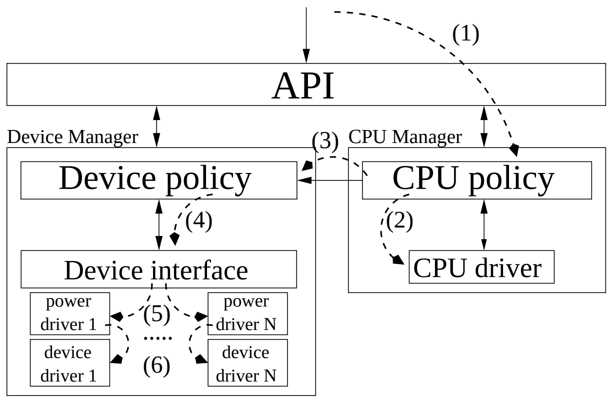

Fig. 4. First example of modules interaction.

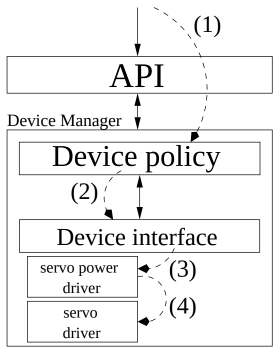

Fig. 5. Second example of modules interaction.

Power driver (3), which translates the communicated state in an appropriate set of commands for the specific servo driver, modifying the actuator performance (4).

III. Implemented policies

This section presents the policies implemented inside the CPU Manager and the Device Manager. Such policies are configured off-line and are automatically invoked at runtime without any user interaction.

A. CPU policies

The policies implemented in the CPU Manager work for a single CPU and adopt a discrete frequency set. In the following, the term *speed* refers to a frequency normalized with respect to the highest one ($s = f/f_{max}$) and s^* denotes the lowest speed ensuring the task set feasibility.

The following three policies are implemented:

- *OnLine Dynamic Voltage Scaling* (OLDVS) is a policy proposed by Lee and Shin [9] that selects the minimum available speed to prolong a task execution time up to its WCET;
- *Bonus Sharing DVFS* (BSDVFS) is a variant of OLDVS proposed in this work to take switching overheads into account;
- BSDVFS* is a variant of OLDVS*, originally introduced by Gong et al. [10], extended in this work to take switching overheads into account.

All the analyzed policies compute the most suitable frequency to exploit tasks early terminations. Note that the tasks deadlines are not considered to slow the CPU down: all policies exploit the unused computation time, if any, from the previous jobs, prolonging the execution of the current job (at a lower speed) until its worst-case finishing time e_i (that is, the time at which the task would finish in the worst case at speed s^*).

To better illustrate the implemented approaches, the three policies are instantiated on a CPU with a set S of three speeds $S = \{0.5, 0.75, 1\}$ and are applied to a task set consisting of two tasks with WCETs equal to $C_1 = 40$ and $C_2 = 30$ (note that all WCETs values refer to the tasks executing at the highest speed $s = 1$). For the sake of simplicity, we assume that task set parameters are such that $s^* = 1$.

Figure 6 shows a schedule in which each task executes for its WCET on the CPU running at speed s^*. Having no early terminations, the speed is not changed and no energy can be saved in this case.

If τ_1 and τ_2 arrive at $a_1 = 0$ and $a_2 = 5$, their worst-case finishing times will be at $e_1 = 40$ and $e_2 = 70$, respectively.

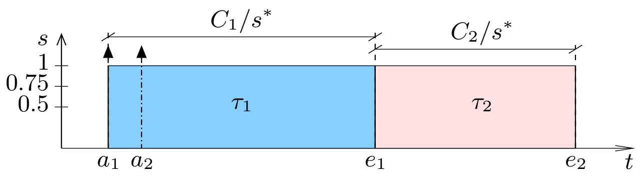

Fig. 6. Tasks executing for their WCETs.

To apply any of the policies listed above, the system has to keep track of the actual execution time used by a task τ_i. Such a monitoring can be efficiently implemented by starting a timer each time a task becomes running and stopping it when the task is preempted or completed. If ε denotes such an interval executed at a speed s, the remaining WCET of the task can be computed as

$$c_i = C_i - \varepsilon\, s. \tag{3}$$

Moreover, a *bonus time*, denoted as B, is introduced to account for the unused time accumulated by previous tasks' executions: when τ_i finishes, the saved time c_i is added to B, which can be exploited as an extra time available for the next scheduled task.

Figure 7 shows the schedule produced by OLDVS when τ_1 finishes at time $t = 8$. At the beginning, τ_1 runs at the highest speed since no computation time is saved at time $t = 0$ (thus $B = 0$). At time $t = 8$, τ_1 completes, saving $c_1 = 32$ units of time. Thus, B is incremented by c_1 and τ_2 can exploit B to execute at a slower speed such that $C_2/s = (C_2 + B)/s^*$. In general, having a bonus time B, a task τ_i with residual WCET c_i can still meet its deadline by running at the speed

$$s_{\scriptscriptstyle OLDVS} = \min_{s_j \in S} \left\{ s_j \geq \frac{c_i}{c_i + B} s^* \right\}. \tag{4}$$

In the example shown in Figure 7, since $B = 32$ and $C_2 = 30$, the lowest feasible speed is $s_{\scriptscriptstyle OLDVS} = 0.5$. With such a speed, τ_2 would finish in the worst-case at $t = 68$.

Figure 8 illustrates the schedule produced by BSDVFS, when taking switching overheads into account. Let $\delta(s_x, s_y)$ be the switching overhead from speed s_x to s_y. Then, the minimum speed under which a task τ_i with residual WCET c_i and bonus time B can still meet its deadline is computed as follows:

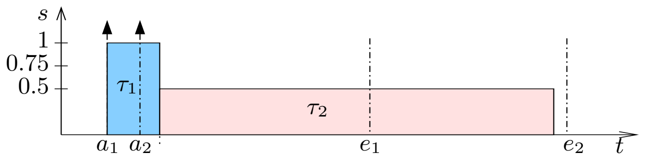

Fig. 7. Schedule produced by OLDVS.

$$s_{\scriptscriptstyle BSDVFS} = \min_{s_y \in S} \left\{ s_y \geq \frac{c_i}{(c_i + B)/s^* - \Delta_{\scriptscriptstyle BSDVFS}(s_x, s_y)} \right\} \tag{5}$$

where

$$\Delta_{\scriptscriptstyle BSDVFS}(s_x, s_y) \triangleq \delta(s_x, s_y) + \delta(s_y, s^*).$$

The term $\delta(s_y, s^*)$ accounts for the overhead needed for restoring the speed at s^* in the case the next running task is not able to slow the CPU down further. In the considered example, the switching overheads are considered symmetric ($\delta(s_x, s_y) = \delta(s_y, s_x)$) and proportional to the speed gap. In particular: $\delta(0.5, 0.75) = 2$, $\delta(0.5, 1) = 5$, and $\delta(0.75, 1) = 2$.

For the given task set, the feasibility test is satisfied only for $s = 0.75$ and $s = 1$, since for $s = 0.5$ τ_2 completes at $t = 78$ (i.e., beyond time $e_2 = 70$). Therefore, the running speed is set to 0.75, causing τ_2 to finish in the worst case at $t = 50$.

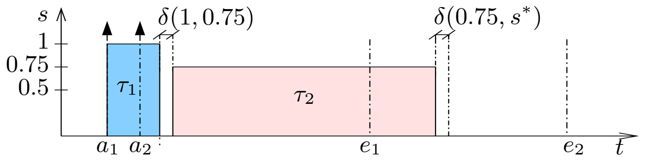

Fig. 8. Schedule produced by BSDVFS.

The idea behind BSDVFS* consists of splitting a task τ_i in two parts, $\tau_i^{\scriptscriptstyle L}$ and $\tau_i^{\scriptscriptstyle H}$, with WCETs $c_i^{\scriptscriptstyle L}$ and $c_i^{\scriptscriptstyle H}$, executed at different speeds, $s_{\scriptscriptstyle L}$ and $s_{\scriptscriptstyle H}$, set as the lower and the higher adjacent speed of $s_{\scriptscriptstyle BSDVFS}$. The switching instant, and therefore the two values ($c_i^{\scriptscriptstyle L}$,$c_i^{\scriptscriptstyle H}$) are computed to prolong τ_i's execution until its worst-case finishing time e_i. Hence, they are computed as follows:

$$\begin{cases} c_i^{\scriptscriptstyle L} + c_i^{\scriptscriptstyle H} = c_i \\ \dfrac{c_i^{\scriptscriptstyle L}}{s_{\scriptscriptstyle L}} + \dfrac{c_i^{\scriptscriptstyle H}}{s_{\scriptscriptstyle H}} + \Delta_{\scriptscriptstyle BSDVFS^*}(s_x, s_{\scriptscriptstyle L}, s_{\scriptscriptstyle H}) \leq \dfrac{c_i + B}{s^*} \end{cases} \tag{6}$$

where

$$\Delta_{\scriptscriptstyle BSDVFS^*}(s_x, s_{\scriptscriptstyle L}, s_{\scriptscriptstyle H}) \triangleq \delta(s_x, s_{\scriptscriptstyle L}) + \delta(s_{\scriptscriptstyle L}, s_{\scriptscriptstyle H}) + \delta(s_{\scriptscriptstyle H}, s^*).$$

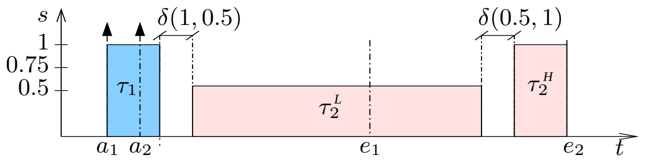

Fig. 9. Schedule produced by BSDVFS*.

It is worth observing that BSDVFS* is the only method able to fully exploit the time bonus to make the current task to complete at its worst-case finishing time e_i. Note that if a task instance finishes earlier, most of the execution is spent at the lower speed s_L, so achieving higher energy reduction.

Figure 9 shows the schedule produced by BSDVFS*, using $s_L = 0.5$ and $s_H = 1$, which are the adjacent speeds of $s_{BSDVFS} = 0.75$. In this example, $\delta(s_H, s^*) = 0$ since $s_H = s^* = 1$.

According to Equation (6), the execution times of τ_2^L and τ_2^H result to be $c_2^L = 22$ and $c_2^H = 8$, respectively. Note that τ_2 finishes exactly at time $e_2 = 70$.

B. Device policies

The policies implemented in the Device Manager support timers and servomotors.

Servomotors are devices driven by Pulse-Width Modulation (PWM) signals, whose absorption peak is concentrated at the beginning of the signal period with a constant intensity and a duration proportional to the detected angle error.

The Power driver offers m states, each one identified by a specific PWM period. The policy inside the Device Manager associates a specific power state to the required torque according to a pre-specified internal look-up table.

To be implemented, the servo driver (or a PWM peripheral) uses a timer to generate the control signals and the Device Manager interacts with such peripherals to vary the PWM period.

Despite of the negligible energy consumption, timers are managed by the module to maintain the consistency of the system time independently of the running speed. The Device Manager does not provide any policy for them and the Power driver offers only two states, ON and OFF, corresponding to the timer active and timer inactive modes, respectively.

IV. Experimental results

The Power Manager has been developed as module of the Erika Enterprise kernel [5] and tested on the Evidence FLEX boards [17] equipped with a Microchip dsPIC33FJ256MC710 microcontroller [18].

A. CPU

A set of experiments has been carried out to evaluate the impact of the three policies on the energy consumption. The experimental measurements refer to the whole board. The CPU driver supports eight different frequencies: 40, 35, 30, 20, 16, 10, 8 and 2 MIPS (Million of Instructions Per Second).

The switching overhead depends on the specific frequency levels, because the lowest frequency is obtained directly from the external clock signal, while the other frequencies are derived by a PLL. Switching the PLL on takes about 1ms, while turning it off or adjusting it to any other frequency takes between 4μs and 40μs.

Figure 10 shows the current consumption of the CPU as a function of the frequency. The upper curve in the figure refers to actual measurements on the entire board, whereas the others two are derived from the datasheet and refer to the CPU only. Note that, for the considered architecture, halving the frequency doubles the execution time, but does not reach a current consumption of 50%. For instance, reducing the frequency from 40 to 20 MIPS, the current consumption goes

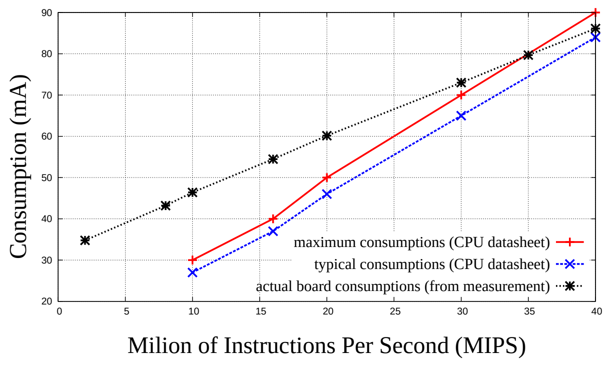

Fig. 10. CPU power consumptions.

from 86.12 mA to 59.12 mA. Such a result indicates that DVFS approaches are not effective on this architecture, where higher saving would be achieved by algorithms that run the application at higher speeds.

In the next experiment the three policies have been tested on ten periodic tasks with a total worst-case utilization $U_{wc} = 0.98$. The lowest frequency which guarantees the task set feasibility in the worst case is 40 MIPS (corresponding to a speed $s^* = 1$).

Figure 11 shows the energy consumption (normalized with respect to the case of no online policy) as a function of the ratio of the actual utilization (U_{real}) and the worst-case one (U_{wc}). As observed above, in the considered architecture, policies using higher speed achieve a lower energy consumption. Therefore, although BSDVFS* is able to exploit slower frequencies than BSDVFS, its average consumption is similar to BSDVFS, because it is compensated by longer execution times. Note that at high utilization ratios ($U_{real}/U_{wc} > 0.5$) OLDVS is characterized by higher energy consumptions because, by neglecting switching overheads, it is able to select lower speeds. On the other hand, at low utilization ratios ($U_{real}/U_{wc} \le 0.5$), both BSDVFS and BSDVFS* achieve higher consumptions because, at the end of each job, they restore the speed to s^* to ensure task set feasibility. Such an effect is enhanced for very low utilization ratios due to the higher overhead (1 ms) introduced when switching to the minimum frequency.

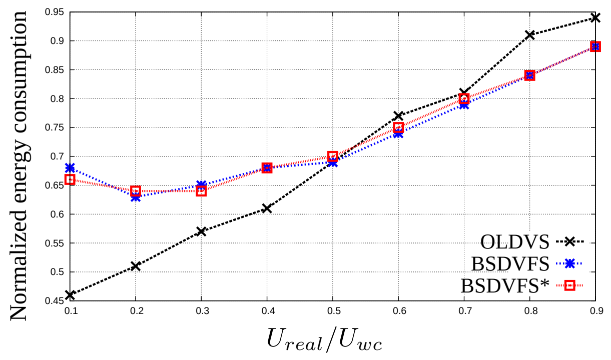

Fig. 11. Normalized energy consumptions of the policies.

B. Devices

Another set of experiments has been performed on a servomotor to derive a policy for driving the device with the minimum average energy consumption. The servomotor used in this test is a Hitec HS-645MG, characterized by a minimum absorption of 12.56 mA, and a peak current of 1 A. The peak occurs at the beginning of the PWM period and has a duration proportional to the detected angular error.

The experimental tests compare the measured mean power consumption as a function of the applied torque, using three different PWM periods: 10, 20 and 40 ms. As shown in Figure 12, the effectiveness of each period depends on the energy needed to correct the accumulated error between two consecutive updates.

Note that for very small torques (< 0.5 kg $\times$ cm) the consumption is not affected by the PWM period, because the angular error on the axis is below the threshold used by the internal position controller. Low torques ($\in [0.5, 1)$ kg $\times$ cm) typically generate similar errors for any PWM period, leading to similar energy costs per update; therefore, longer periods produce less updates per time unit and consume less energy. For torques higher than 1.0 kg$\times$cm, a period of 40 ms copes with higher errors, accumulated between two consecutive updates, resulting in a higher consumption. Moreover, this period cannot guarantee an angular error less than $5°$ with torques greater than 1.5 kg$\times$cm; hence, the measures for such a period are not considered for higher torques. For medium torques ($\in [1, 1.6)$ kg $\times$ cm), the errors produced by 10ms and 20 ms PWM periods are similar, hence the longer period (20 ms) leads to a better performance. The shorter period (10ms) is more suited for heavier loads, because it frequently corrects smaller errors, so leading to lower consumptions.

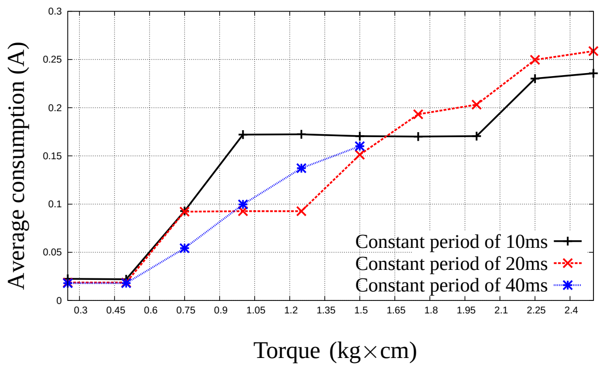

Fig. 12. Servo consumptions with different updating periods.

As a result, the implemented policy binds torque ranges with the period that minimizes the energy consumption, according to the results reported in Figure 12. To optimize the implementation, the results are stored in a look-up table that associates the period leading to the minimum consumption to the corresponding torque range, which also defines a power state. Table I shows the specific state values derived from Figure 12.

Torque range ($kg * cm$)	Power state	Period (ms)
$[0.0, 1.0)$	STATE$_2$	40
$[1.0, 1.6)$	STATE$_1$	20
$[1.6, 2.5]$	STATE$_0$	10

TABLE I
LOOK-UP TABLE USED BY THE SERVOMOTOR POLICY.

V. Conclusions

This paper presented a power management module for real-time embedded systems. The proposed component, designed to be highly modular, implements a set of policies for the CPU and devices using a uniform interface. The power layer has been implemented on an OSEK compliant kernel and tested on an embedded platform based on a dsPic microcontroller. Experimental results show the effectiveness of the approach showing how the module can be used to select the most appropriate policy for a specific application on a given architecture. An example of device policy has been also shown for driving a servomotor with minimum energy consumption.

As a future work, the module will be ported on different systems, for instance on ARM platforms, and expanded to support other policies for the CPU (such as DMP algorithms) and for other devices (such as communication transceivers).

References

[1] "Acpi web site," http://www.acpi.info/.
[2] *Dynamic power management for embedded systems [SOC design]*, November 2003. [Online]. Available: http://dx.doi.org/10.1109/SOC.2003.1241556
[3] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen, "Low power cmos digital design," *IEEE Journal of Solid State Circuits*, vol. 27, pp. 473–484, 1995.
[4] M. Marinoni, G. Buttazzo, T. Facchinetti, and G. Franchino, "Kernel support for energy management in wireless mobile ad-hoc networks ."
[5] "Erika enterprise rtos," http://erika.tuxfamily.org/.
[6] F. Yao, A. Demers, and S. Shenker, "A scheduling model for reduced cpu energy," in *Proc. of the 36th Annual Symp. on Foundations of Computer Science*, ser. FOCS '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 374–.
[7] V. Swaminathan, "Real-time task scheduling for energy-aware embedded systems," 2000.
[8] V. Swaminathan and K. Chakrabarty, "Investigating the effect of voltage-switching on low-energy task scheduling in hard real-time systems," in *Proc. of the 2001 Asia and South Pacific Design Automation Conf.*, ser. ASP-DAC '01. New York, NY, USA: ACM, 2001, pp. 251–.
[9] C.-H. Lee and K. G. Shin, "On-line dynamic voltage scaling for hard real-time systems using the edf algorithm," in *Proc. of the 25th IEEE International Real-Time Systems Symp.* Washington, DC, USA: IEEE Computer Society, 2004, pp. 319–327.
[10] M.-S. Gong, Y. R. Seong, and C.-H. Lee, "On-line dynamic voltage scaling on processor with discrete frequency and voltage levels," in *Proc. of the 2007 International Conference on Convergence Information Technology*, ser. ICCIT '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 1824–1831.
[11] H. Aydi, P. Mejía-Alvarez, D. Mossé, and R. Melhem, "Dynamic and aggressive scheduling techniques for power-aware real-time systems," in *Proc. of the 22nd IEEE Real-Time Systems Symp.*, ser. RTSS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 95–.
[12] Y. Zhu and F. Mueller, "Feedback edf scheduling of real-time tasks exploiting dynamic voltage scaling," *Real-Time Syst.*, vol. 31, pp. 33–63, December 2005.
[13] E. Bini, G. Buttazzo, and G. Lipari, "Speed modulation in energy-aware real-time systems," in *Proc. of the 17th Euromicro Conference on Real-Time Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 3–10.
[14] K. Huang, L. Santinelli, J.-J. Chen, L. Thiele, and G. C. Buttazzo, "Periodic power management schemes for real-time event streams." in *the 48th IEEE Conf. on Decision and Control (CDC)*. Shanghai, China: IEEE, 2009, pp. 6224–6231.
[15] ——, "Adaptive dynamic power management for hard real-time systems," in *the 30th IEEE Real-Time Systems Symp. (RTSS)*. Washington D.C. U.S.: IEEE, 2009, pp. 23–32.
[16] R. Jejurikar, C. Pereira, and R. K. Gupta, "Leakage aware dynamic voltage scaling for real time embedded systems," in *In Proc. of the Design Automation Conference*, 2004, pp. 275–280.
[17] "Evidence srl," http://www.evidence.eu.com/.
[18] "Microchip web site," http://www.microchip.com/.