Predicting Energy and Performance Overhead of Real-Time Operating Systems

Sandro Penolazzi, Ingo Sander and Ahmed Hemani
Dept. of Electronic Systems, School of ICT, KTH, Stockholm, Sweden
{sandrop, ingo, hemani}@kth.se

***Abstract*—We present a high-level method for rapidly and accurately estimating energy and performance overhead of Real-Time Operating Systems. Unlike most other approaches, which rely on Transaction-Level Modeling (TLM), we infer the information we need directly from executing the algorithmic specification, without needing to build any high-level architectural model. We distinguish two main components in our approach: first, an accurate one-time pre-characterization of the main RTOS functionalities in terms of energy and cycles; second, the development of an algorithm to rapidly predict the occurrences of such RTOS functionalities. Finally, we demonstrate the feasibility of our approach by comparing it against gate level for accuracy and against TLM for speed. We obtain a worst-case energy error of 12% against a mean speedup of 36X.**

I. Introduction

Real-Time Operating Systems (RTOS) are critical infrastructural components of embedded SoCs. They enable and manage sharing of hardware resources: computational, storage, interconnect and I/O. However, such services have an overhead, both in latency and energy. This overhead needs to be accounted for when doing design-space exploration and energy/performance estimation at system level, early in the design cycle. Without factoring in this overhead, the accuracy of the estimates would be severely compromised.

Raising abstraction is fundamental to get improvement in design productivity and this pattern has been consistent for the last three decades [1]. Transaction Level Modeling is part of this trend and raises the abstraction above the RTL. TLM models an architecture by abstracting away the implementation details, especially the cycle-accurate-level details, of the architectural resources; the modeling is done in terms of abstract transactions between architectural resources. This allows to cut down the modeling time and to increase the simulation capacity, thus being a good candidate to perform early design-space exploration.

While TLM is a progress, it still implies a substantial additional engineering effort, because TLM is not the starting point of automatic synthesis or system build and, as such, the TLM step is not an essential step in implementing the system. In addition, when TLM is used as the basis for estimating the impact of RTOS, the actual RTOS and application software are simulated in the SoC architectural model and often an instruction-set simulator simulates the RTOS and the application software at instruction-set level. This makes even a TLM-based model slow.

Funtime is an early estimation methodology framework that aims at overcoming these problems. It fundamentally differs from TLM in that it does not simulate the architecture, but it infers the architectural implications while executing only the algorithmic model of the application on a common PC. The abstraction level of the Funtime methodology is higher than TLM. We have previously shown Funtime to be 30X faster than TLM and within 15% accuracy of post-layout gate-level simulation [2]. Unlike TLM, the Funtime methodology relies on an engineering step that has always been part of the SoC engineering cycle, i.e. the algorithmic level models of the application.

The Funtime methodology that we published in the past [2] ignored the impact of RTOS and assumed that each application had the resources to itself. This paper extends the Funtime methodology to factor in the impact of RTOS as well, both for latency and energy, by introducing two additional steps:

- *RTOS characterization*: we characterize the main components of a generic RTOS in terms of typical number of execution cycles and power consumption. This characterization is a *one-time activity* that we propose to be done by the RTOS provider. This is detailed in section IV.
- *RTOS activity prediction*: we propose a static analysis strategy to predict how many occurrences of such OS components we would count if we were actually executing the OS in a real use-case scenario. This prediction method is invoked by the Funtime user for each use-case scenario. A detailed description of this step is presented in section V.

II. Related Work

Yi et al. [3][4] present an approach to RTOS modeling that has some similarity to our work in that they also include a pre-characterization phase of some key RTOS components like the tick interrupt (triggered by the system timer) and the context switch. However, this characterization is done only to factor in the extra latency induced by the RTOS and ignores the impact on energy. Besides, the authors rely on a trace-driven cosimulation in SystemC to run their applications and thus measure the task duration, in order to derive the actual number of clock ticks. Our approach predicts instead the RTOS activity by means of a static analysis, resulting in faster estimates.

In [5], Brandolese et al. also propose a method to pre-characterize system calls of an OS for embedded applications. To do that, they rely on measurements "*based on executing*

stubs". This is also the approach that we use, as detailed in section IV. However, unlike us, they only characterize the impact on latency and neglect the impact on energy. In contrast to [4], [5] and the method proposed by us, Hessel et al. [6] do not pre-characterize the RTOS, but propose an RTOS model in SystemC, by extending the built-in scheduler that also does the power estimation. In [7], an extension of the SystemC simulation engine is proposed to build an RTOS model, where a new simulation library is added to implement the so called T-THREADs. In a similar fashion, in [8] the SystemC SC_THREAD processes are also exploited to create an OS model.

The key difference between the Funtime methodology for estimating the impact of RTOS and the related research that we have reviewed here is that in Funtime we do not simulate the RTOS using an ISS or a SystemC transaction-level model of a SoC architecture: we infer the impact of the RTOS by running the application code at functional untimed level. This difference gives the Funtime methodology three advantages:

1) we avoid the time-consuming process of building the TLM of the architecture.
2) being more abstract, the functional untimed level has a significant speed advantage over TLM.
3) we do not have to deal with synchronization issues among multiple simulation models [4].

III. The Funtime Methodology

Funtime operates at 3 different levels of abstraction, as summarized in the present section and shown in Figure 1. We have highlighted in red color the components of the Funtime flow that concern accounting for the Operating System overhead, which is also the contribution and novelty of this paper.

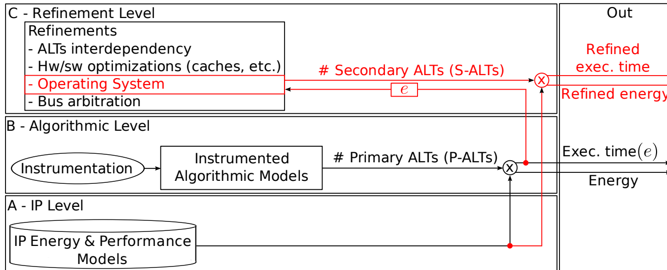


Fig. 1. The Funtime methodology flow (A-IP Level, B-Algorithmic Level, C-Refinement Level)

A. IP Level

At this level, we consider the availability of IP energy and performance models. Such models, which are the result of a one-time effort made by the IP provider, characterize each IP Architectural-Level Transaction (ALT) in terms of energy and number of cycles, where an architectural-level transaction directly expresses one of the IP functionalities. For instance, for a processor IP, each instruction represents one such transaction. Energy and performance characterization is done by the IP provider only once on a back-annotated gate-level netlist, thus conferring high accuracy to the model. In essence, for each IP in the library, the IP model expresses energy and cycles properties for all the transactions of that IP.

In our approach, an OS is also considered an IP, whose transactions correspond to very high-level functionalities such as executing a context switch, serving a clock tick interrupt, etc. The OS characterization process is detailed in section IV.

B. Algorithmic Level

Once each IP transaction has been characterized for energy and number of cycles, energy and execution time e (see Figure 1) for a full application can be calculated provided that the total number and type of transactions triggered by such an application are known.

Traditionally, this information can be collected from Instruction Set Simulation (ISS), transaction-level simulation or any other architecture-based simulation. In our approach instead, since the idea is to avoid architectural simulation, the inference of architectural transactions is achieved by instrumenting the application (algorithmic specification), itself devoid of any architectural detail, to be architecture aware. The instrumented application is then executed natively on a common PC. Transactions inferred at this level are defined *Primary* transactions (P-ALTs). The process flow is illustrated in Figure 1B. For a thorough description of how instrumentation and inference of P-ALTs are done in practice, the reader is referred to our previous work [2].

C. Refinement Level

From Subsections III-A and III-B, once each IP transaction has been characterized for energy and number of cycles, and the total occurrence of such transactions for the execution of an application has been determined, energy and execution time e (see Figure 1) for that application can finally be estimated.

This estimate, while it accurately reflects the contribution of all the Primary transactions, it ignores the impact of other transactions – which we call *Secondary* transactions (S-ALTs) – induced by the following three main reasons:

1) The interdependency of Primary ALTs, especially processor instructions.
2) Architectural optimization measures like caches, power management, etc.
3) Sharing of resources due to Operating System (OS) and bus arbitration.

Refining the estimate from Level B is done at Level C, which is naturally called Refinement Level. In essence the Level C refines the trace of Primary ALTs from Level B by adding the induced Secondary ALTs. Of the three reasons listed above, we have previously addressed the first and in part the second one in [2]. In this paper, we tackle partially the reason 3 by factoring in the OS-related Secondary ALTs that are induced when multiple applications share a processor. This is symbolically shown in Figure 1 and elaborated in the next two sections.

IV. RTOS CHARACTERIZATION

In this section, we propose a general approach to characterize the S-ALTs due to the RTOS overhead in terms of latency and energy. RTOS overheads are implied by its components that are called to implement the RTOS functionality. Characterization is a *one-time activity* done by the RTOS provider. The process consists in simulating the RTOS in a post-layout, back-annotated gate-level netlist of the representative SoC architecture. Although time consuming, this characterization process is justified because it is done only once and the precision gained is important for the accuracy of estimates at high-level. Although in this paper we present the characterization of the RTEMS RTOS [9] for a Leon-based SoC [10], the methods themselves are generic and do not depend on a specific RTOS implementation or SoC architecture.

The characterization process not only considers atomic RTOS calls, as it is done in [5], but also considers coarse-grained RTOS calls like clock tick interrupts and scheduler invocation. The number and type of atomic RTOS calls involved both in the clock tick interrupt and in the scheduler are OS-dependent. However, thanks to the general approach that we have adopted, it is easy to specify the sequence of atomic OS routines that have to be included in the performance and energy characterization.

A. *Characterizing a group of RTOS routines*

We explain in 5 steps the characterization process using RTEMS for the Leon3-based SoC shown in Figure 2.

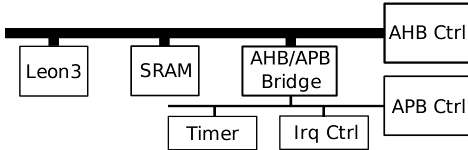


Fig. 2. Leon3-based platform

1 – A representative SoC, with essential hardware for hosting an RTOS, is composed. An example is given in Figure 2. The platform is then synthesized and all the wire delays and parasitics are back-annotated to the gate-level netlist. Synthesis has been carried out in our case for the TSMC-90nm technology.

2 – The RTOS and its tasks are compiled and loaded into the system. From the object file, a memory dump of the RTOS and the tasks is also taken. A fragment of such a dump is shown in Figure 3. In the RTEMS case, this can be done using the `sparc-rtems-objdump` command. The dump allows us to associate any routine name to its memory addresses.

```
40006944 <rtems_clock_tick>:
40006944: 9d e3 bf 98  save  %sp, -104, %sp
40006948: 40 00 08 09  call  4000896c <_TOD_Tickle_ticks>
4000694c: 01 00 00 00  nop
40006950: 11 10 00 71  sethi  %hi(0x4001c400), %o0
40006954: 40 00 16 77  call  4000c330 <_Watchdog_Tickle>
```

Fig. 3. Example of an objdump output

3 – The system is executed at gate level, with the user-defined tasks, and the RTOS calls are activated a sufficient number of times — around 1000 — to make the characterization statistically relevant.

4 – As an output of step 3, an execution trace file and a VCD (Value Change Dump) file are produced from the gate-level simulation. We require that the execution trace file contains both the address of each instruction that has been executed, as well as its time stamp. The VCD file contains instead the switching activity figures, used by the EDA tools to do power estimation.

5 – By using the information in the dump, execution trace, VCD and by providing the sequence of atomic RTOS routines to be characterized, it is possible to extract the average number of cycles and also the power for the routines under characterization. We have automated this last step by implementing and using a C-based script, which we call RTOS Modeler and whose overall functionality is summarized in Figure 4. The advantage of having such a script is that it allows to characterize any atomic routine or sequence of routines present in the object dump file, without needing any intervention from the user. As a consequence, the whole characterization process gets a significant speedup. In addition, this script is completely OS-independent, thus it can be easily reused.

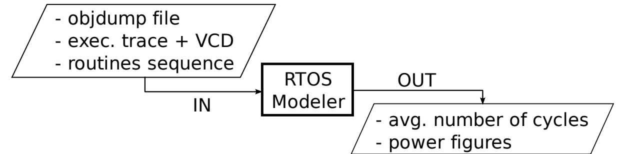


Fig. 4. RTOS Modeler

A summary of the coarse-grained RTOS calls that we have characterized for RTEMS is shown in Table I. For each of them, we show the corresponding number of CPU instructions (S-ALTs), cycles and energy with the associated standard deviation σ. Note that the small value of σ is an index of the characterization reliability. The energy values in the table only refer to the Leon3, configured without cache.

TABLE I
ENERGY AND PERFORMANCE CHARACTERIZATION FOR RTEMS ON LEON3 (NO CACHE)

RTOS calls	# Leon3 S-ALTs	# Cycles	σC	Energy [nJ]	σE
clock tick interrupt	272	2241	24.00	80.14	0.96
scheduler with context switch	880	8434	30.01	263.58	0.96
scheduler without context switch	327	2545	3.53	89.94	0.20
idle task	3	22	0.04	0.76	0.00

Table I only lists a subset of all the possible coarse-grained functionalities associated to an RTOS. This subset is sufficient to represent the case studies described in the next sections of this paper. Characterizing and including more RTOS functionalities is however part of our future work, together with the discussion of other case studies.

V. RTOS ACTIVITY PREDICTION

Once the one-time RTOS characterization activity is done, it is still necessary to predict how many times the OS calls get triggered during the OS execution in order to estimate the total OS overhead. How Funtime allows this is elaborated next.

Some new quantities are here introduced and categorized, where possible, according to the Funtime abstraction level where their value is assigned. We also assume independent tasks, i.e. not having to contend for shared resources, scheduled using Round Robin (RR).

User-defined parameters: when using RR, all tasks run in turn for the same amount of time t_{slice}, which is a multiple of the clock tick period t_{tick}. We further define $t_{rel,i}$ as a task release time, i.e. the time when a task becomes available for execution. t_{slice}, t_{tick} and $t_{rel,i}$ are user-defined parameters.

From Level A: we define e_{sc} and e_{ct} as the execution time of a scheduler call and of a clock tick interrupt respectively. They are determined during the OS characterization phase.

From Level A and B: given a generic task T_i, its execution time e_i is defined as the ideal execution time required to complete T_i when T_i has all the resources available to itself. This quantity was already introduced in section III and shown in Figure 1. The actual number of tasks T_i and all the required runtime information are also provided at these two levels.

From Level C: we define SC_{slice} as the number of scheduler calls during t_{slice}. We assume that such invocations always result in a context switch. We also define CT_{slice} as the number of clock tick interrupts occurring during t_{slice}, excluding the clock tick interrupt that invokes the scheduler. Thus, we have $SC_{slice} = 1$ and $CT_{slice} = (t_{slice}/t_{tick} - 1)$. Figure 6 illustrates this for the time range $30 \leq t < 60$, where we can identify one scheduler invocation in the red bar and two clock tick interrupts in the thinner blue bars.

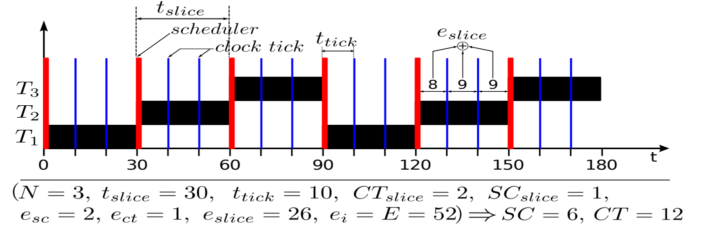
We also define e_{slice} as the slice time that is actually dedicated to executing a generic task T_i and not spent in executing OS calls. This leads to $e_{slice} = t_{slice} - (SC_{slice} \cdot e_{sc} + CT_{slice} \cdot e_{ct})$. In Figure 6, the calculation of the e_{slice} value is shown for the time slice in time range $120 \leq t < 150$.

Finally, we define SC and CT as the number of total calls to the scheduler and of clock tick interrupts happening when running an arbitrary number N of concurrent tasks.

Using such definitions, we show how e_i can be used at Level C to predict SC and CT. To ease the explanation, we use 3 successively more realistic examples. First, we assume that all tasks are released simultaneously and have the same e_i. Second, we allow tasks with different e_i, but we still assume that they are released simultaneously. Finally, we allow tasks released at different times, as well as the possibility that RTOS overheads may be a function of the number of ready tasks.

Example 1: in this ideal case, all the N tasks have the same release time $t_{rel,i} = 0$ and the same execution time $e_i = E$ with $E = n \cdot e_{slice}, n \in \mathbb{N}$. The values for SC and CT can be calculated by Eq. 1. This scenario is exemplified in Figure 6, where we also extract numerical values for SC and CT.

$$SC = SC_{slice} \cdot N \cdot \frac{E}{e_{slice}}; \qquad CT = CT_{slice} \cdot N \cdot \frac{E}{e_{slice}} \tag{1}$$



$(N = 3,\ t_{slice} = 30,\ \ t_{tick} = 10,\ CT_{slice} = 2,\ SC_{slice} = 1,$
$e_{sc} = 2,\ e_{ct} = 1,\ e_{slice} = 26,\ e_i = E = 52) \Longrightarrow SC = 6,\ CT = 12$

Fig. 6. RR scheduling: $\forall \mathbf{T_i}$, $\mathbf{t_{rel,i} = 0}$ and $\mathbf{e_i = E}$

Example 2: we consider a more general case where all N tasks have different execution times e_i and e_i is not an exact multiple of t_{slice}. In this case, each task T_i is completed before the end of its slice time. This scenario, shown in Figure 5 for N=4, is highlighted in grey boxes. When this condition occurs, the scheduler is called at the next clock tick interrupt and, unless the ready task queue is empty, a context switch occurs, even if the slice time has not yet expired. This is done to minimize the performance loss. Note that the behavior described is valid for RTEMS. In this case, we need a more general formula to determine SC and CT.

We define Q_i and R_i as the quotient and the remainder of the division e_i/e_{slice} for a generic task T_i. R_i is the portion of T_i for which the execution time is smaller than a slice time, that is $0 \leq R_i < e_{slice}$. The value of R_i is shown in Figure 5 for each of the 4 tasks considered in this example. Let then $SC_{rem,i}$ and $CT_{rem,i}$ be the number of calls to the scheduler and of clock ticks occurring during the time R_i. As we assumed that a time slice always starts with a call to the scheduler, the values of $SC_{rem,i}$ and $CT_{rem,i}$ can be expressed as in Eq. 2 and 3 respectively.

$$SC_{rem,i} = \begin{cases} 0 & \text{if } R_i = 0 \\ 1 & \text{if } R_i > 0 \end{cases} \tag{2}$$

$$CT_{rem,i} = \begin{cases} 0 & \text{if } R_i \leq (t_{tick} - e_{sc}) \\ \left\lceil \dfrac{R_i - (t_{tick} - e_{sc})}{t_{tick} - e_{ct}} \right\rceil & \text{if } R_i > (t_{tick} - e_{sc}) \end{cases} \tag{3}$$

The total number of calls to the scheduler SC and clock tick interrupts CT is calculated now through an iterative process, where the number of iterations corresponds to the number of tasks N. During each iteration, the value of SC and CT is incremented. The main steps are shown in Algorithm 1 below.

Algorithm 1 Assumptions: $t_{rel,i} = 0$, $e_i \neq e_j$

```
SC, CT ← 0;
for i = 1 to N do
    SC ← SC + SC_slice · Q_i + SC_rem,i;
    CT ← CT + CT_slice · Q_i + CT_rem,i;
end for
```

Example 3: we consider a more real case where N tasks have arbitrary release times $t_{rel,i}$ and execution times e_i, as shown in Figure 7. In addition, we assume that the execution time of some OS functionalities is related to the number of ready tasks $rdy(t)$ at the time t when such functionalities get

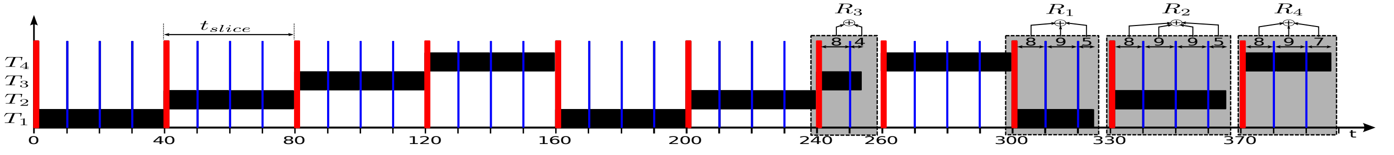


Fig. 5. Round Robin scheduling: $\forall \mathbf{T_i}$, $\mathbf{t_{rel,i} = 0}$, $\mathbf{e_i \neq E}$ and $\mathbf{e_i \neq e_j}$

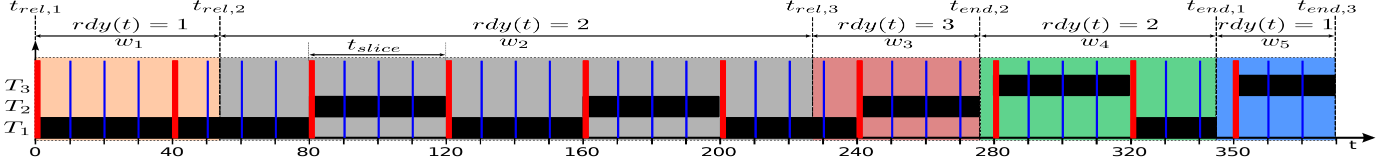


Fig. 7. Round Robin scheduling: $\forall \mathbf{T_i}$, $\mathbf{t_{rel,i} \neq 0}$, $\mathbf{e_i \neq E}$, $\mathbf{e_i \neq e_j}$, $\mathbf{e_{sc}(rdy(t))}$ and $\mathbf{e_{ct}(rdy(t))}$

triggered. Thus, in this case e_{sc} and e_{ct} are not constant figures any more, but they depend on $rdy(t)$ and can be denoted as $e_{sc}(rdy(t))$ and $e_{ct}(rdy(t))$ respectively.

When calculating the value of SC and CT, it is therefore also necessary to determine the value of $rdy(t)$. This process can be eased by restricting the evaluation of $rdy(t)$ to specific time points, which correspond either to a task release time $t_{rel,i}$ or completion time $t_{end,i}$. The reason is that the value of $rdy(t)$ remains unchanged for all the other values of t. The result is a static decomposition of the whole scheduling into multiple time windows w_i. This is shown in Figure 7, where 5 time windows $[w_1, w_5]$ are identified for 3 tasks and highlighted by different-color areas. In detail, whenever a new time window starts and $rdy(t)$ is calculated, the ending time of the same time window is identified as the minimum time between the $t_{end,i}$ for each ready task and the next closest release time t_{rel}. Once the time window has been defined, the values of SC and CT for that time window are calculated and added to the old ones calculated in the previous window. This process is iterated until all tasks are completed. Note that, even in this more elaborated implementation, our OS prediction algorithm has still an execution time proportional to the number of tasks involved, not to their execution time e_i. Thus, it is still extremely fast. The pseudo-code for the main steps described above is reported in Algorithm 2.

Algorithm 2 $t_{rel,i} \neq 0$, $e_i \neq e_j$, $e_{sc}(rdy(t))$, $e_{ct}(rdy(t))$

```
SC(rdy(t)) = CT(rdy(t)) = 0;
total execution time left e_tot_left ← Σ_{i=1}^{N} e_i;
time t ← 0;
while e_tot_left > 0 do
    for i = 1 to N do
        rdy(t) ← number of ready tasks at time t;
        T_rdy[i] ← which tasks are ready at time t;
    end for
    for each T_rdy[i] at time t do
        calculate its ending time t_end,i;
    end for
    t_end_min ← min between each t_end,i and next closest t_rel;
    update SC(t_end_min − t, rdy(t));
    update CT(t_end_min − t, rdy(t));
    update e_tot_left;
    t ← t_end_min
end while
```

VI. Validation

A. Accuracy: Funtime vs. gate level

Energy and timing figures from back-annotated gate-level simulation were taken as a reference for Funtime accuracy validation. The reason is two-fold: first, gate level is very accurate; second, IP-level energy models and the OS refinements were achieved based on gate-level characterization.

The SoC platform used for validation is the same used for the OS characterization and shown in Figure 2. The goal is to verify the accuracy of Funtime in predicting energy and execution time for a software environment where two tasks T_1 and T_2 run on top of the RTEMS OS and are scheduled using Round Robin. The values of the clock tick period t_{tick} and of the time slice period t_{slice} have been set to 1ms and 3ms respectively.

T_1 and T_2 are synthetic applications that have been chosen on purpose with different execution times $e_1 \neq e_2$, but with the same release times $t_{rel,1} = t_{rel,2} = 0$. Such applications are different from those used during the OS characterization.

We have collected the results in Table II, which is itself split into 3 horizontal subtables considering an increasing number of sources of inaccuracy. Subtable 1 only considers the error deriving from the OS characterization described in section IV, while e_1 and e_2, as well as the OS activity are measured from gate level. Subtable 2 has two sources of inaccuracy: the OS characterization and the OS activity prediction, described in section IV and V respectively, while e_1 and e_2 are measured; finally, Subtable 3 considers three sources of inaccuracy: the first two are the same as for Subtable 2, while the third comes from using Funtime (Level A and B) to derive also e_1 and e_2. Table II is further split vertically into a left and a right side: the left side reports figures related to the two tasks T_1 and T_2, independently of the OS; the right side is instead meant to show the OS overhead. The energy values shown refer to the Leon3 processor.

The RTEMS functionalities that we account for in Table II are, from left to right, the clock tick interrupt, the invocation to the scheduler resulting in a context switch and the invocation to the scheduler not resulting in a context switch. For each such a functionality, we compare energy and time values extracted

TABLE II
VALIDATING FUNTIME VS. GATE LEVEL FOR ENERGY ESTIMATION: 2 RR-SCHEDULED TASKS OF DIFFERENT LENGTH

	Applications			**RTEMS OS**					
	$\mathbf{T_1}$	$\mathbf{T_2}$		**clock ticks**		**sched. + c.s.**		**scheduler**	
Subtable 1 (1 source of inaccuracy: OS characterization)									
Real e_i [ms]	37.60	59.35	Real #OS macro-func.		71		28		8
Real energy [μJ]	53.24	84.73	Real energy [μJ], time [ms]	5.67	3.98	7.47	5.27	0.77	0.53
			Inferred energy [μJ], time [ms]	5.69	3.98	7.38	5.20	0.72	0.51
			Error [%]	+0.35	-0.14	-1.27	-1.33	-6.83	-4.48
Subtable 2 (2 sources of inaccuracy: OS characterization + OS activity prediction)									
Real e_i [ms]	37.60	59.35	Real #OS macro-func.		71		28		8
			Inferred #OS macro-func.		72		29		7
			Error [%]		+1		+1		-1
Real energy [μJ]	53.24	84.73	Real energy [μJ], time [ms]	5.67	3.98	7.47	5.27	0.77	0.53
			Inferred energy [μJ], time [ms]	5.77	4.03	7.64	5.39	0.63	0.44
			Error [%]	+1.76	+1.27	+2.26	+2.19	-18.48	-16.41
Subtable 3 (3 sources of inaccuracy: OS characterization + OS activity prediction + $\mathbf{e_i}$ value)									
Real e_i [ms]	37.60	59.35	Real #OS macro-func.		71		28		8
Inferred e_i [ms]	36.25	58.00	Inferred #OS macro-func.		70		28		8
Error [%]	-3.59	-2.28	Error [%]		-1		0		0
Real energy [μJ]	53.24	84.73	Real energy [μJ], time [ms]	5.67	3.98	7.47	5.27	0.77	0.53
Inferred energy [μJ]	46.73	74.77	Inferred energy [μJ], time [ms]	5.61	3.92	7.38	5.20	0.72	0.51
Error [%]	-12.23	-11.76	Error [%]	-1.06	-1.55	-1.27	-1.33	-6.83	-4.48

from a gate-level simulation versus those inferred by Funtime. In any of the three subtables, the results show that the Funtime accuracy increases with the number of OS system calls. For example, the accuracy achieved for the clock tick interrupt, which occurs 71 times, is much higher than the one achieved for the scheduler invocation without context switch, that occurs only 8 times. From the OS activity prediction perspective, it is straightforward that a 1-unit error weighs more on a total of 8 units rather than 71. However, we recall that the Funtime methodology is meant for being used on extensive use-case scenarios, where billions of transactions take place. Under these assumptions, the 18% energy error and the 16% time error obtained with respect to the scheduler invocation without context switch become insignificant.

B. Speed: Funtime vs. TLM-PV

For speed comparison, TLM-PV has been chosen as a reference. The reason is that this is the fastest high-level methodology for system-level estimation commonly used at present. For this purpose, we built our own TLM-PV in SystemC for the reference SoC architecture. The implementation has been as abstract as possible, since it exclusively represents the transactions occurring across the platform among the different IPs. A set of applications has been chosen with a very high number of executed instructions, ranging between 80M - 1.6B. The applications chosen are the image compression codec JPEG2000 and the video compression codec H264. Applications and data have been combined in different ways to show some possible use-case scenarios, where two applications always run concurrently on top of the RTEMS OS. The results are reported in Table III and show a mean speed improvement of 36X for Funtime compared to TLM-PV. This confirms the capacity of Funtime to be used for complex and real use-case scenarios.

TABLE III
SPEED COMPARISON: FUNTIME VS. TLM-PV

Tasks	**TLM[s]**	**Funtime[s]**	**Speedup**
$T_1 \rightarrow$ Jpeg2k_128x128 $T_2 \rightarrow$ Jpeg2k_512x512	46.66	1.86	25
$T_1 \rightarrow$ H264_176x144 $T_2 \rightarrow$ H264_352x288	147.52	3.19	46
$T_1 \rightarrow$ Jpeg2k_128x128 $T_2 \rightarrow$ H264_352x288	93.56	2.40	39

VII. CONCLUSIONS AND FUTURE WORK

We have presented a high-level method for rapidly and accurately estimating energy and performance overhead of Real-Time Operating Systems. We have distinguished two main components in our approach: first, a one-time pre-characterization of the main RTOS functionalities in terms of energy and cycles; second, the development of an algorithm to predict the occurrences of such RTOS functionalities. As a result, we are able to achieve a significant mean speedup (36X) compared to TLM, while only losing 12% of the gate-level accuracy when doing energy and performance estimation.

As part of the future work, we intend to extend the OS prediction algorithm to work also for a priority-driven scheduling policy and to account for inter-dependent tasks that need to compete for shared resources.

REFERENCES

[1] F. Ghenassia, *Transaction-Level Modeling with SystemC*, 2005.
[2] S. Penolazzi, A. Hemani, and L. Bolognino, “A General Approach to High-Level Energy and Performance Estimation in SoCs,” in *VLSI*, 2009.
[3] Y. Yi, D. Kim, and S. Ha, “Virtual Synchronization Techniques with OS Modeling for Fast and Time-accurate Cosimulation,” 2003.
[4] ——, “Fast and accurate cosimulation of mpsoc using trace-driven virtual synchronization,” *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 2007.
[5] C. Brandolese and W. Fornaciari, “Measurement, Analysis and Modeling of RTOS System Calls Timing,” in *Euromicro*, 2008.
[6] F. Hessel, V. M. da Rosa, I. M. Reis, C. A. M. Marcon, and A. A. Susin, “Abstract RTOS Modeling for Embedded Systems,” in *RSP*, 2004.
[7] M. A. Hassan, K. Sakanushi, Y. Takeuchi, and M. Imai, “Enabling RTOS Simulation Modeling in A System Level Design Language,” in *ASP-DAC*, 2005.
[8] Z. He, A. Mok, and C. Peng, “Timed RTOS Modeling for Embedded System Design,” in *RTAS*, 2005.
[9] “RTEMS Homepage. http://www.rtems.com.”
[10] “AEROFLEX GAISLER. http://www.gaisler.com.”