Low-Power Scheduling with DVFS for common RTOS on Multicore Platforms

Shuai Li

THALES Communications & Security
4 Avenue des Louvresses
92622 Gennevilliers, France
shuai.li@fr.thalesgroup.com

Florian Broekaert

THALES Communications & Security
4 Avenue des Louvresses
92622 Gennevilliers, France
florian.broekaert@thalesgroup.com

ABSTRACT

This paper is on a low-power real-time scheduler integrated into a common Linux operating system. The low-power scheduler aims at reducing energy consumption in a system and uses Dynamic Voltage and Frequency Scaling (DVFS) to achieve its goal. The developed solution was implemented as a layer above the Linux OS scheduler. A framework was also developed to integrate the scheduler above Linux without modifying the kernel. We investigate the advantages, challenges, and viability of such a solution in the real-time embedded systems domain.

Categories and Subject Descriptors

D.4.1 [**OPERATING SYSTEMS**]: Process Management—*Scheduling*

General Terms

Concurrency, Multitasking, Threads

Keywords

Real-Time Embedded System, Real-Time Scheduling, Dynamic Voltage and Frequency Scaling, Linux, POSIX

1. INTRODUCTION

An embedded system is a computing component part of a larger system. An embedded system is autonomous and runs on a limited energy source so it thus has energy constraints. A real-time system is one with time constraints. To guarantee time constraints, one common solution applied by system integrators is to take margin by setting the system's processor at its maximum speed (i.e. maintaining the system at its maximum computing capacity at all time). This has a major drawback as it costs more energy which goes against the limited energy philosophy of embedded systems. One possible solution to solve this conflict in Real-Time Embedded Systems (RTES) design is to decrease the system's processor speed when maximum computing capacity is not needed.

Dynamic Voltage and Frequency Scaling (DVFS) is one possible technique to scale the system's processor speed. DVFS consists in changing a processor's speed through increasing/decreasing the processor's (voltage, frequency) couple. Decreasing the (voltage, frequency) means an increase in computing time but leads to a decrease in energy consumption. In today's processors using CMOS circuitry, the energy dissipated per cycle (i.e. power) scales quadratically to the supply voltage ($E \propto V^2$) [2]. We see that one of the main advantages of using DVFS is that it can provide potentially large energy savings.

When using DVFS with RTES, it must be assured that the time constraints are still met. Traditional non-real-time OS power managers do not consider parameters part of real-time scheduling [7], e.g. Worst Case Response Time (WCRT), task deadline. For example in the Linux power manager, the implemented heuristics rely on past activities to take decisions. The consequence of this *a-posteriori* scheme is that future time constraints may be violated if the performance level was lowered too much. In the case of RTES, it is thus important that power managing strategies are coupled with real-time scheduling policies. This results in what is called low-power scheduling.

Unfortunately power management solutions in Real-Time Operating Systems (RTOS) are not commonly available and, as we saw, traditional techniques are not suited for RTES. This is why we investigate in this paper the challenges and viability of implementing a low-power scheduler in a common Linux OS.

The rest of the paper is organized as follows. Section 2 is on works related to our research on low-power scheduling. Section 3 exposes the system model and the notations used in this paper. Section 4 explains the algorithms inside the low-power scheduler we have established. Section 5 shows how it was implemented above the Linux scheduler. Section 6 exposes experimental results on the implementation, in terms of performance and architecture choice. We conclude in section 7 by listing our future works.

2. RELATED WORKS

Low-power scheduling policies using DVFS for real-time systems have been studied in the past. The solutions in the literature can be grouped into three categories: inter-task, intra-task, and statistical.

In the inter-task approach, the task is the atomic entity on which DVFS decisions are taken. This means there is no knowledge of what happens inside a task. In [10], the authors compute an offline energy-optimal schedule that the online scheduler tries to follow. In [6] the author establishes online DVFS algorithms that reclaim slack time (unused computing time).

EWiLi'13, August 26–27, 2013, Toulouse, FRANCE

The intra-task approach is intrusive and needs code instrumentation. Frequency is scaled according to what happens during a task's execution. In [8], the authors suggest a method that uses a task's control-flow composed of nodes. According to the node being executed, the frequency for the next node is chosen. In [1] the authors suggest to add offline-determined voltage-scaling checkpoints in the code.

In the statistical approach, no information on tasks are available. The system monitors events happening during execution and scales the frequency according to these events. In [9] cache hit/miss ratio and memory access counts are used to determine the frequency. In [3] the rate of instructions being executed is used to choose the frequency.

This paper's contribution is based on an intra-task approach similar to the one presented in [8]. Our contribution relies in our low-power scheduler that takes into account application modes (e.g. quality of service) and multitasking on multicore platforms with specific configurations (e.g. Symmetric Multiprocessing [4]). We also developed a framework to integrate such kind of DVFS algorithms in a common Operating System (OS), i.e. Linux.

3. SYSTEM MODEL AND NOTATIONS

The low-power scheduler uses a new task model to ensure its functionality. It also runs on a platform (RTOS and hardware) with specific characteristics. In this section we specify the platform on which the tasks are run, and we define the new task model.

3.1 Platform Model

In order to use the low-power scheduler, it has to be run on a platform with the following hardware entities:

- Core ($CORE_m$): A processing unit that executes atomic instructions.
- Processor (CPU_p): A group of cores (one or more) that also structures their interaction.

Each core has a table of voltage/frequency pair defined as the following:

Definition 1 (Voltage/Frequency Pair). *A core $CORE_m$'s voltage/frequency pair $(V,F)_{um}$ (called "pair" from now on) is a u identified tuple where V is the core's input voltage and F the core's frequency. V and F of $(V,F)_{um}$ are designated as V_{um} and F_{um}.*

The voltage is not necessarily static, i.e. for a same frequency F, V is defined within an interval and several V may exist for the same F. For the rest of the paper, the voltage is assumed to be static, i.e. there is only one V for each F since they are linked. So saying "a frequency has changed" is equivalent to saying the (V,F) pair has changed.

In our work, we consider processors in Symmetric Multiprocessing (SMP) mode, with and without frequencies linked. When frequencies are linked, cores have the same frequency at all time, i.e. changing a core's frequency also changes the frequency of all other cores on the same processor.

3.2 Task Model

A task is a entity executing sequential instructions. For real-time scheduling, one of the first task model was presented in [5]. In the classic task model, a task has a period, priority, deadline and execution time. Classic scheduling analysis techniques use the unitary Worst-Case Execution Time ($WCET$) for the task's execution time. Tasks do not always execute in their $WCET$ so in this case we say that the task has run in its Actual Execution Time (AET). The difference between $WCET$ and AET is called Slack Time (ST), i.e. $ST = WCET - AET$.

We emphasize, for our work, the importance to not to mistaken execution time with response time. All the execution times in our work are measured/computed by only considering time during which the task is effectively executing (e.g. no preemption time is taken into account).

The classic task model needs to be modified for the low-power scheduler presented in this paper and we define a new task called the "low-power task". In order to define the low-power task, let us first define a segment:

Definition 2 (Segment). *A segment S_i^j is a block of sequential instructions of a task τ_i. S_i^j is identified with j. S_i^j has the following properties:*

- *$WCET_i^j$: Worst-Case Execution Time of the segment.*
- *AET_i^j: Actual (unitary) Execution Time of the segment (observed during execution).*
- *ST_i^j: Slack Time of the segment.*
- *$MAET_i^j$: Maximum (unitary) Execution Time Allowed, i.e. the measured AET_i^j must not exceed this value [1].*

We call $(X_i^j \blacktriangleright F_{um})$, segment S_i^j's X_i^j property when S_i^j is executed on $CORE_m$ at frequency F_{um}.

The low-power task is then defined as:

Definition 3 (Low-Power Task). *A low-power task τ_i retains the period, priority and deadline properties of a task. In addition it has a control flow graph where nodes are segments S_i^j. When τ_i is run on $CORE_m$ at pair $(V,F)_{um}$, the pair is be designated by $(V,F)_{um}^i$.*

Figure 1 shows an example of a low-power task's control flow graph.

Through the control-flow model, it is possible to represent several execution paths for a low-power task and thus several execution times.

For the rest of the article, when a task is evoked, we talk about a low-power task.

3.3 Configuration

The entities presented in the previous two sections are regrouped into what are called software and platform configurations.

Definition 4 (Software Configuration). *A software configuration is a structure with the following properties:*

- *Application mode: The application mode determines the timing information in the system.*

[1] We do not define a deadline for a segment and a $MAET$ is not to be mistaken with a deadline. A segment's monitored AET only increases when the segment is executing, not during preemption. The $MAET$ is then this increasing unitary execution time's upper bound (not to be exceeded).

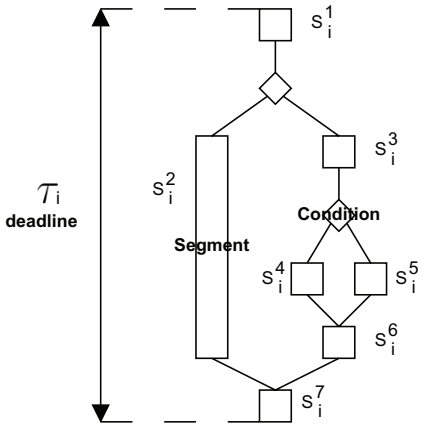

Figure 1: Low-Power Task τ_i's Control Flow Graph

- *Task sets: Sets of low-power tasks with their properties as defined in section 3.2.*

Several application modes may exist and for each mode there is an associated group of task sets. Each task set have timing information for a given frequency. A group of task sets have specific timing information for a mode. For example $MAETs$ can be increased when the application mode is a desired low Quality of Service (QoS).

Definition 5 (Platform Configuration). *A platform configuration is a structure with the following properties*

- *Processor and core set: A set of processors and cores with their (V, F) table.*
- *Task-processor affinity: Allocations of tasks on processors.*

From a platform configuration's task-processor affinity property, we see that we exclude task migration between cores on different processors.

4. LOW-POWER SCHEDULER

The low-power scheduler is a "power-aware" real-time scheduler. In this sense, its goal is not only to respect timing constraints in the system, but also limit power usage. The low-power scheduler is compatible with fixed and dynamic priority policies. It takes as input parameters a software and platform configuration.

In the following sections we first present the concept behind this scheduler, before showing its DVFS algorithm. Finally we will finish with an example.

4.1 Overview

Before the scheduler can be used, the task set at each frequency F_{um} must be assured to be schedulable. This is done by using scheduling analysis. Each task τ_i, composed of segments S_i^j, is assigned a $WCET_i$ that is the sum of all segment S_i^j's $WCET_i^j$ on the task's worst case execution path. If the task set is schedulable, it is possible to assign each segment S_i^j's $MAET_i^j$ to its $WCET_i^j$. If the task set at F_{um} is not schedulable, then F_{um} is discarded. This is done for each application mode.

The low-power scheduler relies on monitoring a task's AET. If a task's AET is smaller than its $WCET$, slack time is observed. As slack time is observed progressively, the frequency is scaled so the task's AET tends towards its $WCET$ (at maximum frequency) without exceeding its $MAET$. This can be achieved thanks to a task's division into segments. Each segment's frequency is set according to the last segment's slack time. Said differently, the last segment's slack time allows the current segment to have more time to achieve its instructions before its $MAET$. The example in Figure 2 illustrates the concept.

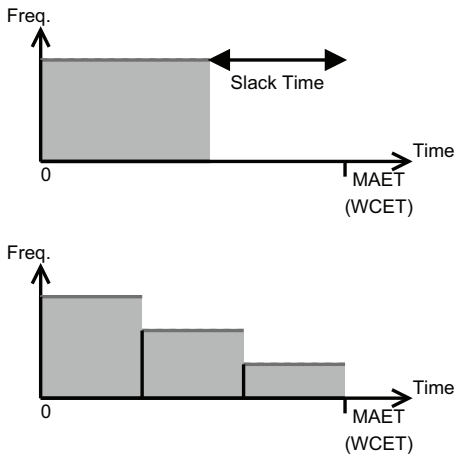

Figure 2: Frequency Scaling: In the graphs, y axis is the frequency, x axis is the time. Top graph is a task's execution at fixed frequency. Bottom graph is the execution of the same task, divided into three segments, with frequency scaled at each segment.

DVFS decisions are also taken according to the current application mode because it affects timing information (e.g. $MAET$). The mode can be changed "on-the-fly", which will result in new timing information as input parameters to the DVFS algorithm. The DVFS algorithm guarantees task deadlines when the application mode stays the same. When the application mode changes, a transient overload [7] may happen. It is up to the scheduling policy to handle the transient overload.

Decreasing the frequency cannot be done blindly without considering software and platform configurations. Handling these parameters is explained in the DVFS algorithm presented in the next section.

4.2 DVFS Algorithm

The low-power scheduler does DVFS at segment-level. The DVFS decision is taken each time a different segment is starting execution. This does not necessarily mean entering a new segment (e.g. it can be returning from preemption). There exist two kinds of decisions to take:

- Standard case: Entering a new segment that follows a segment of the same task and task instance.
- Context switch case: The previous segment executing on the same core belongs to a different task or task instance.

4.2.1 Standard case

In the standard case, the decision is taken when entering segment S_i^j of task τ_i running on $CORE_m$. Segment S_l^k is the last segment running before S_i^j. The DVFS decision is divided into the following steps:

1. Add previous segment S_l^k's slack time to current segment S_i^j's maximum allowed execution time. I.e. $MAET_i^j = MAET_i^j + ST_l^k$.
2. Depending on the platform's processor mode:
   - On a monocore, non-SMP, or frequencies non-linked, multicore platform: choose $CORE_m$'s lowest pair that won't make S_i^j exceed its maximum allowed execution time. I.e. choose $(V,F)_{um}$ so that $(WCET_i^j \blacktriangleright F_{um}) \leq MAET_i^j$.
   - On a SMP, with frequency linked, multicore platform (cores have the same (V,F) pair at all time): choose $CORE_m$'s lowest pair that won't make any segment $S_{i'}^{j'}$, running on any core $CORE_{m'}$, exceed its maximum allowed execution time. I.e. choose $(V,F)_{um}$ so that $(WCET_{i'}^{j'} \blacktriangleright F_{um}) \leq MAET_{i'}^{j'}$ for all segments.
3. Apply the new pair $(V,F)_{um}$ to $CORE_m$ and store it for task τ_i.

4.2.2 Context switch case

In the context-switch case, the decision is taken when segment S_i^j (of task τ_i running on $CORE_m$) starts executing. Segment S_l^k is the last segment running before S_i^j and it does not belong to the same task or the same task instance. There are three cases for this situation:

- Return from preemption: Segment S_i^j belongs to task τ_i, that was preempted by S_l^k of task τ_l. The core's pair is changed to what it was before preemption. I.e. $CORE_m$'s $(V,F)_{um}$ is set to the $(V,F)_{um}^i$ (stored for τ_i).
- Queued task release: Task τ_i is released and it was already queued (i.e. higher priority tasks were running before). Segment S_i^j is then τ_i's first segment. The frequency is modified like in the standard case, i.e. as if previous segment S_l^k belongs to the same task and instance as S_i^j. This way S_l^k's slack time is exploited.
- Other task release: Task τ_i is released and it was not queued. Segment S_i^j is still τ_i's first segment. Apply maximum pair $(V,F)_{um}$ to $CORE_m$ and store it for task τ_i. I.e. $(V,F)_{um}^i = (V,F)_{um}$.

4.3 Example

Figure 3 is an example of a schedule produced by the low-power scheduler.

In this example, tasks τ_1, τ_2, and τ_3 run on a processor with frequency modes 150, 200 and 300Mhz. Priorities are ordered as: $P_1 > P_2 > P_3$. Task τ_2 has 2 segments S_2^1 and S_2^2. Tasks τ_1 and τ_3 have one segment each, S_1^1 and S_3^1 respectively. The processor starts at 300Mhz. The frequency changes, in chronological order, are the following:

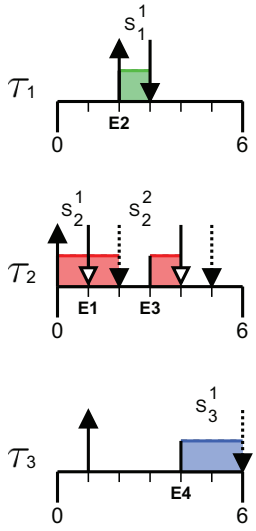

Figure 3: DVFS Example: Vertical up filled arrows are task releases. Vertical down dotted arrows are segment absolute MAETs. Vertical down hollow arrows are segment absolute AETs. When vertical down arrows overlap, AET = MAET. Text below a timeline represents a frequency change.

E1 200Mhz, canonical case: S_2^1 finished at $AET_2^1 = 1$, i.e. 1 time unit earlier than its maximum allowed execution time $MAET_2^1 = 2$. The next segment S_2^2 uses the slack time $ST_2^1 = 1$. We then have $MAET_2^2 = 3$, i.e. it is increased by 1/3. The processor can be set to 2/3 speed, i.e. 200Mhz.

E2 300Mhz, other task release: The processor is set to maximum frequency 300Mhz because a non-queued task (τ_3) is released.

E3 200Mhz, return from preemption: The processor goes back to stored frequency 200Mhz for τ_2.

E4 100Mhz, queued task release: τ_3 was queued and released after τ_2 finished execution. S_2^2 finished earlier so there is slack time. After computation we have $MAET_3^1 = 2$ so the processor is set to 1/2, i.e. 150Mhz.

5. LINUX IMPLEMENTATION

In order to integrate the DVFS algorithm into a Linux OS running on a SMP board, the "low-power framework" was developed. Figure 4 shows the architecture of a system with the low-power framework integrated.

The modules in the framework can either be implemented in kernel space or user space. When implemented in kernel space, the low-power scheduler can be used for hard real-time applications. When implemented in user space, it should only be used for soft real-time applications. Implementation in user space has the interoperability advantage, as the framework can be simply integrated with operating systems sharing the POSIX interfaces. In this paper we made the choice to implement in user space.

The architecture shown in Figure 4 is divided into several layers. In the following sections we will describe the modules in each layer.

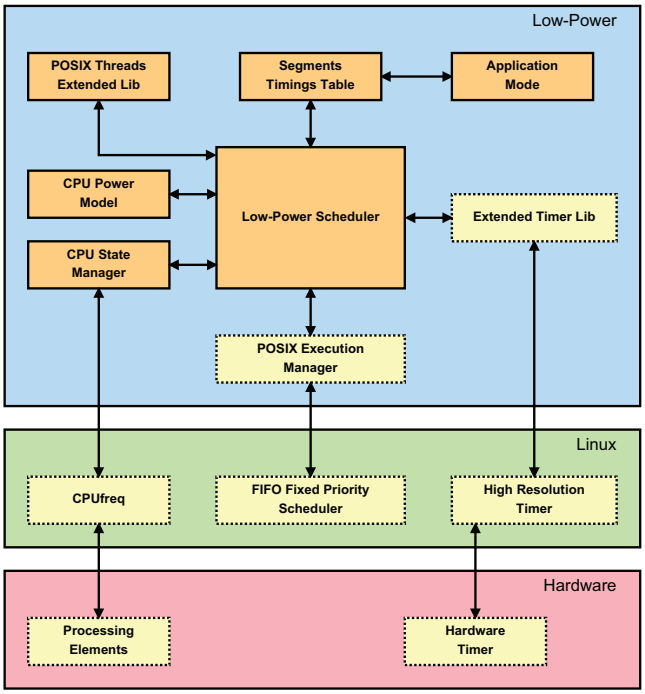

Figure 4: Low-Power Architecture: Solid line modules are part of the low-power framework. Arrows indicate interactions between modules.

5.1 Linux and Hardware Layer

The hardware layer is composed of the following modules:

- `Processing Elements`: Hardware cores and processors with different voltage/frequency pairs.
- `Hardware Timer`: High precision hardware timer.

The Linux OS layer is composed of the following modules:

- `FIFO Fixed Priority Scheduler`: The Linux fixed priority SCHED_FIFO scheduler that schedules POSIX threads.
- `High Resolution Timer`: A high resolution timer driver that is able to count the elapsed time by using the `hardware timer`.
- `CPUfreq`: A DVFS driver that is able to change the hardware `processing elements`' frequency/voltage pairs.

5.2 Low-Power Layer

The application sees the low-power layer as the scheduling modules in an OS. The `low-power scheduler` is the main scheduler in the system and it uses the Linux `FIFO fixed priority scheduler` as a slave to achieve its scheduling requests. This is why translations are needed between this layer and the real Linux OS layer.

The low-power layer is composed of a number of database modules:

- `Segments Timings Table`: This module contains all timing information on the segments in the system.
- `Application Mode`: This module contains the application's modes. It can send a mode change request to the `segments timings table` so the current timings table to use is updated.
- `CPU Power Model`: This module is used to store an abstract model of the processor and cores. The model includes the cores' voltage/frequency pairs.
- `POSIX Thread Extended Lib`: This module is used to create low-power tasks that extend the POSIX threads. When a thread is created in the application, it uses this library instead of the POSIX library.

The layer also contains modules that use the information contained in the database modules, and interact with the Linux OS modules:

- `The CPU State Manager`: This module can send frequency/voltage pair change requests to `CPUfreq`.
- `Extended Timer Lib`: This module translates time given by the Linux `high resolution timer` to time understandable by the `low-power scheduler`.
- `POSIX Execution Manager`: This module translates scheduling requests taken by the `low-power scheduler` on low-power tasks, to scheduling requests taken by the Linux `FIFO fixed priority scheduler` on POSIX threads. It then sends the scheduling requests to the Linux scheduler. For example if a dynamic priority policy is run, modifying a low-power task's dynamic priority translates to modifying the fixed priority of its matching POSIX thread.
- `Low-power Scheduler`: This module schedules the low-power tasks, defined in the `POSIX thread extended lib`. It runs a scheduling policy and handles synchronization (e.g. semaphore, mutex). When a scheduling decision is taken, it sends the scheduling request to the `POSIX execution manager`. When a DVFS decision is taken, it sends the frequency/voltage pair change request to the `CPU state manager`.

6. EXPERIMENT

The low-power scheduler's proof of concept prototype was tested on a multimedia domain application running on a multicore platform. To evaluate the scheduler's performance, we decided to focus on two metrics: the energy consumption gains and the overheads from the low-power scheduler.

In the following sections we first present the case-study application and platform. We then expose our measured results and we discuss the architecture choice for the Linux implementation.

6.1 Case Study

The case-study application is a multitask H.264 decoder. An image arrives periodically every 40ms in the decoder's buffer. An image is composed of 2 slices that both need to be decoded before the image is decoded. Each slice is decoded by a task, i.e. there are 2 decoding tasks. A slice is either an I-type slice or a P-type slice. I-type slices take longer to decode than P-type slices. The decoding of both slices must finish before 40ms, i.e. before the next image arrives. Figure 5 sums up the H.264 decoder.

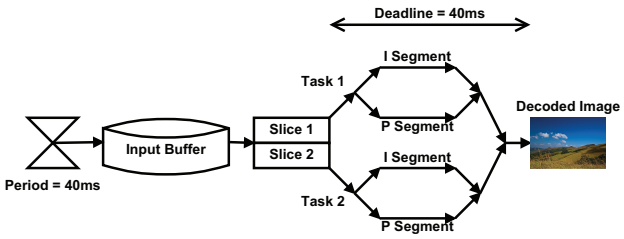

Figure 5: H.264 Decoder Application

The H.264 decoder tasks run on Cortex A8 cores with 5 frequency modes: 600, 550, 500, 250 and 125Mhz. Switching between frequencies takes between 150 and 250μs. The low-power scheduler runs an Earliest Deadline First [5] policy. The low-power framework was implemented in user space above a common Ubuntu Linux.

6.2 Results

After running the application, we observed that the low-power scheduler adds an overhead of 11% to the total execution time. The overheads include the DVFS algorithm's extra operations and the frequency switch overheads. At frequencies 250 and 125Mhz, task deadlines were missed when the application executed at these frequencies alone. Table 1 shows the measured energy consumption when using DVFS and at fixed frequencies. The measured consumption is that of the core and the L1 and L2 caches.

Table 1: Energy Consumption Gain: In the fourth column, energy consumption gain considering overheads, is in parenthesis.

Freq. (Mhz)	Cons. (J)	Gain (J)	Gain (%)
DVFS	1.108	/	/
600	1.579	0.471	30 (27)
550	1.362	0.254	19 (17)
500	1.184	0.076	6 (5)

The overheads depend on the number of segments added in the code. In the general case, more segments will result in more energy gain but more overheads. The difficulty of using the low-power scheduler thus comes in defining the instrumentation strategy.

The overheads are not only due to operations in the DVFS algorithm. Extra overhead is added due to the low-power scheduler's implementation as an user space layer above the Linux OS layer. Furthermore we noticed that since the low-power scheduler is implemented in the user-space, it is not possible to use it with low-power tasks running on different processes. This is due to the fact that the POSIX threads - corresponding to the low-power tasks - cannot communicate with the low-power scheduler if they are on a different process' address space. On the other hand the implementation favors interoperability because it can run on any Linux kernel with POSIX interfaces.

7. CONCLUSION

A multicore-supported low-power scheduler was presented in this paper. This scheduler has a DVFS algorithm based on an intra-task intrusive approach and exploits slack times when the system is executing. A low-power framework was developed to integrate the low-power scheduler as an user space layer above the common Ubuntu Linux OS kernel. This is to favor interoperability between Linux kernels using POSIX interfaces. This implementation was evaluated on a H.264 multitask decoder. Results show energy consumption gains up to 27%.

In the future we would like to evaluate the low-power scheduler on a software radio application (telecommunication domain) with more complex control flows. We would also like to port the low-power scheduler to RTOS other than Linux and evaluate its performance, especially with a kernel space implementation.

8. ACKNOWLEDGMENTS

This work is performed in the framework of the FP7 funded European project PHARAON.

9. REFERENCES

[1] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Proceedings of the 2002 conference on Design, automation and test in Europe*, page 168. IEEE Computer Society, 2002.

[2] T. Burd and R. Brodersen. Energy efficient CMOS microprocessor design. In *Proceedings of the 28th Hawaii International Conference on System Sciences*, pages 288–297. IEEE Comput. Soc. Press, 1995.

[3] S. Ghiasi, J. Casmira, and D. Grunwald. Using IPC variation in workloads with externally specified rates to reduce power consumption. In *Proceedings of the 2000 Workshop on Complexity Effective Design*, 2000.

[4] A. Hung, W. Bishop, and A. Kennings. Symmetric multiprocessing on programmable chips made easy. In *Proceedings of the 2005 Design Automation and Test in Europe Conference*, pages 240–245. IEEE, 2005.

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

[6] P. Pillai and K. G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. *ACM SIGOPS Operating Systems Review*, 35(5):89, 2001.

[7] L. Sha, T. Abdelzaher, K.-E. Arzen, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[8] D. Shin, J. Kim, and S. Lee. Intra-task voltage scheduling for low-energy hard real-time applications. *IEEE Design & Test of Computers*, 18(2):20–30, 2001.

[9] A. Weissel and F. Bellosa. Process cruise control - event-driven clock scaling for dynamic power management. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems*, page 238. ACM Press, 2002.

[10] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *Proceedings of the 36th Annual Symposium on Foundations of Computer Science*, pages 374–382, CA, USA, 1995. IEEE Comput. Soc. Press.