EURASIP Journal on Embedded Systems
a SpringerOpen Journal

RESEARCH Open Access

Accurate energy characterization of OS services in embedded systems

Bassem Ouni[1*], Cécile Belleudy[1] and Eric Senn[2]

Abstract

As technology scales for increased circuit density and performance, the management of power consumption in embedded systems is becoming critical. Because the operating system (OS) is a basic component of the embedded system, the reduction and characterization of its energy consumption is a main challenge for the designers. In this work, a flow of low power OS energy characterization is introduced. The variation of the energy and power consumption of the embedded OS services is studied. The remainder of this article details the methods used to determine energy and power overheads of a set of basic services of the embedded OS: scheduling, context switch and inter-process communication. The impacts of hardware and software parameters like processor frequency and scheduling policy on the energy consumption are analyzed. Also, models and laws of the power and energy are extracted. Then, to quantify the low power OS energetic overhead, the obtained models are integrated in the system level design. Our method allows estimating the energy consumption of the low power OS services when running an application on a specific hardware platform.

Introduction

Nowadays energy consumption in embedded systems has become one of the key challenges for the software and hardware designers. The embedded operating system (OS) serves as an interface between the application software and the hardware. It is an important software component in many embedded system applications since it drives the exploitation of the hardware platform by offering a wide variety of services: task management, scheduling, inter-process communication (IPC), timer services, I/O operations and memory management. Also, the embedded OS manages the overall power consumption of the embedded system components. It includes many power management policies aiming at keeping components into lower power states, thereby reducing energy consumption.

The embedded systems become so complex as they contain various hardware devices and software application which interacts with the users to handle the system. The complexity of the hardware and software layers necessitates the use of a specific support allowing application to exploit efficiently the hardware platform. This support is the embedded OS; it includes libraries and device drivers and offers a wide variety of services. Figure 1 shows the disposition of embedded systems' different layers. The application is represented by a set of tasks $\{T_i, 1 \leq i \leq n\}$.

Under the French research program Open-PEOPLE project [1], we aim at characterizing and optimizing the energy consumption of the embedded OS services. In this article, variation of the scheduling routines, IPC and context switch energy consumption as a function of hardware and software parameters is studied. The remainder of this article is organized as follows: Related works are described in Section "Related works". Energy characterization and estimation flow is presented in Section "Energy characterization and estimation flow". Section "Hardware platform" introduces the hardware platform and explains the measurement setup for measuring the energy consumption. Then, Sections "Embedded OS power and energy models" and "Experimental results" explains our methodology to characterize the embedded OS services energy overhead and describes experimental results and derived models. Section "Embedded OS services models integration in the system level design flow" shows the low power OS services's models integration in the system level

*Correspondence: Bassem.Ouni@unice.fr
[1] University of Nice Sophia-Antipolis, LEAT CNRS, 250 rue Albert Einstein, bât 4 - 06560 Valbonne - Cedex, France
Full list of author information is available at the end of the article

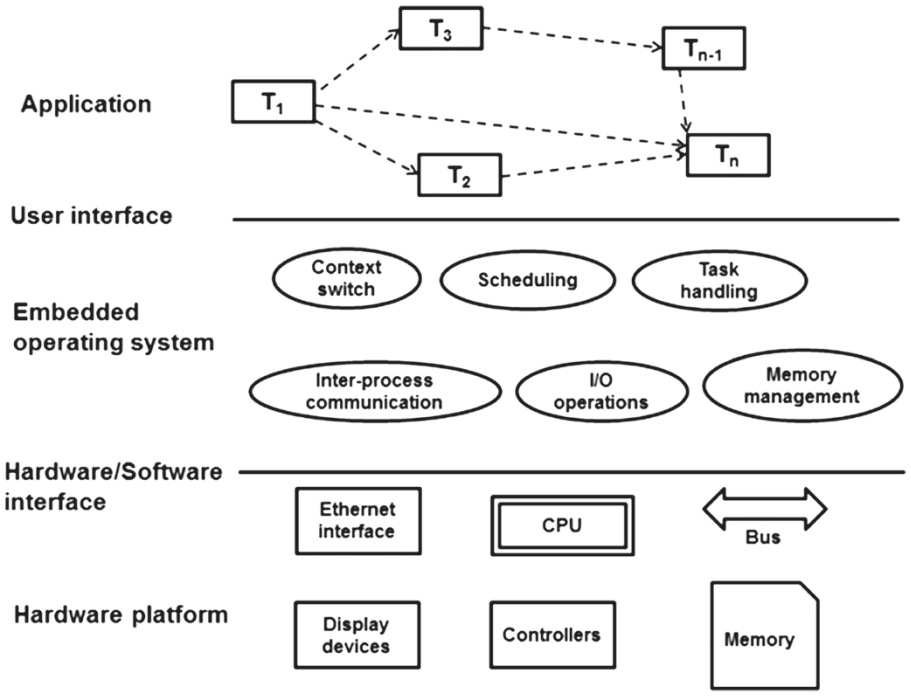

Figure 1 The different layers of an embedded system.

design flow. Finally, Section "Conclusion" concludes the article and proposes the future work.

Related works

In order to characterize energy and power overhead of embedded OSs, several studies have proposed evaluating embedded OS energy consumption at different abstraction levels.

Li and John [2] introduced a routine level power model. According to them, the elementary unit is the OS service routine. So, they consider the energy consumed by the OS services as the sum of those consumed by each routine. They proposed a model of power consumption based on the correlation that they found between the power and the instruction per cycle (IPC) metric.

Acquaviva et al. [3] proposed a new methodology to characterize the OS energy overhead. They measured the energy consumption of the *eCos* real time OS (RTOS) running on a prototype wearable computer, HP's *SmartBadgeIII*. Then, they studied the energy impact of the RTOS both at the kernel and at the I/O driver level and determine the key parameters affecting the energy consumption. This work studied the relation between the power and performance of the OS services and the CPU clock frequency. Acquaviva et al. perform analysis but they did not model the energy consumption of the OS services and drivers.

Tan et al. [4] modeled the OSs energy consumption at the kernel level. They classify the energy into two groups: the explicit energy which is related directly to the OS primitives and the implicit energy which results from the running of the OS engine. The authors explained their approaches to measure these classes of energy and they proposed energy consumption macro models. Then, Tan et al. validated their methodology on two embedded OSs, μCOS and Linux OS. However, the scope of the proposed work in [4] is limited in some ways as it targets the OS's running on a single processor. Also, the authors do not consider the I/O drivers in the proposed energy consumption model.

Dick et al. [5] analyzed the power consumption of the μCOS OS which is running several embedded applications on a *Fujitsi SPARClite* processor based embedded system. The authors demonstrated that the OS functions have an important impact on the total energy consumption. This impact depends on the complexity of the applications. The presented work represents only an analysis of OS power consumption. Dick et al. did not propose an energy consumption model.

Baynes et al. [6] described their simulation environment, *Simbed*, which evaluates the performance and energy consumption of the RTOS and embedded applications. The authors compared three different RTOS's: μCOS, *Echidna* and *NOS*. They found that the OS overhead depends on the applications. It is so high for

the lightweight applications and diminishes for more compute-intensive applications. Nevertheless, Baynes et al. presented high level energy measurements (simulation), the extracted models are not realistic because they are not deduced from measurements on actual hardware platform. Also, the energy consumption of OS services compared with the total application energy consumption was not calculated. Guo et al. [7] proposed a novel approach using hopfield neural network to solve the problem of RTOS power partitioning; they target to optimally allocate the RTOS's behavior to the hardware/software system. They defined a new energy function for this kind of neural network and some considerations on the state updating rule. The obtained simulation results show that the proposed method can perform energy saving up to 60%. This work does not consider energy macro-modeling and RTOS services. Zhao et al. [8] propose a new approach to estimate and optimize the energy consumption of the embedded OS and the applications at a fine-grained level. The work is based on power model and a new estimation model for OS energy consumption. Zhao et al. demonstrate that the approach can characterize and optimize the energy consumption of fine-grained software components. Fournel et al. [9] present a performance and energy consumption simulator for embedded system executing an application code. This work allows designers to get fast performance and consumption estimations without deploying software on target hardware, while being independent of any compilation tools or software components such as network protocols or OSs.

Fei et al. [10] interest in reducing the energy consumption of the OS-driven multi-process embedded software programs by transforming its source code. They minimize the energy consumed in the execution of OS functions and services. The authors propose four types of transformations, namely process-level concurrency management, message vectorization, computation migration and IPC mechanism selection. Fei et al. evaluate the applicability of theses techniques in the context of an embedded system containing an Intel StrongARM processor and embedded Linux OS. They manage process-level concurrency through process merging to save context switch overhead and IPCs. They modify the process interface by vectorizing the communications between processes and selecting an energy-efficient IPC mechanism. This work attempt to relocate computations from one process to another so as to reduce the number and data volume of IPCs. These transformations provide complementary optimization strategies to traditional compiler optimizations for energy savings. Dhouib et al. [11] propose a multi-layer approach to estimate the energy consumption of embedded OSs. The authors start by estimating energy and power consumption of standalone tasks. Then they add energy overheads of the OS services which are timer interrupts, IPC and peripheral device accesses. They validate the multi-layer approach by estimating the energy consumption of an M-JPEG encoder running on linux 2.6 and deployed on a XUP Virtex-II pro development board. In the recent works, low power OSs, which are the embedded OS allowing the binding of the application on hardware platform adapting low power techniques, are not considered. It is not mentioned what are the processor capabilities and which low power policy is used. In fact, the energy consumption of OS services depends on low power policy used. For example, we will see in this article that the context switch energy overhead is important due to the use of specific technique, aiming at reducing the power consumption, that stimulates this service. Creative chip designers have come up with a variety of methods and techniques to reduce power without impacting system performance. For instance, the embedded system used in this work, OMAP35x EVM board, has three basic methods to reduce the power consumption: dynamic voltage and frequency scaling (DVFS), adaptive voltage scaling (AVS) and dynamic power switching (DPS). This article relies on studying the energy overhead of embedded OS adapting DVFS technique.

Energy characterization and estimation flow

Power and energy consumption are important performance metrics for embedded systems. In electrical circuits, the power is the product of the potential difference (voltage) and the electric current. Formally, the energy consumed by a system is the amount of power dissipated during a certain period of time. For instance, if a task T occupies a processor during an execution interval of [a,b] then the energy consumed by the processor E_T during this time interval is given by this Equation (1):

$$E_T = \int_a^b P(t)\,dt \tag{1}$$

The proposed method consists in extracting an energy model of the OS power overhead. The inputs are the embedded OS, the application, and the hardware platform. As showed in Figure 2, to characterize the energy consumed by the services of the embedded OS, a set of benchmarks, which are test programs that stimulate each service separately, are implemented. These programs are compiled and linked to the OS. In the energy analysis step, a set of parameters are varied: hardware and software parameters which influence the energy consumption are identified then energy profiles are traced. The energy traces obtained are able to characterize the energy overhead of the OS services and then to model the power and energy consumption. After extracting the energy models, we determine the energy and power laws, as showed in Figure 3.

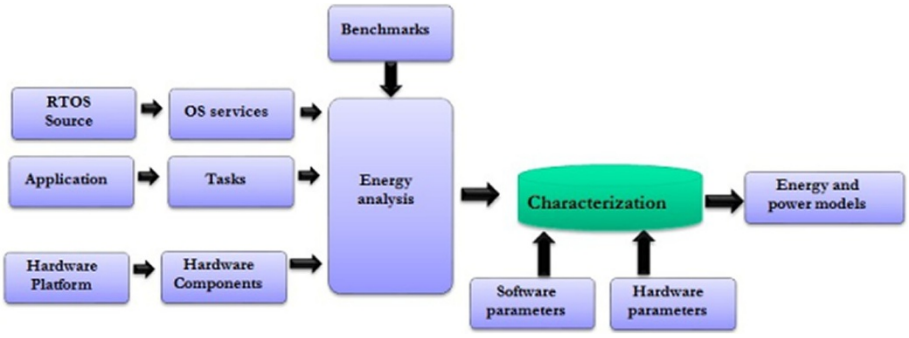

Figure 2 The methodology of OS energy characterization.

We focus on the correlation between the energy consumed by the application and the services of the OS. The energy link proposed in the model is depicted by Equation (2).

$$\forall T_i, E_{T_i} = E_{\text{intertask}} + \left(\sum_{1 \le j \le p} \delta_{i,j} \times E_{S_j} \right) \quad (2)$$

Where E_{T_i} represents the energy consumed by the task T_i, $E_{\text{intertask}}$ is the energy consumed by the task routines and operations, p is the number of services used by the task T_i, $\delta_{i,j}$ is energy consumption rate of the task T_i using the service S_j and E_{S_j} the energy consumption of the service S_j.

We consider t the total number of the OS services, x_j the number of the parameters that influence E_{S_j} the energy consumption of the service $S_j, 1 \le j \le t$.

The set of parameters appropriate to the energy overhead of the service S_j is $\{ Param_{j,k}, 1 \le j \le t, 1 \le k \le x_j \}$. The function f_k describes the variation of E_{S_j} with $Param_{j,k}$. We compute the energy consumption of the service S_j following this Equation (3).

$$E(S_j) = \sum_{1 \le k \le x_j} f_k(\text{Param}_{j,k}) \quad (3)$$

According to the amount of energy consumed by each service when executing the application, we classify the OS's energy overhead into two groups: a basic energy of the services which consume an amount of energy bigger than an energy threshold E_{th}, this threshold is the average energy consumed by the different OS services. The remaining services are the secondary services. The set of basic and secondary services are respectively $\{\text{BS}_j, 1 \le j \le p\}$ and $\{\text{SS}_k, 1 \le k \le q\}$ where p and q are respectively the number of basic and secondary services. This expression (4) depicts the energy consumed by the OS E_{OS} when running a task T_i. The Equation (5) verifies whether the totality of the energy consumed by the OS when running the application, having n tasks, is well distributed between the services.

$$E_{\text{OS}} = \left(\sum_{1 \le j \le p} \alpha_{i,j} \times E_{\text{BS}_j} \right) + \left(\sum_{1 \le k \le q} \beta_{i,k} \times E_{\text{SS}_k} \right) \quad (4)$$

Where

$$\sum_{1 \le i \le n} \left(\sum_{1 \le j \le p} \alpha_{i,j} + \sum_{1 \le k \le q} \beta_{i,k} \right) = 100\% \quad (5)$$

$\alpha_{i,j}$: energy consumption rate of the task T_i using the service BS_j. $\beta_{i,k}$: energy consumption rate of the task T_i using the service SS_k. E_{BS_j} and E_{SS_k} represent respectively the energy consumed by the service BS_j and SS_k.

In the next section, we will explain the approach used to characterize the embedded OS services energy overhead

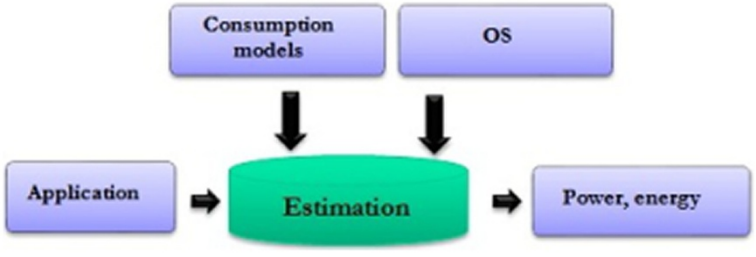

Figure 3 Estimation of the OS energy consumption.

and its variation with hardware and software parameters. Next section describes the hardware platform used and details the energy measurement setup.

Hardware platform

The target hardware platform is the OMAP35x EVM board from MISTRAL (TEXAS INSTRUMENTS). It is equipped with the OMAP 3530 processor [12], an advanced superscalar ARM Cortex-A8 RISC Core. The embedded OS used is Linux which is supported for use with the OMAP35x EVM. As shown in Figure 4, in The measurement platform includes a dedicated server to configure the OMAP EVM and to control the energy consumption measurements on this board. It consists of a computer wire-connected to the board. We use the DHCP protocol to obtain an IP address of the board from the server. The TFTP (Trivial File Transfer Protocol) and NFS (Network File System) protocols are used to load and boot the OS image from the server and through the Ethernet. The test programs are executed on the hardware platform and the energy dissipated by the processor is determined: The voltage drop V_{drop} across a jumper J6 pins connected in series with the OMAP 3530 processor is measured. Then, the current consumed is calculated after dividing V_{drop} by a shunt resistance R in parallel with the jumper pins.

Embedded OS power and energy models

In this section, embedded OS services energy characterization approaches are introduced, three important services are studied: the scheduling, the context switch and IPC.

The scheduling routines

Scheduling routines and operations could generate power overhead on the processor and/or memory components. They are considered as system calls and only consist in switching the processor from unprivileged user mode to a privileged kernel model. To quantify power and energy overhead of embedded OS scheduler routines and operations, we have to build test programs containing threads with different priorities, we measure in a first step the average energy consumed by the standalone tasks without scheduling routines, and then with scheduling routines.

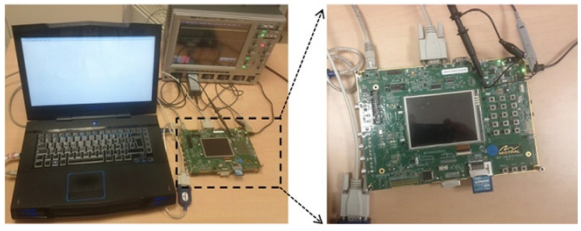

Figure 4 The measurement framework.

$E_{\text{Scheduling}}$ represents the energy consumed by the scheduling operations. It is calculated as showed in this Equation (6):

$$E_{\text{Scheduling}} = E_{\text{withsch}} - E_{\text{withoutsch}} \tag{6}$$

Where E_{withsch} and $E_{\text{withoutsch}}$ represent respectively the energy consumed by the benchmarks with scheduling routines and without scheduling routines.

We vary several parameters when running the test programs. The applicative parameter that we can change is the scheduling policy. We also modify the processor frequency as a hardware parameter. We are interested in studying the influence of three scheduling policies: *SCHED_FIFO*, *SCHED_RR* and *SCHED_OTHER*.

SCHED_FIFO policy is used with static priorities higher than 0, it is a scheduling algorithm without time slicing. Under this policy, a process which is preempted by another process having higher priority will stay at the head of the list for its priority and will resume execution as soon as all processes of higher priority are blocked again. If there are two *SCHED_FIFO* processes having the same priority, the process which is running will continue its execution until it decides to give the processor up. The process having the highest priority will use the processor as long as it needs it.

SCHED_RR policy enhances the *SCHED_FIFO* one; hence, everything described above for *SCHED_FIFO* also applies to *SCHED_RR* except that each process is only allowed to run for a maximum time called quantum. If a *SCHED_RR* process has been running for a time period equal to or greater than the time quantum, it will be put at the end of the priority list. A *SCHED_RR* process that has been preempted by a higher priority process subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum.

SCHED_OTHER policy is only used at static priority 0. To ensure a fair progress among the processes, the *SCHED_OTHER* scheduler elects a process to run from the static priority 0 list based on a dynamic priority that is determined only inside this list. The dynamic priority is based on the nice level and increased for each time quantum the process is ready to run, but denied to run by the scheduler.

Figure 5 shows the evolution of the power overhead of the scheduler routines $P_{\text{Scheduling}}$ over the scheduling policy. We can see that the energy consumed when we use *SCHED_OTHER* is important compared to *SCHED_FIFO* and *SCHED_RR*. This is due to the additional operations (nice or setpriority() system calls) used when the scheduler *SCHED_OTHER* calculates and increases the dynamic priority for each time quantum. $P_{\text{Scheduling}}$

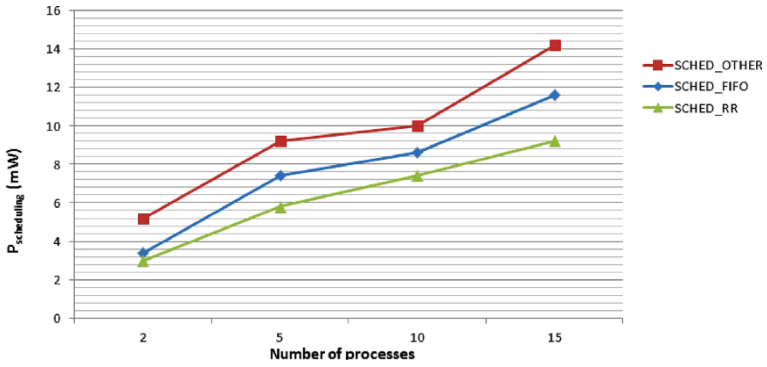

Figure 5 Scheduling routines power consumption versus the number of processes for different scheduling policies.

increases with the rise of the number of processes, this is due to the increase of the scheduler of routines, such as the assignment of the priorities, with the number of processes.

Figure 6 depicts the variation of measured and estimated scheduling routines energy consumption with processor frequency. The scheduling policy is *SCHED_OTHER* and the number of processes is 10. The energy consumption law for the scheduling routines is depicted by Equation (7). The average estimation error is around 0.355%.

The obtained results are explained by considering that the energy is the product between the average power and the total execution time. If we consider that the steady state current (and hence the power) profile obtained when running this experiment is almost flat since the processor does not access the external bus, the energy cost of the scheduler is proportional to the execution time of the scheduling routines which decrease with the increase of the frequency.

$$E_{\text{Scheduling}}(f) = (-59.649 \times 10^{-3} \times f) + (3.106 \times 10^{2}) \quad (7)$$

The context switch

The context switch is a mechanism which occurs when the kernel changes the control of the processor from an executing process to another that is ready to run. The kernel saves the state of current process including the processor register values and other data that describes this state. Then, it loads the saved state of the new process for execution.

In the majority of recent work presented previously, the authors do not take into account the energy and time overheads of this service when studying the energy consumption of the OSs. They include it with the scheduling service, but the two services are distinct. Actually, in the embedded systems, the processor has two operating modes: the kernel mode and user mode. The processes running on kernel and user mode are called kernel and user processes respectively. The user process runs in a memory space which can be swapped out when

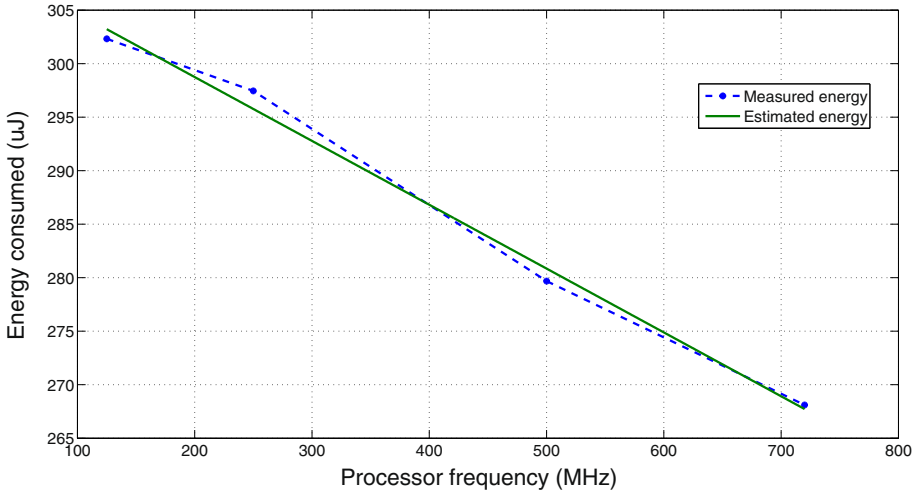

Figure 6 Scheduling routines energy variation as a function of CPU frequency.

necessary. When the processor needs the user process to execute a kernel code, the process become in kernel mode with administrative privileges. In this case, the processor has no restrictions while executing the instructions and will access to key system resources. Once the kernel process finishes its workload, it returns to the initial state as a user process. The scheduler switches the processor from the user mode to a kernel mode via system calls; this mechanism is named the mode switch. Unlike the mode switch, the context switch consists in switching the processor from one process to another.

Context switching introduces direct and indirect overheads [13]. Direct context switch overheads include saving and restoring processor registers, flushing the processor pipeline, and executing the OS scheduler. Indirect overheads involve the switch of the address translation maps used by the processor when the threads have different virtual address spaces. This switch perturbs the TLB (CPU cache that memory management hardware unit uses to improve virtual address translation speed) states. Also, the indirect context switch includes the perturbation of the processor's caches. In fact when a thread T_1 is switched out and a new thread T_2 starts the execution, the cache state of T_1 is perturbed and some cache blocks are replaced. So, when T_1 resumes the execution and restores the cache state, it gets a cache misses. Besides, the OS memory paging represents a source of the indirect overhead since the context switch can occur in a memory page moved to the disk when there is no free memory. Prior research has shown that indirect context switch overheads [14], mainly the cache perturbation effect, are significantly larger than direct overheads.

To characterize the energy consumption of the context switch, we create a set of threads in a multitasking environment using the POSIX standard. As depicted in Figures 7 and 8, the test-bench consists in creating two threads P1 and P2 and generating a number of context-switches as detailed in our recent work [15]. In fact in step 1, only one context switch is generated and in step n, n context switches are generated. In the remainder of this article, *Tcs* represents the time of the context switch, $S_{i,j}$ the j-th section of the process P_i and $T_{i,j}$ is the execution time of the section $S_{i,j}$.

The total execution time of the benchmark in step 1 and step n are, respectively T_{step1} and T_{stepn}. They are depicted by these Equations (8) and (9):

$$T\text{Step } 1 = T\text{exec}_1 + T\text{cs} + T\text{exec}_2 \tag{8}$$

$$T\text{Stepn} = \sum_{1\le i\le p} T_{1,i} + \sum_{1\le j\le q} T_{2,j} + (n \times T\text{cs}) \tag{9}$$

Where p and q represents respectively the number of sections of P_1 and P_2. The context switch time *Tcs* and the context switch power overhead *Pcs* are calculated following these Equations (10) and (11):

$$Tcs = (T_{\text{stepn}} - T_{\text{step1}})/(n-1) \tag{10}$$

$$Pcs = (P_{\text{stepn}} - P_{\text{step1}})/(n-1) \tag{11}$$

The context switch energy overhead is computed as (12):

$$Ecs = ((P_{\text{stepn}} * T_{\text{stepn}}) - (P_{\text{step1}} * T_{\text{step1}}))/(n-1) \tag{12}$$

Where P_{step1} and P_{stepn} are, respectively the average power consumption of the benchmarks in step 1 and step n.

We execute the test programs following the characterization approach. Then, we vary the scheduling policy and the frequency, we note the power and performance variations and we extract energy models.

The scheduling policy impact on the context switch overhead

In our experiments, the scheduling policy and the number of context switches are varied and the energy consumed

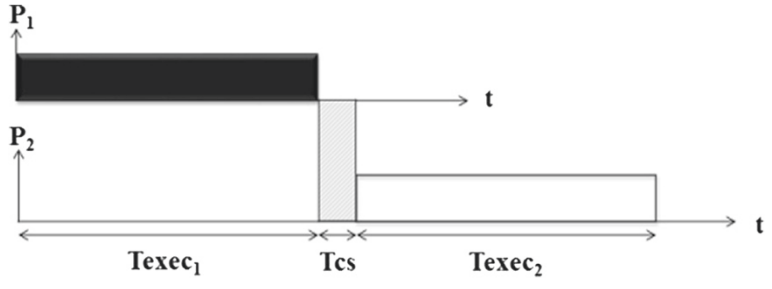

Figure 7 Step 1.

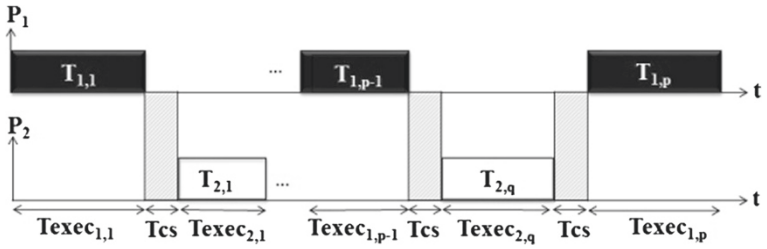

Figure 8 Step n.

is measured when a context switch occurred. The variation of the energy dissipated according to the number of context switches and the scheduling policy is presented in Figure 9. This figure compares the decrease of the context switch energy overhead for the two processes by varying the number of context switches.

It is noted that the context switch energy overhead decreases with the increase of the number of context switches. In fact, to switch from one process to another, the state of each process should be saved in a data structure named process control block (*PCB*). The energy overhead of the creation of the PCB is accounted with the context switch energy overhead and is divided between the context switches so that if the number of context switches increases the average Ecs per context switching decreases. Also, when the scheduling policy used is *SCHED_FIFO*, the context switch energy overhead is more important than the energy for the *SCHED_RR* scheduling policy. In fact, under the round robin scheduling policy, the processor assigns time slices (quantum) to each process. So, before the context switches that we generate, there is another context switches that occurred automatically due to the expiration of the quantum of the process P1. Consequently, the PCB is created during the automatic context switch. The energy overhead of the PCB creation is not accounted with the energy of the context switch that we generate: Ecs. But, under the FIFO scheduling policy, the processor does not switch automatically from the process P1 to P2 only if P1 terminates its execution so that the energy overhead of the PCB creation is accounted with Ecs.

We note that *SCHED_OTHER* processes are non real time processes, but, *SCHED_RR* and *SCHED_FIFO* are real time processes. So, *SCHED_RR* and *SCHED_FIFO* processes need more memory than *SCHED_OTHER* processes to save the processor registers because they execute more operations and calculations in order to respect the real time constraints so that they consume more time to change the context. Then, the context switch of the *SCHED_OTHER* processes consume less energy than the *SCHED_RR* and *SCHED_FIFO* ones.

The processor frequency impact on the context switch overhead

In this section, the impact of processor frequency on the context switch overhead for static and dynamic frequency cases is discussed.

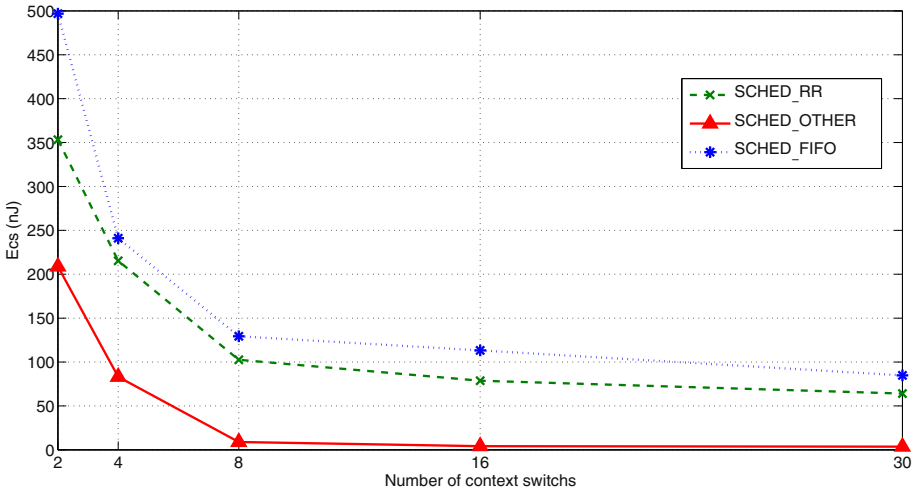

Figure 9 Context switch energy consumption versus the number of context switching for different scheduling policies.

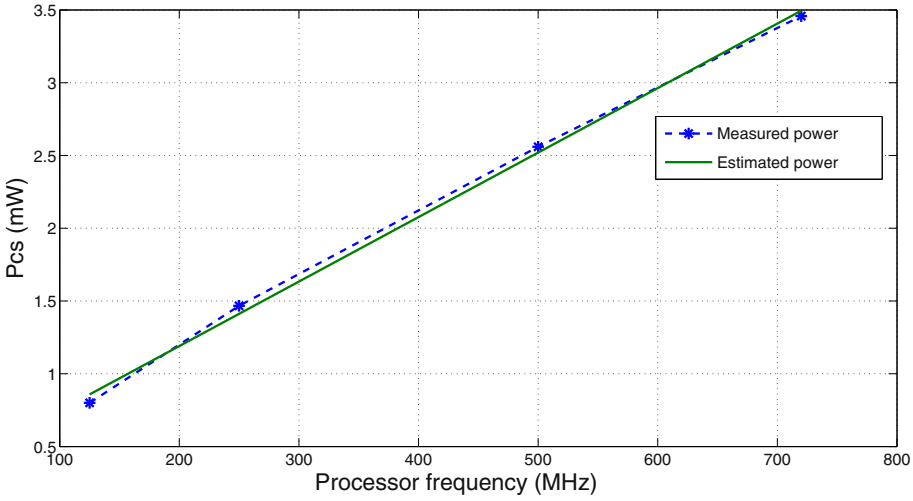

Figure 10 Context switch power variation as a function of CPU frequency.

Static frequency case

For this experiment, the scheduling policy and the number of context switches are fixed. The benchmarks of step 1 and step n with a static frequency are executed. The CPU frequency is varied afterwards and the benchmarks are re-executed.

In complementary metal-oxide semiconductor (CMOS) technology-based systems, there are two principle sources of power dissipation: dynamic power dissipation, which arises from the repeated capacitance charge and discharge on the output of the hundreds of millions of gates in modern chips, it depends on the processor frequency, and static power dissipation which arises from the electric current that leaks through transistors even when they are turned off. The hardware platform used in the current work reduces standby power consumption by reducing power leakage so that static power is negligible compared to the dynamic power. Figure 10 plots the measured and estimated context switch power, mainly the dynamic power, overhead as a function of the frequency. Context switch power variation with processor frequency follows the law presented in the equation below (13). The average error of the proposed methodology results against the physical measurements is about 3.4%.

$$Pcs(f) = (4.4 \times 10^{-3} \times f) + 0.3041 \tag{13}$$

Where f is the CPU frequency, the unit of Pcs and f is respectively mW and MHz.

The voltage V_{drop} across the processor increases with the rise of the processor frequency so that the power consumption increases with the frequency.

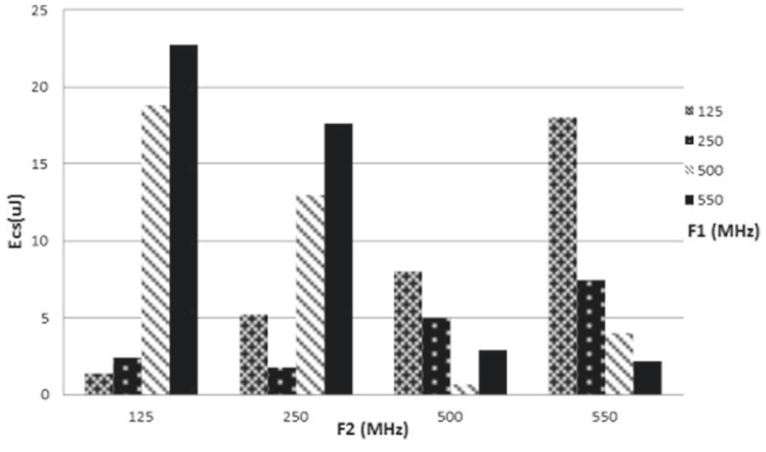

Figure 11 Context switch energy variation as a function of dynamic CPU frequency scaling.

Dynamic frequency case

The core frequency is dynamically changed during the test of the benchmarks: The test programs are executed in step 1 and step n. The processes P1 and P2 are executed respectively at a frequency F1 and F2. So, when the processor preempts the process P1 and executes the process P2, the core frequency changes from F1 to F2; and inversely.

Figure 11 illustrates the context switch energy variation which depends on the frequency difference. Actually, for the processor core, a set of voltage and frequency couples is specified, named operating points. Running on high frequency requires also high voltage and inversely. For raising the frequency and supply voltage, the microprocessor sets a new voltage identifier) code to have a higher output voltage than the current one, and conversely. This operation leads to time and energy overhead [16]. Also, the more important the difference between F1 and F2 is, the higher context switch energy is. This is due to the perturbation of the processor's cache memory resulting from the frequency of the processor bus which varies with the processor frequency.

Inter-process communication

Inter-process communications (IPC) allow threads in one process to share information with threads in other processes, and even processes that exist on different hardware platforms. The embedded OS explicitly copies information from a sending process's address space into a distinct receiving process's address space. Examples of IPC mechanisms are pipes, message passing through mailboxes and shared memory. To characterize the power and energy consumption of IPC, we have to execute test programs, each one repeatedly calling an IPC mechanism. The aim is to get an average power and performance overhead of the IPC mechanism when running the programs. The set of parameters that we could vary are: the type of IPC mechanism, the amount of data shared through the IPC (applicative parameters), the processor frequency. Then, we build the power models of the IPC mechanisms. The test programs were developed for three communication mechanisms which are shared memory, named pipes and anonymous pipes. The message length varies from 1 B to 8 kB, which is the maximum size allowed by the Linux kernel, communications are performed within the same process to avoid process context switching. We have also executed test programs with a high real time priority to avoid preemptive context switch. Figure 12 shows that power consumption P_{ipc} is varying with the processor frequency F. Thus, the power model of IPC mechanisms is:

$$P_{\text{ipc}}(F) = (\alpha \times F) + \beta \tag{14}$$

Where α and β are coefficients of the model. The unit of P_{ipc} and F is, respectively mW and MHz. Power models are presented in Table 1.

Table 1 IPC power models according to processor frequency

IPC mechanism	Power model: P_{IPC} (mW)	Average error (%)
Anonymous pipe	$0.347 \times F + 6.474$	0.293
Named pipe	$0.333 \times F + 4.968$	2.113
Shared memory	$0.217 \times F + 8.542$	2.27

Figure 13 depicts the influence of the message size *msz* on the energy consumption of IPC: E_{ipc}. The energy

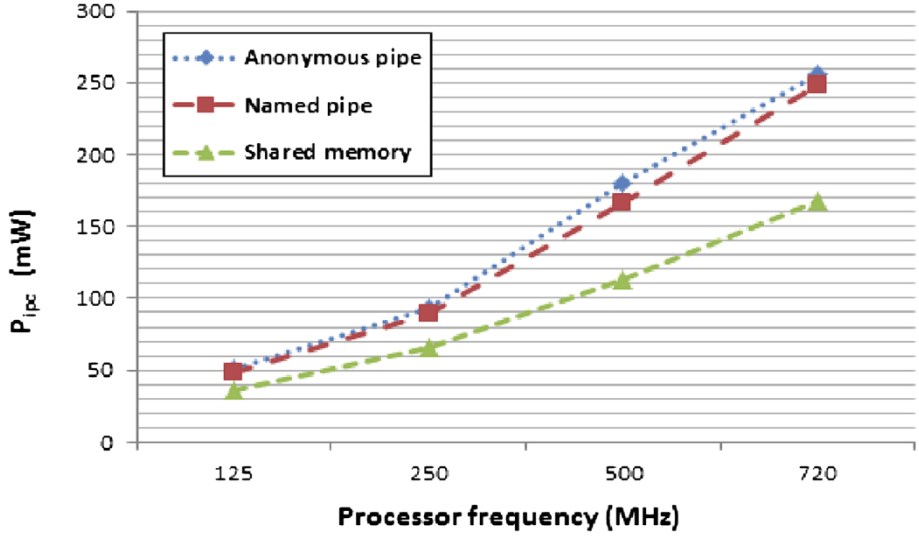

Figure 12 IPC power variation as a function of CPU frequency.

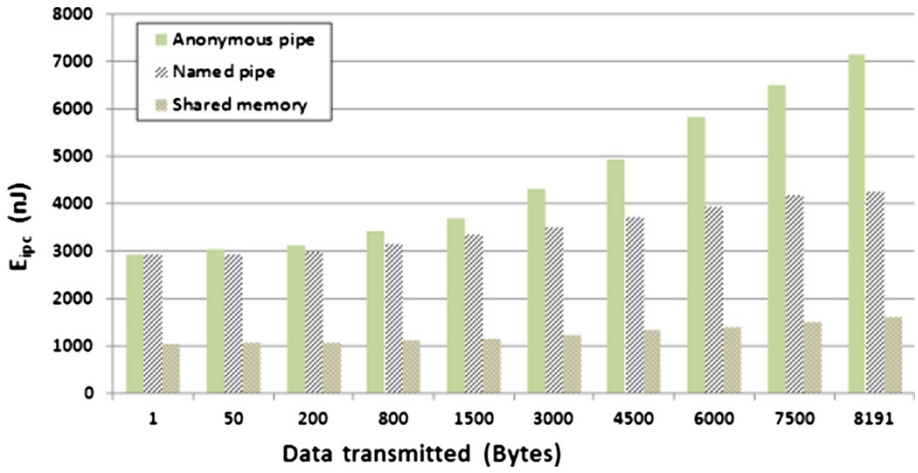

Figure 13 IPC energy variation as a function of message size.

overhead increases exponentially with the rise of the size of data transmitted. Equation (15) represents the energy model:

$$E_{\text{ipc}}(\text{msz}) = \lambda \times e^{(\delta \times \text{msz})} \tag{15}$$

Where λ and δ are coefficients depending on the message size and the IPC mechanism. The unit of E_{IPC} and *msz* is respectively nJ and Bytes. Energy models are presented in Table 2.

Embedded OS service's models integration in the system level design flow

The OS energy and power models are integrated in the system level design flow. The energy and power estimation is targeting the system design including the software and hardware components.

The integration of models of the OS in energy and power estimation tools is necessary to achieve estimations at system level and to quantify the energy and power overhead of the embedded OS energy overhead. Accordingly, we propose an approach, as demonstrated in Figure 14, revolving around three focal concepts: Modeling, code transformation and energy and power estimation.

An AADL [17] model relies on the platform model that contains all the components and connections instances of the application. It also references the implementation models of all the components instances, found in the AADL models library. The AADL model describes also the hardware of the physical target platform: the processor, the memory, and the bus entity which are necessary to processes and threads execution. Taking into account the intra-task properties, such as the deadline and worst case execution time, and the intra-task aspects such as the events and IPC, we define the binding properties that are necessary to the deployment of the application's tasks and embedded operating services on the target platform. Using a textual and graphical modeling tool OSATE, we automatically generate the corresponding textual deployment file: The AADL model is mapped to an XML file. To achieve the simulations, we use a multiprocessor simulation tool named STORM [18] (Simulation TOol for Real-time Multiprocessor scheduling). As shown in Figure 15, The input of this tool is the specifications of the hardware and software architectures together with the scheduling policy; it simulates the system behavior using all the characteristics (task execution time, processor functioning conditions, etc.) in order to obtain the chronological track of all the scheduling events that occurred at run time, and compute various real-time metrics in order to analyze the system behavior and performances from various point of views. It is described in a XML input file in which specific tags and attributes have to be used, and where references to predefined components are made.

As a result, simulated outputs can be computed as: either user readable in the form of diagrams or reports, or machine readable intended for a subsequent analysis tool. The user interacts with STORM through a user-friendly graphical user interface which is composed of command and display windows. The XML file generated from the AADL model having the extension ".aaxl" is not recognized by the STORM simulator. For this reason, in the code transformation step, we adapt the file generated to the simulator structure by parsing existing file ".aaxl" and extracting the data needed to generate the input file of the

Table 2 IPC energy models according to message size

IPC mechanism	Energy model: E_{IPC} (nJ)	Average error (%)
Anonymous pipe	$3068 \times e^{103.9*10^{-6}\times \text{msz}}$	2.149
Named pipe	$3003.8 \times e^{44.96*10^{-6}\times \text{msz}}$	1.728
Shared memory	$1060.9 \times e^{48.9*10^{-6}\times \text{msz}}$	1.468

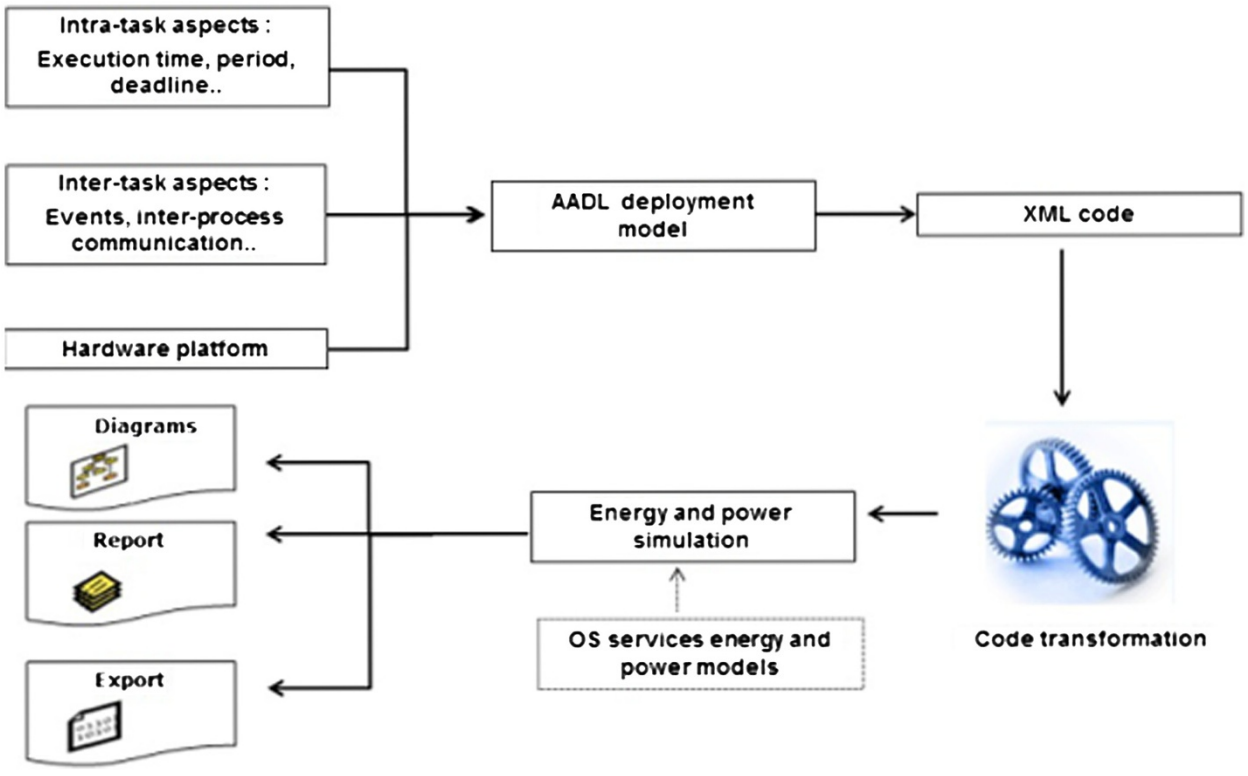

Figure 14 Os power and energy models integration in the system level design flow.

simulator. To extract the required data from the "aaxl" file, we use the java API JDOM which allows us to manipulate, and output XML data from Java code so that we can read and write XML data without the complex and memory-consumptive options that current API offerings provide. Because JDOM uses the Java Collections API to manage the tree, we transform the "aaxl" file to a JDOM tree, and then we extract each data by walking the tree and iterating the document.

Thanks to the significant evolution in processor technology over the last few years, processors with variable voltages and frequencies are now available, they adapt low power and energy techniques to minimize the energy consumption. Reduction in supply voltage requires reduction in operating frequency. That is why, when calculating the overhead of OS services, we execute the application on the hardware platform while adapting a low power technique: The dynamic voltage frequency scaling (DVFS) technique,

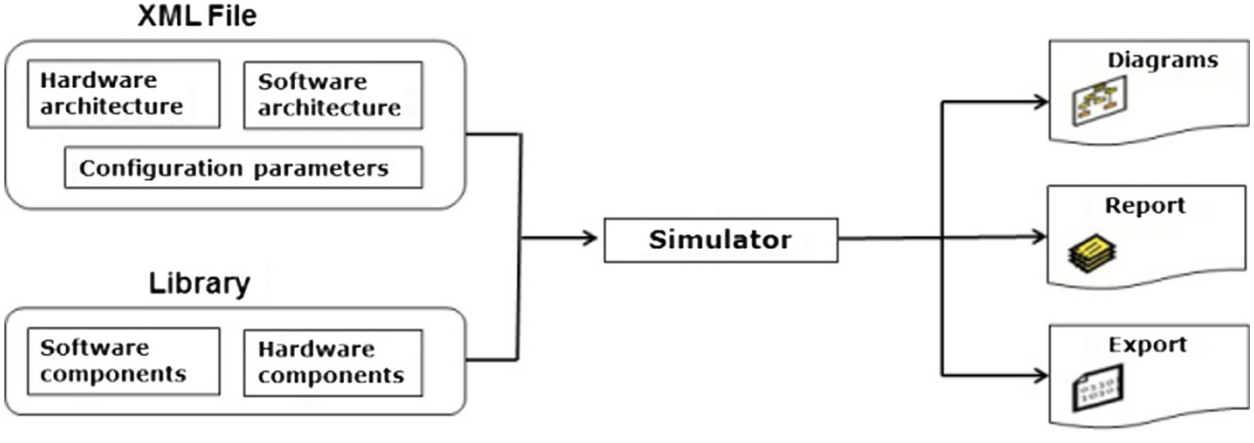

Figure 15 STORM simulator input and output file system.

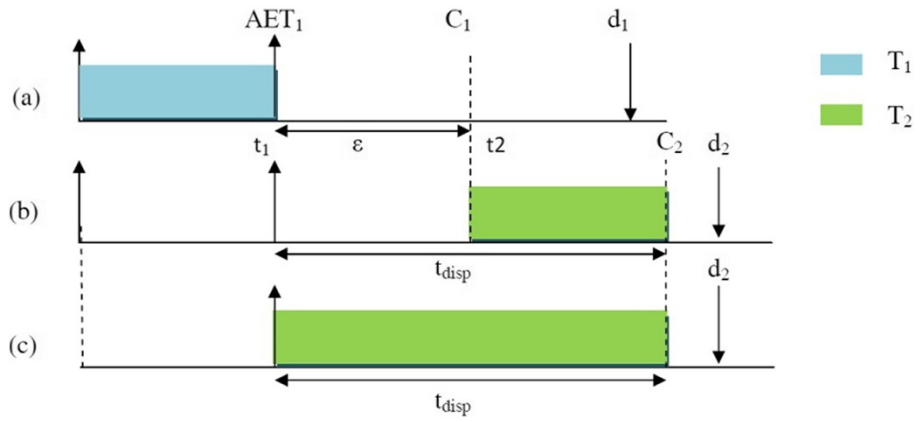

Figure 16 Slack reclamation using the DSF technique.

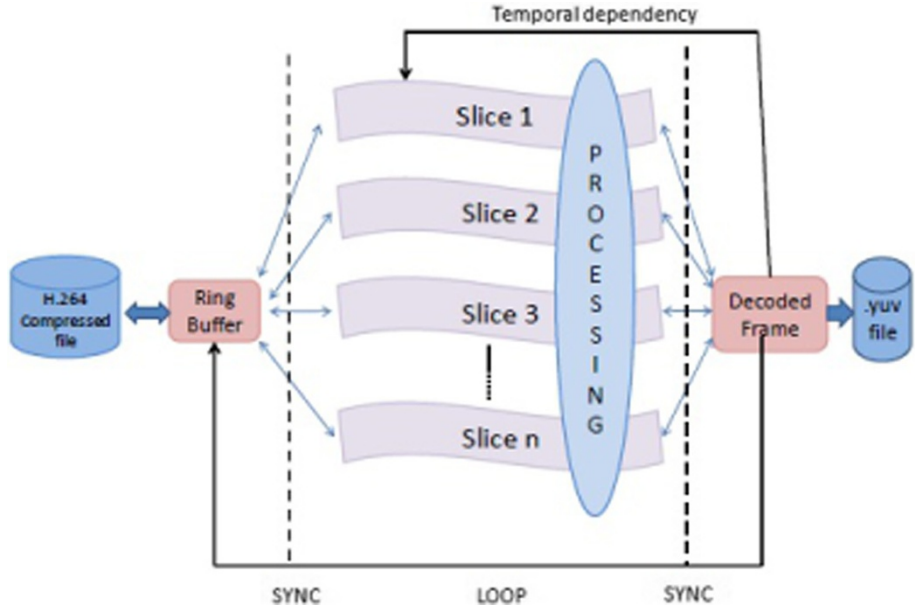

Figure 17 Block diagram of H.264 decoding scheme slices version.

Table 3 H.264 video decoder application tasks features

Task name	WCET (ms)	BCET (ms)	Period (ms)	Deadline (ms)	Activation date (ms)
New_frame	1	1	19	19	0
Nal_dispatch	2	1	5	5	0
Slice1_processing	42	21	66	66	0
Slice2_processing	42	21	66	66	1
Slice3_processing	42	21	66	66	2
Slice4_processing	42	21	66	66	3
Rebuild_frame	2	1	66	66	66

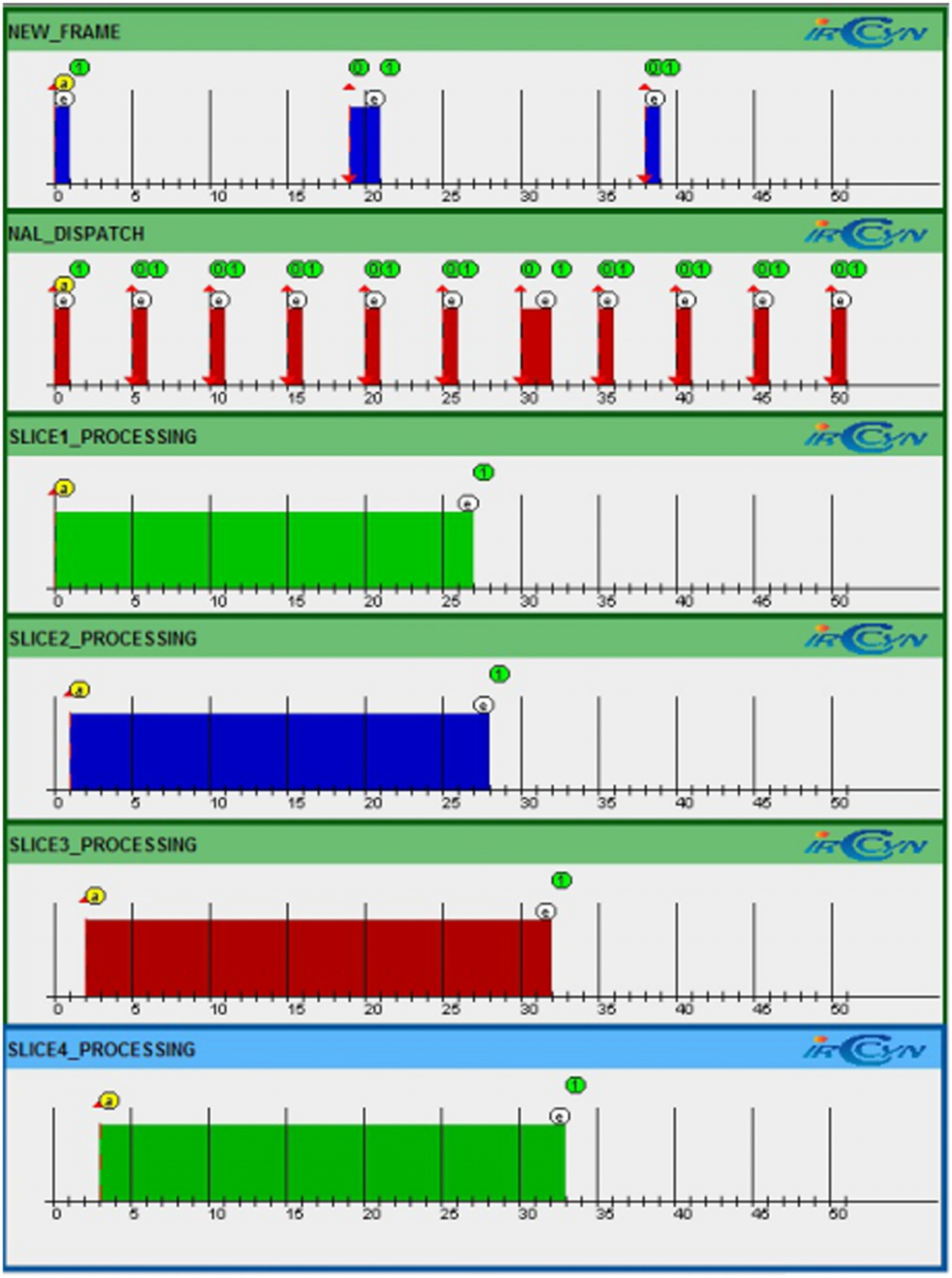

Figure 18 Schedule of application tasks using DSF technique.

it has been particularly distinguished by its efficiency to reduce CPU power consumption. It can execute various tasks of an application at different couples of voltage / frequency depending on the workload of the processor. Several strategies have been proposed to exploit certain aspects of DVFS and offer a particular method to build pseudo intermediate frequencies for use in conjunction with the techniques of dynamic voltage scaling (DVS) [19].

In this article, we adapt an instance of DVFS technique: the deterministic stretch-to-fit technique (DSF) proposed in [20], it is based on the slowdown strategy of reducing the processor power consumption. Slowdown is known to reduce the dynamic power consumption at the cost of increased execution time for a given computation task. It detects early completion of tasks and exploits the processor resources to reduce the energy consumption. As showed in Figure 16, by comparing the actual execution time (AET) of a task T_1 with its worst-case execution time (WCET) C_1, (DSF) technique determines the value of the dynamic slack (ϵ). This slack time is exploited by the method to reduce the energy consumed, by stretching the execution of T_2, having C_2 as WCET, and reducing the frequency of the processor. t_{disp} is the available time at current processor frequency f. t_1 and t_2 represent, respectively the activation date of T_1 and T_2.

The total energy consumed E is calculated as in the equation below (16)

$$E = E_{\text{running}}[f_m] + E_{\text{idle}}[f_m] + E_{\text{OS}} = \sum_{1 \le i \le n} E_{T_i} + E_{\text{idle}}[f_m] \tag{16}$$

Where $E_{\text{running}}[f_m]$ and $E_{\text{idle}}[f_m]$ represent respectively the energy consumed by the processor, at a frequency f_m, when it is in running and idle mode, n is the number of the tasks, E_{OS} and E_{T_i} are previously presented in Equations (2) and (4).

In the next section, taking as use case the H.264 application, the energy consumption of the OS services will be determined following the approach described previously.

Experimental results

The H.264 video decoder application is taken as main use case application. It is a high quality video compression algorithm relying on several efficient strategies extracting spatial (within a frame) and temporal dependencies (between frames). This application is characterized by a flexible coding, high compression and high quality resolution. Moreover, it is a promising standard for embedded devices. The main steps of the H.264 decoding process consist in the following: First, a compressed bit stream coming from the Network application layer (NAL), which formats the representation of the video and provides header information in a manner appropriate for conveyance by particular transport layers, is received at the input of the decoder. Then, the entropy decoded bloc begins with decoding the slice header where each slice consists of one or more 1616 macroblocks, and then it decodes the other parameters. The decoded data are entropy and sorted to produce a set of quantized coefficients. These coefficients are then inverse quantized and inverse transformed. Thereafter, the data obtained are added to the predicted data from the previous frames depending upon the header information. Finally the original block is obtained after the de-blocking filter to compensate the block artifacts effect.The H.264 video decoder application can be broken down into various tasks sets corresponding to different types of parallelization. In our experiments, we use the slices version, one of the task models of H.264 proposed by Thales Group, France [21] in the context of French national project Pherma [22].

The main characteristic of this version is that the algorithm is parallelized on the slices of the frame as illustrated in Figure 17 from this diagram; we consider that we have four types of tasks. First, we start with the *NEW_FRAME* tasks (T_1) that can access only sequentially to the input data buffer. Therefore, the *NAL_DISPATCH* task (T_2) which provides access to a shared resource is protected by a semaphore. Then, *SLICE_PROCESSING* tasks (T_3, T_4,.., T_n) are launched simultaneously. Due to temporal dependencies between frames, it is not possible to compute the next frame if the previous one has not been completely decoded. Thus, at the end of each slice computation, tasks need to be resynchronized using task named SYNC before running the *REBUILD_FRAME* (T_{n+1}) task.

Hence, H.264 slices version, comprising seven periodic tasks as shown in Table 3, is used as use case.

Using AADL, the properties of the system architecture, including the application's tasks and the hardware platform, are modeled. Then, after performing the code transformation step described in the last section, the execution of the application tasks are simulated using the STORM environment. Figure 18 shows the scheduling of the application tasks between 1 and 50 ms. We note the stretching of tasks and then the reduction of energy consumed using the DSF technique. The OS services energy consumption rates are presented in Table 4, we note that the context switch is a basic service because the DSF technique

Table 4 OS services energy consumption rates

OS service S_j	Energy rate $\left(\sum_{1 \le i \le 7} \sum_{1 \le j \le 3} \delta_{i,j}\right)$
Context switch	27.2%
Inter-process communication	3.45%
Scheduling	2%

performs processor frequency changes. The total energy consumed by the low power OS services is significant.

Conclusion

We have presented power and energy models of three basic services of the embedded OS adapting a low power technique: the scheduling, the context switch and IPC. These services are chosen to be characterized because they are stimulated when adapting the DVFS technique. The models are based on measurements on the hardware platform OMAP35x EVM board and allow the characterization of the energy overhead of the low power OS. Experiments show that these services consume a significant part of energy. For this reason, we plan, in the future work, to characterize other basic services of the OS such as the I/O operations and task management, then to compare the overhead of the low power OS using DVFS technique with those using other techniques, for example the DPM (Dynamic power management) technique. The future works of this project will focus on handling the use of the OS by application tasks to optimize the energy consumption of the embedded systems.

Competing interests
The authors declare that they have no competing interests.

Acknowledgements
The authors would like to thank the national agency of research (ANR) in France that sponsors our research project OPEN-PEOPLE.

Author details
[1]University of Nice Sophia-Antipolis, LEAT CNRS, 250 rue Albert Einstein, bât 4 - 06560 Valbonne - Cedex, France. [2]European University of South Brittany-UBS, Lab-STICC CNRS, BP92116 - 56321 Lorient - Cedex, France.

Received: 31 January 2012 Accepted: 5 July 2012
Published: 25 July 2012

References

1. Open-PEOPLE project: Open Power and Energy Optimization Platform and Estimator (2011), http://www.open-people.fr/France,2011
2. T Li, LK John, in *Proceedings of the International Conference on Measurement and Modeling of Computer Systems,* Run-time modeling estimation of operating system power consumption, (SIGMETRICS, San Diego, California, USA, 2003). pp. 160–171
3. A Acquaviva, L Benini, B Riccó, Energy characterization of embedded real-time operating systems, J. ACM SIGARCH Comput. Archit. News. **29**, 13–18 (2001)
4. TK Tan, A Raghunathan, NK Jha, in *Proceedings of the 2002 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD'02),* Embedded operating system energy analysis and macro-modeling, (2002). Washington, DC, USA, pp. 515–520
5. RP Dick, G Lakshminarayana, A Raghunathan, NK Jha, in *Proceedings of the 37th Annual Design Automation Conference,* Power analysis of embedded operating systems, Los Angeles, CA, USA, 2000, pp. 312–315
6. K Baynes, C Collins, E Fiterman, B Ganesh, P Kohout, C Smit, T Zhang, BL Jacob, The performance and energy consumption of embedded real-time operating systems. IEEE Trans. Comput. **52**, pp. 1454–1469 (2003)
7. B Guo, D Wang, Y Shen, Z Li, A Hopfield neural network approach for power optimization of real-time operating systems. Neur. Comput. Appl. **17**, 11–17 (2008)
8. X Zhao, Y Guo, H Wang, X Chen, in *Proceedings of the 2008 International Conference on Embedded Software and Systems Symposia,* Fine-grained energy estimation optimization of embedded operating systems, (2008). Chengdu, Sichuan, China, pp. 90–95
9. N Fournel, A Fraboulet, P Feautrier, in *IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks,* eSimu: a fast and accurate energy consumption simulator for real embedded system, (2007). Espoo, Finland, pp. 1–6
10. Y Fei, S Ravi, A Raghunathan, N Jha, Energy-optimizing source code transformations for operating system-driven embedded software. . ACM Trans. Embed. Comput. Syst. (TECS). **7**, 1–26 (2007)
11. S Dhouib, E Senn, J Diguet, J Laurent, in *Proceedings of the 2009 12th International Symposium on Integrated Circuits, ISIC'09,* Modelling and estimating the energy consumption of embedded applications and operating systems, (2009). Singapore, pp. 457–461
12. OMAP35x Evaluation Module (EVM) (2011), http://focus.ti.com/docs/toolsw/folders/print/tmdsevm3530.html
13. F Liu, F Guo, Y Solihin, S Kim, A Ekerm, in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques,* Characterizing and modeling the behavior of context switch misses, Toronto, Ontario, Canada, 2008, pp. 91–101
14. D Tsafrir, in *ACM Workshop on Experimental Computer Science (ExpCS),* The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops), (2007). San-Diego, California, USA, p. 4
15. B Ouni, C Belleudy, S Bilavarn, E Senn, in *Conference on Design and Architectures for Signal and Image Processing, DASIP,* Embedded operating systems energy overhead, (2011). Tampere, Finland, pp. 52–57
16. J Park, D Shin, N Chang, M Pedram, in *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design,* Accurate modeling and calculation of delay and energy overheads of dynamic voltage scaling in modern high-performance microprocessors, (2010). Austin, Texas, USA, pp. 419–424
17. Architecture Analysis and Design Language (AADL) standard (2011), http://www.aadl.info/
18. STORM simulation tool (2011), http://storm.rts-software.org
19. P Pillai, KG Shin, in *Proceedings of the eighteenth ACM symposium on Operating systems principles,* Real-time dynamic voltage scaling for low-power embedded operating systems, (2001). Banff, Alberta, Canada, pp. 89–102
20. MK Bhatti, C Belleudy, M Auguin, in *Proceedings of the 2010 Conference on Design and Architectures for Signal and Image Processing (DASIP'10) Edinburgh, Scotland, United Kingdom, 2010,* An inter-task real time DVFS scheme for multiprocessor embedded systems, pp. 136–143
21. Thales group (France), France, 2011, [http://www.thalesgroup.com]
22. ANR project Pherma, France, (2007–2010), [http://pherma.irccyn.ec-nantes.fr]

doi:10.1186/1687-3963-2012-6
Cite this article as: Ouni *et al.*: Accurate energy characterization of OS services in embedded systems. *EURASIP Journal on Embedded Systems* 2012 **2012**:6.